Chapter 1

Set-UID Privileged Programs and Attacks on Them

Privileged programs are an essential part of an operating system; without them, simple things such as changing password would become difficult. Because of the privileges carried by these programs, they often become attack targets. In this chapter, we use one type of privileged program, Set-UID programs, as a case study to show how privileged programs gain their privileges, what common mistakes exist for these programs, and how to write safer privileged programs.

Contents

1.1 The Need for Privileged Programs

To understand why privileged programs are needed for operating systems, we use Linux as an example, and show how operating systems allow users to change their passwords without compromising security.

1.1.1 The Password Dilemma

In Linux, users' passwords are stored in /etc/shadow (the shadow file). If a user needs to change her password, the shadow file will be eventually modified to store the new passwords. A closer look at the shadow file shows that the file is only writable to root, not to normal users. See the following:

```
-rw-r----- 1 root shadow 1443 May 23 12:33 /etc/shadow
 ⬆ Only writable to the owner
```

The question is how to allow normal users to change their passwords. This is a dilemma that we face: changing passwords requires changing the shadow file, but the file is not modifiable by normal users. An easy solution is to simply make the shadow file writable to everybody. This is not a safe solution. If normal users can write to the shadow file, they can change other people's passwords to something that they know, so they can log into other people's accounts. Therefore, writing to the shadow file must be restricted.

Another solution is to provide a finer-grained access control mechanism that supports the change-password functionality. Operating systems can implement an access control that allows users (non-root) to only modify the password field of their own records in /etc/shadow, but not the other fields or other people's records. The current access control in most OSes is only enforced at the file level, i.e., it can decide whether a user can access a file or not, but it does not have the sufficient granuality to restrict what part of a file can be accessed. Increasing the granuality of the access control can certainly solve this particular problem, but it will significantly increase the complexity of the operating system as well.

Most operating systems choose not to implement such an over-complicated access control mechanism; they instead choose a simplistic two-tier design (see Figure 1.1): they implement a simple and generic access control model, which allows us to express simple access control rules, such as the read, write, and execute accesses. Many more specific, sophisticated, and application-dependent access control rules cannot be directly expressed using the built-in access control mechanism. To enforce these rules, OSes have to rely on extensions, which are usually in the form of privileged programs. They enforce application-specific access control rules using program logic. For example, to support the above rule on the shadow file, Unix-based operating systems make the shadow file writable only to root, so if a normal user tries to access the file, it will be denied. To allow users to change passwords, Unix implements an extension, a privileged program called passwd, which can modify the shadow file for users.

We can look at the above solution from a different angle. Because of the lack of granuality, access control in operating systems tends to be over-protective. For example, it completely disallows non-root users to modify the shadow file. This is too restrictive, because users should be able to change their passwords, and thus need to modify the shadow file. To support these "exceptions" raised by application-dependent requirements, operating systems will "poke a hole" on its protection shell, allowing users to go through that hole, follow a specific procedure, and make an authorized modification of the shadow file. This hole and its corresponding procedure are usually in the forms of programs.

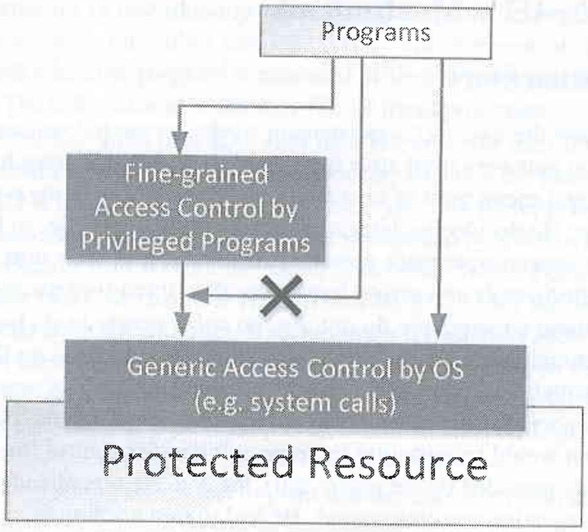
Programs
Fine-grained Access Control by Privileged Programs
Generic Access Control by OS (e.g. system calls)
Protected Resource

Figure 1.1: Two-Tier Approach for Access Control

However, these programs are not usual programs. They actually provide extra privileges that a normal user does not have. In the shadow file example, the program `passwd`, when invoked, allows a user to modify the shadow file; if a user wants to modify the shadow file directly without using the program, the user will not be able to succeed, because of the access control protection on the shadow file. We call such a program a *privileged program*. Any program that has extra privileges can be considered as a privileged program.

1.1.2 Different Types of Privileged Programs

There are two common approaches for privileged programs: daemons and `Set-UID` programs. A daemon is a computer program that runs as a background process. To become a privileged program, a daemon needs to run with a privileged user ID, such as root. In the password-changing example, the system can use a root daemon to do the task. Basically, whenever a user needs to change her password, she will send a request to this daemon, which will modify the shadow file for her. Since this daemon is a root process, it has the permission to modify the shadow file. Many operating systems use the daemon approach for privileged operations. In Windows, they are not called daemons; they are called services, which, just like daemons, are computer programs that operate in the background.

