Security and Mobile Code

Spring 2002

The Problem

- More and more code will arrive from the internet
  - Applets
  - Installation images
  - Mobile agents
  - ...

Unknown code access

- Can an unknown code-snip be allowed to write
  - The screen
  - The printer
  - The modem
  - The network
  - The disk
  - ?

Unknown code access

- Can an unknown code-snip be allowed to read
  - The screen
  - The microphone
  - The keyboard
  - The network
  - The disk
  - ?

Security Issues

- Protect the environment form malicious code
- Protect code from malicious environments

Protecting against malicious code

- Protect against the code gaining unwanted access/control
- Protect against unwanted information flow

Solutions

- Trust
- Sandboxing
- Signing compilers
- Proof carrying code
- Trusted environments

Trust

- The trivial solution is simply to place trust in one or more ”respected” players
- We need to know that the code arises from such a respected player or
- That a respected player verifies the origin of the code
  - Verisign

Trust

- Can we trust code that comes from a trusted player?
- Code that is trusted by a trusted player?
- Code that is trusted by player that is trusted by a trusted player?
- Code that is trusted by player that is trusted by player that is trusted by a trusted player?

Trust

- So we may choose to trust code from a given provider
- But what do we trust
  - That their code is benign?
  - That their code is free from errors that may be exploited by others?

Sandboxing

- Make a protected environment to ensure
  - Qualitative restrictions
  - Quantitative restrictions
- Very similar to a monitor and/or an operating system
  - ”Parental guidens” on hazardous operations

Qualitative restrictions

- No file open
- No format
- No writing to raw devices
- No reading to raw devices
- No execution of protected instructions

Quantitative restrictions

- Don’t eat the whole memory
  - Don’t overflow buffers
- Don’t hog the CPU
- Don’t tie up the network

Requirements for a Sandbox

- Must have reliable access to information about what the program is about to do.
  - e.g., what instruction is it about to execute?
- Must have the ability to “stop” the program

Requirements for a Sandbox cont.

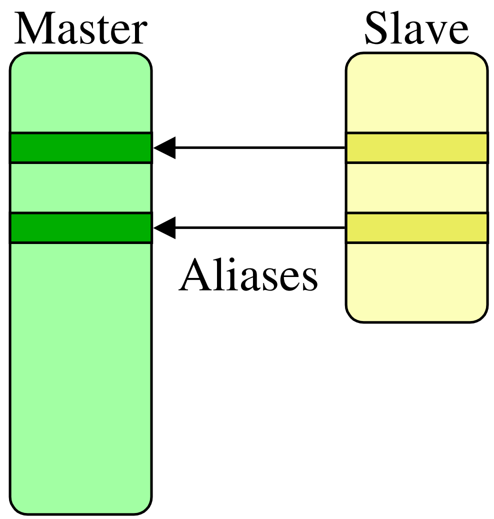
- Must protect the sandbox’s state and code from tampering.
  - key reason why a kernel’s data structures and code aren’t accessible by user code.
- In practice, must have low overhead.

What Policies?

