Exploiting Unix File-System Races via Algorithmic Complexity Attacks

Xiang Cai
xcai@cs.sunysb.edu

Yuwei Gui
ygui@ic.sunysb.edu

Rob Johnson
rob@cs.sunysb.edu

Stony Brook University

Abstract

We defeat two proposed Unix file-system race condition defense mechanisms. First, we attack the probabilistic defense mechanism of Tsafrir, et al., published at USENIX FAST 2008[26]. We then show that the same attack breaks the kernel-based dynamic race detector of Tsyrklevich and Yee, published at USENIX Security 2003[28]. We then argue that all kernel-based dynamic race detectors must have a model of the programs they protect or provide imperfect protection. The techniques we develop for performing these attacks work on multiple Unix operating systems, on uni- and multi-processors, and are useful for exploiting most Unix file-system races. We conclude that programmers should use provably-secure methods for avoiding race conditions when accessing the file-system.

1. Introduction

Time-Of-Check-To-Time-Of-Use (TOCTTOU) file-system race-conditions have been a recurring problem in Unix operating systems since the 80s, and researchers have proposed numerous solutions, including static detectors[3], [30], [5], [7], dynamic detectors[28], [17], [8], [29], [15], [20], extensions to the Unix file-system interface[10], [33], and probabilistic user-space defenses[10], [26].

This paper presents attacks on two proposed file-system race-condition defense mechanisms. First, we attack the probabilistic user-space defense mechanism of Tsafrir, et al.[26]. This defense, which we call atomic k-race, was awarded "Best Paper" at USENIX FAST 2008 and builds on Dean and Hu's k-race algorithm[10]. Borisov, et al's[4] maze attack defeated k-race by forcing the victim to sleep on I/O. Atomic k-race avoids sleeping on I/O with very high probability, so a maze attack is not feasible. Instead, our attack uses an algorithmic complexity attack on the kernel's filename resolution algorithm. This slows down the victim's file-system operations, enabling the attacker to win races with high probability. We also develop two new tools for manipulating the OS scheduler: clock-syncing, which enables the attacker to reliably sleep for a specific amount of time, and sleep-walking, which enables the attacker to do work while sleeping. We use these basic building blocks to defeat the atomic k-race defense mechanism with high probability on multiple operating systems, as shown in Table 1.

k	TY-Race	OS	Attacker Wins/Trials	Success Rate
9	No	Linux 2.6.24	60/ 70	0.85
9	No	FreeBSD 7.0	16/ 20	0.80
9	No	OpenSolaris 5.11	20/ 20	1.00
9	No	OpenBSD 3.4	53/100	0.53
20	Yes	OpenBSD 3.4	65/100	0.65

Table 1. The success rates of our attacks against atomic k-race and TY-Race.

We then use the same tools to attack the dynamic race-condition detector of Tsyrklevich and Yee[28], which we call TY-Race. This detector modifies the OpenBSD kernel to maintain a table of the results of recent file access operations and verify that those results are consistent. However, the attacker can indirectly manipulate the entries in the table to circumvent the defense mechanism. We further argue that any kernel-based dynamic race detector must have at least one of the following limitations: it must have side information about the programs it protects, it must protect only a subset of all programs, it must be vulnerable to DoS attacks, it must have false-positives, or it must fail to prevent some race condition exploits. These limitations stem from a fundamental state management problem: without information about the programs it is protecting, the kernel does not know when it can safely remove an entry from the table. Every dynamic defense system we analyzed used some less-than-perfect strategy to solve this state management problem.

This paper focuses on the `access(2)`/`open(2)` race, but the algorithmic complexity attacks we use work on any name-based Unix system call, e.g. `stat(2)`, `chdir(2)`, `chmod(2)`, etc., so our attack is applicable to other Unix file-system races. Algorithmic complexity attacks can slow down lookups on non-existent filenames, so our attack should also be applicable to insecure temp-file creation races.

Our attacks could be thwarted by fixing the algorithmic complexity vulnerabilities within the OS kernels, but there are several problems with this solution. Every OS we tested is vulnerable to these attacks, so fixing them would require a huge, coordinated effort among OS vendors. Finding all the algorithmic complexity bugs is not trivial (e.g. we found two different bugs in OpenBSD name resolution), so it would be difficult to have confidence in the results. Even if all the algorithmic complexity attacks are fixed, other OS features may still make our attack feasible (see, for example, the discussion of Linux's inotify mechanism in Section 7). It would be more reliable to fix the Unix system call interface to enable race-free file-system access. Until that happens, programmers should only use race-free file-access methods, at a potential cost in code portability.

In summary, we develop new tools for exploiting Unix file-system races, show that atomic k-race is insecure, show that TY-Race is insecure, argue that kernel-based dynamic race detectors must have a model of the programs they protect or provide imperfect protection, and argue that probabilistic race defense mechanisms are likely to remain insecure for the foreseeable future.

2. The `access(2)`/`open(2)` Race

The `access(2)` system call was added to Unix to enable setuid-root programs to safely access files owned by the process' invoker. When a regular user invokes a setuid-root program, the program runs with root privileges and hence can access any file on the system. In certain applications, the setuid-root program wishes to access a file only if its invoker can access that file. The `access(2)` system call enables a setuid-root program to test this condition. Figure 1 shows the typical `access(2)`/`open(2)` usage pattern.

The attacker's goal is to trick the victim program into opening a file that the attacker cannot read. The victim program in Figure 1 is not secure because the state of the file-system may change between its `access(2)` and `open(2)` calls. If the OS scheduler switches to an attacker process after the victim has performed `access(2)` but before it performs `open(2)`, then the attacker process can modify the file-system so

```
void 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 that uses the insecure `access(2)`/`open(2)` design pattern. If an attacker changes the file referenced by the given filename between the `access(2)` and `open(2)` calls, then this program may open a secret file.

```
void 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. An attacker that exploits the vulnerable program in Figure 1.

that the setuid-program opens a secret file. The attacker code in Figure 2 attempts to trick the victim program into reading the system's secret password file. This attack will only succeed if the operating system decides to perform a context switch after the `access(2)` call but before the `open(2)` call, as shown in Figure 3.

3. Proposed Run-time Defense Mechanisms

Researchers have proposed numerous solutions to the general TOCTTOU race problem, but we will only discuss the proposals attacked in this paper.

3.1. Atomic k-Race

Figures 4 and 5 show the main routines of the atomic k-race defense mechanism. For brevity, Figure 4 omits

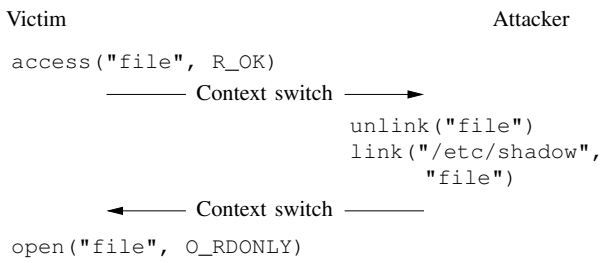

Figure 3. Sequence of events in a successful `access(2)`/`open(2)` attack.

code for handling absolute paths, but our attack does not use them anyway. The `atomic_krace` routine performs path traversal, including resolving symbolic links, in user-space. The `is_symlink` routine simply calls `lstat(2)` to determine whether a path component is a symbolic link. For each non-symbolic-link path component, `atomic_krace` uses `atom_race` to obtain a file-descriptor for that component. The `atom_race` routine uses a modified version of the original k-race algorithm to open its `atom` argument. This function is the only part of atomic k-race that is vulnerable to possible race conditions. To defend against attempts to exploit this vulnerability, the `atom_race` algorithm repeats the `lstat(2)`, `access(2)`, `open(2)`, `fstat(2)` sequence of system calls `krounds` times, verifying that `lstat(2)` and `open(2)` always see the original file.

