

Set-UID Privileged Programs

1 How Set-UID Mechanism Works

- Motivations
 - You want other people to be able to search some words in your file, but you don't want them to be able to read the file. How do you achieve this?
 - Users' passwords are stored in `/etc/shadow`, which is neither readable nor writable to normal users. However, the `passwd` program allows users to change their passwords. Namely, when users run `passwd`, they can suddenly modify `/etc/shadow`. Moreover users can only modify one entry in `/etc/shadow`, but not the other people's entries. How is this achieved?
- Set-UID programs
 - The concept of *effective uid* and *real uid*.
 - For non Set-UID programs, the effective uid and the real uid are the same.
 - For Set-UID programs, the effective uid is the owner of the program, while the real uid is the user of the program.
- Effective User UID and Real User UID
 - At login time, the real user ID, effective user ID, and saved user ID of the login process are set to the login ID of the user responsible for the creation of the process. The same is true for the real, effective, and saved group IDs; they are set to the group ID of the user responsible for the creation of the process.
 - When a process calls one of the `exec` family of functions to execute a file (program), the user and/or group identifiers associated with the process can change. If the file executed is a set-user-ID file, the effective and saved user IDs of the process are set to the owner of the file executed. If the file executed is a set-group-ID file, the effective and saved group IDs of the process are set to the group of the file executed. If the file executed is not a set-user-ID or set-group-ID file, the effective user ID, saved user ID, effective group ID, and saved group ID are not changed.
 - Access control is based on effective user IDs and group IDs.
- Why do `passwd`, `chsh` and `su` programs need to be Set-UID programs?
- Are there Set-UID programs in Windows NT/2000? If not, how is the same problem solved in Windows?
 - Windows does not have the notion of Set-UID. A different mechanism is used for implementing privileged functionality. A developer would write a privileged program as a service and the user sends the command line arguments to the service using Local Procedure Call.
 - A service can be started automatically or on-demand.
 - Each service has a security descriptor specifying which users are allowed to start, stop, and configure the service.
 - Services typically run under the Local System account.

- How to turn on the Set-UID bit?

```
% chmod 4755 file      ---> -rwsr-xr-x
```

- How is Set-UID implemented in Minix?

```
/* This is the per-process information */
EXTERN struct fproc {

    uid_t fp_realuid;           /* real user id */
    uid_t fp_effuid;           /* effective user id */
    gid_t fp_realgid;          /* real group id */
    gid_t fp_effgid;           /* effective group id */
    ...
}
```

- Malicious use of Set-UID mechanism:

- An attacker is given 10 seconds in your account. Can he plant a backdoor, so he can come back to your account later on?

```
% cp /bin/sh /tmp
% chmod 4777 /tmp/sh
```

- By doing the above, the attacker creates a Set-UID shell program, with the you being the owner of the program. Therefore, when the attacker later runs the shell program, it will run with your privilege.

2 Vulnerabilities of Set-UID Programs

2.1 Hidden Inputs: Environment Variables

A privileged program must conduct sanity check on all the inputs. Input validation is actually part of the access control that a privileged program must conduct to ensure the security of the program. A lot of security problems are caused by the mistakes in input validation.

If inputs are explicit in a program, programmers might remember to do the input validation; if inputs are implicit, input validation may be forgotten, because programmers may not know the existence of such inputs. Environment variables are such kind inputs.

Every UNIX process runs in a specific environment. An environment consists of a table of environment variables, each with an assigned value. Some programs use these environment variables internally; shell programs are examples of these programs. In other words, the value of some environment variables can affect the behavior of a shell program.

Since environment variables are controlled by users, if a program relies on these variables, users can indirectly affect the behavior of such a program by changing the values of some environment variables. Therefore, it is very important to understand whether a privileged program relies on the values of any environment variable. One way a program can be affected by environment variables is for this program to use the values of environment variables explicitly in the program. In C, a program can use `getenv()` to

access the values of environment variables. However, there are many cases when a program *implicitly* uses environment variables; that is where we have seen many vulnerabilities in Set-UID programs. We will present several examples in this section.

- PATH Environment Variable

- When running a command in a shell, the shell searches for the command using the PATH environment variable, which consists of a list of directories. The shell program searches through this list of directories (in the same order as they are specified in the PATH environment variable. The first program that matches with the name of the command will be executed.
- What would happen in the following? Note that system (const char *cmd) library function first invoke the /bin/sh program, and then let the shell program execute cmd.

```
system ("mail");
```

- The attacker can change PATH to the following, and cause “mail” in the current directory to be executed.

```
PATH=".:$PATH"; export PATH
```

- IFS Environment Variable

- The IFS variable determines the characters which are to be interpreted as white spaces. It stands for Internal Field Separators. Suppose we set this to include the forward slash character:

```
IFS="/ \t\n"; export IFS  
PATH=".:$PATH"; export PATH
```

- Now call any program which uses an absolute PATH from a Bourne shell (e.g. system()). This is now interpreted like the following that would attempt to execute a command called bin in the current directory of the user.

```
system("/bin/mail root"); ---> system(" bin mail root");
```

- The IFS bug has pretty much been disallowed in shells now.

- LD_LIBRARY_PATH Environment Variable

- In Linux, unless explicitly specified via the -static option during compilation, all Linux programs are incomplete and require further linking to the dynamic link libraries at run time. The dynamic linker/loader ld.so/ld-linux.so loads the shared libraries needed by a program, prepares the program to run, and then runs it. You can use the following command to see what shared libraries a program depends on:

```
% ldd /bin/ls
```

- `LD_LIBRARY_PATH` is an environment variable used by the the dynamic linker/loader (`ld.so` and `ld-linux.so`). It contains a list of directories for the linker/loader to look for when it searches for shared libraries. Multiple directories can be listed, separated with a colon (:). This list is prepended to the existing list of compiled-in loader paths for a given executable, and any system default loader paths.
- Virtually every Unix program depends on `libc.so` and virtually every windows program relies on DLL's. If these libraries can be replaced by malicious copies, malicious code can be invoked when functions in these libraries are invoked.
- Since `LD_LIBRARY_PATH` can be reset by users, attackers can modify this variable, and force the library loader to search for libraries in the attacker's directory, and thus load the attacker's malicious library.

```
% setenv LD_LIBRARY_PATH .:$LD_LIBRARY_PATH
```

- To make sure Set-UID programs are safe from the manipulation of the `LD_LIBRARY_PATH` environment variable, the runtime linker/loader (`ld.so`) will ignore this environment variable if the program is a Set-UID root program, unless the real UID is also zero.
- Secure applications can also be linked statically with a trusted library to avoid this.
- In Windows machines, when loading DLLs, generally, the current directory is searched for DLLs before the system directories. If you click on a Microsoft Word document to start Office, the directory containing that document is searched first for DLLs.

- `LD_PRELOAD` Environment Variable

- Many Unix systems allow you to "pre-load" shared libraries by setting an environment variable `LD_PRELOAD`. These user specified libraries will be loaded before all others. This can be used to selectively override functions in other libraries. For example, if you have already built a library, you can preload it using the following command:

```
% export LD_PRELOAD=./libmylib.so.1.0.1
```

If `libmylib.so.1.0.1` contains a function `sleep`, which is a standard `libc` function, when a program is executed and calls `sleep`, the one in `libmylib.so.1.0.1` will be invoked.

- Here is a program that override the `sleep()` function in `libc`:

```
#include <stdio.h>
void sleep (int s)
{
    printf("I am not sleeping!\n");
}
```

We can compile the program using the following commands (assume that the above program is named `a.c`):

```
% gcc -fPIC -g -c a.c
% gcc -shared -Wl,-soname,libmylib.so.1 \
    -o libmylib.so.1.0.1 a.o -lc
```

Now, we run the following program:

```
int main()
{
    sleep(1);
    return 0;
}
```

If the environment variable `LD_PRELOAD` is set to `libmylib.so.1.0.1`, the `sleep()` in the standard `libc` will not be invoked; instead, the `sleep()` function in our library will be called, and “I am not sleeping!” will be printed out.

- To make sure Set-UID programs are safe from the manipulation of the `LD_PRELOAD` environment variable, the runtime linker/loader (`ld.so`) will ignore this environment variable if the program is a Set-UID root program, unless the real UID is also zero.

2.2 Invoking Other Programs

When a privileged invokes other programs, attention must be paid on whether unintended programs would get invoked. We know that environment variables are places where we should focus on our attention; there are other places that we should also focus on.

- What are the potential problems if a Set-UID program does the following?

```
// The contents of User_Input are provided by users.
sprintf(command, "/bin/mail %s", User_Input);
system(command);
```

- `User_Input` might contain special characters for the shell: (e.g. `|`, `&`, `<`, `>`). Remember, the `system()` call actually invokes shell first, and then asks the shell program to execute `"/bin/mail"`. If we are not careful, attackers might cause other commands to be executed by letting `User_Input` be the following string:

```
xyz@example.com ; rm -f /* ; /bin/sh
```

2.3 Other Well-Known Vulnerability Patterns

Other than the above input validation vulnerabilities, there are several other well-known patterns of vulnerabilities. We will discuss each of them in separate lectures. Here is a summary of these patterns.

- Bufferflow Vulnerability
- Race Condition Vulnerability
- Format-String Vulnerability

2.4 Miscellaneous Vulnerabilities

There are many other vulnerabilities that are not easy to put into any of the categories that we have discussed above. Some might be categorized broadly as “input validation vulnerability”, but because of their unique features, we discuss them separately here. We cannot enumerate all the vulnerabilities. We only give a few examples to show various mistakes programmers have made in their program logic, and show how such mistakes can be turned into vulnerabilities.

- `lpr` vulnerability: It generates temp files under the `/tmp` directory. The file names are supposed to be random; however, due to an error in the pseudo-random number generation, file names will repeat themselves every 1000 times. The program is a Set-UID program. Linking the predictable file name to `/etc/password` will cause `lpr` to overwrite `/etc/password`.
- `chsh` vulnerability: `chsh` asks users to input the name of a shell program, and save this input in `/etc/passwd`; `chsh` does not conduct sanity checking. The program assumes that the users’ inputs consist of only one line. Unfortunately, this assumption can be made false: users can type two lines of inputs, with the second line being something like “`xyz::0:0::`”, i.e., users can insert a new superuser account (uid: 0) with no password.
- `sendmail` vulnerabilities
 - `sendmail`: (1) incoming emails will be appended to `/var/mail/wedu`. (2) If the owner of the `/var/mail/wedu` is not `wedu`, `sendmail` will change the owner to `wedu` using `chown`.
 - Can you exploit this to read `wedu`’s email?
 - Can you exploit this to cause more severe damage to `wedu`?

3 Improving the Security of Set-UID Programs

- The `exec` functions
 - The `exec` family of functions runs a child process by swapping the current process image for a new one. There are many versions of the `exec` function that work in different ways. They can be classified into groups which
 - * Use/do not use a shell to start child programs.
 - * Handle the processing of command line arguments via a shell (shell can introduce more functionalities than what we expect. Note that shell is a powerful program).
 - Starting sub-processes involves issues of dependency and inheritance of attributes that we have seen to be problematical. The functions `execlp` and `execvp` use a shell to start programs. They make the execution of the program depend on the shell setup of the current user. e.g. on the value of the `PATH` and on other environment variables. The function `execv()` is safer since it does not introduce any such dependency into the code.
 - The `system(cmd)` call passes a string to a shell for execution as a sub-process (i.e. as a separate forked process). It is a convenient front-end to the `exec`-functions.
 - The standard implementation of `popen()` is a similar story. This function opens a pipe to a new process in order to execute a command and read back any output as a file stream. This function also starts a shell in order to interpret command strings.

- How to invoke a program safely?
 - Avoid anything that invokes a shell. Instead of `system()`, stick with `execve()`: `execve()` does not invoke shell, `system()` does.
 - Avoid `execlp(file, ...)` and `execvp(file, ...)`, they exhibit shell-like semantics. They use the contents of that file as standard input to the shell if the file is not valid executable object file.
 - Be wary of functions that may be implemented using a shell.
 - * Perl's `open()` function can run commands, and usually does so through a shell.
- Improve security of `system()`
 - Recall that `system()` invokes `/bin/sh` first. In Fedora, it `execv /bin/sh` with arguments `"sh", "-c"` and the user provided string.
 - In Fedora, `/bin/sh` (actually `bash`) ignores the `Set-UID` bit option. Therefore, when invoking `system(cmd)` in a `Set-UID` program, `cmd` will not be executed with the root privilege, unless `cmd` itself is a `Set-UID` program. The following code in `bash` drops the `Set-UID` bit. Actually, I cannot think of any legitimate reason why we need to allow `Set-UID` shell program. Fedora is doing the right thing; many other Unix OSes have not done this.

```

if (running_setuid && privileged_mode == 0)
    disable_priv_mode ();

...
void disable_priv_mode ()
{
    setuid (current_user.uid);
    setgid (current_user.gid);
    current_user.euid = current_user.uid;
    current_user.egid = current_user.gid;
}

```

4 Principle of Least Privilege

Principle of Least Privilege (originally formulated by Saltzer and Schroeder):

Every program and every user of the system should operate using the least set of privileges necessary to complete the job.

The most important reason for limiting the security privileges your code requires to run is to reduce the damage that can occur should your code be exploited by a malicious user. If your code only runs with basic privileges, its difficult for malicious users to do much damage with it. If you require users to run your code using administrator privileges, then any security weakness in your code could potentially cause greater damage by the malicious code that exploits that weakness.

- Questions to ask when writing a privilege program:

- Does the program need the privileges?
 - * If a program does not need any special privileges to run, it should not be a privilege program.
- Does the program need all the privileges?
 - * We only give the program the least set of privileges necessary to complete the job.
 - * Many operating systems do not give us with many choices; we can choose either a set that includes all the root privileges or a set that does not include any privilege. Most Unix systems are like this, you are either root or non-root. there is nothing in between.
 - * Most modern Unix systems (and Windows) introduces more choices. These systems divide the root privileges into a number of sub-privileges. With such a finer granularity, we can better apply the least-privilege principle.
- Does the program need the privileges now?
 - * A program usually does not need certain privileges for some time; they become unnecessary at the point of time. We should temporarily disable them to achieve the least-privilege principle. The advantage of doing this is that in case the program makes an accidental mistake, it cannot cause the damage to the things that require the disabled privileges. The figure below illustrates this point.
 - * At a later time, the disabled privilege might become necessary again, we can then enable it.
 - * Keep in mind that disabling/enabling can reduce the damage in a situation when adversaries cannot inject code into a vulnerable program; if adversaries can inject code into the vulnerable programs, the injected code can enable the privileges by itself.
- Does the program need the privileges in the future?
 - * If a privilege will not be used any more, it becomes unnecessary, and should be permanently removed, so the least set of privileges is adjusted based on the future needs.
- What mechanisms does Unix provide for us to achieve the least-privilege principle?
 - Useful system calls: `setuid()`, `seteuid()`, `setgid()`, and `setegid()`.
 - `seteuid(uid)`: It sets the effective user ID for the calling process.
 - * If the effective user ID of the calling process is super-user, the uid argument can be anything. This is often used by the super-user to temporarily relinquish/gain its privileges. However, the process's super-user privilege is not lost, the process can gain it back.
 - * If the effective user ID of the calling process is not super-user, the uid argument can only be the effective user ID, the real user ID, and the saved user ID. This is often used by a privileged program to regain its privileges (the original privileged effective user ID is saved in the saved user ID).
 - `setuid(uid)`: It sets the effective user ID of the current process. If the effective user ID of the caller is root, the real and saved user IDs are also set.
 - * If the effective user ID of the process calling `setuid()` is the super-user, all the real, effective, and saved user IDs are set to the uid argument. After that, it is impossible for the program to gain the root privilege back (assume uid is not root). This is used to permanently relinquish access to high privileges.
 - * A `setuid-root` program wishing to temporarily drop root privileges, assume the identity of a non-root user, and then regain root privileges afterwards cannot use `setuid()`. You can accomplish this with the call `seteuid()`.

- * If the effective user ID of the calling process is not the super-user, but uid is either the real user ID or the saved user ID of the calling process, the effective user ID is set to uid. This is similar to seteuid().
- Examples (in Fedora Linux): A process is running with effective user ID=0, and real user ID=500, what are the effective and real user IDs after running
 - * `setuid(500); setuid(0);` Answer: 500/500 (the first call generates 500/500, and the second call fails).
 - * `seteuid(500); setuid(0);` Answer: 0/500 (the first call generates 500/500, and the second call generates 0/500).
 - * `seteuid(600); setuid(500);` Answer: 500/500 (the first call generates 600/500, and the second call generates 500/500).
 - * `seteuid(600); setuid(500); setuid(0);` Answer: 0/500 (the first call generates 600/500, the second generates 500/500, and the third generates 0/500).