RACE CONDITIONS

Software Security

Pedro Adão 2022/23
(with Ana Matos & Miguel Pupo Correia)

TÉCNICO LISBOA

Book: Chapter 4 (v1) / Chapter 6 (v2)

Motivation

Search Results (Refine Search)

There are **162** matching records.
Displaying matches **1** through **20**.

Search Parameters:

- **Keyword (text search):** race
- **Search Type:** Search Last 3 Years
- **Contains Software Flaws (CVE)**

1 2 3 4 5 6 7 8 9 > >>

CVE-2015-3247

Summary: Race condition in the worker_update_monitors_config function in SPICE 0.12.4 allows a remote authenticated guest user to cause a denial of service (heap-based memory corruption and QEMU-KVM crash) or possibly execute arbitrary code on the host via unspecified vectors.
Published: 9/8/2015 11:59:02 AM

CVSS Severity: 6.9 MEDIUM

CVE-2015-5189

Summary: Race condition in pcsd in PCS 0.9.139 and earlier uses a global variable to validate usernames, which allows remote authenticated users to gain privileges by sending a command that is checked for security after another user is authenticated.
Published: 9/3/2015 10:59:02 AM

CVSS Severity: 4.9 MEDIUM

CVE-2015-3212

Summary: Race condition in net/sctp/socket.c in the Linux kernel before 4.1.2 allows local users to cause a denial of service (list corruption and panic) via a rapid series of system calls related to sockets, as demonstrated by setsockopt calls.
Published: 8/31/2015 6:59:06 AM

CVSS Severity: 4.9 MEDIUM

Race conditions (I)

Attack: violation of an **assumption of atomicity**

–During a window of vulnerability or window of opportunity or window of inopportunity

–When 2 entities access concurrently the same object

Vulnerability: problem of concurrency / lack of proper synchronization

–between a target and malicious process(es); or

–between several processes/threads of the target

The attacker races to **break the assumption** during the window of vulnerability

Race conditions (II)

Example: service that produces unique sequential numbers

- Called by several threads concurrently
- Vulnerability allows returning the same number twice

```
int count = 0;    // shared


int getticket() {
   count++;
  ----------------
   return count;
}
```

Assumption of atomicity?
Window of vulnerability

Race conditions (III)

Sources of races

- Shared data: files and memory
- Preemptive routines (signal handlers)
- Multi-threaded programs

Mainly 3 kinds

- TOCTOU
- Temporary files
- Concurrency and reentrant functions

Example of running a race

TOCTOU

Quick review: symbolic links

Symbolic link, symlink or soft link

- old concept, Posix, Unix (including Linux, macOS), Windows

Symlink is a special file that references another file or directory

- full path of the other file
- akin to a pointer or reference in programs

Create a link in a shell:

```
ln -s original_path link_path
```

Other operations equal to files: `rm`, `mv`

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

window vulnerab

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

Run this until success:

```
$ touch dummy
$ ln -s dummy pointer
$ program pointer &
$ rm pointer;\
ln -s /etc/passwd pointer
```

Program may test one file and open another

access

Designed for setuid programs (!)

- Does privilege **check using the process' real UID** instead of the effective UID
  - i.e., checks if the user has access to the file, even if the program is running with **`effective UID <> real UID`**

However it is vulnerable to race conditions / TOCTOU

- The file it checks can be altered before it is used
- Why? Because it is identified by a path, so the object in the end can change

Should never be used

A real example (I)

Broken `passwd` command (old, SunOS, HP/UX)

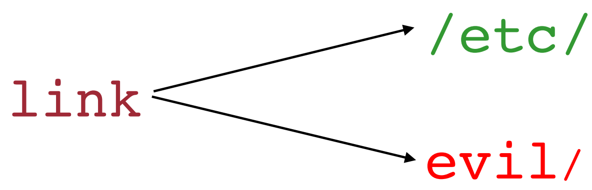
takes as parameter the password file (!) then:

window of vulnerability

1. Open password file, retrieve user's entry, close 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: 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 somehow
- Insert some random delays between operations

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

stat family

The culprits behind many TOCTOUs

```
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, if pathname is a link

These functions give information 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)

They test the file/link, but when it is used it may be another one

Problem is atomicity, not order (I)

setuid program has to open a file but does not want to be tricked into opening a symbolic link

- idea: to use `lstat()` instead of `access()`

Vulnerable code in Kerberos 4 lib used in login daemon:

is it a regular file (not a link)?