Altogether, `atomic_krace`, `is_symlink`, and `atom_race` perform the sequence of system calls $LAOF(LAOFC)^k$ on each non-symlink atom, where L stands for `lstat(2)`, A stands for `access(2)`, etc. An attacker that wishes to trick this algorithm into opening a file that he cannot access must trick `atom_race` on some atom. Since atomic k-race checks that all its `lstat(2)` and `open(2)` calls see the same file, the attacker must ensure that the atom refers to his desired file when each of these system calls execute. On the other hand, the attacker must ensure that the atom refers to a file he can access whenever the victim calls `access(2)`. If a represents the attacker's action of switching the atom to point to an accessible file, and s represents the act of switching atom to point to the secret file, then the attacker must cause the kernel to see the sequence of interleaved actions

$$sLaAsOF(LaAsOFC)^k$$

and hence the attacker must win $2k + 2$ races to fool the victim. If the attacker only has a probability p of winning each race and each race is independent, then the attacker's total success probability is p^{2k+2}.

```
int atomic_krace(char *path)
{
  int fd;
  char *suffix, target[MAXPATHLEN];
  struct stat s;
  bool is_sym;

  /* Handle absolute paths */
  /* Code omitted ... */

  while(true) {
    suffix = chop_1st(path);
    DO_SYS(is_symlink(path, target,
                      &s, &is_sym));
    DO_SYS(fd =
           (is_sym ?
              atomic_krace(target) :
              atom_race(path, &s)));
    if (suffix) {
      DO_SYS(fchdir(fd));
      DO_SYS(close(fd));
      path = suffix;
    } else
      break;
  }
  return fd;
}
```

Figure 4. The main atomic k-race routine. This routine iterates over each atom in `path` and performs a k-race on that atom. It also handles symbolic links by calling itself recursively. The `is_symlink` routine calls `lstat(2)` on the given atom to fill in the `stat` structure `s` and returns whether the file is a symlink in `is_sym`.

The `lstat(2)` calls in atomic k-race thwart the maze attack of Borisov, et al.[4]. The maze attack succeeded against the original k-race by forcing the kernel to follow a long sequence of symbolic links, and consequently to perform disk I/O, on each of the victim's system calls. The `lstat(2)` calls in atomic k-race will not follow symbolic links, so it is more difficult for an attacker to reliably force the kernel to perform I/O. Instead of using I/O, our attack slows down the victim's system calls by performing an algorithmic complexity attack on the kernel's name cache. See Section 5.1.

Even without the new techniques developed in this paper, atomic k-race is not as strong as it appears. The calls to `open(2)`, `fstat(2)`, `close(2)`, and `DO_CHK(DO_CMP(s0,&s2))` inside the for-

```
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.

loop are unnecessary – the attacker would have to win just as many races if they were removed. The `access(2)` and `open(2)` calls are still vulnerable to maze attacks – if the attacker can get scheduled after the victim's `lstat(2)` calls, then he can use mazes to get scheduled after the victim's `access(2)` and `open(2)` calls. Thus $k+1$ of the races are already known to be easy, so the security of atomic k-race is really p^{k+1}, not p^{2k+2}. In this paper, we attack the "official" atomic k-race scheme with all system calls and consistency checks in place, although our attack should work just as well with the redundant operations removed. Our attack does not use mazes, although an attacker could use them to increase his success rate if needed.

3.2. Kernel-Based Dynamic Race Detectors

Researchers have proposed several kernel-base dynamic race detection schemes[28], [17], [8], [29], [15], [20], but all the proposals are based on a similar idea. Each proposal modifies the kernel to maintain a table, T, of the form $(pid, dirID, fname, status)$, which indicates that the last system call made by process pid that referenced entry $fname$ inside the directory identified by $dirID$ yielded a file with the given $status$. The $dirID \in Inodes$ is a unique identifier for the referenced directory, e.g. the $dirID$ could be the device and inode numbers of the directory. The $status \in Inodes \cup \perp$ should uniquely identify the file or indicate that the file did not exist.

Whenever process pid performs a path-based system call, the kernel traverses the path, looking up the inode for each element, starting with the process' current working directory. After looking up the $status$ of element $fname$ inside directory $dirID$, the kernel looks up $(pid, dirID, fname)$ in T. If there is no corresponding entry in T, then the kernel adds an entry $(pid, dirID, fname, status)$ to T. If it finds an entry $(pid, dirID, fname, status') \in T$, then the kernel compares $status$ and $status'$. If they are equal, then the kernel proceeds to the next piece of the path. Otherwise, the kernel may abort the calling process, return an error to the caller, generate a log message, or take some other action. Whenever a process updates the status of some $fname$ within $dirID$, the kernel performs the above check *before* making the update, then makes the update to the file-system, and finally updates T to match.

All kernel-based dynamic race detectors must remove entries from T or it will eventually fill all available memory. The kernel may delete a process' entries when it exits, but a long-running process, such as a server, will continue to accrue table entries. If the kernel has side-information about the process' actions, e.g. if the process informs the kernel when it begins and ends file-system transactions, then the kernel can remove all the process' entries as it completes each transaction. Without this information, though, the kernel cannot predict whether a process will reference a given filename in T at some point in the future.

Such a defense mechanism can still offer protection to "well-behaved" applications, but it cannot protect arbitrary programs. For example, the kernel could give each process n dedicated entries in T. As long as a process never references more than n files within one transaction, it will be protected. However, there exists a program that cannot be protected by any kernel-based dynamic race detection scheme. Imagine a setuid-root

program that calls `access(2)` on every file in a user's home directory, then rescans the directory, calling `open(2)` on each file. On the first pass through the user's directory, the kernel will create entries in T for each file. On the second pass, the `open(2)` calls will be checked against these entries. An attacker can circumvent this defense by creating so many files in his home directory that T becomes filled before the setuid-program finishes its first pass. The kernel will necessarily discard some entries for the process. The attacker can change one of the files in his home directory to point to a file which he cannot access. The setuid-program will then open that file on its second pass, violating its security goals. The kernel could prevent this attack by aborting the setuid program or causing all its future system calls to fail, but this will lead to false positives.

Some dynamic detectors, including TY-Race, assume that all file races involve a single "check" system call followed by a single "use" system call. With this assumption, the kernel can remove an entry as soon as its corresponding process makes a second reference to the same filename. This assumption is invalid for two reasons. First, some programs perform several operations predicated on a single check. If the kernel deletes the result of the check operation after the first use operation, then the second use operation will not be guaranteed to be consistent with the check. More seriously, though, if a program performs two different check/use pairs on distinct paths with a common prefix, then the kernel will not ensure that the prefixes are resolved consistently. This occurs because the first check will bring entries for all the prefix elements into T and the first use operation will cause the kernel to remove those entries from T. The next check operation will load new entries in the table, and hence may not be consistent with the first check/use pair.

The above flaw prevents TY-Race from protecting *some* programs, but it contains a second flaw that enables us to circumvent its protection entirely. TY-Race flushes entries in its table after about 2-3 seconds. If an attacker can cause more than 2-3 seconds to pass between the victim's system calls, then TY-Race will provide no protection. Our techniques enable the attacker to get scheduled, and perform an arbitrary amount of work, between each of the victim's system calls, so it should also be possible to defeat dynamic race detectors that maintain a system-wide table and flush entries using LRU. However, other table-flushing policies, including maintaining small per-process tables, as done in RaceGuard[8], are not generally vulnerable to our attack.