Another approach for privileged programs is to use the `Set-UID` mechanism, which is widely adopted in `Unix` operating systems. It uses a special bit to mark a program, telling the operating system that such a program is special and should be treated specially when running. The `Set-UID` bit was invented by Dennis Ritchie [McIlroy, 1987]. His employer, then Bell Telephone Laboratories, applied for a patent in 1972, and the patent was granted in 1979. We will explain this mechanism in details in the next section.

1.2 The `Set-UID` Mechanism

1.2.1 A Superman Story

Before explaining how the `Set-UID` mechanism works, let me tell you an "untold" superman story first. Superman gets very tired after fighting evils and saving lives for over eighty years. He wanted to retire and spend most of his time lying on the beach, doing nothing, but the world still depends on him. He decided to delegate his tasks to other people, so he invented a power suit, which gave the wearer superman's power. He made many of such suits, and hired a team of people to go out fighting evils and saving lives. He called them *superpeople/superperson*.

To ensure that these superpeople do not use the super power to do bad things, Superman conducted very thorough background checks and psychological tests on them. Unfortunately, regardless how thorough they are, once in a while, some superpeople went rogue and did bad things. When that happened, Superman had to interrupt his vacation, and fought them. Although every time Superman would successfully put everything under control (because his power was still stronger than that provided by the power suit), the damage was already done. Moreover, he hated that when his vacation was interrupted. He had to find a solution.

After many days of thinking, he came up with an idea. In the power suit version 2.0, he embedded a computer chip. When superpeople put on their power suits, their behaviors are completely controlled by the embedded chip. For example, when an instruction in the chip says "go north", they will go north, not the other directions. The actions in each chip are pre-programmed, i.e., before sending a superperson off to carry out a task, Superman programmed the chips, so the person wearing it would only perform the intended tasks, and nothing else. Even if a superperson wanted to do bad things, he/she could not do it, because there was no way to deviate from the pre-programmed tasks. Superman was very excited about this new invention; he even filed a patent for it.

1.2.2 How It Works

Let us temporarily come back from the fictional world to our cyber world, and see how we can build a "power suit" for computer users, so whoever "put on" this suit can gain the super-user power, but without being able to do bad things.

In a typical computer system, although we do not fight evils or save lives, we do need superuser's power to do some routine tasks, such as changing our passwords. One way is to ask a superuser to do that for us, but that is going to drive the superuser crazy. Just like Superman, superusers want to delegate these tasks to others, but they do not want to simply grant the super-power to normal users. If superusers do that, some normal users may go rogue and do bad things with their super-power. A superuser can run a background process to serve all the password-changing requests. This is the daemon approach, and has been adopted by many systems.

`Unix` adopted another approach for privileged operations in addition to the daemon approach. This approach, very similar to the superman's approach, is called `Set-UID` [Wikipedia, 2017s]. With this approach, the superuser power is directly granted to a normal user, i.e., the process running the privileged operations belongs to a normal user, not the superuser as in the daemon approach. However, the behaviors of such a process are restricted, so it can only perform the intended tasks, such as changing the user's own password, nothing else. This type of program is just like the program inside the computer chip embedded in the power suits made by Superman.

A Set-UID program is just like any other Unix program, except that it has a special marking, which is a single bit called Set-UID bit. The purpose of this bit is to tell the operating system that when the program is executed, it should be treated differently than those without such a bit. The difference is in the user IDs of these processes.

In Unix, a process has three user IDs: *real* user ID, *effective* user ID, and *saved* user ID. The real user ID identifies the real owner of the process, which is the user running the process. The effective user ID is the ID used in access control, i.e., this ID represents what privilege a process has. For a non-Set-UID program, when it is executed by a user with user ID 5000, its process's real and effective user IDs are the same, both being 5000. For a Set-UID program executed by the same user, the real user ID will still be 5000, but the effective user ID will depend on the user that owns the program. If the program is owned by root, the effective user ID will be 0. Since it is the effective user ID that is used for access control, this process, although executed by a normal user, has the root privilege. That is how a program gains privileges. Regarding the saved user ID, it is used to help disable and enable privileges; it will be discussed later.

We can use the /bin/id command to print out the user IDs of a running process. First, we copy the program to our current directory and rename it to myid. We change its owner to root (using the chown command), but we do not turn on its Set-UID bit yet. The program is still a non-privileged program, even though it is owned by root. We now run the program. From the result, we can see that only one user ID is printed out, i.e., the real user ID, indicating that the effective user ID is the same as the real user ID.

```
$ cp /bin/id ./myid
$ sudo chown root myid
$ ./myid
uid=1000(seed) gid=1000(seed) groups=1000(seed), ...
```

We now turn on the Set-UID bit of this program using the "chmod 4755 myid" command (the number 4 in 4755 turns on the Set-UID bit); this step needs to be performed using the root privilege, because the file is owned by root. We run the program again, but this time, we see a different result: the program also prints out the effective user ID euid; its value is 0, so the process has the root privilege.

```
$ sudo chmod 4755 myid
$ ./myid
uid=1000(seed) gid=1000(seed) euid=0(root) ...
```