```
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 file is not a link it is safe to open…

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

Problem is atomicity, not order (II)

Still vulnerable if done in the opposite order (`open` first)

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

Attacker creates link, then substitutes it by some file after the `open()` and before the `lstat()`

- The file that is opened and processed is the one that is linked
- Some other file is checked and understood not to be a link

The problem is atomicity, not order

Preventing file race conditions (I)

Most file races have to do with the resolution of 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*

Solution: avoid the two sequential resolutions by avoiding using filenames 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*

a file descriptor is an integer, which is an index for an entry in a kernel array data structure containing the details of open files

Unsafe: `access`, `stat`, `lstat`, `chmod`, `chown`

Safe: `fstat`, `fchmod`, `fchown`

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…

mktemp, tmpnam,…

`mktemp()` creates unique, currently unused, filename from template

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

After `mktemp()` and before `open()` an attacker can link `temp` to some file

`open()` would then not create but open an existing file

`tmpnam` and `tempnam` are similar

NAME top

tempnam - create a name for a temporary file

SYNOPSIS top

```
#include <stdio.h>

char *tempnam(const char *dir, const char *pfx);
```

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

```
tempnam():
    Since glibc 2.19:
        _DEFAULT_SOURCE
    Glibc 2.19 and earlier:
        _BSD_SOURCE || _SVID_SOURCE
```

DESCRIPTION top

Never use this function. Use mkstemp(3) or tmpfile(3) instead.

mkstemp, tmpfile,…

`mkstemp` - much safer than `mktemp()` since it

1. **atomically checks for uniqueness,**
2. **creates and**
3. **opens with rw privileges**

`mkdtemp()` is similar but creates a directory

Temporary files (cont)

Problematic solutions:

Random file names - often the attacker can try the race many times…

Locks - many implementations are enforced by convention, not mandatory

Solutions:

Use long random numbers (e.g. 64 bits)

set *umask* appropriately (e.g. 0006)

open with *fopen*

Safe calls seen before

CONCURRENCY AND REENTRANT FUNCTIONS

Concurrency

In the previous cases, *concurrency is created by the attacker*, with malicious intention

In many cases in which there is normal concurrency of operations on objects, *o*perations may have to be executed **atomically**, i.e., without interruption

Solution: mutual exclusion mechanisms

locks, mutexes, semaphores, transactions, etc.

Problems:

1. Starvation: a thread is never scheduled for execution (problem of fairness)
2. Deadlock: threads inter-block themselves
3. Race conditions…

Reentrancy (I)

A function is reentrant

Definition: A function is reentrant If it works correctly even if its thread is interrupted by another thread that calls the same function

i.e., if several instances of the function can be executed in parallel in the same address space

Example of a non-reentrant function supposed to give unique tickets

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

atomic?

Reentrancy (II)

Similar notions

- Thread-safety – reentrancy in relation to multi-threading
- Async-signal-safety – reentrancy in relation to signals

Sufficient conditions for function being reentrant:

- (i.e., if these are satisfied, then function is reentrant)
- Cannot use: static variables, global variables, other shared resources like libraries (i.e., uses only local non-static variables and function parameters)
- Can only call reentrant functions

Necessary conditions for function being reentrant:

- The sufficient conditions don’t need all to be satisfied for the function to be reentrant
- Using static/global variables or other shared resources is ok if they are only read

Signal handlers

Signals are a Unix/POSIX mechanism used to indicate asynchronous events to a process

Can be treated by a *signal handler* (a function) or ignored or blocked

Signal handlers have to be **asynchronous-safe** (or **signal-safe**):

- Have to run safely and correctly even if interrupted by asynchronous events (i.e., by a signal)

Otherwise they may be vulnerable, and often are

Signal vulnerability example

```
char *user;
int cleanup(int sig) {
   printf("Cleaning up\n");
   free(user);
   exit(1);
}

int main(int argc, char **argv) {
   signal(SIGTERM, cleanup);
   signal(SIGINT, cleanup);
   …
   process_file(fd);
   free(user);
   close(fd);
   printf("bye!\n");
   return 0;
}
```

```
int process_file(int fd) {
    char buffer[1024];
    …read from file into buffer…
    user=malloc(strlen(buffer)+1);
    strcpy(user, buffer);
    …
    return 0;
}
```

What if there is a signal…

1. after `free(user)`: there is a double free (corrupts heap)
2. during `strcpy`: data would be copied into a free heap block (mem corruption, arbitrary code exec)
3. anywhere: if SIGINT then SIGTERM fast (or vice-versa) → double free

Signals

Some solutions

- Make signal handler simple; ideally only set a flag
- Use system calls safe for signal handlers
- Block signals
  - inside signal handlers and
  - during non-atomic operations in the program

Java servlets

Usually

- Called by several threads concurrently (typically)
- Each servlet is only one object in memory…
- …so class attributes are shared resources (like global vars in C)
- Bad solution: implementing SingleThreadModel interface; bad performance

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!");
    }
}
```

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 was longer…)

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

Prevention

Identify resources that are used by several threads

- Typically global variables, global data structures

Use proper mutual exclusion mechanisms

- Sometimes the mechanisms are used but
  - e.g. a lock is released too early
- Example: sys_uselib in the Linux kernel used a semaphore but released it before a call that should also be in the critical section → a race could be used to get root privileges

Summary

Race conditions are violations of assumptions of atomicity during a *window of vulnerability*

Usual objectives are *privilege escalation* and attacks against web applications

Common vulnerabilities:

- TOCTOU – File access
- Temporary files
- Non-reentrant functions – signals, servlets