```
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
```

Figure 6. High-level pseudo-code for our attack on atomic k-race. The value syscall-duration is the time for the victim to perform one system call and can be measured before beginning the attack. The prepare command updates the file-system so that fname points to the specified file, and performs any other actions necessary to prepare for the next race. If the attacker does not know k in advance, the algorithm can loop until the victim exits.

4. Atomic k-Race Attack Overview

Figure 6 shows the overall structure of our attack on atomic k-race. Before executing the attack, the attacker must measure the time required to perform the `lstat(2)`, `access(2)`, and `open(2)` system calls. In our full attack, all these system calls will take approximately the same amount of time, so the pseudo-code just has one value, syscall-duration, to cover them all. We also assume that the `fstat(2)` and `close(2)` system calls complete very quickly, which is true in practice. The prepare function modifies the file system so that fname is a hard link to the given target, and makes any other preparations to win the next race.

A successful run of this attack will generate the execution trace shown in Figure 7. This attack relies on two features of current Unix implementations. First, the OS must properly serialize system calls by the victim and attacker. The attacker's sleep timer expires in the middle of the victim's system call and, when the attacker process runs, it modifies the file-system in preparation for the victim's next system call. The attacker's preparations must not affect the results of the victim's current system call.

The attack also assumes that the OS runs the scheduler at least once between each of the victim's system calls. This gives the attacker the opportunity to get scheduled between each of the victim's system calls.

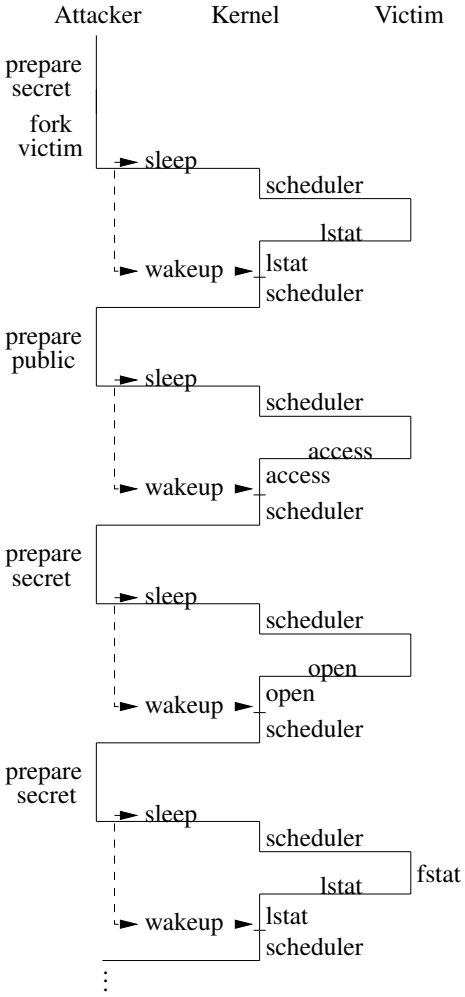

Figure 7. The sequence of events during the execution of the basic attack on atomic k-race.

The four versions of Unix that we tested all exhibited the same behavior: the scheduler runs at the end of each system call, as shown in Figure 7. We believe the attack could easily be adjusted to systems that run the scheduler at the beginning of each system call, or at both the beginning and end, although we could not verify this belief since none of the tested systems do so. The attack may even be feasible on operating systems that only run the scheduler when a process blocks on I/O or reaches the end of its time quantum. In this case, the attacker would have to force the victim's system calls to take longer than one time quantum so that the victim can only make one call each time it is scheduled. An adversary may be able to accomplish this attack by scaling up the algorithmic complexity attack described in the next section.

This attack will only succeed if each of the attacker's sleep timers expires during the execution of the victim's subsequent system call. If the timer expires too late, then the attacker process will still be asleep when the scheduler runs and the kernel will return control to the victim. Since the atomic k-race algorithm only calls `lstat(2)`, `access(2)`, and `open(2)` on atoms, these system calls only take a few microseconds to execute, making the attacker's wakeup window quite small. Making matters worse, the sleep timers on FreeBSD, OpenBSD, and OpenSolaris have much coarser resolution – over 1 millisecond. See Table 2 for system-call times and sleep resolutions of the kernels tested with our attack. Thus this attack is infeasible unless the attacker can slow down the victim's system calls by several milliseconds. The attacker must also ensure that the scheduler chooses to run his process whenever it is runnable. We solve these problems in the next section.

5. Preparing for the Race

To win multiple races in succession, an attacker must slow down the victim's system calls and he must arrange to get scheduled after each of the victim's system calls. For the first task, we use an algorithmic complexity attack against the kernel name lookup algorithm. For the second task, we use `nanosleep(2)` and multiple threads to effectively control the scheduler.

5.1. Algorithmic Complexity Attacks on Name Resolution

Unix kernels maintain a cache of recently referenced file and directory names in order to speed up future name resolution operations. Whenever the kernel needs to look up the inode for a given name, it first checks the name cache. It only looks in its buffer cache or performs I/O when the name cache lookup fails. Every call to `lstat(2)`, `open(2)`, `access(2)`, or any other system call that takes a filename argument performs lookups in the name cache. If we can make name cache lookups slow, then we can make all these system calls slow, too.

Every operating system we examined – Linux, FreeBSD, OpenBSD[1], and OpenSolaris – uses the same basic data structure for its name cache: a hash table with a hard-coded hash function. This data structure is known to be vulnerable to algorithmic complexity attacks[9]. To see why, consider the pseudo-code for

1. OpenBSD has two algorithmic complexity vulnerabilities in its name lookup algorithm. The second vulnerability, which is described later, is easier to exploit, so we used it in our implementation.

OS	File-system	CPU	Timer Res. (μs)	No. of Files	Prepare Time(s)	`lstat(2)` Time (μs) Normal	Attack
Linux 2.6.24	ext3	2.8GHz Pentium D	1	6000	46	6	3120
FreeBSD 7.0	ufs	2.0GHz AMD Turion64	1000	8000	90	20	2500
OpenSolaris 5.11	zfs	2.0GHz AMD Turion64	10000	8000	190	30	3600
OpenBSD 3.4 (A)	ffs	1.0GHz Pentium III	10000	4000	50	70	7000
OpenBSD 3.4 (B)	ffs	2.4GHz Pentium IV	10000	18000	124	8	11100

Table 2. Algorithmic complexity attacks against name resolution in multiple operating systems.

