Language-Based Security

Scribe: Sam Kumar

In this lecture, we discuss the design of programming languages as it relates to security, and in particular to access control. To motivate this discussion, we begin by reviewing traditional approaches to access control.

1 Motivation for Language-Based Security

Access-control lists (ACLs) are a traditional way to enforce permissions for access control. A trusted entity (e.g., operating system) knows which principals (e.g., users, programs) have access to which resources (e.g., files).

One problem with ACLs is that they do not keep track of *why* each principal was granted a certain set of permissions. This allows a malicious entity to misuse permissions, i.e., use permissions for purposes other than those for which they were intended.

For example, consider the following scenario. The owner of a server offers the server as a computation resource to many users. The system has a compiler installed, and the owner of the system wants to bill users according to how much they are using the compiler. To this end, the compiler appends an entry to a log file, `/var/billing.log`, every time a user compiles a file. At the end of every month, the owner of the system bills users according to the entries in `/var/billing.log`.

To protect against malicious users, the system owner does not give users direct access to `/var/billing.log`. Instead, the system owner gives the compiler program, `cc`, permission to open and write to `/var/billing.log`. The idea is that users can only open the file `/var/billing.log` through the compiler—which is trusted to only modify the file in a correct way.

To make this concrete, when the user executes the command

```
$ cc foo.c -o foo.out
```

the operating system sees three `open` system calls: `open("foo.c")`, `open("foo.out")`, and `open("/var/billing.log")`. The operating system allows the first two system calls because the user has read permission for `foo.c` and write permission for `foo.out`. The operating system allows the third system call because the compiler has permission for `/var/billing.log`.

If the user were to execute the command

```
$ echo "Hello, world!" > /var/billing.log
```

then the operating system will see the system call `open("/var/billing.log")`, and deny it because the call was made by the shell, not by the compiler.

However, a malicious user could issue the following command:

```
$ cc foo.c -o /var/billing.log
```

In this scenario, the operating system will see three `open` system calls: `open("foo.c")`, `open("/var/billing.log")`, and `open("/var/billing.log")`. When the operating sees the second system call, it allows it, because it is being executed by the compiler. As a result, the program binary is written to the log file, erasing the history of how much each user has been using the compiler.

This sort of vulnerability is known as the *Confused Deputy Problem*. The compiler is a privileged entity that acts as a deputy for the user, regulating access to `/var/billing.log`. In this example, the user

managed to trick the deputy into accessing `/var/billing.log` for the wrong reason. CSRF attacks on web applications are another example of the confused deputy problem.

One way to fix this problem would have been for the compiler to pass to the operating system the reason for opening a file, and operating system to validate system calls using both the operation and the reason for that operation. Rather than making system calls like `open(filename)`, the compiler would make system calls like `open(filename, powergrant)`, where `powergrant` describes the permission that should be used to validate the system call (in this example, either `USER_PERMISSION` or `EXECUTABLE_PERMISSION`). This approach works because it gives the operating system more information regarding resource access. It allows it to validate not only whether the resource could be accessed for a valid reason, but whether it is being accessed for the *correct* reason.

In order for such an approach to be feasible, however, the application must pass around filenames and powergrants together, as single units. That way, when it comes time to issue a system call, it is clear what permission (or permissions) should be used to justify that action. Adding primitives in the programming language to handle security can make writing code like this less cumbersome, and can help identify security holes at compile-time. This motivates us to study language-based security.

2 Exercise: Sandboxing an Existing Program

Another important aspect of Language-Based Security is securing existing applications. Ideally, it would be easy to restructure existing applications to be more secure, not just to write secure applications from scratch. To explore the challenges in this space, we discuss some of the challenges in sandboxing existing programs.

Suppose you are trying to sandbox a program like Emacs. Specifically, consider its use of the filesystem. You would like to allow Emacs access to the file(s) being edited, as well as `/tmp/*`, `/.emacs/*`, etc. In order to implement this, you have the ability to intercept system calls made by Emacs and do your own processing, making system calls to the operating system as part of that processing.

For standard system calls like `open` and `unlink`, you need to check if the program is permitted to modify that file. There are some fine points—for example, one must take care to handle relative paths, especially ones with `..` components, correctly.

A trickier case has to do with symbolic links. There are two approaches we could take:

1. Do not let the program make symlinks to files they do not know; let the program follow existing symlinks, if the symlink is in a directory to which the program has access.

2. Allow program to make symlinks, but verify that they have permission to see the destination file before following the symbolic link.

Unintuitively, it turns out that the first approach is seriously flawed. Suppose the application creates a file at `/tmp/etc/passwd`, which the application *can* do because it has full access to `/tmp/*`. Then, it creates the symlink `/tmp/d1/d2/l`, which points to the relative path `../../etc/passwd`. This is a valid symlink, because it points to the file `/tmp/etc/passwd`. Then, the application renames the directory `/tmp/d1/d2` to `/tmp/d3`. Now, the symbolic link points to `/etc/passwd`, which the applications should not have permission to see. Therefore, a malicious or compromised application could abuse symbolic links to access files that they should not be able to access.

The second approach is also tricky. A simple approach is to get the filename when an application makes an `open` system call, check if that file is a symbolic link, and verify the destination before making the system call to the operating system. However, this has a *time-of-check-to-time-of-use* (TOCTTOU) vulnerability; the file could change in between checking if it is a symbolic link, and passing the call to the operating system.

Luckily, it is possible to open a file with the flag `O_NOFOLLOW`, which will cause the system call to fail if the file is a symbolic link. So, after intercepting the application's `open` system call, the intermediate layer checks if the specified file is a symbolic link. If it is, it gets the destination and opens it manually. If not, it makes the `open` system call with `O_NOFOLLOW`, and repeats the whole process if the call fails.

There is one more problem: components earlier in the filepath may be symbolic links, but `O_NOFOLLOW` only causes the operating section to check the last component of the filepath. For example, if the filepath is `a/b/c`, and `b` is a symbolic link, the operating system will still follow `b` when resolving the filepath, even if `O_NOFOLLOW` is set. Therefore, the intermediate layer has the open the directories in the filepath one at a time, using `O_NOFOLLOW` each time to ensure that symbolic links are properly validated.

This example shows some of the subtleties in sandboxing an existing application.

3 Approaches to Language-Based Security

Here, we discuss two instantiations of Language-Based Security. First, we discuss Joe-E, a subset of Java that is designed to make it easier to reason about access control and permissions within different modules of a single program. Then, we discuss JFlow, which extends the Java language to statically analyze information flow to ensure that sensitive data is not leaked. The content on JFlow is based on a presentation in class by Wen Zhang. An important feature in both of these papers is that they are based on Java, a language used to build real systems that has seen wide adoption. While there have been existing approaches that build languages from scratch, these approaches are novel in that they build on an already-existing, widely used language, lowering the barrier to adoption in real systems.

3.1 Joe-E: A Security-Oriented Subset of Java

The combination of a resource (e.g., an open file) and a set of permissions governing its use is called a *capability*. Joe-E is a security-oriented subset of Java, which supports a programming paradigm in which each object represents a capability. A reference to an object represents permission to use that capability.

For example, resources such as open files or open network sockets are encapsulated in objects. Furthermore, objects may be nested in order to restrict permissions. For example, given an object that allows read/write access to a file, one could create a wrapper object that only allows reading, not writing. This allows objects to restrict permissions in programmatically interesting ways.

A module of the program that is not given access to a reference to an object cannot access the resource that it represents. This is important for two reasons. First, it makes it possible to easily perform security review on a specific module. Privilege is passed around explicitly via object references, making it easy to track the flow of trust. Furthermore, it makes it possible to securely use untrusted code; even if the code is malicious or has bugs, it can only use resources explicitly given to it.

In order to enforce this, Joe-E removes features from the Java language that would allow malicious code to circumvent the security properties of an object, such as finalize(), reflection, mutable static variables, mutable exceptions, catching of errors, and "finally" clauses. It also restricts the standard library, so that all permissions are granted via the object-capability model. Normally, any piece of Java code can invoke the standard library to open any file, open any socket, etc.; in Joe-E, a module must receive a reference granting a capability in order to be able to use it.

In this sense, passing a reference to an object to another module gives that module permission to use the encapsulated resource.

3.2 JFlow: Practical Mostly-Static Information Flow Control (based on Wen Zhang's presentation)

Joe-E enforces capability discipline, which regulates how code can interface with outside resources. In contrast, JFlow enforces how code can manipulate data. JFlow allows variables to be marked as secret via *label annotations*, and statically analyzes the program to track the use of such variables, to ensure that they are not leaked outside of the program. This is useful to ensure that sensitive information (e.g., a string containing a plaintext password) is not leaked. For example, if one uses a library (untrusted) to manipulate strings, JFlow can be used to statically ensure that sensitive data passed in to the library is not leaked through some hidden channel. Whereas the Trusted Computing Base (TCB) would normally contain this

library (library is trusted not to leak data passed to it), JFlow shrinks the TCB. With JFlow, the TCB consists of only the label checker, compiler, and static analyzer.

Label checking is done conservatively. Assignment of a variable transfers its labels. When multiple variables are combined (e.g., by arithmetic), the result has the union of the labels of the operands. A program counter label is used to track dependencies resulting from control flow (e.g., loops, branches).

One final point is that sometimes, restrictions need to be relaxed. For example, if data is encrypted, the labels from that data can be removed, because it is OK to send the encrypted data over the network. To account for such cases, JFlow provides a way to relax permissions. Label annotations consist of a policy, readers, and an owner. Code running with the owner's permission can declassify labels owned by that principal.