RaceGuard: Kernel Protection From Temporary File Race Vulnerabilities

Crispin Cowan, Steve Beattie, Chris Wright, and Greg Kroah-Hartman

WireX Communications, Inc. http://wirex.com/

Abstract

Temporary file race vulnerabilities occur when privileged programs attempt to create temporary files in an unsafe manner. "Unsafe" means "non-atomic with respect to an attacker's activities." There is no portable standard for safely (atomically) creating temporary files, and many operating systems have no safe temporary file creation at all. As a result, many programs continue to use unsafe means to create temporary files, resulting in widespread vulnerabilities. This paper presents RaceGuard: a kernel enhancement that detects attempts to exploit temporary file race vulnerabilities, and does so with sufficient speed and precision that the attack can be halted before it takes effect. RaceGuard has been implemented, tested, and measured. We show that RaceGuard is effective at stopping temporary file race attacks, preserves compatibility (no legitimate software is broken), and preserves performance (overhead is minimal).

1 Introduction

Attacks exploiting concurrency problems ("race vulnerabilities") are nearly as old as the study of computer system security [1, 5]. These are called TOCTTOU ("Time of Check To Time Of Use") errors [6]. Of particular interest is the temporary file creation vulnerability: programs seeking to create a temporary file first check to see if a candidate file name exists, and then proceed to create that file. The problem occurs if the attacker can *race* in between the file existence check and the file creation, and the attacker creates the file that the victim program expected to create.

In concrete terms, this problem occurs on UNIX systems when programs use `stat()` or `lstat()` to probe for the existence of files, and `open(O_CREAT)` to create the files. An encapsulated means to create temporary names is the `mktemp()` library function.[1] The `mktemp()` library function simply encapsulates the `lstat()` call, and thus `mktemp()` followed by `open(O_CREAT)` is vulnerable to race attacks.

This *race condition* becomes a security vulnerability if the victim program creating the temporary file is privileged (i.e. running as `root` or some other privileged user-ID) and the attacker creates a link pointing to a security sensitive file such as `/etc/passwd` or `/etc/hosts.allow`. When this occurs, the `open(O_CREAT)` will obliterate the data contained in the sensitive file. The `fopen()` library function, being a wrapper around `open(O_CREAT)`, is similarly vulnerable.

There are two commonly accepted mechanisms that exist to prevent this race condition: using `open()` with the `O_CREAT` and `O_EXCL` flags, or using the `mkstemp()` library function (which is a wrapper around `open(O_CREAT|O_EXCL)`). When `open(O_CREAT|O_EXCL)` is called on a file that already exists, it will fail and prevent the race attack. Unfortunately, because these mechanisms are not ubiquitously available and portable, common programs (such as Apache [3, 13]) still continue to use `mktemp()` and friends, despite the fact that the Linux `mktemp` man page says "Never use mktemp()."

This paper presents RaceGuard: a kernel enhancement that detects attempts to exploit temporary file race vulnerabilities, and does so with sufficient speed and precision that the attack can be halted before it takes effect. RaceGuard functions by detecting the change in circumstances between the stat() call and the open() call. If the stat() "fails" (the file does not exist), then RaceGuard caches the file name. If a subsequent open() call provides the same name, and discovers that the file *does* exist, then RaceGuard detects a race attack, and aborts the open() operation.

The rest of this paper is organized as follows.Section 2 elaborates on the temporary file vulnerability issue. Section 3 presents the RaceGuard design and implementation. Section 4 presents our security testing against known race vulnerabilities in actively used software. Section 5 presents our compatibility testing, showing that RaceGuard protection does not interfere with normal system operations. Section 6 presents our performance testing, showing that the performance costs of

1. and related library functions tmpnam() and tempnam().

This work supported in part by DARPA contract N66001-00-C-8032.

RaceGuard protection are minimal. Section 7 describes related work in defending against temporary file race vulnerabilities. Section 8 presents our conclusions.