```
procedure name-lookup(name)
  h := hash(name)
  listnode := C[h]
  while listnode ≠ nil
    if listnode.name = name
      return listnode.inode
    listnode := listnode.next
  return nil
```

Figure 8. Pseudo-code for hash-table lookups.

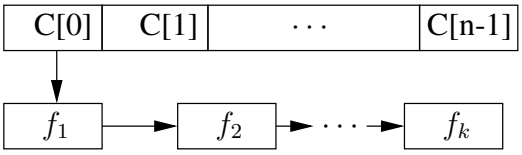

Figure 9. The state of the name-cache hash table during an algorithmic complexity attack.

a hash-table lookup function in Figure 8. If an attacker can discover ℓ distinct filenames, $f_1, \ldots, f_\ell$, that all hash to the same value, say 0, then he can create files with those filenames, causing the kernel to create an entry for each f_i in the name cache. All these entries will go into the linked-list stored in C[0], putting the name cache into the state shown in Figure 9. Future name lookups on f_ℓ will traverse the entire linked list, requiring $O(\ell)$ time. If all the filenames have a long common prefix, then the traversal will be even slower because the comparison "listnode.name = name" will examine the entire prefix for each node in the list.

We can use this algorithmic complexity attack to attack atomic k-race as follows. The adversary creates $f_1, \ldots, f_{\ell-1}$ as described above. He also creates f_ℓ as a hard link to the target file he wants the victim to open. He launches the victim program, passing f_ℓ as the filename argument that it will open with atomic k-race, i.e. f_ℓ is the fname argument in Figure 6. As long as f_ℓ remains at the end of its linked-list in the kernel name cache, the victim's system calls will all be slow, enabling the attacker to successfully execute that attack outlined in Figure 7. Note that in this attack, the filenames $f_1, \ldots, f_\ell$ do not have to collide with the "canonical" name of the attacker's target file. Since the victim performs all its accesses through the hard link f_ℓ, this is the only filename that needs to be slow.

To perform this attack on a given operating system, the adversary must first construct a list of filenames that collide under that operating system's hash function. We could have just used the generator-based filename generation algorithm of Crosby, et al.[9], but we noticed that all the hash functions on the systems we tested are vulnerable to straightforward birthday attacks. By exploiting this weakness, we were able to generate tens-of-thousands of colliding filenames on each OS in only a few seconds (instead of half an hour as with the Crosby attack). Our filenames all had long (> 240 characters) common prefixes, maximizing the effect of the algorithmic complexity attack. Full details on the filename generation are given in Appendix A.

Given the colliding filenames $f_1, \ldots, f_\ell$, Figure 10 shows our algorithm for initializing f_ℓ to point to a given target file and setting up the kernel name cache so that queries for f_ℓ will be slow[2]. We discovered that some operating systems do not add entries to the cache for all operations, e.g. calling `open(2)` with `O_CREAT` did not seem to create a cache entry on FreeBSD. The init-hash-table algorithm in Figure 10 uses both `lstat(2)` and `open(2)`, so it works on Linux, FreeBSD, and OpenSolaris.

The init-hash-table routine runs in $O(\ell^2)$ time because, every time the kernel creates a new entry in its name cache, it first traverses all the entries in the cache to ensure that the name doesn't already exist. As a result, setting up the kernel data structures may take several minutes, as shown in Table 2. Recall that, to win each race, the attacker must make the victim's system calls take longer than the OS timer resolution. Section 5.2 describes techniques for relaxing this requirement, enabling us to use fewer files in the attack.

Table 2 shows the efficacy of this algorithmic complexity attack on Linux, OpenSolaris, and FreeBSD

2. Figures 10–12 present the FreeBSD version of the attack. The other operating systems require slight variations on this attack, which are described in the text.

```
procedure init-hash-table(target)
  unlink(f_ℓ)
  link(target, f_ℓ)
  close(open(f_ℓ, O_RDONLY))
  lstat(f_ℓ)
  for i = 1, ..., ℓ − 1
    unlink(f_i)
    close(open(f_i, O_CREAT))
    lstat(f_i)
```

Figure 10. Algorithm for setting up the FreeBSD kernel name cache so that f_ℓ points to target and ensuring that subsequent accesses to f_ℓ will be slow. This algorithm also works on OpenSolaris.

(the OpenBSD measurements exploit a different algorithmic complexity attack described below). By using only a few thousand filenames, the attacker can make the victim's system calls take several milliseconds. The attacker can then arrange to wake up during the victim's system call, as described in Section 5.2.

Updating f_ℓ. On FreeBSD and OpenSolaris, the init-hash-table routine in Figure 10 can also be used to update f_ℓ to point to a new target. The first 4 lines will update f_ℓ and ensure that it is in the hash table. The for-loop then causes the kernel to delete its cache entries for $f_1, \ldots, f_{\ell-1}$ and recreate them. Afterwards, their cache entries will all be in front of the entry for f_ℓ, so future lookups on f_ℓ will still be slow.

The Linux filename cache is more complicated. It maintains both regular cache entries and "negative" entries, performs some updates in place, and maintains several auxiliary data structures that complicate its behavior. Our attack sidesteps this complexity by using `rename(2)` to move $f_1, \ldots, f_{\ell-1}$ to another filename with a different hash value, and then renames them back to their original name. This causes Linux to move the cache entries for these files to a different bucket. When the attack renames them back to their original names, the kernel moves them to the front of the bucket.

OpenBSD. We use a slightly different algorithmic complexity attack on OpenBSD. The OpenBSD name cache omits names over 31 characters long, so our attack uses 255 character filenames. This causes the kernel to look in its disk-block cache for each name lookup. Directory entries are stored on disk in order of creation and the kernel simply scans through each block until it finds the desired entry. We therefore create directory entries $f_1, \ldots, f_\ell$ in order, ensuring that the kernel will have to scan through all the entries to find f_ℓ. When our attack updates f_ℓ to point to a new target, the kernel performs the changes in place, so future accesses to f_ℓ will still be slow.

OpenBSD's directory traversal algorithm introduces one small wrinkle in the above attack. The kernel begins its traversal where the last name lookup finished. If the victim runs immediately after the attacker updates f_ℓ, the the victim's lookup will start at the block containing f_ℓ and hence will complete very quickly. Our attack solves this problem by performing a lookup on f_1 after updating f_ℓ, moving the search pointer back to the beginning of the directory and causing the victim's subsequent lookup on f_ℓ to be slow.

Buying time. When the attacker gets scheduled after one of the victim's system calls, he only has one time quantum to update f_ℓ, but the algorithm in Figure 10 may take several minutes. Our attack solves this problem by suspending the victim as soon as the attacker gets scheduled. The POSIX.1-2001 standard specifies that an unprivileged process may send a signal to a privileged process whenever the real or effective userid of the sender matches the real or saved userid of the receiver. Thus, the attacker can send SIGSTOP to a setuid-root victim on any POSIX-compliant system. After init-hash-table completes, the attacker can send the SIGCONT signal to the victim, allowing it to continue.

On Linux, FreeBSD, and OpenBSD, the victim is removed from the run-queue as soon as the attacker sends the SIGSTOP signal, so the above strategy works as expected, but on Solaris, the victim is not removed from the run-queue until after its next system call. The attacker therefore has at most one time quantum in which to update f_ℓ, which is not enough time to run the init-hash-table algorithm. This difference may appear to be a problem at first, but actually allows the attacker to win two races for the price of one. Before launching the victim, the attacker sets up the name cache so that the victim's first system call will be slow and the attacker will get scheduled immediately after it completes. When the attacker gets to run, he sends SIGSTOP to the victim and calls `unlink(2)` and `link(2)` to update f_ℓ to point to the new target. These operations are fast and can complete within one time quantum. On Solaris, the victim's subsequent access to f_ℓ will now be very fast, but this no longer matters. After the victim performs its next system call, the kernel will process the pending SIGSTOP and move it off the run queue. The attacker will then get to run and may take as much time as it needs to set up to repeat this process.

Fast file switching. The usual idiom for changing f_ℓ to point to a new file, t, requires two system

calls, `unlink(`f_ℓ`)` followed by `link(`t`, `f_ℓ`)`, but we can do better by using `rename(2)`. For each file, t, to which the attacker might need to link f_ℓ, the attacker creates a directory of files $t_1, \ldots, t_m$, all of which are links to t. The attacker can create these files in advance of the actual attack, so this step is not time-critical. The actual names $t_1, \ldots, t_m$ are not important, and the attacker can create these files anywhere on the same file-system as t. Later, whenever the attacker needs to switch f_ℓ to point to t, he can simply call `rename(`t_i`, `f_ℓ`)`. The `rename(2)` system call replaces f_ℓ if it exists, and will remove t_i, so the attacker can use each t_i at most once.

By using `rename(2)`, the OpenBSD prepare algorithm needs to perform only two system calls: `rename(`t_i`, `f_ℓ`)` and `lstat(`f_1`)`. These two operations are extremely fast, requiring only a few microseconds each, so the attacker does not need to send SIGSTOP (or SIGCONT) to the victim. From our reading of the Linux, FreeBSD, and OpenSolaris source codes, fast file switching provides no benefit on those systems, since the prepare algorithm still needs to perform several thousand other system calls.

The experimental results on Linux, FreeBSD, OpenSolaris, and OpenBSD configuration (A) all use SIGSTOP/SIGCONT and the standard `unlink(2)`/`link(2)` idiom. The experiments on OpenBSD configuration (B) use fast file switching and do not send any signals to the victim.

5.2. Scheduler Management

Once the attacker has arranged to have his sleep timer expire during the victim's next system call, he needs to ensure that the scheduler will choose to run his process. The attacker must also avoid being descheduled after starting the victim but before calling `nanosleep(2)`. The attacker must consistently sleep for one tick of the sleep timer, but if he calls `nanosleep(2)` right before one tick of the timer, he may sleep for two ticks of the timer. Finally, the OS scheduler may behave unpredictably on multi-processor. We now describe solutions to these problems.

Priority laundering. Our attacker launches the victim with the `fork(2)`/`execvp(3)` idiom, so before calling `execvp(3)`, the forked child can use `nice(2)` to decrease its priority. The victim will inherit this reduced priority when the child process execs it. On some operating systems, this is sufficient to ensure the attacker will get scheduled whenever the victim and attacker are both runnable. On some operating systems, such as Linux, the priority of a process only affects the size of its time-slice within one scheduling epoch[18], so adjusting priorities is not enough.

Fortunately, most operating systems include heuristics to schedule I/O-bound processes whenever they are runnable. Tsafrir, et al, exploited this scheduler behavior to monopolize the CPU[24]. Our attack exploits this feature by launching a sub-process to perform all the file-system manipulations needed to prepare for the next race, effectively laundering the main attacker's scheduling priority by dumping all the dirty work on its child. The main attacker thread sleeps while the sub-process is working. The main attacker process spends almost all of its time sleeping – either waiting on its worker sub-process or the victim – and hence looks like an I/O-bound process and gets preferential scheduling. Although the victim also spends most of its time suspended, it apparently does not get the same priority boost. This is probably because it is suspended as a result of a signal instead of I/O or voluntarily sleeping.

Sleep-walking. After the attacker process has finished preparing for the next race, it needs to perform two actions nearly simultaneously: (1) call `nanosleep(2)` and (2) awaken the victim by calling `kill(2)`. The attacker cannot call `nanosleep(2)` and then awaken the victim, since it would not be able to awaken the victim until the end of the `nanosleep(2)`. However, if the OS returns control to the attacker process after its `kill(2)` system call, then the attacker can simply call `kill(2)` followed by `nanosleep(2)`. If, on the other hand, the kernel transfers control to the victim at the end of the attacker's `kill(2)` system call, then this method will not work.

Our attack solves this problem by using two processes, as shown in Figure 11. At startup, the attacker creates a segment of memory that will be shared with a helper process. As long as the parent does not get descheduled between writing "1" to the shared segment and calling `nanosleep(2)`, then this technique will ensure that the victim is awakened immediately after the parent calls `nanosleep(2)`. Since the main attacker process writes 0 to the shared segment before forking the child, the shared memory segment should be in memory, so writing the non-zero value should not cause a page fault, which could cause the main attacker process to get descheduled. Since the parent sleeps briefly before writing 1 to the shared segment, it should have a full time quantum to perform the second write and call `nanosleep(2)`, making it extremely likely that these two actions will execute atomically.

In our experiments with Linux on a multi-processor,

```
procedure sleep-walk(duration, func, arg)
  shared-flag = 0
  if fork() == 0
    // Child: wait for parent to sleep
    while shared-flag = 0
      do nothing
    func(arg)
    exit(0)
  sleep()              // sync with clock
  s = now()            // busy-wait to reduce sleep time
  while now() - s + duration < clock-resolution
    do nothing
  shared-flag = 1   // let child run after we sleep
  sleep()              // sleep until clock tick
```

Figure 11. Algorithm for simultaneously executing func(arg) while sleeping for duration. The requested duration must be less than the clock-resolution. This algorithm works on FreeBSD, OpenBSD, and OpenSolaris.

sleepwalking resulted in unreliable scheduling behavior and sleep times, so our Linux attack program calls `kill(2)` followed by `nanosleep(2)`. Our experiments on a multi-processor suggest that Linux schedules the awakened victim on a separate CPU, so the attacker gets to continue executing at the end of its `kill(2)` system call.

Syncing with the clock. The first sleep in the above algorithm serves a second purpose: it synchronizes the attacker with the kernel's clock ticks. Our initial experiments showed that calling `nanosleep(2)` with a short sleep duration would often cause the attacker program to sleep for two clock ticks. We determined that this occurred whenever the attacker called `nanosleep(2)` right before a clock tick. By calling `nanosleep(2)` twice in a row, we ensure that the second `nanosleep(2)` occurs immediately after a clock tick and hence will yield a much more reliable sleep time.

OpenBSD and OpenSolaris have very low-resolution sleep timers, so our attack uses busy-waiting to shrink the effective sleep time. If the clock has resolution $r$ms, then the attacker can reduce his sleep time to r'ms by busy-waiting for $r - r'$ milliseconds before calling `nanosleep(2)`. This step is not strictly necessary for the attack to succeed, it just reduces the effective sleep time, and hence the number of files required in the algorithmic complexity attack, and hence the running time of the attack. Figure 11 includes this busy-waiting step.

Multi-processors. This attack is able to stay synchronized with the victim because, on a uni-processor, the attacker and victim cannot both run simultaneously. On Linux, the attack can be trivially adapted to multi-processor and multi-core systems by using `sched_setaffinity(2)` to bind the attacker process to a specific CPU or core. This binding is inherited by the attacker's child processes, including the victim, so that the attacker and victim effectively run in a single-processor environment. Solaris appears to support similar functionality through its `processor_bind` system call, and FreeBSD has `cpuset_setaffinity`, so this approach is reasonably general.

However, it is not always necessary to bind all the processes to a single CPU. The next section presents experimental results on a dual-core Linux machine, both with and without binding all processes to a single core.

5.3. Summary

Figure 12 shows the revised, complete attack algorithm for FreeBSD. The Linux, OpenSolaris, and OpenBSD versions are similar, except for the caveats described above. After setting up the initial state of the kernel name cache, the attacker uses the sleep-walk method to launch the victim. By using sleep-walk to launch the victim, the attacker will get to sleep before the victim starts, regardless of how the kernel schedules processes after `fork(2)` and `execvp(3)` system calls. After the victim has started, the attacker gets to run once after each of the victim's `lstat(2)`, `access(2)`, and `open(2)` calls. Each time the attack program runs, it calls prepare, which puts the victim to sleep, launches a separate process to update f_ℓ, sleeps while that process does its work, and then uses sleep-walk to simultaneously go to sleep while awakening the victim.

6. Evaluation

Table 3 shows the success of our attack against atomic k-race on various operating systems[3]. Our attack breaks atomic k-race with the recommended security parameter $k = 9$ on all the tested operating systems with probability at least 0.5. Increasing the security parameter is not a viable response to this attack, as Table 3 shows that we can defeat atomic k-race with $k = 20$ with probability at least 0.5, too. Surprisingly,

3. Some systems have fewer trials because the attack takes several hours to execute.

```
procedure prepare(target)
  kill(victim-pid, SIGSTOP)
  if fork() == 0
    init-hash-table(target)
    exit(0)
  else
    wait for child to exit
  sleep-walk(syscall-duration,
             kill, (victim-pid, SIGCONT))

procedure attack(victim, secret-file, public-file)
  init-hash-table(secret-file)
  sleep-walk(syscall-duration, exec, "victim f_ℓ")
  for i = 0, ..., k
    prepare(public-file)
    prepare(secret-file)
    prepare(secret-file)
```

Figure 12. The prepare and attack algorithms for FreeBSD. The prepare routine updates f_ℓ and prepares to win the next race. The attack algorithm defeats k rounds of atomic k-race by repeatedly calling prepare.

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.

CPU Binding	No	No	Yes	Yes
Sleep-walking	No	Yes	No	Yes
No. of Files	4000	4000	6000	4000
`lstat(2)` Time(μs)	2980	2100	3100	2089
Observed Sleep Times(μs)	506	500, 2000, 4000, 6000	3170	2220
Wins/Trials	54/70	1/100	0/100	93/95
Success Rate	0.77	0.01	0.00	0.98

Table 4. The interaction of CPU binding and sleep walking on a dual-core Pentium D Linux 2.6.24 machine. In all cases, the attacker requested to sleep for 500μs, and the victim used $k = 9$.[4]

the success rate on OpenBSD (configuration (A)) with $k = 20$ appears to be higher than with $k = 9$. This is because, in all our OpenBSD experiments, if the attacker won the first race, then he always won all subsequent races. Hence the success rate on OpenBSD is just the probability of winning the first race and is thus largely independent of k. The results on OpenBSD (configuration (B)) demonstrate that by using fast file switching, we can defeat atomic k-race on OpenBSD without using SIGSTOP/SIGCONT. The success rate is much higher than in configuration (A) because we spent more time tuning it to win the first race. The Linux results in this table were generated on a dual-core CPU and did not use `sched_setaffinity` or sleep-walking.

This attack also breaks TY-Race. Recall that TY-Race flushes entries from its table after about 2 seconds. Our attack was able to completely evade detection by pausing for 5 seconds between sending SIGSTOP to the victim and modifying f_ℓ. We tested our attack on TY-Race by attacking the atomic k-race victim with the kernel-based TY-Race activated. We also verified that, if our attack did not pause for 5 seconds, then the TY-Race kernel module detected the race condition and generated a log message. Table 3 shows that TY-Race had no appreciable impact on our success rate. TY-Race never detected our attack, even when it failed.

Table 4 shows how sleep-walking interacts with CPU binding on a dual-core Linux machine. In the first experiment, the attacker's observed sleep time is less than the time required for the victim to complete its system call, proving that, on a multi-processor Linux system, the attacker can wake up in the middle of the victim's system call. The Linux results in Table 3 use the same attack and achieved a similar success rate. The second experiment shows that sleep-walking on a multi-processor produces unreliable scheduling, and hence unreliable sleep times. The success rate is consequently very low. The third and fourth experiments use CPU binding to effectively reduce a multi-processor system to a uni-processor system. In both experiments, the attacker's sleep times show that it never wakes up in the middle of the victim's system call. The third experiment shows that sleep-walking is necessary on uni-processor systems because the Linux scheduler gives control to the victim as soon as the attacker sends it SIGCONT. The fourth experiment shows that sleep-walking is very effective on uni-processor Linux systems.

4. These experiments were run in parallel on two different machines: a 2.4GHz CPU with Ubuntu kernel 2.6.24-19, and a 2.8GHz CPU with Ubuntu kernel 2.6.24-21. We do not believe the differences affected the outcome of the experiments.

7. Discussion

Although we have developed this attack in the context of the `access(2)`/`open(2)` race, the tools we use and develop in this paper – algorithmic complexity attacks, priority laundering, and sleep-walking – are generally useful for exploiting Unix file-system races. For example, the algorithmic complexity attack on the kernel name cache will also slow down name lookups for non-existent names. Thus, an attacker could use this technique to exploit a temporary file creation race in which the victim checks for the existence of a predictable filename before creating this file.

One could attempt to fix atomic k-race by adding randomized busy-waits between each system call. On a uni-processor system, the attacker could respond by scaling up the algorithmic complexity attack until it can reliably wake up during victim system calls. On a multi-processor, the attacker can poll a system-call distinguisher[4] to detect when the victim begins each system call. In this case, the algorithmic complexity attack is used to make the system call take long enough to observe via polling. The attacker can then use the same prepare algorithm to win the race.

On OpenBSD, the attacker does not need to send SIGSTOP and SIGCONT to the victim. Thus, the attack is not just applicable to setuid-root programs, but to any program, including privileged servers. This also implies that the victim cannot defend itself from our attack by registering a handler for SIGCONT and aborting if it ever receives that signal during the atomic k-race algorithm, as suggested by Dan Tsafrir[23].

As with the original k-race algorithm, one could consider a randomized version of atomic k-race that randomly chooses to perform either `lstat(2)` or `access(2)` on each iteration of strengthening. Borisov, et al. defeated randomized k-race by causing the victim to sleep on I/O in the middle of each system call, enabling them to update the file-system so the system call would succeed. This technique cannot be directly applied to convert the present attack on atomic k-race into an attack on randomized atomic k-race because our attack does not pause the victim in the middle of its system calls. Defeating randomized atomic k-race is therefore still an open problem. Despite this, given the repeated failures of previously proposed probabilistic race defense mechanisms, we do not recommend randomized atomic k-race for any application.

Fixing the algorithmic complexity vulnerabilities exploited in this attack would be straightforward – simply replace the hash tables with balanced binary trees. However, this still would not give us great confidence in the security of atomic k-race, since the OS may contain some other algorithmic complexity vulnerability that an attacker could use instead. For example, as mentioned in Section 5.1, OpenBSD contains two algorithmic complexity vulnerabilities on its name-lookup code path.

Probabilistic race defenses are brittle because they make so many unspecified assumptions about the underlying OS. By leaving a non-zero chance that the attacker can win races against the victim, probabilistic schemes pull the performance characteristics of the OS and hardware into the trusted computing base. Probabilistic defense mechanisms are also likely to interact poorly with new OS features, implying that over time, these schemes will become less secure. For example, the relatively new Linux "inotify" feature allows one process to "watch" the opens, closes, reads, and writes of other processes to files in a watched directory. Our experiments with inotify revealed that, when a process opens a watched file, the watching process gets to run as soon as the open completes but before control is returned to the original process. This gives an attacker great control over the scheduling of certain system calls – `open(2)`, `read(2)`, `write(2)`, and `rename(2)`, among others. Future OS enhancements may give attackers even more power.

