Java

- Java is a type-safe language in which type safety is security-critical
- Memory safety: programs cannot fabricate pointers to memory
- Encapsulation: private fields, methods of objects cannot be accessed without using object operations
- Bytecode verifier ensures compiled bytecode is type-safe

Java stack inspection

- Java goal: execute untrusted code on same machine, address space as trusted code
- Early Java security model based on “sandbox” model
  - Applets isolated from each other (sort of) by inability to name each others’ classes
  - Access mediated by capability model
  - Need type safety + inability to generate arbitrary object refs (enforce encapsulation)
  - Hard to apply applet-specific security policies, and capabilities leak
- Stack inspection intended to fix it...

Objects as capabilities

- Single Java VM may contain processes with different levels of privilege (e.g. different applets)
- Some objects are capabilities [DV66] to perform security-relevant operations:
  ```java
  FileReader f = new FileReader("/etc/passwd");
  // now use "f" to read password file
  ```
- Original 1.0 security model: use type safety, encapsulation to prevent untrusted applets from accessing capabilities in same VM
- Problem: tricky to prevent capabilities from leaking (downcasts, reflection, …)

Security operations

- Each method has an associated protection domain
  - e.g., applet or local
  - `doPrivileged(P) {S};`
    - fails if method’s domain does not have priv. P.
    - switches from the caller’s domain to the method’s while executing statement S (think setuid).
- `checkPrivilege(P)` walks up stack S doing:
  ```java
  for (f : pop(S); !empty(S) ; f := pop(S)) {
    if domain(f) does not have priv. P then error;
    if f is a doPrivileged frame then break;
  }
  ```
- Very operational description! But ensures integrity of control flow leading to a security-critical operation
Example

Font Library:

\[\text{doPrivileged(ReadFiles)}\{\text{load("Courier");}\}\ldots\]

FileIO:

\[\ldots\text{checkPrivilege(ReadFiles);}\text{read();}\ldots\]

Requires:

- Privilege enabled by some caller (applet can’t do this!)
- All code between enabling and operation is trustworthy

Some pros and cons?

- **Pros:**
  - rich, dynamic notion of context that tracks some of the history of the computation.
  - low overhead, no real state needed.
- **Cons:**
  - implementation-driven (walking up stacks)
  - policy is smeared over program
  - possible to code around the limited history
    - e.g., by having applets return objects that are invoked after the applet’s frames are popped.
  - danger of over/under-amplification

Logic model

- Paper: uses ABLP authentication logic to describe stack inspection
- Code, stack frames, targets represented by principals
- Logic: principal P can speak for P' (P = P') and can say things
  - Models relationship between code signer, code:
    - \( K_{\text{owner}} \Rightarrow \text{Signer} \)
    - \( K_{\text{owner}} \Rightarrow \text{Code} \Rightarrow \text{Signer} \)
    - Frame \Rightarrow \text{Code}
    - Frame \Rightarrow \text{Signer}
  - Models relationships between principals and groups
  - Models relation between targets (macro targets, implies)

Reasoning procedure

\( E_{F} \) is environment of frame F:

- Frame credentials \( \Phi \) established by code signing
- Belief set \( \mathcal{B}_{F} \) from enablePrivilege(...) calls
- Access matrix \( \mathcal{A}_{VM} \) expressed as set \( P \Rightarrow T \)

Result: success of stack inspection implies existence of ABLP proof of \( E_{F} \Rightarrow \text{Ok}(T) \) for target T

- If we have \( F_{1} \Rightarrow F_{2} \Rightarrow \ldots \Rightarrow F_{k} \Rightarrow \text{Ok}(T) \)
  - via \( \mathcal{B}_{F_{i}} \)
  - And \( F_{i} \Rightarrow T, 1 \leq i \leq k \)
  - via \( \Phi(F_{i} \Rightarrow P), \mathcal{A}_{VM}(P \Rightarrow T) \)
  - derive T says Ok(T)

Security-passing style

- Idea: do reasoning ahead of time, pass authorizations or belief set down the stack
  - no special JVM support needed
  - permits more compiler optimization via dead-code elimination, inlining, tail calls?

Stack inspection over RPC

- Idea: use security-passing style to support stack inspection across RPC
  - Send belief set with remote call
  - Beliefs are “said” by caller, i.e. signed by \( K_{\text{CVM}} \)
  - Receiver gets
    - \( K_{\text{CVM}} \Rightarrow K_{1} \Rightarrow \ldots \Rightarrow K_{k} \Rightarrow \text{Ok}(T) \)
    - where \( F_{i} \Rightarrow K_{i} \) and \( K_{i} \Rightarrow P_{i} \Rightarrow \ldots \Rightarrow T \)
  - Effect: beliefs from untrusted machine are ignored
  - Equivalent to distributed stack walk?
Some questions

• Is this a useful formalization?
• disablePrivilege = revocation?
• What doesn’t this do?
• Is security-passing style an optimization? Can we do better?
• Is proposed RPC mechanism flexible enough?