2 Temporary File Race Vulnerabilities

The basic form of a temporary file race vulnerability is that a privileged program first probes the state of the file system, and then based on the results of that probe, takes some action. The attacker can exploit the vulnerability by "racing" between the probe and the action to change the state of the file system in some critical way, such that the victim program's action will have an unintended effect.

The simple form of this attack is temporary file creation. The victim program seeks to create a temporary file, probes for the existence of the file, and if the nominated file name is not found, proceeds to create the file. The attacker exploits this by creating either a symbolic link that matches the name of the file about to be created, and points to a security sensitive file. The result is that the victim program will unwittingly over-write the security sensitive file with unintended content.

A variation on this scheme is the "dangling symlink". The victim program performs the same sequence as above. The attacking program races in and creates a symlink or hard link from the matching name to a *non-existent* file whose existence has security implications, such as `/etc/hosts.allow` or `/etc/nologin`.

Another variation is the "file swap." Here the victim program is a SUID root program that can be asked to write to a specific file [6]. The victim program defensively checks to see if the requesting user has access to the file, and then only does the write if the user has permission. The attacker provides a file that they have access, to, and between the access check and the write operation, the attacker swaps the file for a symlink pointing to a security sensitive file.

3 RaceGuard: Dynamic Protection from Race Attacks

RaceGuard detects attempts to exploit race vulnerabilities at run time by detecting a change in the environment between the time the program probes for the existence of a file, and the time it tries to create it: if the file named "foo" does *not* exist at the time of the stat, but *does* exist at the time of the open, then someone tried to race us, so abort the operation. RaceGuard achieves this by caching the file names that are probed, and when creation attempts occur that hit existing files, the names are compared to the cache. Section 3.1 describes the RaceGuard algorithm. Section 3.2 describes the RaceGuard implementation and the cache management policy.

3.1 RaceGuard Design

RaceGuard seeks to detect pertinent changes in the file system between the time an application probes for a nominated temporary file name, and the time the file is actually created. "Pertinent" means changes with respect to the nominated name. The RaceGuard algorithm to achieve this is as follows:

- Each process keeps a cache of potential temporary file races. This cache is a list of file names, associated with each process control block within the kernel.
- If file probe result is "non-existent file," then cache the file name in the process's RaceGuard cache.
- If file creation hits a file that already exists, *and* the name matches a name in the RaceGuard cache, then this is a race attack: abort the open attempt.
- If file creation succeeds without conflicts, and matches a name in the RaceGuard cache, then clear that entry from the cache. This prevents "false positive" RaceGuard events when a program uses the same name for a file more than once.

This caching mechanism serves to detect and differentiate between the sequence "probe; create", and "probe; attacker meddling; create". To defend against the "dangling symlink" variant attack described in Section 2, RaceGuard does *two* resolves on the name provided to open that are in the RaceGuard cache: the first follows symlinks, while the second does not. If the two resolve differently, and the argument name matches an entry in the RaceGuard cache, then this is treated as a race attack.

RaceGuard does not defend against the "file swap" attack. Because the attack concerns an already existent file, this is not really a temporary file race attack. In practice, such vulnerabilities appear to be relatively rare: searching Securityfocus.com's vulnerability database [18] for "race" produced 75 hits, while searching for "race & !tmp & !temp" produced only 24 hits. Even among the 24, random sampling indicates that many of them are actually temporary file issues, but did not say so in the name of the vulnerability.

3.2 RaceGuard Implementation & Cache Management Policy

The RaceGuard implementation is in the kernel, facilitating both per-process and inter-process RaceGuard cache management. RaceGuard mediates three basic types of system calls:

- those which can inform the program that a file system entry does not exist -- `stat()`, `lstat()`, `access()`, `newstat()`, and `newlstat()`.
- those which enable the program to actually create file system entries -- `open()`, `creat()`, `mkdir()`, `mknod()`, `link()`, `symlink()`, `rename()`, and `bind()`.
- those which create and remove processes -- `fork()` and `exit()`.

These system calls are often called indirectly via library wrappers. For example, an insecure program may use the C library function `mktemp()`, a wrapper for `lstat()`, followed by `fopen()`, a wrapper for `open()`. Placing RaceGuard mediation in the kernel provides protection for such a programs, in an effort to provide mediation of temporary file creation that is as complete as possible [17].

The interesting part of RaceGuard's implementation is the cache management policies: when to place a cache entry, when to clear it, and the cache replacement policy. We take an aggressive position on cache clearing, and a conservative position on cache populating. This results in some potential race vulnerabilities getting past RaceGuard, in exchange for assuring that no legitimate software is disrupted by RaceGuard. We do this because RaceGuard is an intrusion *rejector* in addition to an intrusion detector, making false positives much more critical than false negatives.

The RaceGuard cache is small (7 entries per process) to keep the kernel memory footprint small, as there is one cache per process, one cache entry per file, and each cache entry is large (`MAX_PATH_LEN`). We hypothesize that most race situations occur with little file system activity occurring in the process between the stat() and the open(), thus a small cache will be sufficient.

The assumption that programs will do the probe and creation in close sequence also affects our cache eviction policy. We considered using LRU (Least Recently Used) and FIFO (First In, First Out). LRU is not appropriate because the expected use is one creation and one reference, so a recent reference is not a good basis for retention. We settled on this fast approximation to FIFO:

1. The cache is a circular buffer.
2. Scan the cache for empty slots, and take the first empty slot found. *Note*: empty slots occur naturally for RaceGuard because of the heavy use of cache invalidation upon successful creation of temporary files (see Section 3.1).
3. The above scan is started from the most recently created entry. If no empty slots are found, then eject the entry just before the most recently created entry slot in the circular buffer.

This cache eviction policy is fast, and avoids the pathology of evicting the most recently created entry.

Races sometimes occur between processes, especially for shell scripts. RaceGuard partially deals with this by inheriting the cache from parent to child (which is why `fork()` is mediated). If the parent tested a file's existence with a common shell built-in function such as `[ -f tempfile ]`, this information is shared with its subsequent child processes. Employing our aggressive cache clearing policy, child processes clearing entries from their cache notify their parent to also clear entries. Likewise, children do not try to populate their parents' cache as this would violate our conservative cache population policy and could pollute the parent's cache or cause false positives.

Some system calls which create file system entries are not subject to race conditions because they fail when the entry already exists, i.e. `mkdir()`, `link()`, etc. However, we clear matching cache entries on *any* successful file system entry creation, even those which we do not need to monitor for races. Similarly, many system calls return `ENOENT` informing the user that no file system entry exists. However, we have carefully selected a small subset of these calls to mediate based on real world code. It is common for applications to use `stat()` or `access()` to check for a file's existence, while it is uncommon for applications to use `chmod()` for such a check. This conservative approach to cache population also helps ensure the cache is not polluted.

This approach of cautiously only caching entries from a few file probing system calls is largely effective. However one pathological case exists: when a shell script executes a program, the shell typically `stat`'s for that program file in every directory in the `$PATH`. If the script probes many directories before finding the file it is looking for, it has the effect of flooding the RaceGuard cache with useless entries. Thus, a shell script that probes for a file, executes an external program, and then creates the file, may not be protected by RaceGuard. In our experience, most shell scripts find the executables they are looking for in `/bin` or `/usr/bin` so this problem does not occur in practice.

Note that this problem does *not* occur for native programs and dynamic linking. The GNU/Linux `ld.so` loader uses `open()` rather than `stat()` when searching the `$LD_LIBRARY_PATH` for a `.so` file. The

```
 <<< set up our target to overwrite >>>
[steve@reddwarf .elm]$ echo "please dont hurt me" > ~/dont_hurt_me

        <<< run our vulnerable program >>>
[steve@reddwarf .elm]$ LD_PRELOAD=~/libmktemp.so rcsdiff -u elmrc > /dev/null
===================================================================
RCS file: RCS/elmrc,v
retrieving revision 1.3
unsafe_mktemp[20038]: ImmunixOS unsafe mktemp - about to pass back /tmp/T0LZ388D
        <<< In another shell, do "ln -s ~/dont_hurt_me /tmp/T0LZ388D" >>>
diff -u -r1.3 elmrc

        <<< and what does our target now contain? >>>
[steve@reddwarf .elm]$ cat ~/dont_hurt_me | head -5
#
# .elm/elmrc - options file for the ELM mail system
#
# Saved automatically by ELM 2.5 PL1 for Steve Beattie
#
```

Figure 1 Successful Attack Against RCS Without RaceGuard

`execlp`/`execvp` system calls search the path, but they do not `stat` files; instead, they call `execve` (the system call), and if it fails they move on to the next directory in the path.

4 Security Testing

Rigorous testing of temporary file race vulnerabilities is problematic, because the vulnerability is fundamentally non-deterministic: the outcome of the attack depends on whether the victim program or the attacking program wins the race to the file in question. Therefore, the security testing in this paper will not be as cleanly definitive as we would like. To do deterministic, repeatable testing, we had to create a situation in which the attacker would *reliably* win the race. We did this by creating a doctored version of the `mktemp` library call that does two key things:

Pause the Program: our doctored `mktemp` function pauses the caller for 30 seconds, giving the attacker ample time to deploy the race attack.

Print the Created File Name: The file names produced by `mktemp` are easy enough to guess that a determined attacker can get a hit and violate security, *eventually*. Our doctored `mktemp` function shortens this task by printing the name of the temporary file that it will create to syslog. This allows the attacker to precisely deploy a race attack, rather than repeatedly guessing the file name.

While we recognize the limited value of security testing against such a straw-man, we felt it necessary to get repeatable experiments. We view the above concessions as largely immaterial to the validity of RaceGuard defense, because they only make the programs *more* vulnerable. However, it is interesting to note that while exploits for buffer overflow [9], format bug [7], and CGI [8] vulnerabilities are readily available, exploits for race vulnerabilities are extremely rare. We conjecture that the relative scarcity of race exploits is related to the relative difficulty in successfully deploying race attacks: “script kiddies” aren’t interested in attacks that are hard to do, and so race attacks remain the purview of the relatively serious attacker.

Using this doctored `mktemp` function, we attacked four programs: RCS version 5.7 [15], rdist Version 6.1.5 [14], sdiff - GNU diffutils version 2.7 [2], and shadow-utils-19990827 [12]. In each case, without RaceGuard protection, we succeeded in duping the victim program into over-writing an unintended file. Figure 1 shows such a successful attack against RCS. With RaceGuard protection, the identical attack produces a RaceGuard intrusion alert and aborts the victim program, while the file that would have been over-written is unharmed, as shown in Figure 2.

5 Compatibility Testing

RaceGuard is intended to be a highly transparent security solution, which means that it may not break much (if any) legitimate software that is not being actively

```
 <<< set up our target to overwrite >>>
[steve@kryten .elm]$ echo "please dont hurt me" > ~/dont_hurt_me

        <<< run our vulnerable program >>>
[steve@kryten .elm]$  LD_PRELOAD=~/libmktemp.so rcsdiff -u elmrc > /dev/null
===================================================================
RCS file: RCS/elmrc,v
retrieving revision 1.3
unsafe_mktemp[1456]: ImmunixOS unsafe mktemp - about to pass back /tmp/T0POjIdZ
        <<< In another shell, do "ln -s ~/dont_hurt_me /tmp/T0POjIdZ" >>>
/usr/bin/co: Killed
rcsdiff aborted

        <<< and what does our target now contain? >>>
[steve@kryten .elm]$ cat ~/dont_hurt_me
please dont hurt me

        <<< RaceGuard intrusion alert in syslog >>>
[steve@kryten .elm]$ dmesg | tail -1
Immunix: RaceGuard: rcsdiff (pid 1458) killing before opening /tmp/T0POjIdZ!
```

Figure 2 Failed Attack Against RCS With RaceGuard

subjected to actual race attacks. To test this compatibility requirement, we exercised RaceGuard under a wide variety of software. To that end, RaceGuard has been running on various developers workstations day-to-day since January 1, 2001. This section describes the various compatibility faults induced by the original RaceGuard design, and how we addressed them. The current implementation exhibits no known compatibility faults.

The first problem we encountered was manifested by the Mozilla web/mail client. Mozilla makes heavy use of temporary files for caching web content. Re-use of some of these names induced false positive reports from RaceGuard. This problem is what spurred us to add the cache clearing feature, where RaceGuard cache entries are flushed when the corresponding file creation succeeds.

A related problem was induced by the script Red Hat Linux uses to preserve `/dev/random`'s entropy pool across re-boots. This is a shell script in which the parent process does the probe, and a child process creates the temporary file. Adding the feature where clearing the cache entry from a process also clears the entry from its parent's cache (see Section 3.2) fixed this problem.

The third problem encountered was induced by CVS [4] checkout. Here, CVS frequently probes for the same file name in various directories. The rough sequence of "`probe("foo"); chdir("bar"); creat("foo")`" induced a false positive RaceGuard event for the file "`foo`". Changing RaceGuard cache entries from simply the name presented to each system call to a fully resolved absolute path addressed this problem.

Finally, the VMWare virtual machine emulation system [11] manifested a minor compatibility problem with RaceGuard. Portions of the VMWare system periodically make calls to the `stat()` system call with a *null* argument for the pathname, i.e calling "`stat("")`" which is meaningless. Initially, RaceGuard reported a debugging error when this occurred (thinking that it was some kind of error copying syscall arguments to kernel space. However, once we satisfied ourselves that this behavior is harmless, we disabled that debugging feature.

The RaceGuard kernel has been in use on various developer workstations (now up to half a dozen) for the last six weeks. Workloads include editing files, compiling & testing code, reading e-mail, surfing the web, playing MP3s[1], and compiling large systems such as the kernel itself (see Section 6). The above are the only compatibility issues found to date, and all of them are addressed by the current implementation.

1. Essential for software development :-)

Table 1: RaceGuard Microbenchmark Results

System Call	Without RaceGuard	With RaceGuard	% Overhead
Stat non-existent file	4.3 microseconds	8.8 microseconds	104%
Open non-existent file	1.5 milliseconds	1.44 milliseconds	-4%
Fork	161 microseconds	183 microseconds	13%

6 Performance Testing

Any run-time security defense will impose performance costs, due to additional run-time checks that it is performing. However, a security enhancement must be efficient enough that these overhead costs are minimal with respect to the defense they provide. Ideally, the cost should be below noticability for the intended user base.

RaceGuard achieves this level of performance. Overhead is only imposed on the run-time cost of a handful of mediated system calls. The cost on each system call is cache insertion or lookup to see if the proposed name is in the RaceGuard cache. Section 6.1 presents microbenchmarks that show the precise overhead imposed on these system calls. Section 6.2 shows macrobenchmarks that measure the imposed overhead on programs that make intensive use of many temporary files.

6.1 Microbenchmarks

Here we measure the marginal overhead of RaceGuard protection on each of the mediated system calls. We measure the overhead with programs that call the affected system call 10,000 times in a tight loop, the test is run 10 times, the lowest and highest are thrown away, and the remainder are averaged. We ran these tests with and without RaceGuard protection, and computed the percent overhead. The performance results are shown in Table 1. Some commentary on the results:.

Stat a non-existent file: measure the overhead to create a RaceGuard cache entry. The marginal overhead is substantial, because the work of the non-RaceGuard case is minimal, while the RaceGuard case is doing some work.

Open non-existent file: measure the overhead to find and clear a RaceGuard cache entry. We do not actually believe there is a speedup due to RaceGuard, and regard this as experimental error, as the differences within the tests exceeded the differences between the tests. We believe this is because the cost of creating a non-existent file is dominated by the state of the file system on disk.

Fork: measure the overhead of copying the RaceGuard cache. This test exhibited significant variances, and so we enhanced measurement to run the test 100 times and took the average. The variance was present in both the RaceGuard and non-RaceGuard tests, so it was not induced by RaceGuard.

Thus there is substantial overhead only in stat'ing non-existent files, and that cost is dwarfed by the cost of creating files. This operation does not represent a large amount of time in a real workload, as we show in our macrobenchmarks in Section 6.2.

6.2 Macrobenchmarks

To stress-test RaceGuard at the macro level, we sought an application that incurred a substantial amount of run time, used many temporary files, and did a lot of forking. Our first selected test is what we call the Khernelstone[1]: the time to build the SRPM of the Linux kernel, which builds the kernel from its 1800 C and assorted assembly source files, several times. Thus this test incorporates several thousand forks and temporary files. The test was run on a single-processor 700 MHz Athlon machine with 128 MB of RAM.

We ran this test four times each with and without RaceGuard. The results showed *very* little variation. The averages of the four runs are shown in Table 2. In all cases (real time, user time, and system time) the overhead due to RaceGuard was always below 0.5%.

Our second macrobenchmark was the Apache web server [3] measured by the Webstone benchmark tool [16]. Webstone simulates various web server workloads, varying the number of concurrent clients and the size of the requests. Webstone measures a variety of factors, including connection latency, and server throughput in terms of both connections per second and bytes per second. Webstone only stresses the web server machine, which in this case was a dual processor 700 MHz Pen-

1. After the venerable Dhrystone integer performance benchmark [20], which in turn is a reference to the Whetstone floating point benchmark.

Table 2: Khernelstone Macrobenchmark, in Seconds

	Real Time	User Time	System Time
Without RaceGuard	10,700	8838	901
With RaceGuard	10,742	8858	904
%Overhead	0.4%	0.2%	0.3%

tium III, 256 KB of cache, 256 MB of RAM, and a 100 Mbit network to the client.

We ran the Webstone test through a range from 25 concurrent clients to 200 concurrent clients. While performance varies substantially across this range, the difference between the RaceGuard and non-RaceGuard cases was always in the noise. WebStone runs each test case three times, and when ever one test run showed one of the cases to be ahead, another test case showed the opposite. In almost all cases, the RaceGuard average is within the non-raceguard average +/- the non-raceguard standard deviation. There are only three cases in which the RaceGuard average is outside the deviation and in each of those three instances we're outside because we do slightly better.[1]

We draw two conclusions from these tests. First, they are apparently not particularly effective at highlighting the costs of RaceGuard. Second, and more importantly, it shows that the performance overhead of RaceGuard is imperceptible for common, heavy workloads.

7 Related Work

The study of temporary file race vulnerabilities is old: Abbott et al [1] and then Bisbey & Hollingsworth [5] described them as a subclass of timing or synchronization flaws. Yet despite the depth of past study of this problem, a practical solution is apparently still needed: temporary file race vulnerabilities were found in core Internet infrastructure tools such as Apache in 2001 [13].

Bishop's seminal paper [6] formally defined the notion of a TOCTTOU (Time Of Check To Time Of Use) error as being two sequential events in which the second depends on the first, and that there is a faulty assumption that results from the first operation will persist to the second operation. Bishop presents a partial solution to TOCTTOU vulnerabilities in the form of a program scanning program to detect some potential TOCTTOU vulnerabilities in C code, but also presents theorems showing that detecting statically TOCTTOU flaws is undecidable.

Bishop discusses the possibility of a run-time TOCTTOU detector that modifies system call interfaces to track the arguments to system calls, and the association of file descriptors and names, abstractly similar to RaceGuard. Bishop does not elaborate this proposal due to performance concerns. RaceGuard overcomes these performance difficulties by narrowing the scope and duration of the information to be tracked, showing that *near* precise file system race attacks can be detected at run time with very low performance costs.

"Solar Designer" [10] takes a different approach to combating temporary file race vulnerabilities. Rather than attacking the "race" aspect, this enhancement to the Linux kernel attacks the propensity for privileged programs to create and follow links. Two restrictions are imposed: processes will not follow links in directories with the +t ("sticky") bit set unless the owner is trusted (same UID as the process, or owns the directory), and processes are not allowed to create hard links to files that they do not own.

While effective in many cases, this approach unfortunately gets mixed results. Some programs (wrongly) create temporary files in other file systems, e.g. the current working directory. We have also observed real programs that insist on linking to files that they do not own, e.g. the Courier mail server [19] which uses links to optimize the order of mail delivery, and makes links to files with a different UID but a shared GID.