Until a general solution is found, we recommend that programmers choose security over portability when accessing the file-system. Most operating systems offer some non-portable but secure way to accomplish the equivalent functionality of `access(2)`/`open(2)`. Alternatively, if programmers are confident that their code will be deployed only on systems that use the user/group/other read/write/execute permission bits with no extensions (such as ACLs), then the deterministic `access(2)`/`open(2)` algorithm of Tsafrir et al. looks correct and secure on those systems[25].

8. Related Work

Static detectors. Static bug finders detect security flaws in source code before it is compiled, and can therefore offer very strong levels of protection[3], [5], [22], [7], [30], [11], [12], [13]. Some static analysis tools strive for soundness, i.e. a guarantee that they will not miss any bugs, but file-system race conditions require both data and control flow analysis, making them difficult to analysis precisely and soundly. Therefore, most practical static checkers are unsound.

Dynamic detectors and preventers. Dynamic race condition detectors come in two basic varieties: preventers[8], [28], [29] and detectors[1], [14], [16],

[17], [32]. As argued in Section 3, run-time prevention of race-condition attacks must always face a state management problem. Some detectors, though, merely log information at run-time and perform the analysis off-line. This mitigates the state management problem, but sacrifices prevention power.

Probabilistic defenses. Both probabilistic defenses for file-system races are broken[10], [26], [4].

Interface changes. Bishop suggested that `access(2)` be replaced with an `faccess` system call that takes a file descriptor as its argument[2]. Unfortunately, the result of an `access(2)` call depends on the entire path to the resulting file, but `faccess` would only be able to check permissions on the last component. Dean and Hu suggested an `O_RUID` flag for open, which would solve the `access(2)`/`open(2)` problem quite cleanly, but is not a general solution[10]. Maziéres and Kaashoek propose capability-based interfaces to solve the race-condition problem[19]. Schmuck and Wylie and Wright, et al. have suggested adding support for transactions to the OS interface[21], [33].

User-space solutions. Tsafrir, et al, have subsequently proposed a deterministic solution to the `access(2)`/`open(2)` problem[25]. In their new proposal, programs perform path-traversal and access-control checks in user-space. Their implementation looks portable and correct on systems that only implement the normal user/group/other read/write/execute permissions, but it may lead to incorrect access decisions on systems that use more complex access-control data structures, such as ACLs. In the specific case of POSIX draft ACLs, this could only result in a setuid-root program erroneously refusing to open a file, but other OS extensions could result in falsely granting access.

A setuid-root program can use the `setuid(2)` family of functions to temporarily drop its privileges before calling `open(2)`. This solution is free of file-system races, but could be vulnerable to tractor-beaming attacks[31], and the Unix user ID management interface is notoriously complex and non-portable[6]. There's no reason this functionality couldn't be encapsulated in a library, hiding the portability issues from developers. Tsafrir, et al. have proposed just such an interface for simplifying user ID management[27]. Their implementation correctly handles user IDs, group IDs, and supplemental group IDs, but does not handle any other OS-dependent access-control data structures, such as capabilities.

9. Conclusion

The atomic k-race algorithm is insecure on four of the most common Unix operating systems in use today – Linux, FreeBSD, Solaris, and OpenBSD – so it should never be used. Even if all of these operating systems fixed the algorithmic complexity attack that makes our attack possible, there will likely be some other operating system that is vulnerable, and legacy systems will remain vulnerable. Even if every operating system in the world corrected this problem and all legacy systems were upgraded, there may be other OS features, such as future inotify implementations, that render atomic k-race insecure.

The attack techniques presented in this paper are not specific to the `access(2)`/`open(2)` race. Most Unix race conditions can be exploited using these tools, so programmers should give race conditions the same level of consideration that they would give buffer-overflows or other software vulnerabilities.

Acknowledgements

We are grateful to Nikita Borisov, Sam King, Naveen Sastry, Dan Tsafrir, and David Wagner for their many helpful comments on this research.

Availability

Our atomic k-race implementation and attack code are available at http://splat.cs.sunysb.edu/.

References

[1] Ashish Aggarwal and Pankaj Jalote. Monitoring the security health of software systems. *Software Reliability Engineering, 2006. ISSRE '06. 17th International Symposium on*, pages 146–158, Nov. 2006.

[2] Matt Bishop. Race conditions, files, and security flaws; or the tortoise and the hare redux. Technical Report CSE-95-8, UC Davis, September 1995.

[3] Matt Bishop and Michael Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, Spring 1996.

[4] Nikita Borisov, Rob Johnson, Naveen Sastry, and David Wagner. Fixing races for fun and profit: how to abuse atime. In *Proceedings of the 14th USENIX Security Symposium*. USENIX Association, 2005.

[5] Hao Chen and David Wagner. MOPS: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, pages 235–244, Washington, DC, November 18–22, 2002.

[6] Hao Chen, David Wagner, and Drew Dean. Setuid demystified. In *Proceedings of the 11th USENIX Security Symposium*, pages 171–190, Berkeley, CA, USA, 2002. USENIX Association.

[7] Brian Chess. Improving computer security using extended static checking. *Proceedings of the 23rd Annual IEEE Symposium on Security and Privacy*, 2002.

[8] Crispin Cowan, Steve Beattie, Chris Wright, and Greg Kroah-hartman. Raceguard: Kernel protection from temporary file race vulnerabilities. In *In Proceedings of the 10th USENIX Security Symposium*. USENIX Association, 2001.

[9] Scott A. Crosby and Dan S. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th USENIX Security Symposium*, pages 29–44, August 2003.

[10] Drew Dean and Alan J. Hu. Fixing races for fun and profit: how to use access(2). In *Proceedings of the 13th USENIX Security Symposium*. USENIX Association, 2004.

[11] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI'00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, pages 1–1, Berkeley, CA, USA, 2000. USENIX Association.

[12] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. *SIGOPS Oper. Syst. Rev.*, 35(5):57–72, 2001.

[13] Rob Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 9–9, Berkeley, CA, USA, 2004. USENIX Association.