1.2.3 An Example of Set-UID Program

We use the /bin/cat program to demonstrate how Set-UID programs work. The cat program basically prints out the content of a specified file. We make a copy of the /bin/cat program in our home directory (user ID is seed), and rename it to mycat. We also change its ownership using the chown command, so it is owned by the root. We run this program to view the shadow file. As shown from the following result, our attempt has failed, because seed is a normal user, and does not have a permission to view the shadow file.

```
$ cp /bin/cat ./mycat
$ sudo chown root mycat
$ ls -l mycat
```

```
-rwxr-xr-x 1 root seed 46764 Feb 22 10:04 mycat
$ ./mycat /etc/shadow
./mycat: /etc/shadow: Permission denied
```

Let us make one small change before running the program `mycat` again: we turn on the `Set-UID` bit of this program, and run `mycat` to view the shadow file again. This time, it is successful. When the `Set-UID` bit is on, the process running the program has the root privilege, because the program's owner is root.

```
$ sudo chmod 4755 mycat
$ ./mycat /etc/shadow
root:$6$012BPz.K$fbPkT6H6Db4/B8c...
daemon:*:15749:0:99999:7:::
...
```

If we change the owner back to `seed`, while keeping the `Set-UID` bit enabled, the program will fail again. Even though the program is still a `Set-UID` program, but its owner is just a normal user, who does not have a permission to access the shadow file. It should be noted that in the experiment, we have to run `chmod` again to enable the `Set-UID` bit, because the `chown` command automatically turns off the `Set-UID` bit.

```
$ sudo chown seed mycat
$ chmod 4755 mycat
$ ./mycat /etc/shadow
./mycat: /etc/shadow: Permission denied
```

1.2.4 How to Ensure Its Security

In principle, the `Set-UID` mechanism is secure. Although a `Set-UID` program allows normal users to escalate their privileges, this is different from directly giving the privileges to users. In the latter case, normal users can do whatever they want after getting the privileges, while in the `Set-UID` case, normal users can only do whatever is included in the program. Basically, users' behaviors are restricted.

However, it is not safe to turn all programs into `Set-UID` programs. For example, it is a bad idea to turn the `/bin/sh` program into a `Set-UID` program, because this program can execute other programs specified by users, making its behavior unrestricted. It is similarly a bad idea to turn the `vi` program into a `Set-UID` program, because although `vi` is just a text editor, it can run user-specified external commands from inside the editor.

1.2.5 The `Set-GID` Mechanism

The `Set-UID` mechanism can also be applied to groups, instead of users. This is called `Set-GID`. Namely, a process has effective group ID and real group ID, and the effective group ID is used for access control. Because the `Set-GID` and `Set-UID` mechanisms work very similarly, we will not discuss `Set-GID` in details.

1.3 What Can Go Wrong: What Happened to Superman

The security of the `Set-UID` mechanism depends on the assumption that the user can only do whatever is coded in the program, and nothing else. Unfortunately, this is not easy to guarantee. Very often, developers may make mistakes in their code, and as a result, users may be able to do things that are not intended for a privileged program. Before discussing the technical details of the potential mistakes in `Set-UID` programs, let us continue the Superman story.

After inventing the chip idea, Superman could finally enjoy his time at the beach without frequent interruptions. Unfortunately, such a peaceful time did not last long. It all started from one hostage rescue mission. A bad guy held two hostages in a building, threating to kill them if his requests are not met, so Superman dispatched a superperson named Mallory to rescue the hostages. To Superman, this was a very easy case, so he programmed the chip, and sent Mallory to this rescue. The program is supposed to let Mallory fly north for one mile, and then turn left. After reaching the first building, knock down the wall behind the bad guy, capture him, and hand him to the policeman outside the building. After that, the superpower and the restriction on Mallory will disappear.

After sending Mallory off to the rescue, Superman flew to the Moon, enjoying his sun bath from there. Suddenly, a loud voice came out from his emergency satellite phone; it came from a major bank near the building where the hostages were held. Apparently, somebody with superpower knocked down the wall of the bank, and took all of its gold. Witnesses said that it was done by a superperson and her partners. It must be Mallory, because she was the only one out on a mission that day. Did he make a mistake in the program? Superman immediately checked his program, but everything seemed fine. The calculation of the path was correct: after turning left, Mallory should reach the hostage building; the bank building was on the opposite direction. How could she knock down the bank's building?

Before answering that question, we have to mention Superman's computer background. When Superman grew up, there was not much education on computer security, and he did not major in computer science anyway. He learned programming from several textbooks that he picked up from the bookstores. Even though he has superpower, which he got from Krypto, the planet where he came from, the power did not enable him to write flawless programs. Therefore, in terms of programming, Superman is just like a normal human being. A common nature of human being is that we make mistakes, especially in programming. Mistakes in privileged programs, like the one in Superman's chips, can often lead to security breaches.

