Model Checking An Entire Linux Distribution for Security Violations

Benjamin Schwarz   Hao Chen   David Wagner
{bschwarz, hchen, daw}@cs.berkeley.edu
Geoff Morrison   Jacob West
{gmorrison, jwest}@fortifysoftware.com
Jeremy Lin (jjlin@ocf.berkeley.edu)   Wei Tu (tuwei@berkeley.edu)
University of California, Berkeley

Abstract

Software model checking has become a popular tool for verifying programs' behavior. Recent results suggest that it is viable for finding and eradicating security bugs quickly. However, even state-of-the-art model checkers are limited in use when they report an overwhelming number of false positives, or when their lengthy running time dwarfs other software development processes. In this paper we report our experiences with software model checking for security properties on an extremely large scale—an entire Linux distribution consisting of 839 packages and 60 million lines of code. To date, we have discovered 108 exploitable bugs. Our results indicate that model checking can be both a feasible and integral part of the software development process.

1 Introduction

Software bugs are frequent sources of security vulnerabilities. Moreover, they can be incredibly difficult to track down. Automated detection of possible security violations has become a quickly-expanding area, due in part to the advent of model checking tools that can analyze millions of lines of code [6].

In this paper we describe our experience using MOPS, a static analyzer, to verify security properties in an entire Linux distribution. We use the following recipe for finding security bugs: identify an important class of security vulnerabilities, specify a temporal safety property expressing the condition when programs are free of this class of bugs, and use MOPS to decide which programs violate the property. We have developed six security properties—expressed as finite state automata (FSAs)—and refined them to minimize false positives while preserving high effectiveness. These properties aim at finding security bugs that arise from the misuse of system calls, often vulnerable interaction among these calls. For example, time-of-check-to-time-of-use (TOCTTOU) bugs involve a sequence of two or more system calls acting on the same file (see Section 3.1).

Our primary contribution is the scale of our experiment. We ran MOPS on the entire Red Hat Linux 9 distribution, which contains 839 packages totaling 60.0 million lines of code (counting total lines in all `.h`, `.c`, and `.cc` files). MOPS successfully analyzed 87% (732) of these packages; the remaining 107 packages could not be analyzed because MOPS's parser cannot parse some files in these packages. To the best of our knowledge, our experiment is the largest security audit of software using automated tools reported in the literature. Model checking at this scale introduces major challenges in error reporting, build integration, and scalability. Many of these technical challenges have been addressed in our work; we show how to surmount them, and demonstrate that model checking is feasible and effective even for very large software systems.

As part of this experiment, we have worked out how to express several new security properties in a form that can be readily model checked by existing tools. Earlier work developed simple versions of some of these properties [6], but in the process of applying them at scale we discovered that major revisions and refinements were necessary to capture the full breadth of programming idioms seen in the wild. Some of the properties checked in this paper are novel; for instance, we introduce a TOCTTOU property that turned out to be very effective in finding bugs. In our experiments, we focused on finding bugs rather than proving their absence. Verification is difficult, especially since MOPS is not completely sound because it does not yet analyze function pointers and signals. However, we expect that our techniques could point the way to formal verification of the absence of certain classes of bugs, as better model checkers are developed in the future.

The technical contributions of this paper are threefold: 1) We show how to express six important security properties in a form that can be model checked by off-the-shelf tools; 2) We report on practical experience with model checking at a very large scale, and demonstrate for the first time that these approaches are feasible and useful; 3) We measure the effectiveness of MOPS on a very large corpus of code, characterizing the false positive and bug detection rates for different classes of security bugs.

The full version of this paper [1] contains further detail on our experiments, some of which is omitted from the conference version due to space constraints. MOPS is freely available from `mopscode.sourceforge.net`.

2 The MOPS Model Checker

MOPS is a static (compile time) analysis tool that model checks whether programs violate security properties [7]. Given a security property—expressed as a finite-state automaton (FSA) by the user—and the source code for a program, MOPS determines whether any execution path through the program might violate the security property.

