Race Conditions

Ibéria Medeiros

Departamento de Informática

Faculdade de Ciências da Universidade de Lisboa

Linux Kernel "Dirty COW" Vulner. (Oct 2016)

"Linux users are being urged to patch servers to fix a vulnerability known as "**Dirty COW**." The **privilege elevation flaw** is caused by "**a race condition** ... in the way the Linux kernel's memory subsystem handles the copy-on-write (COW) breakage of private read-only memory mappings." Linux vendor Red Hat says that the flaw is being **actively exploited**. The flaw has been present in the **Linux kernel since 2007**, and is **trivial to exploit**. Linus Torvalds acknowledged that he **tried, unsuccessfully, to fix the problem more than 10 years ago**."

Race conditions (I)

- Violation of an assumption of atomicity
  - ☞ during a **window of vulnerability** or window of opportunity or window of inopportunity
  - ☞ 2 (or more) entities access concurrently the same object
- The vulnerability is always due to a problem of concurrency / lack of proper synchronization
  - ☞ between a target and malicious process(es), ***or***
  - ☞ between several processes/threads of the target

- To exploit this kind of vulnerability, the attacker races to break the assumption during the window of vulnerability

Race conditions (II)

- Example: a service that generates unique sequential numbers
  - ☞ called by several threads concurrently
  - ☞ vulnerability allows returning the same number twice

```
int count = 0;    // global and shared by 2 threads

// Two threads execute this code concurrently
int getticket() {
    count++;
    return count;
}
```

There is an assumption of atomicity, which creates a window of vulnerability!

To address this problem, the access to getticket() needs to be synchronized using locks, semaphores, … or some other mechanism offered by the language.

Race conditions (III)

- Sources of races
  - ☞shared data: files and memory
  - ☞preemptive routines (signal handlers)
  - ☞multi-threaded programs

- We are going to study them in three scenarios
  - ☞TOCTOU (*Time-Of-Check to Time-Of-Use*)
  - ☞Temporary files
  - ☞Internal concurrency and reentrant functions

TOCTOU:
TIME OF CHECK – TIME OF USE

TOCTOU Time-of-check to time-of-use

aka TOCTTOU
also **symlink attack**

- Typical case:
  - A program running *setuid* root is asked to write to a file owned by the user running the program
  - Root can write to any file so the program has to check if the actual user has the right to write to the file

access() checks if the *Real UID* has write access to the file

```
// 0 if the user has write privilege
if(!access(file, W_OK)) {
    f = fopen(file, "w+");
    write_to_file(f);
}
else {
    fprintf(stderr, "Permission denied\n");
}
```

window vulnerab

Run this until success:
$ touch dummy
$ ln -s dummy link
$ program link &
$ rm link;\
  ln -s /etc/passwd link

access()

- int access(const char *pathname, int mode);
  - ☞ checks whether the calling process can access the file <u>pathname</u> according with <u>mode</u>
  - ☞ designed for setuid programs, does privilege check using the process' **real UID** instead of the **effective UID**

- However it is usually vulnerable to race conditions / TOCTOU
  - ☞ the file it checks can be altered before it is used
  - ☞ why? because it is identified by a <u>path</u>, so the object in the end can change
- <u>Should never be used!</u>

A real example (I)

- Broken **passwd** command (SunOS, HP/UX)
  - ☞ it allows for instance to change the password

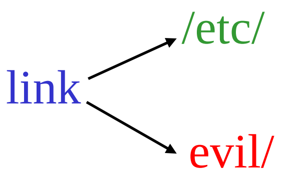
- passwd takes as parameter the password file to manipulate, then:

window vulnerab

1. Opens password file, retrieves user's entry, and closes file
2. Create and open temporary file (*ptmp*) in the same directory as password file
3. Open password file again, copy content to *ptmp* and update modified info
4. Close both files, rename *ptmp* to be the password file

A real example (II)

- Attack script (interleaved with desired passwd execution):



$ mkdir evil

$ cp /etc/passwd evil/passwd

$ echo “hacker::0:0:::/bin/bash” >> evil/passwd

$ ln -s /etc link

$ passwd link/passwd

passwd step 1: open link/passwd, get user entry, close file

passwd step 2: create *ptmp*

$ rm link ; ln -s evil link

passwd step 3: open and copy link/passwd to *ptmp*, update modified info

$ rm link ; ln -s /etc link

passwd step 4: close files; move *ptmp* to link/passwd (i.e. /etc/passwd)

- ***/etc/passwd*** *has a new user hacker with superuser privileges*

Putting it to work

- Running *passwd* and exploiting the window of vulnerability seems hard...
- Solutions:
  - ☞ make a script that tries as many times as needed until the attack works
  - ☞ delay the program someway
  - ☞ insert some random delays between operations

- Easier if the file system is distributed (e.g., NFS)

stat() function family

- Collect information about files and symbolic links
  - ☞ int stat(const char *pathname, struct stat *buf);
    - pathname – file to be checked
    - buf – structure to be filled with data about the file
  - ☞ lstat() – the same but data returned is about the link itself, if pathname is a link

- These calls give info about the file
  - ☞ Owner, owning group, number of hardlinks to the file
  - ☞ Type of file (regular, dir, char device, block device, named pipe, symbolic link, socket)

- Apparently, these functions could be helpful to distinguish real files from links …

Vulnerability with lstat()

- *setuid* program has to open a file but does not want to be tricked into opening a symbolic link
  - ☞ trick: use lstat() instead of access()

- Vulnerable code in Kerberos 4 lib used in login daemon

```
if (lstat(file, &statb) != 0) goto out;
if (!(statb.st_more & S_ISREG) || …) goto out;
if ((fd=open(file, O_RWDR|O_SYNC, 0)) < 0) goto out;
```

If it is **not** a regular file then exit.