8 Conclusions

Temporary file race vulnerabilities have been a pervasive security problem for over a decade. There are safe methods to create temporary files, but they are not portable, and thus common programs continue to use vulnerable-but-portable temporary file methods such as `mktemp`. RaceGuard protects vulnerable programs against this problem, even if the program insists on using unsafe means, and regardless of whether the program is using an unsafe library, or "rolled their own"

1. We do not claim that RaceGuard is a performance enhancement, only that performance cost is smaller than experimental error.

unsafe temporary file creation method. We have shown that RaceGuard is effective in stopping attacks, and imposes minimal compatibility and performance overhead. RaceGuard is available as a GPL'd patch to the Linux kernel, and is incorporated into WireX's Immunix server products.

References

[1] R.P. Abbott, J.S. Chin, J.E. Donnelley, W.L. Konigsford, S. Tokubo, and D.A. Webb. Security Analysis and Enhancements of Computer Operating Systems. NSBIR 76-1041, National Bureau of Standards, April 1976.

[2] Assorted. GNU Diffutils. `http://www.gnu.org/software/diffutils/`, June 2000.

[3] Brian Behlendorf, Roy T. Fielding, Rob Hartill, David Robinson, Cliff Skolnick, Randy Terbush, Robert S. Thau, and Andrew Wilson. Apache `HTTP` Server Project. `http://www.apache.org`.

[4] Brian Berliner, david d 'zoo' zuhn, Jeff Polk, and et al. Concurrent Versions System. `http://www.cyclic.com/`, 1999.

[5] R. Bisbey and D. Hollingsworth. Protection Analysis Project Final Report. Technical Report ISI/RR–78-13, USC/Information Sciences Institute, May 1978. DTICAD A 056816.

[6] M. Bishop and M. Digler. Checking for Race Conditions in File Accesses. *Computing Systems*, 9(2):131–152, Spring 1996. Also available at `http://olympus.cs.ucdavis.edu/ bishop/scriv/index.html`.

[7] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. FormatGuard: Automatic Protection From printf Format String Vulnerabilities. In *USENIX Security Symposium*, Washington, DC, August 2001.

[8] Crispin Cowan, Steve Beattie, Calton Pu, Perry Wagle, and Virgil Gligor. SubDomain: Parsimonious Server Security. In *USENIX 14th Systems Administration Conference (LISA)*, New Orleans, LA, December 2000.

[9] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In *DARPA Information Survivability Conference and Expo (DISCEX)*, January 2000. Also presented as an invited talk at SANS 2000, March 23-26, 2000, Orlando, FL, `http://schafercorp-ballston.com/discex`.

[10] "Solar Designer". Root Programs and Links. `http://www.openwall.com/linux/`.

[11] Mendel Rosenblum et al. VMWare. `http://www.vmware.com/`, February 1998.

[12] Tomasz Kloczko. Shadow Utilities. `http://freshmeat.net/projects/shadow/`, October 2000.

[13] Greg Kroah-Hartman. Immunix OS Security update for lots of temp file problems. Bugtraq mailing list, `http://www.securityfocus.com/archive/1/155417`, January 10 2001.

[14] MagniComp. RDist. `http://www.magnicomp.com/rdist/`, July 1999.

[15] RCS Maintainers. RCS: Revision Control System. `http://www.cs.purdue.edu/homes/trinkle/RCS/`, December 2000.

[16] Mindcraft. WebStone Standard Web Server Benchmark. `http://www.mindcraft.com/webstone/`.

[17] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9), November 1975.

[18] Securityfocus.com. Vulnerability Search. `http://search.securityfocus.com/search.html`, 1997-2001.

[19] Sam Varshavchik. Courier Mail Transfer Agent. `http://www.courier-mta.org/`, 1999.

[20] Reinhold P. Weicker. Dhrystone: A Synthetic Systems Programming Benchmark. *Communications of the ACM*, 27(10):1013–1030, October 1984.