In more detail, the MOPS process works as follows. First, the user identifies a set of security-relevant operations (e.g., a set of system calls relevant to the desired property). Then, the user finds all the sequences of these operations that violate the property, and encodes them using an FSA. Meanwhile, any execution of a program defines a *trace*, the sequence of security-relevant operations performed during that execution. MOPS uses the FSA to monitor program execution: as the program executes a security-relevant operation, the FSA transitions to a new state. If the FSA enters an *error state*, the program violates the security property, and this execution defines an *error trace*.

At its core, MOPS determines whether a program contains any feasible traces (according to the program's source code) that violate a security property (according to the FSA). Since this question is generally undecidable, MOPS errs on the conservative side: MOPS will catch all the bugs for this property (in other words, it is sound, subject to certain requirements [7]), but it might also report spurious warnings. This requires the user to determine manually whether each error trace reported by MOPS represents a real security hole.

Specification of Security Properties. MOPS provides a custom property language for specifying security properties. The MOPS user describes each security-relevant operation using a syntactic pattern similar to a program's abstract syntax tree (AST). With wildcards, these patterns can describe fairly general or complex syntactic expressions in the program. The user then labels each FSA transition using a pattern: if the pattern matches an AST in the program during model checking, the FSA takes this transition.

To extend the expressiveness of these patterns, we introduced *pattern variables*, which can describe repeated occurrences of the same syntactic expression. For instance, if X denotes an pattern variable, the pattern $f(X, X)$ matches any call to the function f with two syntactically identical arguments. In any error trace accepted by an FSA, the pattern variable X has a single, consistent instantiation throughout the trace.

Formally, let Σ denote the set of ASTs. We may view a program trace as a string in Σ^*, and a property B as a regular language on Σ^*. Pattern variables provide existential quantification over the set of ASTs. For instance, the pattern $\exists X.f(X, X)$ matches any call to f whose two arguments, once parsed, yield the same syntax subtree. If $B(X)$ is a language with an unbound pattern variable X, the language $\exists X.B(X)$ accepts any trace $t \in \Sigma^*$ where there exists an AST A' so that $B(A')$ accepts t. In other words, if $L(B)$ denotes the set of error traces accepted by the language B, we define $L(\exists X.B(X)) = \cup_{A'} L(B(A'))$. We use the convention that unbound pattern variables are implicitly existentially quantified at the outermost scope.

Scalability. Since we aim at analyzing hundreds of large, real application, MOPS must be scalable in several senses. First, MOPS must run quickly on large programs. Second, MOPS must run on different application packages without requiring the user to tweak each package individually.

We have put much effort into integrating MOPS with existing build processes, including `make`, `rpmbuild`, and others. By interposing on `gcc`, the model checker sees the same code that the compiler sees. As a result, running MOPS on numerous software packages is as easy as invoking a MOPS script with the names of these packages. This ease of use has been critical to our success in checking such a large number of packages.

Error Reporting. MOPS reports potential errors in a program using error traces. A typical problem with reporting error traces is that a single bug can cause many (sometimes infinitely many) error traces. To avoid overloading the user, MOPS divides error traces into groups such that each group contains all the traces caused by the same bug. More precisely, two traces belong to the same group if the same line of code in both traces causes the FSA to enter an error state for the first time via the same transition[1]. The user can then examine a representative trace from each group to determine whether this is a bug and, if so, to identify the cause of the bug.

Not all error traces identify real bugs: imprecision in the analysis causes spurious traces. MOPS provides an HTML-based user interface where the user can examine traces very rapidly. The user, however, does spend time identifying false positives, so the cost of using MOPS correlates roughly to the number of trace groups, each of which the user has to examine. In our experiments, we quantify the

[1]This implies that both traces enter the same error state. An FSA may contain multiple error states, corresponding to different kinds of bugs.

Property	*Reported Warnings*	*Real Bugs*	*Section*
TOCTTOU	790	41	3.1
Standard File Descriptors	56	22	3.2
Temporary Files	108	34	3.3
`strncpy`	1378	11*	3.4
Chroot Jails	1	0	(full version)
Format String	(too many)	(unknown)	(full version)
Total	2333	108	

Table 1. Overview of Results.

cost of using MOPS by measuring the number of false positives, counting only one per trace group.

Resource Usage. The running time of the model checker is usually dwarfed by the time a human spends perusing error traces. Still, since our goal is to audit entire distributions, we have aimed to make computation time small. We timed the process of model checking several of our properties against all of Red Hat 9. Using MOPS to look for TOCTTOU vulnerabilities (filesystem races) among all Red Hat 9 packages requires about 1 GB of memory and takes a total of 465 minutes—a little less than 8 hours—on a 1.5 GHz Intel Pentium 4 machine. Detecting temporary file bugs takes 340 minutes of CPU time and about the same memory footprint. The observed wall-clock time was between 20% and 40% more than the CPU time. MOPS produces an extraordinary amount of output, and is required to read in extremely large control flow graphs; I/O thus constitutes a significant portion of this running time, although it is dominated by the time needed for model checking itself.

Also of chief concern to us was being able to audit manually all error traces produced by MOPS. Error trace grouping was a huge time saver: a typical group has more than 4 traces, but some groups contain more than 100 traces. The amount of human effort that was spent auditing the error groups is roughly proportional to the total number of groups. We spent about 100 person-hours auditing error reports from the TOCTTOU property, 50 person-hours for the temporary file property, and less for the other properties. Several of us were undergraduate students who had no prior experience with MOPS prior to joining this project.

3 Checking Security Properties

We developed six security properties. Table 1 shows a summary of the bugs discovered. For each property, the table shows the number of warnings reported by MOPS, the number of real bugs, and the section that describes detailed findings on this property. We will describe four properties in detail, explain the model checking results, and show representative bugs and vulnerabilities that we discovered in this section. The other two are discussed in the full version.

3.1 TOCTTOU

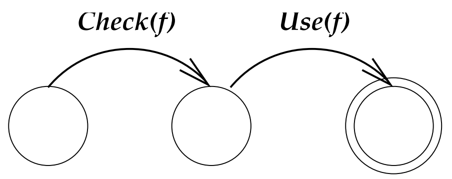
Race condition attacks have perennially plagued secure systems. One common mistake is that a program checks the access permission of an object and, if the check succeeds, makes a privileged system call on the object. For example, one notorious error involves the `access()` and `open()` system calls. Consider a program running as root (e.g., setuid root, or executed by root) executing the following code fragment:

```
if (access(pathname, R_OK) == 0)
  fd = open(pathname, O_RDONLY);
```

The programmer is attempting to enforce a stricter security policy than the operating system. However, the kernel does not execute this sequence of system calls atomically— so if there is a context switch between the two calls or if the program is running on a multiprocessor system, another program may change the permission of the object. When the above program resumes its execution, it then blindly performs `open()` even though the user should no longer have access permission to the object.

Another example comes from UNIX folklore. It is well known that the root user should not recursively remove files inside directories that may be writable by other users. For example, “`rm -rf /tmp`” is a dangerous command, even if root has verified that the directory `/tmp` contains no symlink to other parts of the file system. The reason is that after `rm` verifies that a directory is not a symlink but before it enters the directory to delete the files within, an adversary may replace the directory with a symlink to another part of the file system, therefore tricking `rm` into deleting that part of the file system.

Many of the vulnerabilities that we found are exploitable when two users share access to some portion of the file system, and one user is operating on the shared portion while the other mounts an attack by replacing symbolic links. Although programs commonly attempt to ensure that they do not follow symbolic links before doing dangerous operations like `unlink()`, they often check it incorrectly and are susceptible to TOCTTOU attacks as a result.



Check(f) = {`stat(f), lstat(f), access(f), readlink(f), statfs(f)`}.
Use(f) = {`basename(f), bindtextdomain(f), catopen(f), chown(f), dirname(f), dlopen(f), freopen(f), ftw(f), mkfifo(f), nftw(f), opendir(f), pathconf(f), realpath(f), setmntent(f), utmpname(f), chdir(f), chmod(f), chroot(f), creat(f), execv(f), execve(f), execl(f), execlp(f), execvp(f), execle(f), lchown(f), mkdir(f), fopen(f), remove(f), tempnam(f), mknod(f), open(f), quotactl(f), rmdir(f), truncate(f), umount(f), unlink(f), uselib(f), utime(f), utimes(f), link(f), mount(f), rename(f), symlink(f)`}.
Non-filename arguments are omitted.

Figure 1. A refined FSA for finding TOCTTOU (file system race condition) vulnerabilities.

We have experimented with several different FSAs to capture these types of vulnerabilities. In our first attempt, we chose an FSA that had two transitions—one from the start state to an intermediate state, and the other from the intermediate state to the accepting state. Both transitions were defined on the union of the file system calls that access the file system using a pathname argument. However, we found that this naive property results in too many false positives—for example, `chdir(".")` followed by a `chdir(".")` would trigger a false positive. Typically these situations arise when a single system call is located inside of a loop, and both transitions in the FSA are made as a result of executing the same line of code. Since these are obviously not security holes, we decided to separate out the file system calls that can be classified as "checks" from those that are "uses".

We refined the property by dividing the file system calls into two distinct sets. Figure 1 shows the general structure of the FSA. We assume here, and in subsequent illustrations, that there is an implicit *other* transition from every state back to itself; if none of the normal outgoing transitions match, the *other* transition is taken, and the FSA stays in the same state. The intuition is as follows: a call to *Check(f)* is probably intended to establish an invariant (e.g., "*f* is not a symlink"), and a call to *Use(f)* might rely on this invariant. Of course, these invariants might be violated in an attack, so *Check(f)* followed by *Use(f)* may indicate a TOCTTOU vulnerability. This refined property is much more manageable and finds most of the bugs we are interested in. However, the more general property is capable of finding some bugs which the narrower TOCTTOU cannot—for example, `creat(f)` followed by `chmod(f)`.

The types of vulnerabilities we have found can be classified under the following categories:

1. [*Access Checks*] A check is performed—often by a program running as root—on file access permissions. The result of the check is then used to determine whether a resource can be used. The `access(f)` and `open(f)` race at the beginning of this section illustrates this class of bugs.

2. [*Ownership Stealing*] A program may `stat()` a file to make sure it does not exist, then `open()` the file for writing. If the `O_EXCL` flag is not used, an attacker may create the file after the `stat()` is performed, and the program will then write to a file owned by the attacker. We consider this a vulnerability, because the program may disclose sensitive information.

3. [*Symlinks*] Vulnerabilities due to symbolic links arise when two users share the same portion of the file system. One user can change a file to a symlink to trick the other user to mistakenly operate on an unexpected file. The method of such an attack depends on whether the system call follows symlinks. Broadly, there are two classes of system calls:

   (a) [*Syscalls that follow symlinks*] Many system calls will follow symbolic links that occur anywhere in the pathname passed to them. These present no barrier to attack.

   (b) [*Syscalls that don't follow symlinks*] Other system calls avoid following symbolic links if they occur in the last component of their pathname argument. For instance, if `c` is a symbolic link to `d`, calling `unlink("/a/b/c")` will delete the symbolic link itself rather than the target of the link: it deletes `/a/b/c`, not `/a/b/d`. However, many programmers do not realize that these calls will gladly follow any symlinks that occur in earlier components of the pathname. For example, if `b` is a symlink to `../etc`, then `unlink("/a/b/passwd")` will delete the password file `/etc/passwd`. Consequently, to attack this second class of system calls, it suffices for the attacker to tamper with one of the intermediate components of the path.

Many bugs we found were not previously known. Some were previously reported (but apparently not yet fixed and not known to us at the time of our experiments). To illustrate the kinds of bugs we found with MOPS, we will show three representative examples of TOCTTOU bugs.

```
# binutils-2.13.90.0.18-9 :: ar #
exists = lstat (to, &s) == 0;
/* Use rename only if TO is not a symbolic
   link and has only one hard link.  */
if (! exists || (!S_ISLNK (s.st_mode)
  && s.st_nlink == 1)) {
  ret = rename (from, to);
  if (ret == 0) {
    if (exists) {
      chmod (to, s.st_mode & 0777);
      if (chown (to, s.st_uid,
           s.st_gid) >= 0) {
        chmod (to, s.st_mode & 07777);
      }
  ...
```

In our first example, the program *ar* executes the code fragment above to replace an archive file with one of the same name. It calls lstat on the destination file and then checks if the file is a symbolic link. If it is not, *ar* calls rename on the file and then sets the mode of the file. This code, however, is unsafe. An adversary may change the file to be a symbolic link after *ar* checks for symbolic links. Then, *ar* will happily change the mode of whatever the symbolic link points to—assuming the user running *ar* has permission to do so. The attack is applicable when two users have write access to the directory of the archive file.

```
# initscripts-7.14-1 :: minilogd #
/* Get stat info on /dev/log so we can later
   check to make sure we still own it... */
if (stat(_PATH_LOG,&s1) != 0) {
   memset(&s1, '\0', sizeof(struct stat));
}
...
if ( (stat(_PATH_LOG,&s2)!=0) ||
     (s1.st_ino != s2.st_ino ) ||
     (s1.st_ctime != s2.st_ctime) ||
     (s1.st_mtime != s2.st_mtime) ||
     (s1.st_atime != s2.st_atime) ) {
         done = 1;
         we_own_log = 0;
}
/* If we own the log, unlink it before trying
   to free our buffer. Otherwise, sending the
   buffer to /dev/log doesn't make much sense */
if (we_own_log) {
   perror("wol");
   unlink(_PATH_LOG);
}
```

The second code fragment is taken from the program *minilogd*, which is run by root. This program may unlink _PATH_LOG (which is defined to be /dev/log by default) if it thinks it exclusively owns the file. It compares the timestamps on the file at two different times in the execution of the program and, if they are equal, decides that it exclusively owns the file and then removes the file. However, even if another program modifies the file after *minilogd* checks the timestampes, *minilogd* will still unlink it, possibly corrupting other programs. An additional vulnerability exists when user programs can write to the log file; for instance, if _PATH_LOG is defined as something like /home/alice/log instead. In this case, Alice can trick *minilogd* into removing anything on the file system. We have found many vulnerabilities that are similar to the latter case. It is important that these filename constants be checked very carefully when these programs are built, since it may not be obvious to users that defining _PATH_LOG to a user-writable file can result in a total compromise of the file system.

```
# zip-2.3-16 :: zip #
  d_exists = (LSTAT(d, &t) == 0);
  if (d_exists) {
    /* respect existing soft and hard links! */
    if (t.st_nlink>1 ||
          (t.st_mode&S_IFMT)==S_IFLNK)
       copy = 1;
    else if (unlink(d))
       return ZE_CREAT;
  }
```

The final code snippet comes from the widely-used program *zip*. If the destination file already exists, *zip* will move the file to a new location, unlink the old copy, and write the new copy. The program verifies that the file is not a link before calling unlink on it. The attack is applicable when two users share a portion of the file system and one user is running *zip* to write a new file to the shared space. If the other user is malicious, after *zip* calls stat, the user can change the file to be a symbolic link that points to another part of the file system. Since unlink will not follow the last component of a pathname, the attacker would have to change one of the components in the middle of the pathname to a symbolic link. For instance, if Alice is using *zip* to write a file to /shared/alice/afile[2], Bob can change /shared/alice to be a symbolic link that points to /home/alice. Then the *zip* program running on behalf of Alice will remove /home/alice/afile. Most users will not be aware that using a shared directory enables such attacks, so it seems unfair to blame Alice for doing so. In this case, *zip* does try to do the right thing by checking for symbolic links; it just happens to get the check wrong.

[2]The suggested scenario requires that the sticky bit is not set. The sticky bit prevents deletion of files and directories for anyone except the creator, even if others have write access. Generally, /tmp has the sticky bit set.

3.2 A Standard File Descriptor Attack

The first three file descriptors of each Unix process are called *standard file descriptors*: FD 0 for the standard input (*stdin*), FD 1 for the standard output (*stdout*), and FD 2 for the standard error (*stderr*). Several commonly used C standard library functions read from or write to these standard file descriptors; e.g., `fgets()` reads from *stdin*, `printf()` writes to *stdout*, and `perror()` writes to *stderr*. Programs that print information intended for the user to see, or diagnostic information, typically do so on FDs 1 and 2. Customarily, a program starts with its standard file descriptors opened to terminal devices. However, since the kernel does not enforce this convention, an attacker can force a standard file descriptor of a victim program to be opened to a sensitive file, so that he may discover confidential information from the sensitive file or modify the sensitive file.

For example, suppose a victim program is setuid-root[3] and executes the following code:

```
/* victim.c */
fd = open("/etc/passwd", O_RDWR);
if (!process_ok(argv[0])) perror(argv[0]);
```

Then the adversary can run the following attack program to exploit the standard file descriptor vulnerability in the victim program:

```
/* attack.c */
int main(void) {
  close(2);
  execl("victim",
    "foo:<pw>:0:1:Super-User-2:...", NULL);
}
```

This attack works as follows. First, the attack program closes FD 2 and executes victim.c. A child process will inherit the file descriptors from the parent process; consequently, the victim program starts with FD 2 closed. Then, when the victim opens the password file */etc/passwd*, the file is opened to the smallest available file descriptor—in this case, FD 2. Later, when the victim program writes an error message by calling `perror()`, which writes to FD 2, the error message is appended to */etc/passwd*. Due to the way the attacker has chosen this error message, the attacker may now log in with superuser privileges. These bugs are particularly dangerous when the attacker can influence the data written to the standard FD.

In the previously discussed vulnerability, the attacker is able to append content to an important system file. One can envision similar attacks on the *stdin* file descriptor in which the attacker can read content from a file that is not intended

[3]A setuid-root program runs with root privileges, even if it is executed by an unprivileged user.

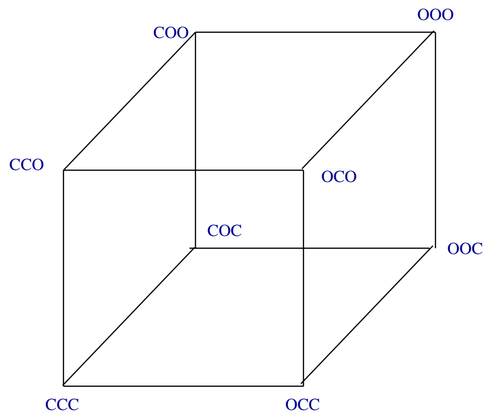


Figure 2. The structure of an FSA for finding file descriptor vulnerabilities. This FSA tracks the state of the three standard file descriptors across `open()` calls.

to be publically available. To stage such an attack, the malicious program would first close FD 0, then execute the privileged program containing code that unwittingly tries to read from *stdin*. The vulnerable program will now instead read data that is from the attacker's choosing, and possibly disclose confidential information in the process. Note that the program must relay the information it has learned back to the attacker, either directly or through a covert channel. The latter means of disclosure is impossible to detect using the single program analysis techniques we employ.

The way to prevent these types of attacks is simple—a program that runs as setuid-root should ensure that the three lowest numbered file descriptors are opened to a known safe state prior to carrying out any important operations. A common way to do this is by opening a safe file, such as */dev/null*, three times when the program first starts up. In the case that someone tries to attack the program by closing one or more of the file descriptors prior to executing the victim program, no harm is done because they are re-opened to point to */dev/null*. This solution is usually acceptable because the overhead is only three system calls. In the case that all 3 FDs were already opened, the program also consumes three file descriptor slots.

Our FSA used in this property (see Figure 2) contains eight states that are used to describe a unique combination of the states of the three standard file descriptors 0, 1 and 2. For example, the state `OCC` represents that FD 0 is open, but FD 1 and FD 2 are closed. The program may start in any of the seven states where at least one of the three standard FDs is closed; the case where all of the standard FDs are initially open is the usual one, and not of interest to an attacker. The starting state will be chosen nondeterministically during the model checking phase to insure all possibilities are explored. Transitions in the FSA occur only along the edges of the cube, as there are no system calls that can change the status of multiple standard file descriptors at once.