- If file is not a link it is safe to open…

  but what if it changes after *lstat()* and before *open()* ?

Vulnerability with lstat (cont)

- Is it still vulnerable if done in the opposite order?

```
fd = open(fname, O_RDONLY);
if (fd == -1) perror("open");
if (lstat(fname, &statb) != 0) die("file not
  there");
if (!S_ISREG(statb.st_mode)) die("it's a symlink");
```

- Attacker creates link, then deletes it after the *open()* and before the *lstat()*

Preventing file race conditions (I)

- Most file races have to do with pathnames
- When a call with a *pathname* is done (open, access, stat, lstat,…), the pathname is resolved until the *inode* is found
- If two calls are made one after the other, the path can lead to different *inodes*

- *So the protection is to avoid the two sequential resolutions and use filenames only when strictly necessary inside the program*

Preventing file race conditions (II)

- Correct example – with *file descriptors*

```
fd=open("/tmp/bob", O_RDWR);
fstat(fd, &sb);
```

- If someone unlinks and re-links /tmp/bob between the two calls, fd would still point to the same *inode*

- Unsafe: access, stat, lstat, chmod, chown
- Safe: fstat, fchmod, fchown

Permission races

- Are we vulnerable in this case?

```
FILE *fp;  //stream
int fd;
if (!(fp = fopen("myfile.txt", "w+"))) die("fopen");
fd = fileno(fp);  // returns fd associated to stream
if (fchmod(fd, 0600)<0)  // fchmod() to prevent race
  die("fchmod");
```

- If file does not exist
  - *fopen()* creates it by calling *open()* with default permissions 0666
    - maybe lower as it is constrained by umask, but umask is inherited by the program
  - Program changes it to 0600 but it is too late, a race is possible
  - Attacker does not have to write in the file during the window, only open it for read/write (for instance)
  - *Solution: set umask*

Memory Races

- The following code is from a device driver in Windows 8.
- Are we vulnerable in this case?

```
PDWORD BufferSize = /* defined by a process in user mode */ ;
PUCHAR BufferPtr = /* defined by a process in user mode */ ;
PUCHAR LocalBuffer;
if (BufferSize > MAXSIZE) goto out;
LocalBuffer = ExAllocatePool(PagedPool, *BufferSize);
if (LocalBuffer != NULL) {
    RtlCopyMemory(LocalBuffer, BufferPtr, *BufferSize);
}
else goto out;
```

Copies an area of memory

- If the user changes the size of BufferSize between the check and its use, then he can write in areas of memory that were NOT supposed to (i.e., he can create a buffer overflow)

TEMPORARY FILES

Temporary files

- Have the same problems as others plus those derived from usually being in a shared dir
  - ☞ */tmp;* */var/tmp*
- Typical attack
  1. Privileged program checks that there is no file X in */tmp*
  2. Attacker races to create a link called X to some file, say */etc/passwd*
  3. Privileged program attempts to create X and opens the attacker's file doing something undesirable that its privileges allow …
- Two characteristics make the attack possible
  - ☞ the filename can be predicted by the attacker
  - ☞ the race condition between the check and the file creation

mktemp(), tmpnam(),…

- Trying to address the problem: *mktemp()* creates unique, currently unused, filename from a template

```
strcpy(temp, "/tmp/tmpXXXXXX");
if (!mktemp(temp))  die("mktemp");
fd = open(temp, O_CREAT | O_RDWR, 0700);
```

  ☞ after *mktemp()* and before *open()* an attacker can link filename to some file

  ☞ *open()* would then not create but open an existing file

- *tmpnam()* and *tempnam()* are similar

Possible solutions

- Non-solutions
  - ☞ Random file names: often the attacker can try the race many times, and if the name generator can be predicted …
  - ☞ Locks: many implementations are enforced by convention, not mandatory

- Acceptable solutions
  - ☞ Use long random number (e.g. 64 bits), set umask appropriately (e.g. 0066), and open the file
  - ☞ Use the safe calls → *see next slide*

mkstemp(), tmpfile(),…

- Possible solutions:

*int mkstemp(char *template)*

much safer than *mktemp()* since it checks for uniqueness, creates and opens with rw privileges only for the owner (0600)

*FILE *tmpfile(void)*

is similar, but in the past sometimes it was implemented on top of *mktemp(),* and therefore could be vulnerable to attacks

