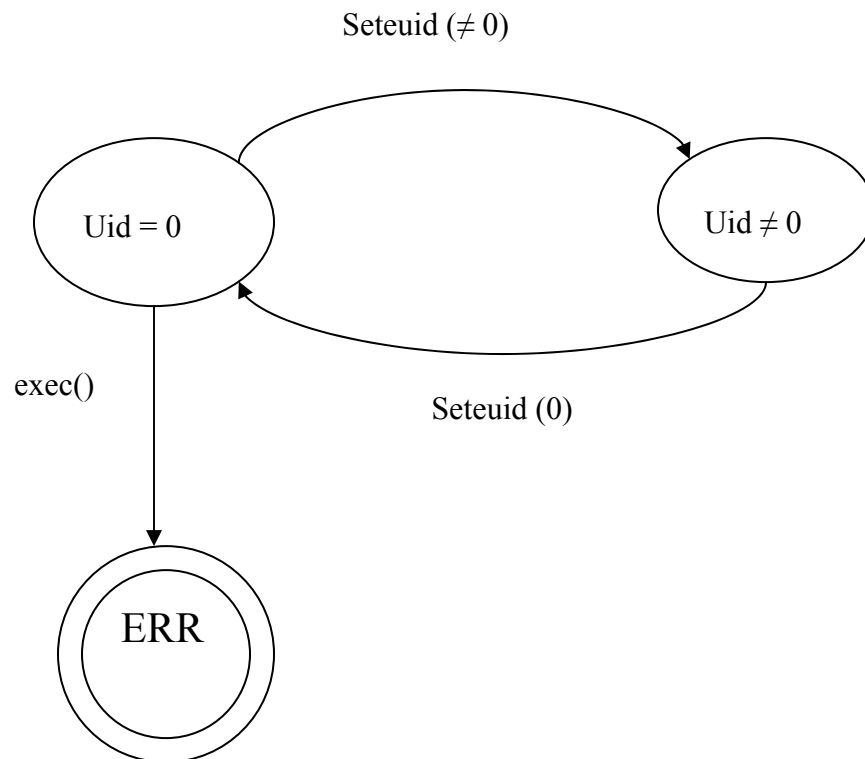


LECTURE NOTES FOR 03/16/2006

STATIC ANALYSIS OF PROGRAMS

Last time the lecture was about Model Checking, and we described bugs and mistakes, like the `seteuid()` and `exec()` mistakes, in some code samples. We ended up by representing each type of mistake that can happen by a State Machine. So, for example, a State Machine for the program containing `seteuid()` and `exec()` commands looks as follows:



(The above figure shows that we have 3 states for the program, namely the state when the user is given root privileges, (`Uid = 0`), the state when the user has non root privileges, (`Uid ≠ 0`) and an error state. The program switches from one state to another by the execution of certain commands. The above figure shows that there is a bug in the program if we have root privileges (, i.e., we are in the state: `Uid = 0`) and we execute the “`exec ()`” command.)

Definition for Model Checking:

Model Checking is a technique, in which we make a State Machine that represents the different states of the program, and that shows the commands that cause a transition from one state to another. After this, we try to look for bugs in the program by checking for the paths that would cause the program to go to an error state.

Example:

Let us look at the following example:

```
void func()
{
1.   while (.....) {
2.       if (.....) {
3.           seteuid(geteuid());
4.           exec(...);
5.           seteuid(0);
5.5          } // end of if
6.       else {
7.           exec(...);
8.           } // end of else
9.       } // closes while()
10. } // Closes func()
```

Assume that code starts with `euid = 0`

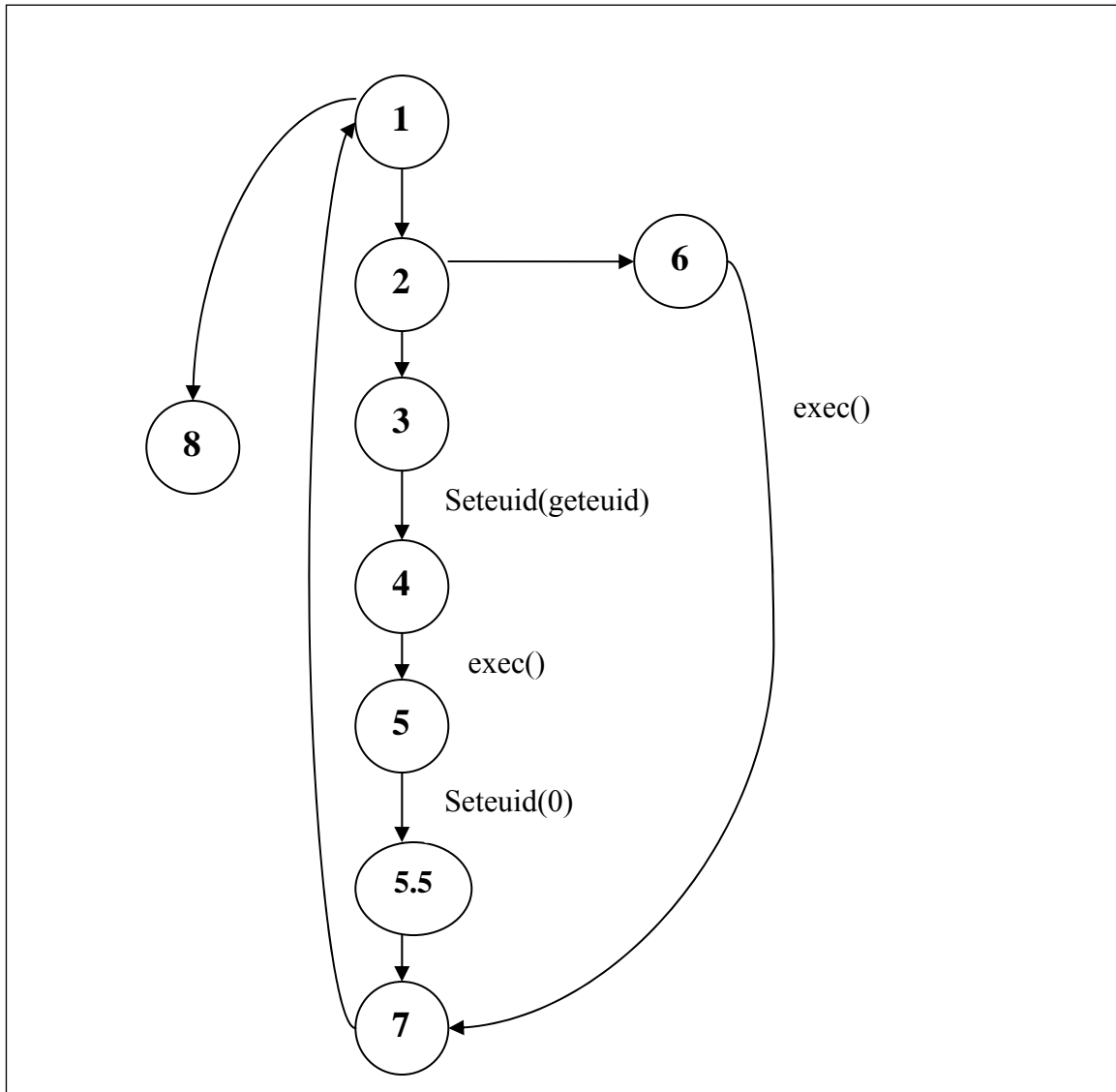
Question) Does this code violate the property shown by the above State Machine?

Answer) Yes. Why? Because, there is a path in the function, which can call “`exec()`” when the user has “`root`” privileges. (If the program fails the “`if`” condition and takes the else “`branch`”, then we end up in an error state.)

So, we can detect this automatically by making a “`Control Flow Graph`” of the program we are interested in.

Control Flow Graph (CFG):

Let us make a CFG of the above given code.



Important Information about CFGs and Security Property:

- CFG is a Finite State Machine that gives a regular language L_{CFG} .
- Security Property is specified as a Finite State Machine also, hence from this we get the Language "Lp." (In our example, this FSM is the very first diagram.)
- Both L_{CFG} and Lp are regular.

- The regular language obtained from the CFG, represents the sequence of statements that the program might execute.
- The Regular language we get from the FSM that represents the Security property, are sequences of statements that must not be executed.

Hence, this leads us to the following:

Requirement That We Would Like to Check:

$$\text{We want: } L_{CFG} \cap L_p = \{\} \dots\dots\dots (1)$$

The above property means that there is no sequence of statements that is bad and would be executed by the program.

Now, since there are infinite number of strings acceptable by the FSM that represents security, therefore we might think that it is very hard to check for (1).

Hence we need to make use of the following:

$$L_{CFG} \cap L_p \text{ is regular, because both the languages are regular.}$$

We can get the intersection of the above two regular languages by making a product FSM from the FSM's that are related to the above two languages.

Algorithm For Model Checking:

1. Specify security property as FSA.
2. Convert input program into FSA.
3. Compute product of machines from 1 and 2.
4. Check if product machine gives empty language.

(Please note that in step 2, we can also use "Push Down Automatas" instead of "Finite State Automatas". The tool MOPS uses PDA instead of an FSA)

Tools Based On the Above Basic Idea:

- MOPS
- SLAM
- BLAST
- MECA

Some Definitions:

1) Soundness:

A program analysis tool is sound if whenever it says a program doesn't have

bugs, then the program doesn't have bugs.

2) False Positive:

False Positive is when an analysis tool claims that a program has a bug, but it doesn't.

3) Completeness:

A complete analysis tool has no false positives.

Theorem:

We can only have 2 properties from the following for any given tool:

- 1) Soundness
- 2) Completeness
- 3) Termination

We can *never* have all three properties together. This is because of the halting problem in Turing Machines.

Choices in Model Checker Designs:

1) Function vs. Whole Program

- We looked at a one function problem before. The function under focus was "void func()." However, a program does not consist of one function. We might want to check for code that performs setuid in one function, and then does exec() in another function. Checking for the latter case is an example of a whole program problem.
- Choosing a function Model Checker or a Whole Program Model Checker depends upon the type of task and taste of the user.
- MECA analyzes functions, and it is quite successful.

2) Soundness vs. Convenience

- SLAM is sound and terminates
- BLAST is sound and complete but not always terminates
- MECA terminates and is convenient.

3) Theorem Proving:

- SLAM and BLAST incorporate lots of Theorem Proving.

How Does MECA Help Security?

- Unsound, because may not be secure afterwards, but still helps a lot.
- With tools that are sound you can eliminate one bug after a long time and effort, but program is still not going to be secure. In contrast with MECA, you can catch a lot of bugs with lesser effort and time.
- Hence, it is a *laser vs. shot gun* approach.
- We want to find all the bugs that the attacker can find by hand; hence it is *Asymmetry of Attacking vs. Defending*. It must be noted that Defender must close all vulnerabilities, but attacker just has to find one.
- MECA has false positives but this problem can be reduced by ranking. We can rank bugs detected as “For Sure it is a Bug” and “It could be a bug.” We can even go further by giving a percentage of how much likely it is for the bug detected to actually be a bug.

MECA Results:

- Detected 1000 bugs in Kernels and System Codes.
- Low false positive rates. About 10% to 100 %.
- Commercialized in Company Coverity. You send your code and they run their tool on it.
- Misses real bugs.
- Does not cover design level bugs.

[Buffer Overflows start on the next page.]

BUFFER OVERFLOWS

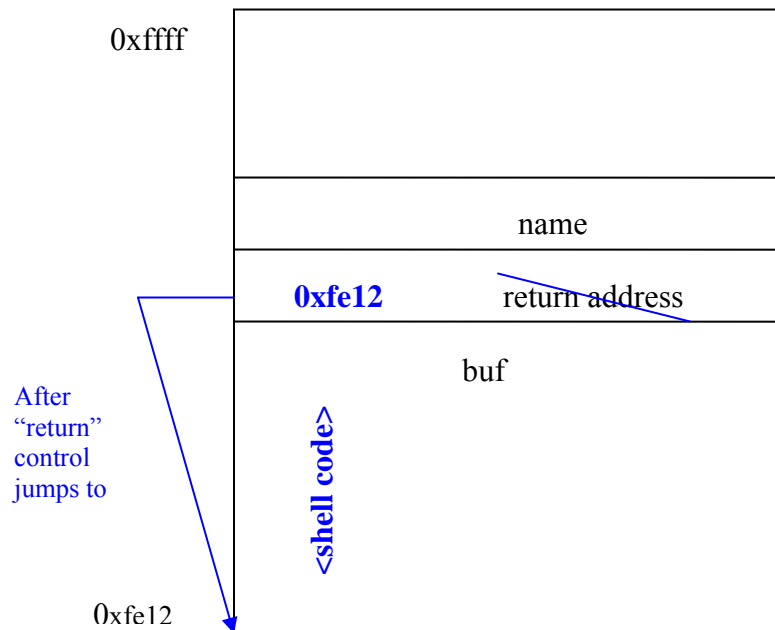
Attack Example 1:

Let us first look at the following code:

```
void func(char *name)
{
    char buf[1024];
    strcpy(buf, name);
    return;
}
```

What Happens When This Method is Called?