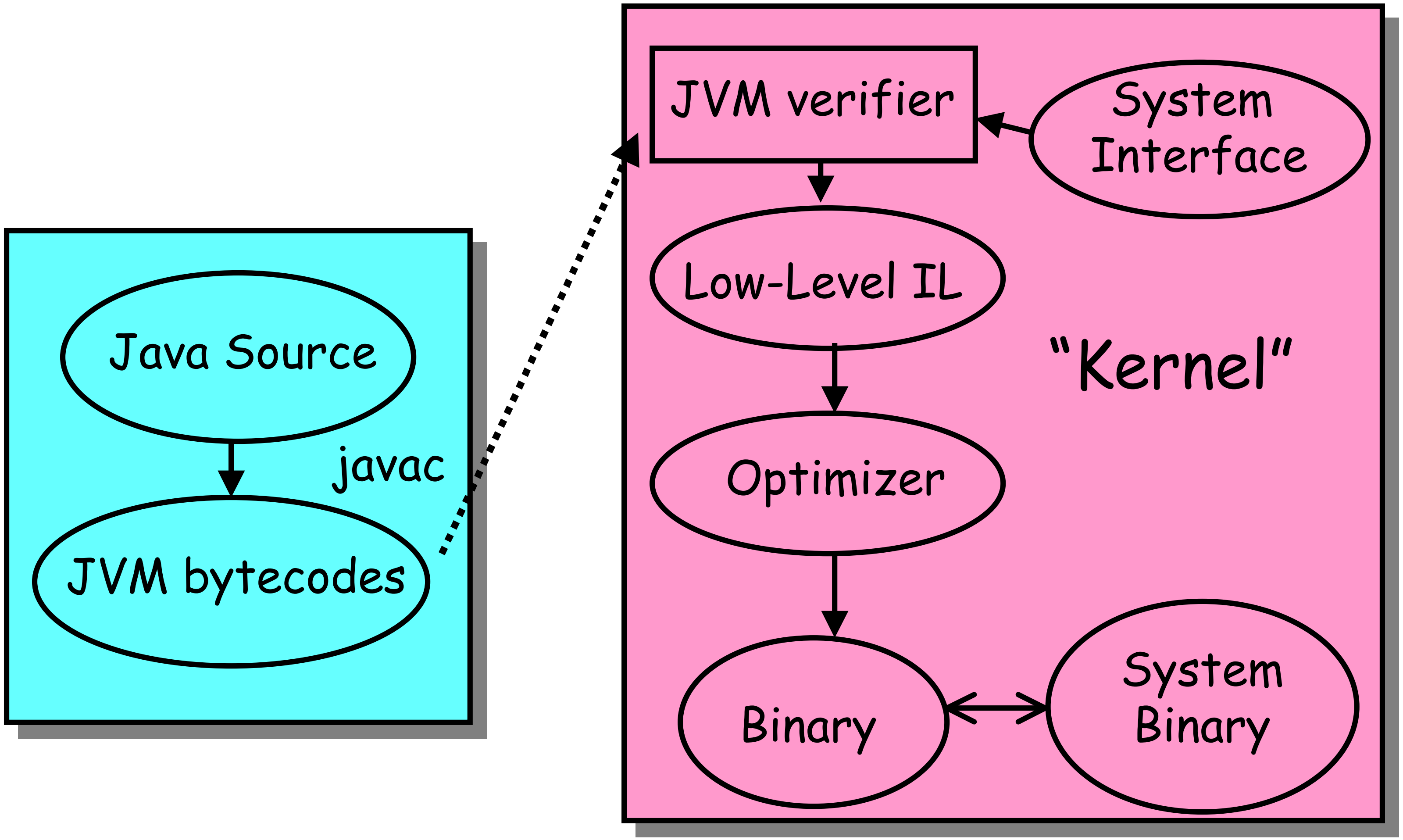
- There's a nice class of policies that reference sandboxes can enforce (safety properties).
- There are desirable policies that no reference sandbox can enforce *precisely*.
  - Rejects a program if *and only if* it violates the policy

Protecting The Environment: Safe-Tcl

- Dual interpreters (like kernel/user space).
- Unrestricted master interpreter (like kernel space) for user and receiving application.
- Incoming Tcl script executes in restricted slave interpreter (like user space): no dangerous commands.
- Aliases transfer control from slave interpreter back to master interpreter (like kernel calls).



Example: JVM



Java Source
javac
JVM bytecodes
JVM verifier
System Interface
Low-Level IL
"Kernel"
Optimizer
Binary
System Binary

JVM Pros & Cons

Pros:

- portable
- hype: $, tools, libraries, books, training

Cons:

- large trusted computing base
  - includes JVM interpreter/JIT
- requires many run-time tests
  - “down” casts, arrays, null pointers, etc.
- only suitable for Java (too high-level)
  - even then, there are mismatches with JVM

Type-Safety

Code is written in a "type-safe" language:

- type safe ~ respects host's abstractions
- Java, Modula-3, ML, Scheme

Host ensures code is legal:

- static checks (e.g., type-checking)
- dynamic checks (e.g., array-bound checks)
  - requires interpreter or JIT compiler

Sandbox Limitations

- Sandbox can have access to entire state of computation
  - But in reality it can’t monitor every transition
- Sandbox can’t have infinite state
- Sandbox can’t guess the future – the predicate it uses to determine whether to halt a program must be computable

Proof Carrying Code

Fundamental idea:

- finding a proof is hard
- verifying a proof is easy

PCC:

- highly optimized code + proof that code respects the host’s integrity.

Advantages of PCC

- Simple, small, and fast trusted computing base
- No external authentication or cryptography
- No need to insert additional run-time checks
- “Tamper-proof”

Disadvantages of PCC

- Still no fully automated proof-generator
- Security is very sensitive to implementation errors in the proof verifier
- Can we test for quantitative measures?

Signed compilers

- We can choose to trust a compiler to generate only non-malicious code
  - All array-accesses are checked
  - No casting of types are allowed
  - No pointers per se – but perhaps references
    - Access to references must be checked that they are in fact of the correct type

Extreme trust

- Extensible operating systems
- SPIN
  - User code may be included in the kernel if a trusted compiler has signed that this code is well-behaved
  - Uses Modula-3
  - Extensions are given privileges using capabilities
  - Provides very high performance

Protecting the Code

- Cannot protect a program from its processor.
- Must authenticate the engine.
- Total system authentication:
  - Each software layer authenticates the layers above it.
  - Boot ROM has secret key.
  - Boot ROM authenticates OS.
  - OS authenticates interpreter.
  - Scripts can then trust the interpreter.
- Simpler solution: trusted sites.

End-to-end

- Most security risks out there aren't because of incorrect protocols – but because of incorrect implementations!!!
  – If you're implementing a security mechanism don't start with main() ☺
- Most security models are state-machines
  – So use the tools for automated test-vector generation!!!