CSE409, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, October 29th, 2011

Principles of secure system design

Separation of privilege

Some **examples** of separation of privilege:

- When you two people need to turn their keys in order to launch a missile
- MIX nets for electronic voting systems. The MIX takes an array of votes, shuffles them and peels off a layer of encryption and sends it to the next MIX. In voting systems, each party (democratic, republican) has a MIX.
- Workflows for document declassification in the NSA or military

Open design

"With many eyes, all bugs are shallow."

There is some evidence to support this, specifically a paper called **"Fuzz revisited,"** which tested the software quality in a lot of UNIX programs by feeding them random input. The authors would pick some common UNIX program like `ls` or `cd` and feed it random input and see what happened. They discovered that the open source tools fail about 13% of the time and the commercial tools failed about 25% of the time.

Even with open design, some security flaws can creep in and remain unobserved for long periods of time. For instance, the **SSL bug** was present for over 10 years in the source code which has been extensively studied and no one noticed the flaw.

Open design is sometimes used to encourage the community to come up with designs and try to break each one to see if a winner emerges. The **DES replacement crypto contest** did this. Various encryption schemes were proposed, people tried hacking all of them and AES remained the winner.

Note: For open design to be effective, you need to have the skilled people or some knowledgeable community to look at your code or design.

Psychological acceptability

Imagine a corporate network, a smart developer and a firewall between him and the internet. >:) That smart developer will feel very restricted so he might dial out over the phone using a modem to get around the firewall thereby compromising the security of the corporate network.

Moral of the story: Security measures can be too restrictive and people will try to get around them because it inconveniences them, creating security holes.

Security measures can be too difficult to use. There was a paper back in the day called **"Why Johnny can't encrypt"** where the authors asked a bunch people to send an encrypted email using PGP. It turns out that the majority of people were unable to successfully do so, because of user interface problems.

Funny fact: There is now a new paper called **"Why Johnny still can't encrypt."**

Password Hash, a plugin for Firefox, allowed you to type in a special/master password instead of your actual password when you're visiting a webpage. However, people were very confused about using it.

Now people are looking into finding an authentication scheme that would allow a user to compute a function on his password and on a challenge sent from the server, and send the result as an authentication token, rather than sending his password.

Client (has secret x)	Server
send $username$ to server	
	send challenge c to client
compute $p = f(c, x)$	
send p to server	
	checks p from client to see if it matches $p' = f(c, x_{user})$

The problem is finding an easy enough hash function f that a human can compute but no computer can break easily is an open research question.

Static analysis for security

Static analysis for security is a research area where people are trying to develop very **smart compilers** that will warn you about bugs in your code.

Sandboxing and Intrusion Detection Systems

On top of designing systems as secure as possible, there is a need to also focus on **catching and preventing exploitation at deployment time**. There are interesting projects in the area of **sandboxing** and **intrusion detection systems** (IDS) that deal with this.

Example: The Google Native Client (will be part of Google Chrome browser) allows you to download x86-code and execute it on the system without letting it take control of your system.

The problem is that you've got a process and you want to prevent it from destroying your system. There are two possibilities:

1. You could **trust the process**, but it might be buggy.
    a. Firefox might have a bug, you trust it, but you'd still like to protect yourself
2. You might **not trust that process at all.**
    a. You have a phone, you're downloading an app, but you don't trust it

The idea is to put the process in some sort of a **containment chamber** that won't let it do any bad things.

If the process is trusted, you should get a **specification** from the program which tells you what the program needs to be able to do. You can then set up the containment chamber to allow the program to do all the things it needs to do and nothing else.

With untrusted programs, you cannot trust the program to tell you what it needs to be able to do, so **the specification should come from you.** The machinery for enforcing the specification is the same, but the source of the specification is different depending on whether you trust the program or not.

What should we have in a specification and how can we enforce it?

We need to have a way to describe the allowed "behaviors" of the application.

CSE409, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, October 29th, 2011

If an application will do any damage whatsoever, then it **has to make a system call** (syscall) of some sort. Otherwise the process just sits there messing with its user-space data and does not interact with the rest of the system or with other applications. *"Who cares? Write to your own memory."* The choke point is in the syscall.

"behaviors" = actions = system calls

With victim one and two, if **the OS knew that those programs should never do an `execve` call**, then it would not allow it, and our exploits would not work.

One **simple model** is to let the specification be the set of system calls the application is supposed to make. Unfortunately, for large programs, the set of system calls it makes gets huge so this specification will not help that much, since the attacker will have plenty of options in terms of the allowed syscalls he can make once he breaks the program.

The next step up is to **look at the sequences of system calls**.

- `strace` is an awesome tool for UNIX, which will run a program and print every syscall the program makes

Question: How can you build a model, since the syscall sequence will differ as you execute the program with different input?

Answer: You can take sliding windows of size 6 from the syscall sequence. You take the sequence and you take the first window of 6, the second, the third, and so on. As a result, you get a whole bunch of sequences of 6 system calls. You collect all these **sixgrams** (n-grams for a sequence of size n) as you execute the program multiple times and you build a set of all the possible ones (approximation really).

When the program makes a syscall, the IDS takes the last 5 calls made and the current one and tries to find a matching sixgram.

