SetUID

In this presentation all references to “setuid syscalls” refer to all syscalls that change a processes uid. These can differ for different distributions of linux.

To understand the SetUID system call we first need to know about linux user ID's:

Linux has three* different user ID's that it uses to to control a users access to certain files, in general use of the OS these are all the same - however when interacting with SUID files and syscalls it is important to understand the roles of each one. These UID's are stored under /proc (in a read-only file) as they can change on a per process basis, to see your current users id run the command "id".

Real user ID (ruid):
This is the true ID of the user, set when the user logs into their account (as found in /etc/shadow)

Effective user ID (euid):
The euid is the id that processes use to check permissions, in most cases this is the same as the ruid (except in SetUID files)

Saved user ID (suid):
The suid is used to temporarily do unprivileged work, how this works will be explained later, but is normally the same as the ruid

**There is a forth (file system uid) but it only exists for compatibility reasons in modern linux.*

What’s the difference between a Suid file and the setuid syscalls?

A file with a Suid bit set will always be executed with the uid of the owner, the setuid (and seteuid) syscalls are to change the uid of an already running process.

A good example of a setuid file is the passwd file, it is used to change a users password - however to do that it needs to alter the file that contains all users passwords (/etc/shadow) which an unprivileged user will be unable to see / edit unless they use the the passwd program.

The setuid and seteuid syscalls are used to alter the processes current uid during execution, this is normally used to temporarily lower permissions when performing tasks for a user (like writing to a file) so the process executes with the users id, this will stop the user writing to files they don’t have access to normally, or to drop the privileges indefinitely when they will no longer be needed again.

Files that use the setuid syscall are generally setuid bit files, as unprivileged users can only change their euid to one of their other id’s (real or saved) and they are all the same.

Demonstration:

Demonstration 1 will show how a processes UIDs are affected if it is run by a normal user, as sudo and with a setuid bit enabled.

Demonstration 2 will show how setuid syscalls can be used to alter a processes UIDs during the runtime.

The setuid and seteuid syscalls:

These syscalls act differently if your euid is set to 0 (root):

When process euid = 0:	When process euid != 0:
The setuid syscall: This changes all three uids to the one supplied in the argument, and is used to drop privileges permanently. The seteuid syscall: This will only change the euid, leaving the real and saved uid in their previous state, this means the saved uid will be 0 and thus privilege can be regained.	When you don't have root privileges you can only change your euid to one of your other uids. So both syscalls only change the euid to another uid.

The setuid syscall is secure since it is in kernel space and the uids of a process are stored in kernel data structures, since the user cannot write to kernel space it is trusted, however security vulnerabilities often come from the program using these syscalls. It is vital that programs using these syscalls check they return successfully, especially when dropping privileges, otherwise the process will run at a different permission level than is intended.

Demonstration:

Unfortunately strace (a program used to trace syscalls) does not behave as intended with setuid files so for this demonstration I will run as root to start (for this example it will have no difference as we are only observing syscalls)

```
void example_3() { // strace example
  // These are just two simple syscalls:
  setuid(1000); // Drop all privileges
  seteuid(0); // Try to regain them
  // This simple function is to make the syscalls easier to see
}
```

Through the strace we will be able to see the syscalls being made, however I have written the program to attempt to regain privileges in a position where it will fail so we can see what happens in the strace.

Example vulnerabilities in setuid files:

The actual setuid file and syscalls are secure, however if they are implemented incorrectly it can expose privilege escalation vulnerabilities in a system this is a very basic example:

This is an very basic word logger that could be exploited if set as a setuid file.

```
void example_4() { // append to file user data
  const char* path = getenv("PASSWORD_LOG_FILE");
  char sentence[1000];
  FILE *fptr;
  fptr = fopen(path, "a");

  while(1) {
    printf("Enter sentence to log:\n");
    fgets(sentence, sizeof(sentence), stdin);
    if ( strcmp(sentence, "QUIT\n") == 0 ) { break;}
    fprintf(fptr, "%d : %s", (int)time(NULL), sentence);
  }
  fclose(fptr);
}
```

Preventing misuse:

It is important to note the different implementation of the setuid syscalls between distributions, they may not all act the same and thus a good way to ensure privileges are dropped is to try to regain them if this succeeds the program should exit. This is an example implementation that ensures the process has properly dropped its privileges: (function can be found in suid_examples.c)

```c
void example_5() { // Misuse prevention example:
  // starting euid is 0
  int ret;
  ret = setuid(1000); // drop privileges
  if (ret != 0) { // Check syscall succeeds
    exit(ret); // exit if fail
  }
  if (setuid(0) == 0) { // check if privileges can be regained
    exit(1);
  }
  // Now safe to handle potentially malicious input
}
```

There have been proposals for a new universal API to alter setuids to reduce the misuse (Chen, Wagner and Dean, n.d.) this new API proposed three calls:

drop_priv_temp() - To lower privilege to the users ruid

restore_priv() - To revert back to higher privilege

drop_priv_perm() - To permanently remove elevate privileges

These API call however perform the error checks (similar to the previous example) when they are called instead of leaving it to the process to handle, this means programs using setuid syscalls would be much safer to use

References:

Chen, H., Wagner, D. and Dean, D., n.d. Setuid Demystified. [online] People.eecs.berkeley.edu. Available at: <https://people.eecs.berkeley.edu/~daw/papers/setuid-usenix02.pdf> [Accessed 7 February 2021].