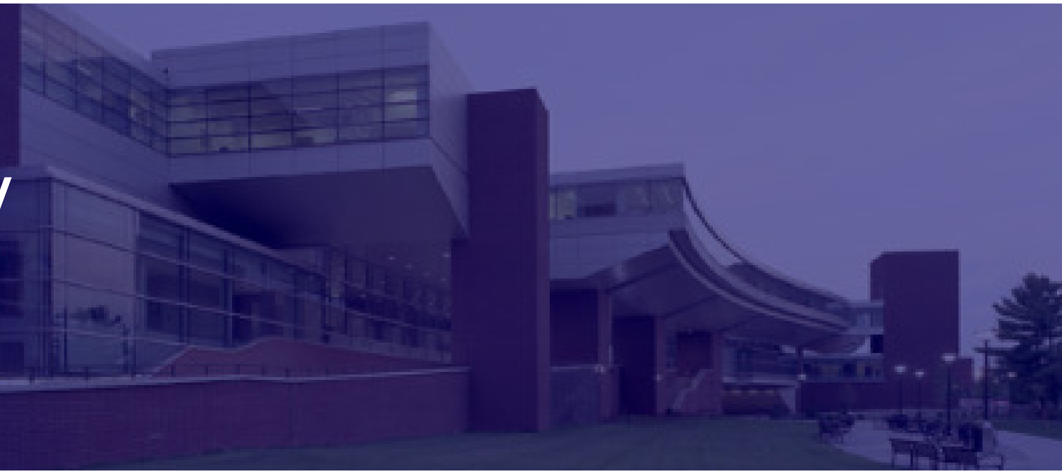
Network and Security Research Center
Department of Computer Science and Engineering
Pennsylvania State University, University Park PA



Exploiting Unix File-System Races via Algorithmic Complexity Attacks

By Cai, Gui, Johnson

Presented By: Philip Koshy

Background

- Unix (and its variants) offers a rich system call interface. Some examples include:

  `access()`

  Checks permissions on a file

  `open()`

  Opens a file

  `link()`

  Creates a link to an existing file

  `unlink()`

  Deletes a link to a file (can also delete the file)

Background

`setuid()`

When a program is executed, run the program with the privilege of the owner (which is typically the root user).

This function is the root of all evil. (Pun intended.)

setuid()

- The ‘passwd’ utility is owned by root but can be run by unprivileged users. (i.e., mode bits are 755)
- This utility needs to modify sensitive system files (e.g, /etc/shadow) which are owned by root.
- When passwd runs, it runs as root, instead of the unprivileged user.
  - Otherwise, no user could change their password!

But…

- It seems as if an unprivileged user can read/modify privileged files?
  - Only in ways authorized by the setuid-root program in question
  - In our example, passwd is part of our TCB

- Utility programmers can use access() to check the permission of the real uid (i.e., unprivileged user), as opposed to the effective user id (i.e., root)

- If we call access() before open(), we should be safe…right?

Simple setuid-root program

```
void main(int argc, char **argv)
{
     int fd;

     if (access(argv[1], R_OK) != 0)
          exit(1);

     fd = open(argv[1], O_RDONLY);
}
```

So far so good…

T
i
m
e

access(“file”, R_OK)

open(“file”, O_RDONLY)

The problem

Victim

Attacker

T i m e

access(“file”, R_OK)

Context Switch →

unlink(“file”)

link(“/etc/shadow”, “file”)

← Context Switch

open(“file”, O_RDONLY)

This file contains passwords (albeit encrypted…)

Code Snippet

Victim Code

```
int main(int argc, char **argv)
{
  int fd;
  /* If my invoker cannot access
     argv[1], then exit. */
  if (access(argv[1], R_OK) != 0)
    exit(1);
  fd = open(argv[1], O_RDONLY);
  /* Use fd... */
}
```

Figure 1:
A setuid-program uses the insecure access(2)/open(2) design pattern.

Attacker Code (Calls the Victim Code)

```
int main(int argc, char **argv)
{
  /* Assume “file” refers to a file
     readable by the attacker. */
  if (fork() == 0) {
    system(“victim file”);
    exit(0);
  }
  usleep(1);

  unlink(“file”);
  link(“/etc/shadow”, “file”);
}
```

Figure 2:
Exploitation of the vulnerable program in Figure 1.

The result

- The attacker can force a poorly written setuid-root program into opening a file for which the user does not have access.