*char *mkdtemp(char *template)*

is similar but creates a directory with permissions 0700

CONCURRENCY AND REENTRANT FUNCTIONS

Concurrency

- In the previous cases, concurrency **is (mainly) created by the attacker**, with malicious intention
- In programs there is also the normal concurrency of operations that need to access shared objects
- Operations may have to be executed atomically, i.e., without interruption
- Mutual exclusion is a solution for <u>some</u> of the problems
  - ☞ mutexs, semaphores, transactions, etc.
- Difficulties with mutual exclusion
  - ☞ starvation: a thread is never scheduled for execution
  - ☞ deadlock: threads inter-block themselves

Reentrancy (I)

- A function is reentrant if several instances of the function can be executed in parallel in the same address space
  - ☞ for example, if it works correctly even if a thread is interrupted by another thread that calls the same function
  - ☞ Example:
    - – non-reentrant function supposed to give unique ticket

```
int count=0;    //global var
int getticket() {
  count++;
  return count;
}
```

atomic?

Reentrancy (II)

- Separating the reentrancy in two main sources of problems
  - ☞ Thread-safety – reentrancy in relation to several threads
  - ☞ Async-signal-safety – reentrancy in relation to signals

- Conditions for a function to be reentrant
  - ☞ <u>does not use</u>: static variables, global variables, other shared resources like libraries (i.e., uses only local non-static variables and function parameters)
  - ☞ <u>only calls</u> reentrant functions (namely libraries)

- Notice: conditions are **sufficient but not necessary**!
  - ☞ the use of global vars may ***not*** prevent the function from being reentrant

Signal handlers

- Signals are a Unix/POSIX mechanism to indicate asynchronous events to a process
  - ☞ often the semantics varies in different Unix flavors
- Can be treated by a signal handler (a function)
  - ☞ or ignored or blocked (except SIGKILL and SIGSTOP)
- Signal handlers have to be **asynchronous-safe** (or **async-safe** or **signal-safe**):
  - ☞ have to run correctly even if interrupted by other asynchronous events (i.e., by a signal) **and**
  - ☞ they should not corrupt the part of the program that was interrupted
- Otherwise they may be vulnerable (or create vulnerabilities) and often are!

Example of Signal Vulnerability

```
int size;
FILE *fd;
Char *bglobal;
int main(int argc, char **argv){
  signal(SIGTERM, complete);
  signal(SIGINT, complete);
  signal(SIGSEGV, complete);
  // open file for input
  fd = fopen("input.txt", "r");
  // now start two threads →
}
void thread_readData(){
  bglobal = malloc(MAX_BYTES);
  while (true) {
    wait_workDone(); // from thrd2
    size = fread(bglobal, 1,
                 MAX_BYTES, fd);
    more_work(); // warn thrd2
    printf("Read %d bytes.\n", size);
  }
}
```

```
void thread_workData(){
  char buffer[MAX_BYTES];
  while (true){
    wait_work(); //wait thread1
    memcpy(buffer, bglobal, size);
    done_work(); // warn thread1
    // does some processing
  }
}
int complete(int sig){
  printf("recv signal. Done!.\n");
  free(bglobal);
  fclose(fd);
  exit(1);
}
```

If the signal is executed :
What if complete is called two times?
What if signal occurs before memory is allocated or the file is open?
What if signal occurs when printf() was being executed?

Signals

- Some solutions
  - ☞ Read very carefully the man pages!!!
  - ☞ Make signal handler simple; ideally only set a flag
  - ☞ Use system calls safe for signal handlers
  - ☞ Block signals
    - (1) inside signal handlers and
    - (2) during non-atomic operations in the program

- Keep in mind that
  - ☞ mutual exclusion mechanisms typically cannot be used because they can lead to deadlocks (depending on how signals are configured)
  - ☞ alternatively, it is possible to use mechanisms that temporarily block the delivery of signals

Servlets

- Usually
  - ☞ Called by several threads concurrently
  - ☞ Each servlet is only one object in memory…

    … so class attributes are shared resources (like global vars in C)

- Servlet that can give two users the same count

```
public class Counter extends HttpServlet {
    int count = 0;    //shared!!
    public void doGet(…, HttpServletResponse out)
                       throws… {
        PrintWriter p = out.getWriter();
        count++;
        p.println(count + " hits so far!");
    }
}
```

There is an assumption of atomicity, which creates a window of vulnerability!

Solutions

```
public class Counter extends HttpServlet {
    int count = 0;    //shared!!
    public synchronized void doGet(…, HttpServletResponse out)
                                 throws… {
        PrintWriter p = out.getWriter();
        count++;
        p.println(count + " hits so far!");
    }
}
```

not efficient (if method longer…)

```
public class Counter extends HttpServlet {
    int count = 0;    //shared!!
    public void doGet(…, HttpServletResponse out) throws… {
        int my_count;  // NOT shared!!!
        PrintWriter p = out.getWriter();
        synchronized(this) {  my_count = ++count;  }
        p.println(my_count + " hits so far!");
    }
}
```

Bibliography

- M. Correia, P. Sousa, Segurança no Software, FCA Editora, 2017 (see chapter 6)

Other references: