SetUID Attack Lab

Brief

Set-UID is an important security mechanism in unix systems. When a Set-UID program is run, the executing program assumes the owner's privileges irrespective of the user running the program. The tasks are exploration based and are to be done on linux systems. The prebuilt VM called `seedubuntu` comes installed with all the required software. Various scenarios are duplicated using Set-UID programs. The problems with each are listed.

Task 1 : Familiarizing with `chsh`, `su`, `sudo`

1. `chsh` - This command helps change the current user's login shell to another desired one. It accepts any executable as long as it is listed in `/etc/shells` file.
2. `su` - This command is to switch user to another user specified one. It can also be used to do the job of sudo using `su -c`.
3. `sudo` - This command is used to specify that the following command is to be executed with the permissions of the super user (super user do). If `root` has given the super user access rights (the ability to elevate current access rights as a normal user to super user which can be equivalent to root), the user can use sudo to elevate the permissions.

NOTE - The sudo and su command combination can be used by a user with access rights to become the root user without knowing the password of the `root` user. `sudo su` is the command that changes to the `root` user if the current user has access rights.

The above commands need to be Set-UID because the actions done by all the above commands are to be executed only with the access rights of `root`. If any normal user is to use these commands, he would not be able to get the results as he does not have sufficient rights to let the execution carry out the actions. Therefore, these executables are made Set-UID by the `root`, so when a user wants to execute these commands, his access rights will be elevated to that of the owner (i.e., `root`), and as a result he will be able to let the executable carry out the actions and see the results.

Task 2 : Run Set-UID shell programs in linux

The permissions parameters as seen after ls command have more 3 bits to the left of the one that specifies if the file in question is a file or a link or a directory. The 1st of these 3 bits signifies if the program is Set-UID or not. This makes the 3 bits 100 i.e., 4. Therefore, to make an executable Set-UID, the command is `chmod 4<><><> <file>`, where <> are general permissions. After an executable has been made Set-UID, the `x` bit that signifies the executable rights of it, becomes `s`. The task is to login as root and make a copy of a shell program called `zsh` to the tmp folder using command `# cp /bin/zsh /tmp/testshell`. This will create the program `testshell` which is a copy of zsh. We make this a Set-UID to root executable by command `chmod 4755 /tmp/testshell`. Now we create another user `BOB` and do not give him any explicit rights. Using this new user we navigate to `/tmp`. Here on checking the output of `$whoami`, we see `BOB`. Now we can execute the `testshell`, upon which, the output of `whoami` changes to `root`. This happens because the `testshell` program is a Set-UID to root executable. Now even a user user who does not have high level of access rights can do anything in the system as he has become the root.

This problem is very grave and exists in the `zsh` shell program. The current default shell in linux is `bash`. On repeating the above task again with `/bin/bash` instead of using zsh, we see that upon executing `testshell`, now the output of `whoami` gives only `BOB`. Therefore, `bash` has protection against this flaw of Set-UID. Zsh on the other hand is not immune to this issue.

Task 3 : Setup for remaining tasks

As seen, `bash` does not share the Set-UID exploit vulnerability with `zsh`. Therefore, the remaining tasks will be explored on zsh itself. For this we create a symbolic link to zsh and remove the link before it if it is a link to bash. The series of commands are →

```
$ sudo su
  Enter password:
# cd /bin
# rm sh
# ln -s zsh sh
```

This has created a symbolic link to zsh as `/bin/sh` .

Task 4 : PATH variable and `system()` vulnerability

The `system()` command is one that executes the command by the command name that is passed into itself treated as a function. The way `system()` does this is by creating or calling the shell program `/bin/sh` . To exploit the vulnerability, consider the situation where the root user has created a program in c using a `system()` command to emulate the `ls` command. This program is owned by root and is compiled as test and made a Set-UID to root executable.

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
  system("ls");
  return 0;
}
```

The `PATH` variable is that which tells the system where commands are located. Since commands are basically just executables, to run them we need to specify a path like `/<directory>/<command>` . Giving the `PATH` variable ensures that to run the command, the entire path need not be given. When the `<directory>` part of the command is added in the `PATH` variable, any command we type on the terminal is searched in the directories listed in the `PATH` variable. If the command is present is one of these mentioned directories, it gets executed. As a result instead of typing `/<directory>/<command>` , we just type `<command>` . This is compiled using `$gcc -o test test.c` . Now basically, any user who runs the executable `test` will actually be executing `ls` in his current working directory with root permissions. The exploit is to set the `PATH` environment variable to just the current directory and then making a script called `ls` which calls a shell. `PATH` environment variable is the one that stores the paths of all the executable so that users need not specify the exact location of that command from the `/` directory or the present working directory to execute that command. Consider the user `BOB` , writes a script -

```
#!/bin/sh/bin/sh
```

This script is named `ls` and is made executable by `BOB` . Now `BOB` , sets the PATH to the current directory by using `$ export PATH=.` . Now he runs the Set-UID executable test and gets a shell. Upon using `whoami` , he sees that he is actually root.

```
$ ./test
# /usr/bin/whoami
root
#
```

On executing test, it actually calls ls on a shell. Since the PATH variable is set to the current directory, the `system()` command looks for `ls` only in the current directory where it finds the script which basically calls for the shell `/bin/sh` . Since the program is a Set-UID program, the shell called is the one with the access rights of the owner i.e., root. Since PATH is changed, the user needs to call all commands by giving absolute paths. But, he got the root shell. A similar exploit can be done by adding `/` to `IFS` environment variable.

Now, the same task is repeated but this time, `/bin/sh` is changed to point to `/bin/bash` . Zsh had the vulnerability, but bash on executing the same command, does give the shell but it has the same access rights as the user who called the `test` executable. Hence, the exploit fails.

Task 5 : Extension of task 4, `system()` vs `execve()`

Background: `BOB` works for an auditing agency, and he needs to investigate a company for a suspected fraud. For the investigation purpose, `BOB` needs to be able to read all the files in the company's Unix system; on the other hand, to protect the integrity of the system, Bob should not be able to modify any file. To achieve this goal, `Vince`, the superuser of the system, wrote a special set-root-uid program, and then gave the executable permission to `BOB`. This program requires `BOB` 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 a root, it can display any file `BOB` specifies. However, since the program has no write operations, `Vince` is very sure that `BOB` cannot use this special program to modify any file.

The code is as follows →

```
#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;
  int q = 0;
  if(q == 0)
  {
    char* command = malloc(strlen(v[0]) + strlen(v[1]) + 2);
    sprintf(command, "%s %s", v[0], v[1]);
    system(command);
  }
  else execve(v[0], v, 0);
  return 0;
}
```

Here, `q` is the variable control for us to shift between `system()` and `execve()` commands.

Initially, `q = 0`. Therefore, `system()` will get executed. This is an extension to the the exploit on Set-UID programs with `system()` command. Here, `BOB` can exploit Set-UID vulnerability to actually modify files (which is contrary to what he should be able to do according to `Vince`). To do this, `BOB` basically makes use of special characters used in shell scripting to modify files. Consider a file called `test.txt` which is owned by root. `BOB` can read the contents of the file by using the above program's executable which is a Set-UID to root executable (calls `/bin/cat` with root permissions). Now `BOB` can actually exploit the program by making use of `&&` and `;` operators. He can execute something like →

```
$ ./test "test.txt;mv test.txt ruty.txt"
<contents of text.txt>
```

After this operation, `test.txt` would have been renamed to `ruty.txt`. The same can also be accomplished using the `&&` operator as well as these have special meaning in shell programs. Another way to check the permissions is

```
$ ./test "test.txt && whoami"
<contents of text.txt>
root
```

Since the `system()` command passes the given parameters onto a shell, the `whoami` gets executed with the permissions of the owner of the program's Set-UID to root executable. Hence, we get root.

One possible way to save from this vulnerability can be to edit the program to include a regular expression match to rule out special characters like `$ & ;` .

Now, when we make the value of q equal to 1 in the c file `q = 1`, the program basically executes `execve()`. This is different from `system()` in the sense that it does not pass the arguments to a shell. It simply executes the program that is passed to it as the first argument. The program must be a binary executable or a script start with shebang directive. The output using `execve()` is →

