Race Conditions

March 27, 2006

Concurrency and Race condition

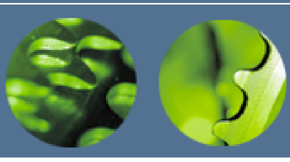
- Concurrency
  - – Execution of Multiple flows (threads, processes, tasks, etc)
  - – If not controlled can lead to nondeterministic behavior
- Race conditions
  - – Software defect/vulnerability resulting from unanticipated execution ordering of concurrent flows
    - E.g., two people simultaneously try to modify the same account (withrawing money)

Race condition

- Necessary properties for a race condition
  - Concurrency property
    - At least two control flows executing concurrently
  - Shared object property
    - The concurrent flows must access a common shared *race object*
  - Change state property
    - Atleast one control flow must alter the state of the race object

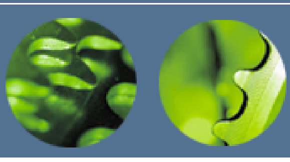
Race window

- A code segment that accesses the race object in a way that opens a window of opportunity for race condition
  - Sometimes referred to as critical section
- Traditional approach
  - Ensure race windows do not overlap
    - Make them mutually exclusive
    - Language facilities – *synchronization primitives (SP)*
  - *Deadlock* is a risk related to SP
    - Denial of service



Time of Check, Time of Use

- Source of race conditions
  - Trusted (tightly coupled threads of execution) or untrusted control flows (separate application or process)
- ToCToU race conditions
  - Can occur during file I/O
  - Forms a RW by first *checking* some race object and then *using* it



Example

```
int main(int argc, char *argv[]) {
        FILE *fd;
        if (access("/some_file", W_OK) == 0) {
                printf("access granted.\n");
                fd = fopen("/some_file", "wb+");
                /* write to the file */
                fclose(fd);
        }  else {
                err(1, "ERROR");
        }
        return 0;
} Figure 7-1
```

- Assume the program is running with an effective UID of root

TOCTOU

- Following shell commands during RW

```
rm /some_file
ln /myfile /some_file
```

- Mitigation
  - Replace access() call by code that does the following
    - Drops the privilege to the real UID
    - Open with fopen()
    - Checks to ensure that the file was opened successfully

TOCTU

- Not all untrusted RCs are purely TOCTOU
  - E.g., GNU file utilities

```
chdir(“/tmp/a”);
chdir(“b”);
chdir(“c”);
//race window
chdir(“..”);
chdir(“c”);
ulink(“*”);
```

  - Exploit is the following shell command

    mv /tmp/a/b/c /tmp/c

    - Note there is no checking here - implicit

File locking

- SP cannot be used to resolve RC from independent processes
  - Don't have shared access to global data
- File locks can be used to synchronize them

```
int lock(char *fn) {
  int fd;
  int sleep_time = 100;
  while (((fd=open(fn, O_WRONLY | O_EXCL | O_CREAT, 0)) == -1)
           && errno == EEXIST) {
    usleep(sleep_time);
    sleep_time *= 2;
    if (sleep_time > MAX_SLEEP)
      sleep_time = MAX_SLEEP;
  }
  return fd;
}

void unlock(char *fn) {
  if (unlink(fn) == -1) {
    err(1, "file unlock");
  }
} Figure 7-3
```

File locking

- Two disadvantages
  - Open() does not block
    - Use sleep_time that doubles at each attempt (also known as *spinlock* or *busy form of waiting*)
  - File lock can remain locked indefinitely (e.g., if the locking process crashes)
    - A common fix is to store the PID in the lock file, which is checked against the active PID.
    - Flaws with this fix
      - PID may have been reused
      - Fix itself may contain race conditions
      - Shared resource may also have been corrupted because of the crash

File System Exploits

- Files and directories are common race objects
- Open files are shared by peer threads
- File systems have exposure to other processes
    - As file permissions
    - File naming conventions
    - File systems mechanisms
  - Most executing programs leave a file in a corrupted state when it crashes (backup is remedy)
- Exploits
  - Symbolic linking exploits
  - Temporary file open exploits
  - ulink() race exploit
  - Trusted filenames
  - Nonunique temp file names

Symbolic linking exploits

- Unix symbolic linking is most common
  - Symlink is a directory entry that references a target file or directory
  - Vulnerability involves programmatic reference to a filename that unexpectedly turns out to include a symbolic link
    - In the RW the attacker alters the meaning of the filename by creating a symlink

Symbolic linking exploits

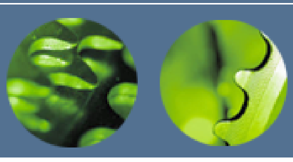
```
if (stat("/some_dir/some_file", &statbuf) == -1) {
   err(1, "stat");
}
if (statbuf.st_size >= MAX_FILE_SIZE) {
   err(2, "file size");
}
if ((fd=open("/some_dir/some_file", O_RDONLY)) == -1) {
   err(3, "open - %s",argv[1]);
} Figure 7-4
```

Attacker does:

rm
/some_dir/some_file

Symbolic linking exploits

- Reason for its wide spread use in exploits
  - Creation of symlink is not checked to ensure that the owner of the link has any permissions for the target file, nor
  - Is it even necessary that the target file exists
  - The attacker only needs write permissions to the directory in which symlink is created
- Further complication introduced by the following
  - Symlink can reference a directory
    - E.g., in some passwd() function – required user to specify a password file as a parameter



Symbolic linking exploits

- Vulnerable segment in passwd()

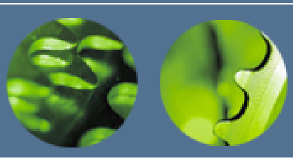
  Open the password file, use it to authenticate the user, and then close the file

  Create and open a temporary file called ptmp in the directory of the password file

  Reopen the password file and copy an updated version into ptmp (which is still open)

  Close both files and rename ptmp as the new password file

- Exploit allows entry to an account
  - A creates a bogus attack_dir/.rhosts – A is a valid user
  - V has real password file in victim_dir
  - A creates symlink to attack_dir called symdir
  - A calls passwd() passing the password file as /symdir/.rhosts



Symbolic linking exploits

- Vulnerable segment in passwd()

  Open the pssword file, use it to authenticate the user, and then close the file

  - attacker changes /symdir to attack_dir

  Create and open a temporary file called ptmp in the directory of the password file

  - allow use of victim_dir

  Reopen the password file and copy an updated version into ptmp (which is still open)

  - attacker changes /symdir to attack_dir

  Close both files and rename ptmp as the new password file

  - allow use of victim_dir

  Result:

  The password file in victim_dir is replace by that from the attack_dir

Symbolic linking exploits

- Slightly different symlink vulnerability
  - Permissions are threatened (elevated)
  - The attack works because of the following
    - When permissions are changed on a symbolic link, the change is applied to the target file rather than the link
- Windows “shortcut” is similar
  - But windows rarely have symlink problem because
    - The API includes primarily file functions that depend on file handles rather than the file names, and
    - Many programmatic windows functions do not recognize shortcuts as links

Temporary file open exploits

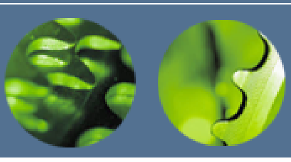
- Temporary files
  - Vulnerable when created in a directory where attacker has access
  - In unix /tmp is frequently used for temporary files
  - Simple vulnerability

```
int fd = open(“/tmp/some_file”,
                O_WRONLY |
                O_CREAT |
                O_TRUNC,
                0600)
```

What if the /tmp/some_file is a symbolic link before the instruction is executed?

Solution:
add O_EXCL flag

File existence check and creation -> atomic!



Temporary file open exploits

- Stream functions in C++ have no atomic equivalent

mitigation

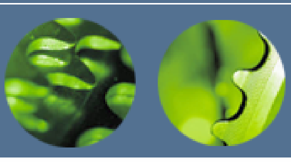
```
int main(int argc, _TCHAR* argv[])
{
  ofstream outStrm;
  ifstream chkStrm;
  chkStrm.open("/tmp/some_file",,
         ifstream::in);
  if (!chkStrm.fail())
         outStrm.open("/tmp/some_file",
                        ofstream::out);
         .
         .
}
```

Race window?

```
int main(int argc, char *argv[])
{ int fd;
  FILE *fp;
  if ((fd = open(argv[1],
      O_EXCL|O_CREAT|O_TRUNC|O_RDWR,
      0600)) == -1) {
    err(1, argv[1]);
  }
  fp = fdopen(fd, "w");
  :
  :

}
```

File descriptor + O_EXCL



Temporary file open exploits

- Exploit would be possible if the filename can be guessed before a process creates it
- Random filename using mkstemp()
  - Each X is replaced by a random character

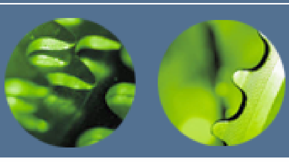
```
char template[] = “/tmp/fileXXXXXX”;
if (fd = mkstemp(template)) = -1) {
   err(1, “random file”);
}
```

ulink Race exploits

- RC is created when
  - A file is opened and later unlinked
  - Key reason, Linux does not support an equivalent to unlink() that uses a file descriptor
    - Replacing the named open file with another file or symbolic link, an attacker can cause unlink() to be applied to the wrong file
    - Mitigation: proper permissions on the directory

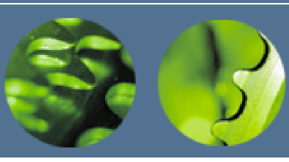
Trusted filenames

- Trusted filename vulnerability
  - Results as a result of unverified filenames
    - Filenames from user or untrusted source
- Goal of exploit
  - Cause a program to manipulate a file of attacker’s choosing
  - Mitigation: verify the filename
- Some difficulties
  - Different length restrictions, remote file systems & shares, etc.
  - Device as a file (some OSs crash)
  - Inclusion of substring “..”
  - General mitigation: transform to canonical form
    - Generate an absolute path without “..”, “.” or symbolic links
    - Unix – realpath()
      - Care must be taken to avoid TOCTOU condition using realpath() to check a filename
    - Another mitigation is to validate ancestral directories.



Nonunique Temp File Names

- Faulty implementation
  - Of tempnam() and tempfile() can produce non unique filenames (using a uers ID)
  - tmpnam_s() generates a valid filename that is not the name of an existing file
    - RC is still possible if the name is guessed before use



Mitigation strategies

- Can be classified based on properties
  - Mitigations that remove concurrency property
  - Techniques that eliminate the shared object property
  - Ways to mitigate by controlling access to the shared object to eliminate the change state property
- Different strategies may/should be combined

Mitigation strategies

- Closing the race window
  - Eliminate RW whenever possible
- Techniques
  - Mutual exclusion
  - Thread safe functions
  - Use of atomic operations
  - Checking file properties safely
  - Use file descriptors not filenames
  - Shared directories
  - Temporary files

Mitigation strategies

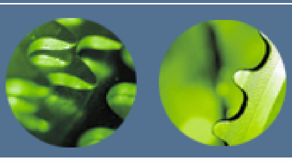
- Mutual exclusion
  - Implement mutually exclusive critical sections
    - Mutex/semaphores
    - Critical issue is to minimize CS size
  - Object-oriented alternative
    - Use decorater module to isolate access to shared resources
    - provides wrapper functions
  - Signal handling poses problems
    - Signals can interrupt normal execution flow at any time
    - Unhandled signals usually default to program termination
    - A signal handler can be invoked at any time, even in the midst of a mutually excluded section of code
    - If the attacker sends a signal to a process within a race window, it is possible to use signal handling to effectively lengthen the window
    - Mitigation:
      - Signal handling should not be used for normal functionality
      - Avoid sharing objects between signal handlers and other program code

Thread safe function

- In Multithreaded applications
  - It is not enough to ensure code is RC free
  - It is possible that invoked functions could be responsible for race conditions
- Thread safe function
  - No RC when concurrent calls to this function
  - If non-thread safe function is called, treat it as a critical section

Use of atomic operations

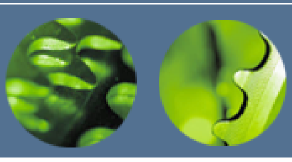
- Atomicity
  - Implemented by synchronization functions
- Entry to critical section
  - Should not be interrupted until completed
  - Concurrent executions of EnterCriticalRegion() should not overlap
  - Concurrent execution of EnterCriticalRegion() should not overlap with the execution of LeaveCritcalSection()
- Open() with O_CREAT and O-EXCL
  - Alternative is to call stat() or access() followed by open() – may introduce TOCTOU



Checking file properties securely

```
struct stat lstat_info;
int fd;
if (lstat(“some_file”, &lstat_info) == -1) {
  err(1, "lstat");
}
if (!S_ISLNK(lstat_info.st_mode)) {
  if ((fd = open(“some_file”, O_EXCL | O_RDWR, 0600)) == -1)
    err(2, argv[1]);
}
```

- lpstat() is a difficult problem
  - Stats a symbolic link
    - No file descriptor alternative
- Mitigation – follow the four steps
  - lpstat() the filename
  - open() the file
  - fstat() the file descriptor from step 2
  - Compare the results from steps 1 and 3



Checking file properties securely

- The four steps are used in the following

```
struct stat lstat_info, fstat_info;
int fd;
if (lstat(“some_file”, &lstat_info) == -1) {
  err(1, "lstat");
}
if ((fd = open(“some_file”, O_EXCL | O_RDWR, 0600)) == -1) {
  err(2, "some_file");
}
if (fstat(fd, &fstat_info) == -1)
{
  err(3, "fstat");
}
if (lstat_info.st_mode == fstat_info.st_mode &&
    lstat_info.st_ino == fstat_info.st_ino)
      //process the file
```

Eliminating the race object

- RC exists because of
  - Concurrent execution flows share some object
- Hence, RC can be eliminated by
  - Eliminating shared objects, or
  - Removing shared access to it
- Mitigation
  - Identify the shared object (file system is key)
  - Use file descriptors, not file name
    - File's directory is key element
    - Once a file is opened, it is not vulnerable to symlink attack if the file descriptor is used instead of file/directory
  - Shared directories – avoid it
  - Temporary files: /tmp is key source (commonly shared)

Eliminating the race object

- Temporary files: some good practices
  - Never reuse filenames, especially temporary files
  - Use random files names for temporary file – avoids conflict and guessing
    - Use cryptographically strong random number generator and seeds
  - Use mkstemp() instead of mktemp(), tempnam(), etc.
  - Unlink temporary files as early as possible
    - Reduces the RW
  - Log temporary file events

Controlling access to the race object

- Some techniques
  - Principle of least privilege
    - Eliminates RC or reduce exposure
      - If possible, avoid running processes with elevated permissions
      - When a process must use elevated permissions, these should be normally dropped (using setuid())
      - When a file is created, the permissions should be restricted exclusively to the owner
  - Trustworthy directories
  - Chroot jail
    - Creates an isolated directory with its own root/tree
      - Avoids symlink, ".." exploits

Race detection tools

- Static analysis
  - Parses software to identify race conditions
  - Warlock for C (need annotation)
  - ITS4 uses (database of vulnerabilities)
  - RacerX for control-flow sensitive interprocedural analysis
  - Flawfinder and RATS – best public domain
- Extended Static checking
  - Use theorem proving technology
- Race condition detection is NP complete
  - Hence approximate detection
  - C/C++ are difficult to analyze statically –
    - pointers and pointer arithmetic
    - Dynamic dispatch and templates in C++

Race detection tools

- Dynamic analysis
  - Detect during execution
  - Disadvantages
    - Fails to consider execution path not taken
    - Runtime overhead
  - Some tools
    - Eraser, MultiRace
    - ThreadChecker (intel) – finds races and deadlocks
    - RaceGaurd for unix – secure use of temp files