CS 711

Advanced Programming Languages Seminar

Language-Based Security and Information Flow

Understanding Stack Inspection

Fall 2003

Andrew Myers

Cornell University

www.cs.cornell.edu/courses/cs711

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 in[tro]spection 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:

```
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, …)

Java Stack Inspection

- Dynamic authorization mechanism
  - close (in spirit) to Unix effective UID
  - attenuation and amplification of privilege
- but with a richer notion of context
  - operation can be good in one context and bad in another
  - Operations represented by *targets*
  - E.g: local file access
    - may want to block applets from doing this
    - but what about accessing a font to display something?

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:

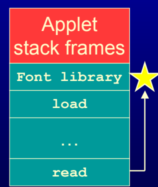
```
for (f := pop(S); !empty(S) ; f := pop(S)) {
  if domain(f) does not have priv. P then error;
  if f is a doPrivileged frame then break;
}
```

- Very operational description! But ensures integrity of control flow leading to a security-critical operation

Example

```
Font Library:
  ...
  doPrivileged(ReadFiles) {
     load("Courier");
  }
  ...
```

```
FileIO:
  ...
  checkPrivilege(ReadFiles);
  read();
  ...
```



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′ ($P \Rightarrow P'$) and can *say* things
  - Models relationship between code signer, code:
    - $K_{signer} \Rightarrow$ Signer
    - K_{signer} says Code $\Rightarrow$ Signer
    - Code $\Rightarrow$ Signer
    - Frame $\Rightarrow$ Code
    - Frame $\Rightarrow$ Signer
  - Models relationships between principals and groups
  - Models relation between targets (macro targets, implies)

Reasoning procedure

- $\mathcal{E}_F$ is environment of frame F:
  - Frame credentials Φ 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 $\mathcal{E}_F \supset$ Ok(T) for target T
  - If we have F_1 says F_2 says…F_k says Ok(T)
    - via $\mathcal{B}_F$
  - And $F_i \Rightarrow T$, $1 \le i \le k$
    - via Φ ($F_i \Rightarrow P$), $\mathcal{A}_{VM}$ ($P \Rightarrow T$)
  - 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_{CVM}
  - Receiver gets

    K_{CVM} says K_1 says…says K_k says Ok(T)

    where $F_i \Rightarrow K_i$ and $K_i \Rightarrow P_i \Rightarrow \overset{\mathcal{A}_{VM}}{\ldots} \Rightarrow T$
- 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?