CS261 Notes: Extensible Security Architectures for Java

Lecturer: David Wagner Scribe: Shoaib Kamil

10 October 2007

This paper is essentially about access control. In particular, it is about using languages and runtimes for access control. The authors wish to use the Java VM as somewhat of an OS, to allow use of apps/code from different users/sources/etc, and have the Java VM provide protection using the type safety of the language as well as software-based methods instead of hardware. The thinking at the time was that the browser would be a "platform-independent" OS, with applications written in Java, each running from different sources and with different amounts of trust.

The goals of the work were 1) Language-based access control and 2) fine-grained access control, and to make it easy to follow the principle of least privilege. In particular, the goal was to make it easier to build privilege-separated applications in Java: each piece of an app would run in a different security domain, each with only necessary privileges.

1 Access Control In General

To review, the basic undergrad version of access control: you have principals (e.g. Alice) and resources (e.g. a printer). Administrators set the policy, and on the system there is a mechanism to check the authorization policy and enforce it.

This conceptualization is overly simplistic. A more complex model: Alice *uses a program* that outputs to printer (`lpr`). By convention in (a Unix-like) OS, `lpr` runs with process privileges of Alice. The OS sees a request from `lpr`, and realizes that the program is running with the privileges of Alice, then looks up to see if Alice has authority to print. But this model is flawed as well, because it is not just one piece of code that Alice is running.

An even more complex model: Alice → shell → `lpr` → printer. So Alice yields privileges to `sh`, which yields to `lpr`, and so on, until the OS checks the permissions like above.

So have to trust all the links in the chain to the printer, if they all get the same privilege. However:

- we could reduce the privilege granted to `lpr`.
- we could use fine-grained access control to assign just enough privilege as required
- we could be really careful and not run anything we don't completely trust

But these models are still too simplistic. In previous models, codes all have the same amount of privilege. It gets more interesting (and realistic) when codes that interact have different levels of privilege: Alice → `sh` → `lpr` → `lpq` → printer, where `lpq` has higher privileges, for example, so Alice can't directly print. Going from `lpr` to `lpq`, there is a crossing of trust boundary. Issue: `lpr` might attack `lpq`. The Unix answer is "be really careful when writing `lpq`," that is, it doesn't help you do anything about such attacks. In Unix, privilege is coarse-grained so `lpq` will have much more privilege than necessary.

In addition, more privileged code can call code with less privilege. Issue: luring attack. This may only occur in Java— not so much in Unix, perhaps because of the lack of callback-type mechanisms in Unix.

2 Unix Privilege Management

Most code runs with `euid=alice, suid=root`. But when the system wants to temporarily elevate privilege, it executes something like `seteuid(0)`...(critical section)....`seteuid(getuid())`. These roughly map to `enablePriv()` and `disablePriv()` calls in Java, but with some significant differences. In Unix, forgetting to `seteuid(getuid())` (i.e. forgetting to de-escalate privilege) will make the rest of program run with elevated privilege. In Java, however, the privilege flag is per stackframe, so when you pop off a frame (i.e. return), it logically inserts `disablePriv()`.

3 Paper

The general policy in Java is that applets get NO privileges, and applications (from local disk) get all privileges. The paper adds 2 things to each stack frame: 1) classloader (which is the label of the code's security) and tells what the privileges this code has and 2) a bit for each privilege that logically is 1 if the privilege has been enabled, and 0 if not.

The basic algorithm:

```
checkpermission(p):
for each stack frame F:
    if classloader(F) forbids p:
       return DENY
    if p enabled in F:
       return ALLOW
return DENY.
```

The paper does the checks in the wrong order! This is a bug in the paper, but the implementation did it correctly.

Microsoft decided to default to ALLOW, while Netscape (more securely) defaulted to DENY. Microsoft's version was backwards-compatible, but Netscape's was more secure.

Disadvantages of the paper's approach:

- doesn't work too well for event-driven code
- cannot safely do function inlining
- cannot perform tail-call optimizations safely

The paper has been very successful because this approach is now integrated into every JVM. However, very few programmers actually use these API calls. If they were followed, one could easily write code that uses the P.O.L.P. If programmers used it, would be useful in code reviews for identifying what was privileged code and needed to be audited.

3.1 Namespace Management

This was a flawed strategy. For example, there seems to be no way to drop privilege. One could have a class loaded in that was actually subclass of class requested (e.g. load "restricted file" instead of File). The problem is that once a class is loaded, it can't be unloaded. So if we load File, can't load restrictedFile. Another problem was problems with subclassing: Java doesn't support multiple inheritance which makes some of the subclass relationships not work.