[14] Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. *SIGOPS Oper. Syst. Rev.*, 39(5):91–104, 2005.

[15] Calvin Ko, George Fink, and Karl Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *In Proceedings of the 10th Annual Computer Security Applications Conference*, pages 134–144, 1994.

[16] Calvin Ko and Timothy Redmond. Noninterference and intrusion detection. In *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 177, Washington, DC, USA, 2002. IEEE Computer Society.

[17] Kyung-Suk Lhee and Steve J. Chapin. Detection of file-based race conditions. *International Journal of Information Security*, 4(1-2):105–119, 2005.

[18] Robert Love. The linux process scheduler. http://www.informit.com/articles/article.aspx?p=101760, November 2003.

[19] Mazières and M. Kaashoek. Secure applications need flexible operating systems. In *HOTOS '97: Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, page 56, Washington, DC, USA, 1997. IEEE Computer Society.

[20] Calton Pu and JinPeng Wei. A methodical defense against tocttou attacks: The edgi approach. In *Proceedings of the 2006 International Symposium on Secure Software Engineering*, March 2006.

[21] Frank Schmuck and Jim Wylie. Experience with transactions in quicksilver. In *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 239–253, New York, NY, USA, 1991. ACM.

[22] Benjamin Schwarz, Hao Chen, David Wagner, Geoff Morrison, Jacob West, Jeremy Lin, and Wei Tu. Model checking an entire linux distribution for security violations. Technical Report UCB/CSD-05-1384, UC Berkeley, April 2005.

[23] Dan Tsafrir. Personal communication. January 2009.

[24] Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson. Secretly monopolizing the CPU without superuser privileges. In *USENIX Security Symposium*, pages 239–256, Boston, Massachusetts, August 2007.

[25] Dan Tsafrir, Tomer Hertz, David Wagner, and Dilma Da Silva. Portably preventing file race attacks with user-mode path resolution. Technical Report RC24572, IBM T. J. Watson Research Center, Yorktown Heights, NY, June 2008. (submitted for publication).

[26] Dan Tsafrir, Tomer Hertz, David Wagner, and Dilma Da Silva. Portably solving file tocttou races with hardness amplification. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, February 2008.

[27] Dan Tsafrir, Dilma Da Silva, and David Wagner. The murky issue of changing process identity: revising "setuid demystified". *USENIX ;login*, 33(3):55–66, June 2008.

[28] Eugene Tsyrklevich and Bennet Yee. Dynamic detection and prevention of race conditions in file accesses. In *Proceedings of the 12th USENIX Security Symposium*, pages 243–256, August 2003.

[29] Prem Uppuluri, Uday Joshi, and Arnab Ray. Preventing race condition attacks on file-systems. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 346–353, New York, NY, USA, 2005. ACM.

[30] J. Viega, J. T. Bloch, Y. Kohno, and G. Mcgraw. Its4: a static vulnerability scanner for c and c++ code. In *Proceedings of the 16th Annual Conference on Computer Security Applications (ACSAC)*, pages 257–267, 2000.

[31] David Wagner. Design principles for security-conscious systems. http://www.cs.berkeley.edu/~daw/teaching/cs261-f07/slides-aug30.pdf.

[32] Jinpeng Wei and Calton Pu. Tocttou vulnerabilities in unix-style file systems: an anatomical study. In *FAST'05: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, pages 12–12, Berkeley, CA, USA, 2005. USENIX Association.

[33] C. P. Wright, R. Spillane, G. Sivathanu, and E. Zadok. Extending ACID Semantics to the File System. *ACM Transactions on Storage (TOS)*, 3(2):1–42, June 2007.

Appendix

Our filename generation algorithm improves upon the brute-force attack of Crosby, et al.[9] by using a birthday attack.

Note that, because each OS mixes in the directory identifier into the hash of a filename, we are not able to compute which bucket our filenames land in. However, we are certain that they all land in the same bucket, and this is all that is needed to mount the algorithmic complexity attack.

Linux 2.6.28. For any 8-bit character c, let $\hat{c} = (c << 4)+(c >> 4)$. Given a string of 8-bit characters $c_1 \cdots c_n$, define

$$t(c_1 \cdots c_n) = 11^n \hat{c}_1 + 11^{n-1}\hat{c}_2 + \cdots + 11\hat{c}_n \bmod 2^{32}$$

The Linux hash for a filename $c_1 \cdots c_n$ in a directory identified by dir is

$$h(dir, c_1 \cdots c_n) = \text{somefunc}(dir, t(c_1 \cdots c_n))$$

The details of somefunc are irrelevant – two filenames in the same directory collide under h if they collide under t, so we only need to find collisions in t.

Observe that if $1 < m < n$, then

$$t(c_1 \cdots c_n) = 11^{n-m} t(c_1 \cdots c_m) + t(c_{m+1} \cdots c_n)$$

Thus, if s_1 and s_2 are strings such that $t(s_2) = -11^{|s_2|} t(s_1) \bmod 2^{32}$, then $t(s_1||s_2) = 0$, where "||" represents string concatenation. We can find a large number of such strings with the following algorithm:

1) Select L random 6-character strings $s_1, \ldots, s_L$, and create the table of pairs $(s_i, t(s_i))$.
2) Sort the table by its second column.
3) For each entry (s, x) in the table, use binary search to look for another entry of the form $(s', -11^6 x \bmod 2^{32})$. If such an entry exists, output $s||s'$.

For $L < 2^{32}$, this algorithm runs in $L \log L$ time, and the expected number of hits is $L^2/2^{32}$.

The strings output by the above algorithm are only 12 characters long and do not necessarily have any common prefix. However, note that if $t(s_1) = t(s_2)$ and $|s_1| = |s_2|$, then for any other string s, $t(s||s_1) = t(s||s_2)$. Therefore we can take the outputs of the above algorithm and simply prepend a long common prefix of our choice.

FreeBSD 7.0. Given an 8-bit character c, define the function $\hat{c}(x) = (16777619x \bmod 2^{32}) \oplus c$, where $\oplus$ represents bitwise XOR of 32-bit words. For a given string $s = c_1 \cdots c_n$, let $\hat{s}(x) = c_n(c_{n-1}(\cdots(c_1(x))\cdots))$. Define $t(s) = s(33554467)$. The FreeBSD hash for a filename $c_1 \cdots c_n$ in a directory identified by dir is

$$h(dir, s) = \text{somefunc}(dir, t(s))$$

As with Linux, we only need to find collisions in t.

Note that the functions $\hat{c}$ are invertible, and hence so are the functions $\hat{s}$. If two strings s and s' satisfy the equation $s(33554467) = s'^{-1}(0)$, then $t(s||s') = 0$. We can find a large number of such strings with a long common prefix as follows.

1) Pick a prefix string, pre, and let $SEED = \widehat{pre}(33554467)$.
2) Pick L random 6-character strings $s_1 \ldots, s_L$ and generate the table T of pairs $(s_i, \hat{s}_i(SEED))$.
3) Pick L random 6-character strings $s'_1, \ldots, s'_L$ and generate the table T' of pairs $(s'_i, \hat{s'}_i^{-1}(0))$.
4) Sort T' by its second column.
5) For each entry (s, x) in T, binary search for an entry of the form (s', x) in T'. If one exists, output $pre||s||s'$.

OpenSolaris 5.11. Let

$$t(c_1 \cdots c_n) = 17^{n-1}c_1 + 17^{n-2}c_2 + \cdots + c_n \bmod 2^{32}$$

and

$$h(dir, c_1 \cdots c_n) = 17^n dir + t(c_1 \cdots c_n)$$

Two strings of the same length and in the same directory collide under h if they collide under t. We can therefore use the same strategy as with Linux.

OpenBSD 3.4. . Let

$$t(c_1 \cdots c_n) = 33^n 5381 + 33^{n-1}c_1 + \cdots + c_n \bmod 2^{32}$$

and

$$h(dir, s) = \text{somefunc}(dir, t(s))$$

Again, we only need to find collisions in t, and we can use the same basic strategy as with OpenSolaris.