```
$ ./test "test.txt;whoami"
/bin/cat: test.txt;whoami : No such file or directory
```

Therefore, as seen `system()` is very dangerous compared to `execve()` for Set-UID programs.

Task 6 : The `LD_PRELOAD` environment variable

For normal executions of programs, the linux dynamic loader finds and loads the shared libraries needed by the program. The shared libraries are loaded in whatever order the loader needs them in order to resolve symbols. An environmental variable called `LD_PRELOAD` contains the paths to shared libraries or shared objects. The loader will load these before any other library, even the C runtime library. This is called preloading a library. When a library is preloaded, the functions defined in it are used before the functions with the same name in other libraries that are loaded later. This enables library functions to be overwritten. As a result we can modify program actions without the need of recompilation. Eg - we write a library to define malloc and free. After loading this into the `LD_PRELOAD`, we can use the new definitions as the library will be preloaded. It is exported in the same way `PATH` variable is exported. This is used many times for Set-UID attacks. There is a mechanism which prevents the attacks which is to be explored in this task.

This task is about making a dynamic link library which will override the `sleep()` function. This will be used for Set-UID attack, and the conditions for which the loader will ignore the `LD_PRELOAD` so as to prevent the Set-UID attacks will be explored. The task has 4 different cases for this exploration.

The dll we make is →

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

We can compile this using →

```
$ gcc -fPIC -g -c mylib.c
$ gcc -shared -Wl,-soname,libmylib.so.1 -o libmylib.so.1.0.1 mylib.o -lc
```

The `LD_PRELOAD` can be set using →

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

Now we create a program to call `sleep()` →

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

The 4 cases are →

1. Run normal program by normal user → Made, compiled and run by normal user, the `sleep()` function does get overwritten. We get the "I am not sleeping message".

2. Run Set-UID to root program by normal user → `myprog.c` compiled by root and executable is Set-UID, the `sleep()` function is not overwritten. There is a wait for 1 second and then prompt returns.
3. Run Set-UID root program by root user → `myprog.c` compiled by root and executable is Set-UID, the `sleep()` function is not overwritten. There is a wait for 1 second and then prompt returns. But if the `LD_PRELOAD` is also exported by the root user, then the `sleep()` function gets overwritten and we see the "I am not sleeping" message.
4. Run Set-UID user program by other non root user → In this case too, `myprog.c` compiled by non-root and executable is Set-UID and `myprog` is called by another non-root user, the `sleep()` function is not overwritten. There is a wait for 1 second and then prompt returns.

The result is hence that the linker ignores the `LD_PRELOAD` environment variable if the real UID of the user id different than that of the same user's effective UID. This is a prevention mechanism against the Set-UID attacks.

Task 7 : Relinquishing privileges and cleanup

Too be secure, the Set-UID programs call `setuid()` system call to permanently relinquish their rot privileges. Sometimes, this is not enough.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

void main()
{
  int fd;
  fd = open("etc/zzz", O_RDWR | O_APPEND);
  if (fd == -1)
  {
    printf("cannot open it \n");
    exit(0);
  }
  sleep(1);
  setuid(getuid());
  if(fork())
  {
    close(fd);
    exit(0);
  }
  else
  {
    write(fd, "malicious data\n", 15);
    close(fd);
  }
  return 0;
}
```

We compile this as root and make the executable Set-UID to root. `setuid()` sets the effective user id of the current process. If the effective UID of the caller is root, the real UID and saved set-user-id are also set. We create a file `/etc/zzz` which is owned by root and the permissions are 0644. Now any other user should not be able to write to the file after the `setuid()` function is called and all the user privileges while the Set-UID was being executed is reset. On making the Set-UID to root executable and executing it by a different user, the result is that the file actually gets modified. The malicious data is actually appended to the file. Despite the presence of a function to drop all privileges, the write succeeded. This is so because the function was not placed properly. The `setuid()` was called only after the file was opened. To prevent other users from accessing files this way, the function should be declared before the opening of the file. This way the real UID and effective UID will be changed from root before the file is opened. If the function is called after the opening of the file, the privileges are dropped, but the program had already opened the file in write (append) mode as `root` itself.
