CSE 409, Fall 2011, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, October 7th, 2011

Operating systems level bugs

Today's topics on UNIX

- User ID management
- Signals
- File systems races

Process user IDs

UNIX has up to four user ID's for a process:

- The **effective user ID:** the ID used for pretty much all access control decisions
- The **real user ID:** the user ID of the invoker of the program (people wanted to know who created the process)
  - *Example:* If you have a setuid program, and *rob* runs it but the owner of the executable file is *root*, then the effective UID will be *root* and the real UID will be *rob*
  - *Paper:* Setuid Demystified: http://www.cs.berkeley.edu/~daw/papers/setuid-usenix02.pdf
- The **saved set-user-ID:** people realized setuid programs can be dangerous, so it would be best for them to lower their effective user ID sometimes, while being able to revert to it later
  - Sometimes a setuid program might want to switch to its real user ID so that it can limit the damage it would do if it the program was exploited.
  - But how would it switch back? (for instance if it the program switched from *root* to *rob* and now it needs root privileges again)
  - … and so the **saved set-user-ID** was added to the process UID list
    - Consider a setuid *root* program started by *rob*
    - When the program switches to its real user ID, the saved set-user-ID will be set to *root* so that the effective UID can be set back to *root*, once the program decides it needs root privileges again
- The **file-system UID**: unless you mess with it, it is always the same as the effective user ID.

APIs for changing UIDs

There are a lot of APIs that are used to change a process' user IDs.

- `int setuid(uid_t uid)`
  - sets all UIDs for programs running as root
  - it can be used to drop all privileges
- `int seteuid(uid_t euid)`
  - sets the effective UID
- `int setruid(uid_t ruid)`
  - sets the real UID
  - non-standard function
- `int setreuid(uid_t ruid, uid_t euid)`
  - sets the real and/or effective UIDs
- `int setresuid(uid_t ruid, uid_t euid, uid_t suid)`
  - non-standard function
  - can permute RES UIDs in any way (as long as they are from your RES set)

CSE 409, Fall 2011, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, October 7th, 2011
Documentations links: setuid, seteuid, setreuid, setresuid

Rule: You can set the real UID to any of the RES UIDs (real, effective or saved set-user-ID), you can set your effective UID to any of the RES UIDs

Bugs in setuid programs

Imagine you have a setuid root program, `program1.c`:

```
try {
	// Sets effective UID to root (temporarily acquire privileges)
	setresuid(-1, root, -1);
	// Do delicate stuff
	do_stuff_that_throws_exception();
	// Sets effective UID to real UID (temporarily dropping privileges)
	setresuid(-1, real_user_id, -1);
} catch(Exception e) {

}
```

A lot of times, setuid root programs do the privileged stuff they need to do and then quickly drop their privileges using `setresuid` by setting their effective UID to their real UID. If the `do_stuff_that_throws_exception()` call fails, then the program keeps running with root privileges.

File system races

File system races are bugs that can occur when the file system changes between successive calls from the victim process.

Consider `lpr.c`, a setuid root program:

```
uid_t euid = root;
uid_t ruid = invoker;

if(access(input_file, R_OK))
{
    // Race condition here!
    fd = open(inputfile, O_RDONLY);
}
```

The `open` function will check the **effective UID** to decide if it can open the file, but the setuid root program only wants to open files that the invoker **(real UID)** can open. That is why it first makes a call to `access`, which **checks if the real UID can open the file**.

Attack:

1. The `access` call checks if the file can be opened using the real user ID.
    a. The call says that the file can be opened.
2. Unfortunately, *time passes between successive system calls*.
3. The attacker can do the following in between the `access` and `open` call

```
$rm inputfile
$ln -s /etc/shadow inputfile
```

If he's lucky the `lpr` program will be preempted after the access call succeeded.

RAII (Resource Acquisition Is Initialization)

```
class Lock {
    public:
        Lock(int * lockId)
        {
            //  acquire lock
        }

        ~Lock()
        {
            //  release lock
        }
}

int main() {
    {
        Lock lock(&someLock);

        //  synchronized block
    }
}
```

Ways around file systems race conditions

- Drop privileges using `setresuid` then you call `open` and then you restore privileges.
- Have two processes, one that deals with the printer, another one that reads the file and sends it to the first one
- Inherited capability using `execve`
- Capability passing using UNIX sockets: On the same system, one process can send a file handle to another process using a socket.

The exceve system call

You've got a process running with data, code, PID, RUID, EUID, SUID, a list of open files, etc. There are a lot of resources associated with your process.

```
int execve(const char *filename, char *const argv[], char *const envp[]);
```

`exceve`, replaces a lot of your process's memory and process control block (PCB) with a new process. Your process dies, and the new process begins. It always copies the euid into the suid.

One way to implement `lpr` is using two programs:

- `lpr.c`, not a setuid root program
    - Opens the input file, succeeds only if the user can open the file
    - Then it just executes `lpr_internal` using `exceve`
- `lpr_internal.c`, which is setuid
    - Once run using `exceve`, its EUID is set to 0 (root) because it is a setuid program
    - Now `lpr_internal` has the file handle in its memory and can send the file to the printer