Mimicry attacks

The attacker now knows he only has a few options for a syscall. **Should he give up? No.** If an attacker has the source code (like Apache) and the six-grams for the program, then:

- he will know what the last 5 syscalls were before his attack (since he can look at the code)
- he will also know what are the next calls that the IDS will allow.
- he just needs to fine tune his attack, by making a few **dummy syscalls**, until he gets into a state from which his target syscall can be made (it is allowed by the IDS)

This is called a **mimicry attack**.

You can build an automaton, each state represents the five previous syscalls and a transition is labeled by the allowed syscall and takes you to another state with the "new" five previous syscalls. The resulting graph models the syscall states and the allowed transitions between them. Therefore the attacker can **find a path in the graph that will allow him to get to a state that will let him execute his syscall**, that is, there will be a transition from that state to another state labeled with the target syscall.

CSE409, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, October 29th, 2011

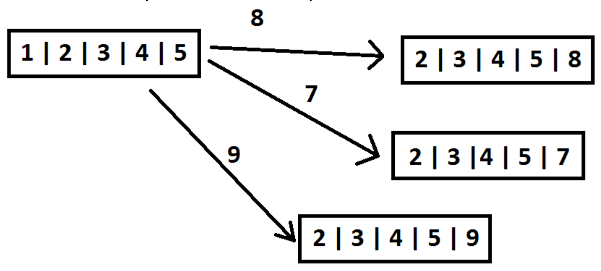


Figure 1: Graph that models the allowed syscall sequence of the program

Downside: Sometimes you need to make 300 dummy calls to get to the state where you can make the syscall you want.

Note that the n-grams will not fully cover all the possible syscalls. There could be some special error path that does not occur when you build the sixgrams. In this case the IDS will complain to the user he's under attack when in fact the program is just taking a different path.

Therefore it would be nice to **construct a model that will not throw false alarms**.

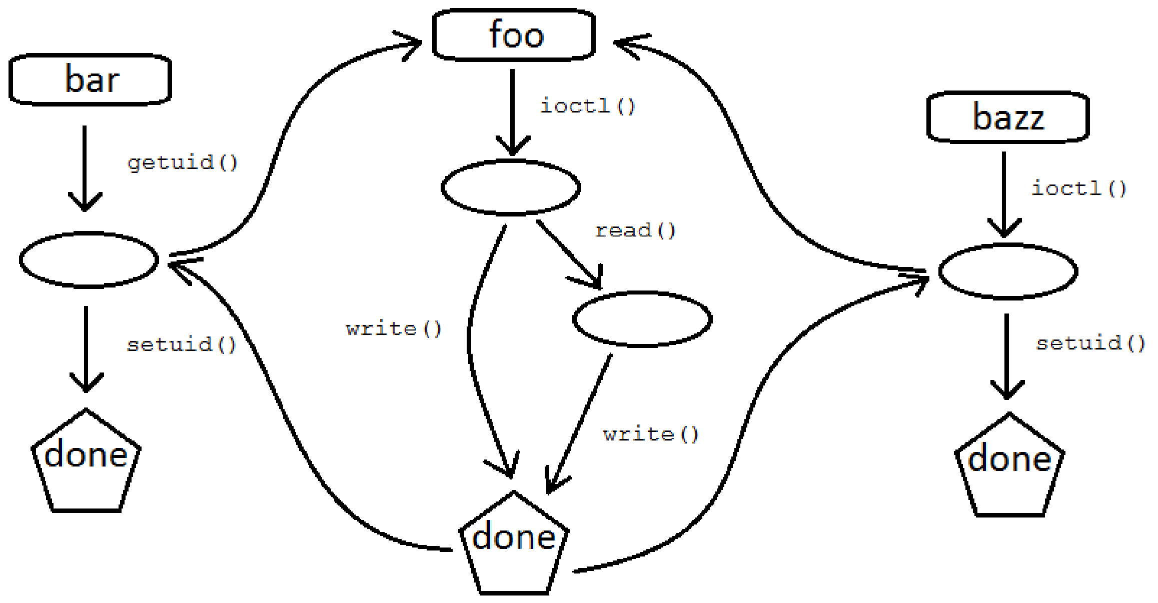
Assuming we have access to the source code we can build a graph that better models the syscall sequence. The main idea is that by looking at and analyzing the source code you will know how the program behaves at any time.

Consider the following code:

```
void foo()
{
    int fd = open();

    if(fd < 0)
    {
        write(fd);
        return;
    }

    read(fd);
    write(fd);
}

void bar()
{
    getuid();
    foo();
    setuid();
}

void bazz()
{
    ioctl();
    foo()
    setuid();
}
```

You can build a **control flow finite automaton**, where each transition is a syscall and takes the program into a different state

CSE409, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, October 29th, 2011



To solve the ambiguity between the `bar`/`bazz` return paths, you can push a 1 when you go into `foo` from `bar` and push a 2 when you go into `foo` from `bazz`. Then you can look at the stack when you exit `foo` to tell which transition to take out. This is a slowdown though.

Note that **mimicry attacks** are still possible even in this model.

Thought: `foo` does a write after `open` only if `open` fails. The OS knows when `open` failed so maybe you can leverage the OS knowledge about syscall failures and add this information to your PDA to incorporate it into the transitions.