The parameter (String name,) and the return address gets saved onto the stack. After that, for the local variable “buf”, space is allocated on the stack. The black parts of the following figure, shows this stage:



Attack:

Assume that attacker got a copy of the code and the attacker found that the base address for the ‘buf’ is 0xfe12.

He then supplies name = “<shell code>\xfe12\0” to the above method.

Question) What happens after this, if the shell code is just of right size?

Answer) The shell code gets copied to the area in the stack that is allocated for 'buf' and then the return address gets overwritten by 0xfe12. So, when "func" wants to return it pops out the hijacked address, which is 0xfe12, from the stack, and jumps to that location and starts to execute code from 0xfe12. [The blue parts of the above diagram show what is explained here.](#)

Question) The return address, in this case 0xfe12, might vary a little bit every time. How would you deal with these variations if you were an attacker?

Answer) To deal with this case, in the name string, we can have a bunch of statements that would not do anything. So, we can have the following:

```
name = "NOPNOPNOPNOP<shell code>\xfe12\0"
```

Question) How can the attacker find the exact starting address for buf?

Answer) The attacker gets a hold of the source code, and then finds out the exact version and type of environment that the victim is using to compile and run his program. So, for example if the victim is using Red Hat, then the attacker also installs the same version of Red Hat on his computer. He then compiles the code and runs it. In this way, he can find the exact starting address for buf.

Question) "strcpy" stops copying after it hits the '\0' byte. How would you ensure that your whole "name" string would get copied till the end?

Answer) Use a decoder to eliminate forbidden bytes.

How to Fix the Above Attack:

Use NX bit. Forbid the execution of code that is present on the stack.

Attack Example 2:

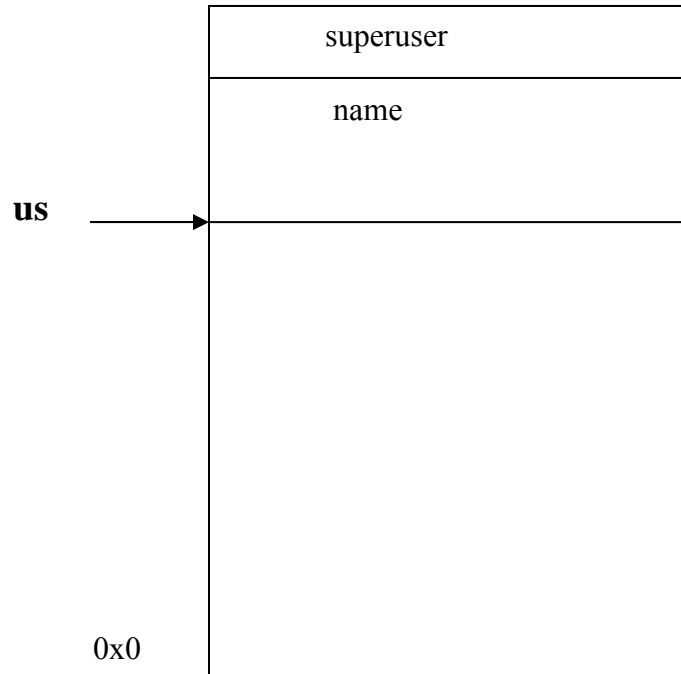
Let us look at the following:

```
struct usestate
{
    char name[64];
    int superuser;
}

void login_user(char *name)
{
    struct usestate *us = malloc(..);
    strcpy(us->name, name);
}
```

How can we exploit this?

The figure below shows how the stack looks like:



Now how can we exploit this?

Attack:

We can just overflow data and get control. We can see that the name is 64 bytes long. We can supply a name that is 68 bytes long and we can easily set the superuser access to true, by supplying all 1's for the last 4 bytes. So, now we can do whatever we want to do in the system.

Attack Example 3 (Return to libc Attack):

Let us again look at the following code:

```
void func(char *name)
{
    char buf[1024];
    strcpy(buf, name);
    return;
}
```

Now assume that we have made the fix, and we are not allowed to run code that is present on the stack. Is there any way we can still exploit this?

Attack:

- We know that we cannot overflow the return address to point at the bottom of the buffer.
- However, we know that we have lots of code in the code section, and we might have code available in the code section that might be able to do what we want.
- So, main Idea is this: Instead of executing on stack, call the library code present in the code section of the program, in order to start the shell program.
- So, when I attack, I overflow the buffer and right above the correct return address I insert the return address that points to the “exec()” function present in the code section. In this overflow attack, we also insert the arguments for the “exec()” function call right above the fake return address.

The figure below explains well what happens in this attack. [What happens as a result of the buffer over flow attack is shown in blue.](#)