- This is due to the imprecision inherent in treating Unix file-system paths as simple strings.

Key Problem

T
i
m
e

access("file", R_OK)

open("file", O_RDONLY)

Programmers incorrectly assume these calls are 'Atomic.'

This leads to the TOCTTOU attack. (Time Of Check To Time Of Use)

A race to the finish!

Victim Attacker

T
i
m
e

If the attacker can interleave code correctly, they win!

Existing Defenses

- This paper invalidates two previously proposed defenses to file-system TOCTTOU attacks.
  - Atomic k-Race
  - TY-Race

- We look mostly at the Atomic k-Race defense

- The TY-Race is trivially broken if any victim system call is delayed by more than 2 seconds.
  - We'll see how this is possible later.

More Background

- We've seen that the (A)ccess/(O)pen design pattern can be broken.
  - This is possible because the attacker can interleave commands.
  - The original call sequence of AO became AsO

    Where s = switch to a secret file

- We need a way to make sure that the file we've checked for access is the file we've actually opened.

- What if we can determine something akin to the identity of a file before and after usage? (Maybe the inode number?)

System Calls

- One measure of a file's identity is the file's status, which takes the form of a 'stat' structure

- The status information includes:
  - ID of the device containing the file
  - inode number
  - Mode bits of the file
  - number of hard links
  - user and group id
  - Many more fields…

Most reliable metric of file 'identity'

System Calls

- The status information can be retrieved using two system calls:

```
lstat( char* path, struct stat* status )
```

Retrieves the status of a file. If the file is a symlink, it retrieves its status (as opposed to the underlying file.)

```
fstat( int fd, struct stat* status )
```

Retrieves the status of an already open file.

A revised approach

- What if we called the following in sequence?

  - Lstat(char* path) // Get unique identity of path
  - Access(char* path)
  - Open(char* path)
  - Fstat(int fd) // Get unique identity of opened file

If the result of lstat() != fstat(),
we likely have a problem.

A revised approach

PENNSTATE 1855

- Unfortunately, the LAOF sequence is still susceptible to file-system races!
- LAOF may becomes sLaAsOF
  Where s = switch to a secret file
  a = switch to an accessible file
- The attacker would need to use lstat() with the same secret file he wants to open().
- The access check is invalidated if the attacker can reroute the check to a file he has access to.

Atomic k-Trace motivation

- We can increase the LAOF sequence's tolerance to failure.
- If we repeatedly apply the LAOF sequence, we can achieve a probabilistic defense.
- If we repeat LAOF k-times, how likely is it that an attacker can interleave code *every* time?
  - It was assumed to be difficult
  - Spoiler: It's not.

Atomic k-Race

```
int atom_race(const char *atom,
              struct stat *s0)
{
  int i, mode;
  int fd1, fd2;
  struct stat s1, s2;

  mode = S_ISDIR(s0->st_mode) ?
           X_OK : R_OK;

  DO_SYS(access(atom, mode));
  DO_SYS(fd1 = open(atom, O_RDONLY));
  DO_SYS(fstat(fd1, &s1));
  DO_CHK(DO_CMP(s0, &s1));

  for (i = 0; i < krounds; i++) {
    DO_SYS(lstat(atom, &s1));
    DO_CHK(!S_ISLNK(s1.st_mode));
    DO_SYS(access(atom, mode));
    DO_SYS(t = open(atom, O_RDONLY));
    DO_SYS(fstat(t, &s2));
    DO_SYS(close(t));
    DO_CHK(DO_CMP(s0,&s1));
    DO_CHK(DO_CMP(s0,&s2));
  }
  return fd1;
}
```

Figure 5. The heart the atomic k-race defense mechanism. Before calling this function, the main atomic k-race routine ensures that `atom` is a single path component (i.e. it contains no "/"), calls `lstat(atom, s0)`, and checks the resulting `s0` to verify that `atom` is not a symbolic link.

Atomic k-Race

```
int atom_race(const char *atom,
              struct stat *s0)
{
  int i, mode;
  int fd1, fd2;
  struct stat s1, s2;

  mode = S_ISDIR(s0->st_mode) ?
           X_OK : R_OK;

  DO_SYS(access(atom, mode));
  DO_SYS(fd1 = open(atom, O_RDONLY));
  DO_SYS(fstat(fd1, &s1));
  DO_CHK(DO_CMP(s0, &s1));

  for (i = 0; i < krounds; i++) {
    DO_SYS(lstat(atom, &s1));
    DO_CHK(!S_ISLNK(s1.st_mode));
    DO_SYS(access(atom, mode));
    DO_SYS(t = open(atom, O_RDONLY));
    DO_SYS(fstat(t, &s2));
    DO_SYS(close(t));
    DO_CHK(DO_CMP(s0,&s1));
    DO_CHK(DO_CMP(s0,&s2));
  }
  return fd1;
}
```

This algorithm is essentially LAOF, repeated 'krounds' times.

The security of Atomic k-Race is p^{2k+2} where p is the attacker's ability to win a *single* race.

The attack vector

- The authors show they can deterministically do an arbitrary amount of work between the victim's system calls.

- They can control the OS scheduler.

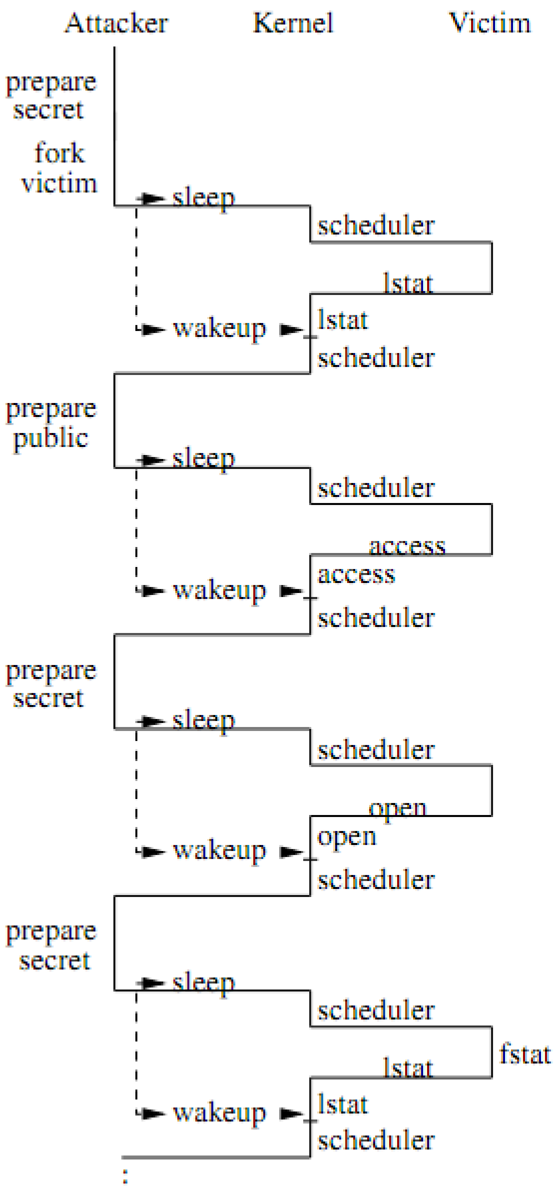
- The paper shows that the successful interleaving in LAOF for *multiple* iterations can be achieved with very high success rates.

The attack vector

- Pseudo-code:

```
setup(secret-file, fname)
if fork() == 0
    exec("victim fname")
for i = 0, . . . , k
    sleep(syscall-duration)          // lstat
    prepare(public-file, fname)
    sleep(syscall-duration)          // access
    prepare(secret-file, fname)
    sleep(syscall-duration)          // open
    prepare(secret-file, fname)
    // fstat, close take negligible time
```

sLaAsOsF



Prepare Secret

Lstat()

Prepare Accessible

Access()