Mallory is a hacker, and she hid this fact in the background checks. She joined the superpeople force with only one goal: to find problems in Superman's programs, exploit them, so she could use the superpower for personal gains. She got the opportunity that she had been waiting for in this rescue mission. In the Superman's code, it said "flying north, and then turn left", but it did not specify how she should fly, so Mallory flew backward to the north. When she got to the turning point, turning left steered her toward the bank building, instead of the hostage building. Before she put on the power suit, she called her friends to wait outside of the bank to help. After she knocked down the wall, her behaviors were still restricted, so she could not pick up any gold bars, but her friends could. The mistake was in the "turning left" instruction, which is relative to the direction one faces. Superman forgot to specify that direction at the first place, but he learned from the mistakes quickly, and vowed not to make any mistake again.

Not very long after the incident, coincidently, another hostage was held in the same building. Learning from the mistakes, Superman changed the instruction to "turning west", instead of "turning left". This time, it did not matter whether one flew backward or not, the turning direction is the same. Superman assigned the task to Malorie, who, just like Mallory, was also a hacker.

She was also good at science, and knew that Superman's chip gets its directions from a built-in magnetic sensor, which calculates the directions based on the earth's magnetic field. She called her friends to strategically place a magnet near the turning point, and changed the magnetic field there. When she got there, the chip was fooled, and steered Malorie towards the bank direction, because based on the magnetic field, that direction was west (it was actually east). The bank lost a lot of gold again.

After these two mistakes, and many other ones later, Superman finally realized that writing code for his chip is not as easy as what he thought. He eventually decided to come back from the retirement, and do everything by himself. That is why we never heard about superpeople any more.

1.4 Attack Surfaces of `Set-UID` Programs

Let us come back from the fictional world to our cyber world again, and see how we, who write `Set-UID` programs, can make similar mistakes like what Superman did. We start from analyzing the attack surface. For a privileged program, the attack surface is the sum of the places where the program gets its inputs. These inputs, if not properly sanitized, may affect the behaviors of the program. Figure 1.2 depicts the main attack surfaces of `Set-UID` programs.

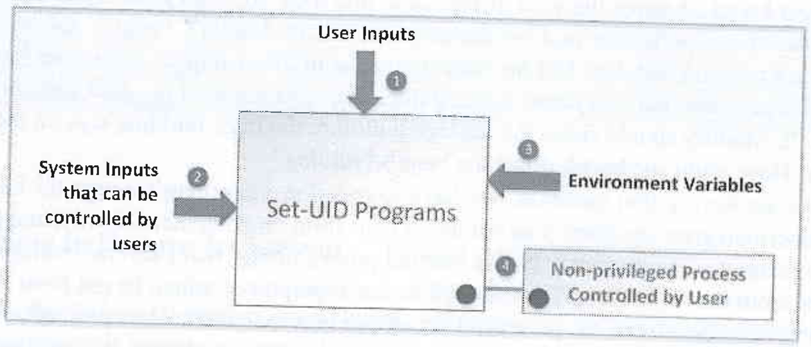
User Inputs
1
System Inputs that can be controlled by users
2
Set-UID Programs
3
Environment Variables
4
Non-privileged Process Controlled by User

Figure 1.2: Attack Surface (inputs and behaviors that are controllable by users)

1.4.1 User Inputs: Explicit Inputs

A program may explicitly ask users to provide inputs. If the program does not do a good job sanitizing their inputs, it may become vulnerable. For example, if the input data are copied into a buffer, it may overflow the buffer, and cause the program to run malicious code. We will discuss this particular buffer overflow vulnerability in Chapter 4. Another example is the format string vulnerability, which is covered in Chapter 6. In this case, user inputs are used as format strings, and they can change a program's behaviors.

Another interesting example is the vulnerability in the earlier version of `chsh`, which is a `Set-UID` program that allows users to change their default shell programs. The default shell information is stored in the `/etc/passwd` file (the password file). To change it, the password file needs to be modified; that is why `chsh` needs to be a `Set-UID` program, because the password file is only writable by root. After authenticating users, the program asks users to provide the name of a shell program, such as `/bin/bash`, and updates the last field of the user's entry in the password file. Each entry consists of several colon-separated fields like the following:

```
bob:$6$jUODEFsfwfi3:1000:1000:Bob Smith,,,:/home/bob:/bin/bash
```

Unfortunately, the `chsh` program did not sanitize the input correctly, and failed to realize that the input may contain two lines of text. When the program writes the input into the password file, the first line replaces the shell-name field in the user's entry, and the second line replaces the next entry. Since each line in the password file contains the account information of one user, by creating a new line of text in the password file, attackers can essentially create a new account on the system. If attackers put 0s in the third and fourth fields (the user ID and group ID fields), they can create a root account.

1.4.2 System Inputs

Programs may get inputs from the underlying system. One may think that these inputs are safe, because they are provided by the system. However, that really depends on whether they are controllable by untrusted users or not. For example, a privileged program may need to write to a file `xyz` in the `/tmp` folder, and the filename is already fixed by the program. Given the name, the target file is provided by the system, so it does not seem that there is any user input here. However, the file is inside the world-writable `/tmp` folder, so the actual target of the file may be controllable by users. For example, a user can use a symbolic link to make `/tmp/xyz` point to `/etc/shadow`. Therefore, although the user does not directly provide any input to the program, she can influence what the program gets from the system. The race condition attack exploits this attack vector, we will cover it in Chapter 7.

1.4.3 Environment Variables: Hidden Inputs

The enemy is never more unnerving than when he's invisible.
By K. J. Parker, *Devices and Desires*

When a program runs, their behaviors can potentially be influenced by many inputs that are not visible from inside the program, i.e., if we look at the code of these programs, we will never see these inputs. Without being aware of their existence, when writing code, many developers may not realize the potential risks introduced by these hidden inputs. One type of hidden input is environment variable. Environment variables are a set of named values that can affect the way a process behaves. These variables can be set by users before running a program, and they are part of the environment in which a program runs.

Because of their stealthy nature, environment variables have caused many problems for `Set-UID` programs. Let us look at an example. This is related to the `PATH` environment variable, which is used by shell programs to find where a command is if a user does not provide the full path for this command. In C programs, if we want to execute an external command, one of the approaches is to use the `system()` function. If a privileged `Set-UID` program simply uses `system("ls")` to run the `ls` command, instead of using the full path `/bin/ls`, it can get into trouble. From the code itself, it seems that no user can change the behaviors of `system("ls")`. A closer look at how `system()` is implemented, we will find out that it does not directly run the `ls` command; instead, it first runs the `/bin/sh` program, and then uses this program to run `ls`. Because the full path to `ls` is not provided, `/bin/sh` uses the

PATH environment variable to find where the ls command is. Users can change the value of the PATH environment variable before running the Set-UID program. More specifically, users can provide their own malicious program called ls, and by manipulating the PATH environment variable, they can affect how /bin/sh finds the ls command, so their ls program is found first and gets executed, instead of the intended /bin/ls program. Attackers can do whatever they want in their ls program, using the privileges provided by the Set-UID program.

There are many examples like this. In Chapter 2, we will conduct a systematic study on how various environment variables affect Set-UID programs, These variables are not directly used by Set-UID programs, but they are used by libraries, dynamic linker/loader, and shell programs that Set-UID programs depend on. A number of case studies will be discussed in the chapter.

1.4.4 Capability Leaking

In some cases, a privileged program downgrades itself during its execution, so the process continues as a non-privileged one. For example, the su program is a privileged Set-UID program, allowing one user to switch to another user, if the first user knows the second user's password. When the program starts, the effective user ID of the process is root (the file is owned by root). After the password verification, the process downgrades itself to the second user, so both real and effective user IDs become the second user, i.e., the process becomes non-privileged. After that, it runs the second user's default shell program. This is the functionality of the su program.

When a privileged process transitions to a non-privileged process, one of the common mistakes is capability leaking. The process may have gained some privileged capabilities when it was still privileged; when the privileges are downgraded, if the program does not clean up those capabilities, they may still be accessible by the non-privileged process. In other words, although the effective user ID of the process becomes non-privileged, the process is still privileged because it possesses privileged capabilities.

We use a program to demonstrate how capability can be leaked. Listing 1.1 shows a Set-UID root program. There are three steps in this program. First, it opens a file /etc/zzz that is only writable by root. After the file is opened, a file descriptor is created, and the subsequent operations on the file can be done using the file descriptor. File descriptor is a form of capability, because whoever carries it is capable of accessing the corresponding file. In the second step, the program downgrades its privileges by making its effective user ID (root) the same as the real user ID, essentially removing the root privilege from the process. In the third step, the program invokes a shell program.

Listing 1.1: Capability leaking example

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

void main()
{
  int fd;
  char *v[2];

  /* Assume that /etc/zzz is an important system file,
   * and it is owned by root with permission 0644.
   * Before running this program, you should create
   * the file /etc/zzz first. */
  fd = open("/etc/zzz", O_RDWR | O_APPEND);
  if (fd == -1) {
     printf("Cannot open /etc/zzz\n");
     exit(0);
  }

  // Print out the file descriptor value
  printf("fd is %d\n", fd);

  // Permanently disable the privilege by making the
  // effective uid the same as the real uid
  setuid(getuid());

  // Execute /bin/sh
  v[0] = "/bin/sh"; v[1] = 0;
  execve(v[0], v, 0);
}
```

Unfortunately, the above program forgets to close the file, so the file descriptor is still valid, and the process, which does not have privileges, is still capable of writing to `/etc/zzz`. From the execution result, we can see that the file descriptor number is 3. We can easily write to `/etc/zzz` using the command `"echo ... >&3"`, where `"&3"` means file descriptor 3. Before running the `Set-UID` program, we were not able to write to the protected file `/etc/zzz`, but after gaining the file descriptor via the `Set-UID` program, we can successfully modify it.