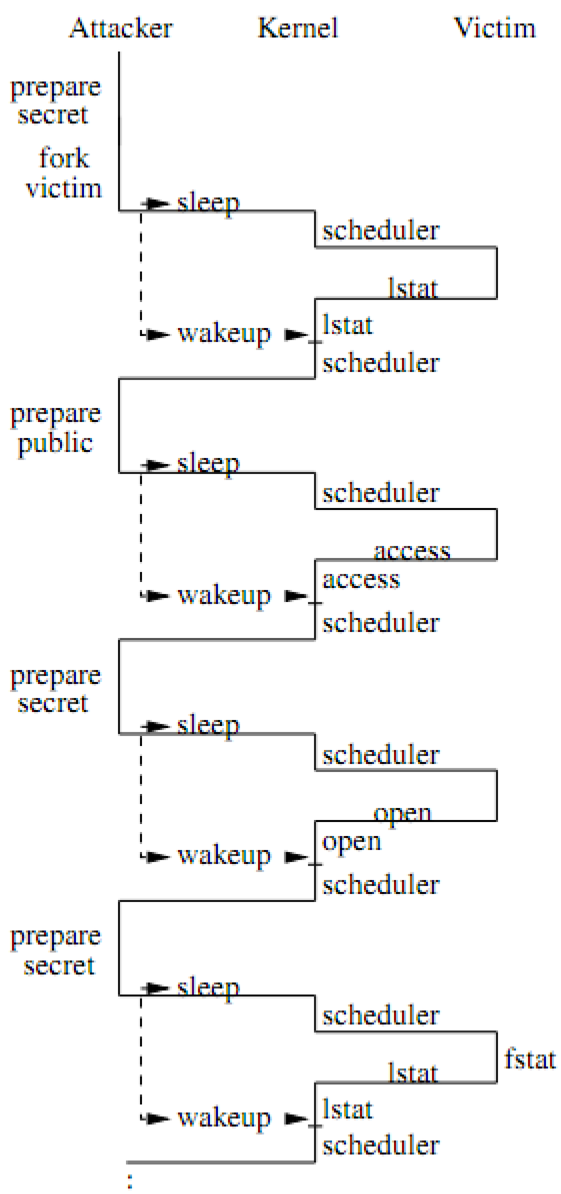
Prepare Secret

Open()

Prepare Secret

Fstat()

Attack Requirements



The attacker's sleep timer must expire in the middle of the victim's system call.

The OS scheduler runs between the victim's system calls.

The scheduler sees the attacker's expired timer and must run the attacker code immediately after.

First Requirement

- The attacker's sleep timer must expire in the middle of the victim's system call.

  - Problem:
    A sleep command has a much higher granularity (measured in milliseconds/seconds) than a system call (measured in microseconds).

  - Solution:
    Slow down system calls until their granularity surpasses that of sleep().

First Requirement

- How do we slow down a system call?

- The kernel keeps an LRU cache of all paths provided to any system call that requires a path using a hash table.

- Since the attacker knows the particular path it wants to access, it also knows the hash digest of the target file.

- The attacker can create a list of thousands of filenames that cause hash collisions with the target file. (This is the “preparation step”)

- This causes the hash table lookups to operate at the worst case of O(n).

First Requirement

- The preparation phase can take up to ten minutes.
  - We can indefinitely suspend the victim (using a POSIX signal) while the attacker prepares, thus moving the victim completely out of the ready-queue.

- We said that the attacker needs to go to sleep in order to be scheduled immediately after a victim system call.
  - How can the sleeping attacker un-suspend the victim?

  - Using a technique they call sleep-walking, the attacker can go to sleep and have a helper process un-suspend the child.

Second Requirement

- The scheduler sees the attacker's expired timer and must run the attacker code immediately after.
  - On every POSIX system they tested, the OS scheduler runs between system calls.

- Problem:
  The attacker must be selected by the OS scheduler between consecutive victim system calls.

- Solution:
  Lower the priority of the victim and increase the priority of the attacker. If both are ready, this guarantees that the attacker will be called before the victim.

Second Requirement

- How do you lower the victim's priority?
  - Easy, start it with a lower nice level.

- How do you increase the attackers priority?
  - ➢ The attacker uses a helper process to:
    1. Do all of its CPU bound work (i.e., the preparation of the hash table).
    2. Sleep-walking

  - ➢ This allows the main attacker process to mostly sleep.

  - ➢ The OS scheduler assumes this is an I/O heavy process and automatically gives it the highest priority.

Evaluation

k	TY-Race	OS	Attacker Wins/Trials	Success Rate
9	N/A	Linux 2.6.24	60/ 70	0.85
9	N/A	FreeBSD 7.0	16/ 20	0.80
9	N/A	OpenSolaris 5.11	20/ 20	1.00
9	No	OpenBSD 3.4 (A)	53/100	0.53
9	No	OpenBSD 3.4 (B)	45/45	1.00
20	N/A	Linux 2.6.24	22/ 30	0.73
20	N/A	FreeBSD 7.0	22/ 40	0.55
20	N/A	OpenSolaris 5.11	20/ 20	1.00
20	No	OpenBSD 3.4 (A)	60/100	0.60
20	Yes	OpenBSD 3.4 (A)	65/100	0.65

Table 3. The success rates of our attacks.