```
$ gcc -o cap_leak cap_leak.c
$ sudo chown root cap_leak
[sudo] password for seed:
$ sudo chmod 4755 cap_leak
$ ls -l cap_leak
-rwsr-xr-x 1 root seed 7386 Feb 23 09:24 cap_leak
$ cat /etc/zzz
bbbbbbbbbbbbbbb
$ echo aaaaaaaaaa > /etc/zzz
bash: /etc/zzz: Permission denied   ← Cannot write to the file
$ cap_leak
fd is 3
$ echo cccccccccccc >& 3            ← Using the leaked capability
$ exit
$ cat /etc/zzz
bbbbbbbbbbbbbbb
cccccccccccc                        ← File modified
```

To fix the above capability leaking problem in the program, we should destroy the capability before downgrading the privilege. This can be done by closing the file descriptor using `close(fd)`.

A case study: capability leaking in OS X. In July 2015, OS X Yosemite was found vulnerable to a privilege escalation attack related to capability leaking [Esser, 2015a]. In OS X 10.10, Apple added some new features to the dynamic linker `dyld`, and one of these features is the new environment variable called `DYLD_PRINT_TO_FILE`. Users can specify a file name in this environment variable to tell the dynamic linker to save error log information to this file. The dynamic linker runs inside the process running the program, so for a normal program, this new environment variable poses no risk, because the dynamic linker runs with the normal privilege. However, for `Set-UID` root programs, the dynamic linker runs with the root privilege, and can open any file. Before running a `Set-UID` program, users can set the environment variable to a protected file, such as `/etc/passwd`. When the `Set-UID` program is executed, the dynamic linker will open the file for write.

Unfortunately, the dynamic linker does not close the file. `Set-UID` programs do not know about the file, so they do not close it either. As a result, the file descriptor (a form of capability) is still valid inside the process. There are two scenarios here. In the first scenario, when a `Set-UID` program finishes its job, its process dies, so all its descriptors are naturally cleaned up; there is no harm. In the second scenario, such as in the case of the `su` program, the `Set-UID` program does not terminate; it invokes another program, usually untrusted, in a child process running with no special privileges. This has been secure, until `DYLD_PRINT_TO_FILE` was introduced: the file that is opened by the privileged `Set-UID` program will still be accessible by the non-privileged child process, because a child process inherits its parent's file descriptors. Using the `DYLD_PRINT_TO_FILE` environment variable and the `su` program, attackers can make arbitrary changes to any file, such as `/etc/passwd`, `/etc/shadow`, and `/etc/sudoer`. Consequently, they can gain the root privilege.

1.5 Invoking Other Programs

Invoking an external command from inside a program is quite common, but doing this needs to be extremely careful in `Set-UID` programs, because the privileged program may end up executing unintended programs provided by users, and can thus completely defeat the security guarantee (the security of a `Set-UID` program requires the program to only run its own code or trusted code, not users' arbitrary code).

In most cases, the external command is decided by the `Set-UID` program, and users are not supposed to choose the command, or there is no way to restrict the behavior of the `Set-UID` program. However, users are often required to provide inputs for the command. For example, a privileged program may need to send an email to users; it invokes an external email program to do this. The name of the email program is predefined by the privileged program, but users need to provide their email addresses, which will be given to the email program as command-line arguments. If the external email program is not invoked properly, these command-line arguments may cause user selected programs to be invoked.

1.5.1 Unsafe Approach: Using `system()`

There are many ways to execute an external command. The easiest way is to use a function called `system()`. We have discussed how environment variables can cause security problems in this approach. We will not repeat that here; we will focus on the argument part of the command.

Let us start with an example. Mallory works for an auditing agency, and she needs to investigate a company for a suspected fraud. For the investigation purpose, Mallory needs to be able to read all the files in the company's `Unix` system. However, to protect the integrity of the system, Mallory is not allowed to modify any file. To achieve this goal, Vince, the superuser of the system, wrote a special `Set-UID` program (see below), and gave the executable permission to Mallory. This program requires Mallory to type a file name at the command line, and then it will run `/bin/cat` to display the specified file. Since the program is running as root, it can display any file Mallory specifies. However, since the program has no write operations, Vince is very sure that Mallory cannot use this program to modify any file.

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  char *cat="/bin/cat";

  if(argc < 2) {
    printf("Please type a file name.\n");
    return 1;
  }

  char *command = malloc(strlen(cat) + strlen(argv[1]) + 2);
  sprintf(command, "%s %s", cat, argv[1]);
  system(command);
  return 0 ;
}
```

After compiling the above program (let us call it `catall`) , changing its owner to root, and enabling the `Set-UID` bit, Vince gives Mallory the executable permission, so she can run the program to view any file, including those that are only readable to root, such as `/etc/shadow`. Everything seems to be fine, but if we understand how the `system()` function works, we can easily use this `Set-UID` program to gain the root privilege.

If we type the `"man system"` command, we can get the manual of the function, which states that `system(command)` executes a command by calling `"/bin/sh -c command"`. In other words, the command is not directly executed by the above program; instead, the shell program is executed first, and then the shell will take command as its input, parse it, and execute whatever command is specified in it. Unfortunately, shell is too powerful; it can do many things beyond executing one single command. For example, in a shell prompt, if we want to type two commands in one line, we can use a semicolon (;) to separate two commands.

With the above knowledge about `system()`, Mallory can easily take over the root account using `catall`. She just needs to feed a string `"aa;/bin/sh"` to the program (the quotation marks should be included). As we can see from the following experiment results, shell actually runs two commands: `"/bin/cat aa"` and `"/bin/sh"`. Since `"aa"` is just a random file name, `cat` complains that the file does not exist, which is not something that we care about. Our focus is on the second command: we would like the `Set-UID` program to execute a shell program for us, so we can get a root shell. As indicated by the pound sign (#), the attack is successful, and we get the root privilege. We further confirm that by typing the `id` command, which shows that the `euid` (effective user ID) is root.

```
$ gcc -o catall catall.c
$ sudo chown root catall
$ sudo chmod 4755 catall
$ ls -l catall
-rwsr-xr-x 1 root seed 7275 Feb 23 09:41 catall
$ catall /etc/shadow
root:$6$012BPz.K$fbPkT6H6Db4/B8cLWb....
daemon:*:15749:0:99999:7:::
bin:*:15749:0:99999:7:::
sys:*:15749:0:99999:7:::
sync:*:15749:0:99999:7:::
games:*:15749:0:99999:7:::

$ catall "aa;/bin/sh"
/bin/cat: aa: No such file or directory
#        ← Got the root shell!
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root), ...
```

1.5.2 Safe Approach: Using `execve()`

Running a shell inside `Set-UID` programs is extremely dangerous, because shell is simply too powerful. The security of `Set-UID` programs depends on the proper restriction of its behaviors; running a powerful shell program inside makes such a restriction very difficult. All we need is to run a command, so why do we run such a powerful program ("middle man") to do that? A much safer approach is to cut out the "middle man", and run the command directly. There are many ways to do that, such as using `execve()` [Linux Programmer's Manual, 2017c]. See the following revised program.

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  char *v[3];

  if(argc < 2) {
    printf("Please type a file name.\n");
    return 1;
  }

  v[0] = "/bin/cat"; v[1] = argv[1]; v[2] = 0;
  execve(v[0], v, 0);

  return 0 ;
}
```

The `execve()` function takes three arguments: (1) the command to run, (2) the arguments used by the command, and (3) the environment variables passed to the new program. It will directly ask the operating system (not the shell program) to execute the specified command. The function is actually a wrapper for a corresponding system call, which does the actual job. If we include some additional commands in the second argument, they will be treated just as an argument, not as a command. That is why in the following experiment, `/bin/cat` complains that file `"aa;/bin/sh"` cannot be found, because this whole string is treated as an argument to the `cat` program.

```
$ gcc -o safecatall safecatall.c
$ sudo chown root safecatall
$ sudo chmod 4755 safecatall
$ safecatall /etc/shadow
root:$6$012BPz.K$fbPkT6H6Db4/B8cLWb....
daemon:*:15749:0:99999:7:::
bin:*:15749:0:99999:7:::
sys:*:15749:0:99999:7:::
sync:*:15749:0:99999:7:::
games:*:15749:0:99999:7:::

$ safecatall "aa;/bin/sh"
/bin/cat: aa;/bin/sh: No such file or directory    ← Attack failed!
```

Notes on the `exec()` family of functions. Several other functions, such as `execl`, `execlp`, `execle`, `execv`, `execvp`, and `execvpe`, behave similarly to `execve`. They all belong to the `exec()` family of functions. While they are similar functions, some of them have special semantics that make them dangerous for privileged programs. For example, according to the manual of `exec` [Linux Programmer's Manual, 2017b], "the `execlp()`, `execvp()`, and `execvpe()` functions duplicate the actions of the shell in searching for an executable file if the specified filename does not contain a slash (/) character. The file is sought in the colon-separated list of directory pathnames specified in the `PATH` environment variable". Basically, just like what we have discussed in the environment variable part of this chapter, these functions allow normal users to affect what programs to invoke via the `PATH` environment variable.

1.5.3 Invoking External Commands in Other Languages

The risk of invoking external commands is not limited to C programs; other programming languages have the same issue. When executing an external command in a privileged program, we should pay a close attention to the underlying mechanism used for command execution. We should avoid problems similar to those caused by the `system()` function. For example, in Perl, the `open()` function can run commands, but it does so through a shell, making it dangerous for privileged programs. PHP also contains a `system()` function, which works just like its C counterpart. It uses a shell to execute commands. Let us look at the following code snippet (`list.php`):

```
<?php
  print("Please specify the path of the directory");
  print("<p>");
  $dir=$_GET['dir'];
  print("Directory path: " . $dir . "<p>");
  system("/bin/ls $dir");
?>
```

The above script is meant to list the contents of a directory on the web server. The path of the directory is stored in the `dir` parameter, which is provided by users in the HTTP request. Since the script uses `system()` to execute an external command, attackers can send the following HTTP request to the server:

```
http://localhost/list.php?dir=.;date
```

After having received the above request, the PHP program will execute the `"/bin/ls .;date"` command, which is equivalent to two commands: `"/bin/ls ."` and `"date"`. The second command is selected by the attacker. In a real attack, the attacker can replace the benign `date` command with something that is more malicious, such as deleting a file, stealing some secrets, or setting up a reverse shell.

1.5.4 Lessons Learned: Principle of Isolation

The difference between `system()` and `execve()` reflects an important principle in computer security:

> **Principle of data/code Isolation:** Data should be clearly isolated from code.

What this implies is that if an input is meant to be used as data, it should be strictly be used as data, and none of its contents should be used as code (e.g. as the name of a command). If there is a mixture of data and code in the input, they should be clearly marked, so the computer systems will not mistakenly treat data as code. In the `system()` case, users are supposed to provide a file name, which should be strictly treated as data. However, the `system()` function does not support code/data isolation, so attackers can embed a new command or special characters (another form of code) in the input, leading to unintended code being executed. The `execve()` function clearly forces developers to break down their inputs into code (the first argument) and data (the second and third arguments), so there is no ambiguity.

There are many other vulnerabilities and attacks that can be attributed to the violation of this principle, including the cross-site scripting attack, the SQL-injection attack, two of the most popular attacks on web applications, and the buffer-overflow attack. We will revisit this principle when we discuss those attacks in the future chapters.

There is a cost when applying this principle: the loss of convenience. The `system()` function is more convenient to use than `execve()`, because you just need to put everything in a single string, as opposed to breaking them up manually into code and data. This kind of cost is quite normal, as we often say "there is no free lunch for security", which means, to be more secure usually requires a sacrifice of some degree of convenience. In this case, the sacrifice is not much, but in many other cases, it may be significant. A real security expert knows how to balance security and convenience.

1.6 Principle of Least Privilege

The `Set-UID` mechanism is quite useful, and `Unix` operating systems have many `Set-UID` programs. However, the design of this mechanism violates an important security principle:

> **Principle of Least Privilege**: Every program and every privileged user of the system should operate using the least amount of privileges necessary to complete the job [Saltzer and Schroeder, 1975].

Most of the tasks performed by a `Set-UID` program only need a portion of the power from root, not all, but they are given the full power of root. That is why when they are compromised, the damage is quite severe. This definitely violates the Principle of Least Privilege. According to this principle, a privileged program should only be given whatever power is necessary for it to perform its tasks. Unfortunately, most operating systems do not provide a sufficient granularity for privileges. For example, earlier `Unix` operating systems had only two levels of privileges, root and non-root. To provide a finer granularity, POSIX capabilities was introduced [Linux Programmer's Manual, 2017a]. They partition the powerful root privilege into a set of less powerful privileges. This way, a privileged program can be assigned the corresponding POSIX capabilities based on its tasks. Modern operating systems, such as Android, also provide fine-grained privileges. For example, Android has more than 100 permissions, each representing a privilege. An Android app needing to access GPS is only given the location permission, while apps requiring access to cameras are only given the camera permission.

There is another implication from this principle: if a privileged program does not need some privileges for part of its execution, it should disable the privileges either temporarily or permanently, depending on whether the privileges are still needed later on. By doing so, we can minimize the risk even if there are mistakes in the code.

`Set-UID` programs can use `seteuid()` and `setuid()` to enable/disable their privileges. The `seteuid()` call sets the effective user ID of the calling process. When a `Set-UID` program uses this call to set the effective user ID to its real user ID, it temporarily disables the privilege. The program can regain the privilege by calling it again to set its effective user ID to the privileged user.

It should be noted that disabling privileges does not make a program immune to all the attacks. Some attacks, such as buffer overflow, involve code injection, i.e., the `Set-UID` program is fooled to execute the code injected by attackers. For these attacks, even if the privileges are disabled temporarily, it does not prevent damages, because the malicious code can enable the privileges.

To permanently disable a privilege, `Set-UID` programs need to use `setuid()`, which also sets the effective user ID of the calling process, but if the effective user ID of the caller is root, the real and saved user ID are also set, making it impossible for the process to regain the privilege. Privileged processes usually use `setuid()` to downgrade their privilege before handling the control to a normal user. We have seen an example in Listing 1.1.

1.7 Summary

`Set-UID` is a security mechanism that allows normal users to gain temporary privileges when executing certain programs, allowing them to do what they cannot do with their own privileges, such as changing the `/etc/shadow` file to update their passwords. Because of the involved privilege escalation, one needs to be very careful when writing `Set-UID` programs; if a developer makes a mistake, normal users may be able to conduct unauthorized actions using the privileges obtained via a `Set-UID` program. In this chapter, we have systematically analyzed the risks faced by `Set-UID` programs, and showed a variety of vulnerabilities in `Set-UID` programs and how attackers can exploit them to gain privileges.

We use the `Set-UID` mechanism as an example to show that when a privileged program makes a mistake, it may lead to security breaches. There are many other types of privileged program, other than `Set-UID` programs. Some of the attacks discussed in this chapter are specific to `Set-UID` programs, but some are not. In future chapters, we will keep using

`Set-UID` programs as examples to demonstrate other types of vulnerability, such as buffer overflow, race condition, and format string vulnerabilities. However, those vulnerabilities are not specific to `Set-UID` programs, other privileged programs, such as OS kernel and root daemons, can also have those vulnerabilities.