The basic FSA structure in Figure 2 is not entirely complete, as we have not shown the error state. For detecting the class of attacks which can cause the vulnerable program to write to arbitrary files, we add a new error state and a transition to the error state when a file that is neither */dev/null* nor */dev/tty* is opened on FD 1 or 2 in a mode other than read-only. For detecting the class of attacks that may disclose the contents of secret files, we add transitions to the error state from the four states in which FD 0 is closed (COO, COC, CCO, and CCC), and a file other than */dev/null* or */dev/tty* is opened for reading. These two transitions are seperated into two different automata, to give the two properties.

To save space, we have not labeled the transitions along the edges of the cube. These transitions are taken for system calls that are considered a "safe" open—that is, when */dev/null* or */dev/tty* is opened. For example, if the current state is COC and a "safe" open is encountered, then the new state is OOC, since the file will be opened on the lowest-numbered available FD.

We have audited the programs that run as setuid root on our Linux distribution, and have identified a number of bugs (but not exploitable vulnerabilities at this time). In many cases, an attacker can cause a setuid program to write data not of her choosing to temporary files, lock files, or PID files (files used to store the process ID of the currently running service). These situations can be potential vulnerabilities if some other program trusts the contents of the PID file. For example, consider a system administration script for restarting some network daemon that executes `kill ‘cat pidfile‘`. If the attacker exploits a setuid program that writes to this PID file to introduce a line of the form "*PID*; `rm /etc/passwd`" into the PID file, then the administration script might unwittingly remove `/etc/passwd` when it is next run. We have not yet found any fully exploitable scenario like this, but the fact that some setuid programs allow corrupting PID files like this is perhaps room for some concern.

An example of a bug we found in the program *gnuchess*, a chess playing application, follows:

```
int main(int argv, char *argv[]){
   ...  BookBuilder(depth, ...); ...
}
void BookBuilder(short depth, ...){
  FILE *wfp,*rfp;
  if (depth == -1 && score == -1) {
    if ((rfp = fopen(BOOKRUN,"r+b")) != NULL) {
      printf("Opened existing book!\n");
    } else {
      printf("Created new book!\n");
      wfp = fopen(BOOKRUN,"w+b");
      fclose(wfp);
      if ((rfp = fopen(BOOKRUN,"r+b"))
            == NULL) {
        printf("Could not create %s file\n",
            BOOKRUN);
       return;
      }
      ...
}
```

The function `BookBuilder` is called to manipulate and read from the playbook used by the game. Although there is no attack to compromise security, it is easy to see the bug. The playbook can become corrupted when a malicious user closes all file descriptors except standard out, and invokes the gnuchess program. The file *BOOKRUN* will then be opened onto standard out, and the subsequent writes from *printf()* can corrupt the book.

The full version of the paper lists results from applying MOPS to all Redhat 9 setuid programs. There were two main sources of false positives: 1) the property does not track the UID privilege changes inside the program, so the program may drop privileges before opening files, and 2) the property did not recognize that the program safely opened */dev/null* three times, due to a nonstandard invocation of safe opens. Unfortunately these are difficult false positives to recognize, because they require the user to look at the trace in its entirety as opposed to the usual points of interest (line numbers that caused transitions in the FSA). The presentation of our results differentiates betweens bugs and exploits. For this property, we classify bugs as programming mistakes that can cause unexpectd program behavior, but not necessarily lead to any compromise of security. For example, an attack that can compromise the contents of a non-important file, such as a lockfile, falls under the category of a bug. An exploit needs to have security concerns—we have found none of these to date. However, it was surprising that many setuid programs did not open */dev/null* three times before performing file operations, given that it has low overhead and guarantees safety with regards to this property.

3.3 Secure Temporary Files

Applications often use temporary files as a means for passing data to another application, writing log information, or storing temporary data. Often times on a Unix system, the files will be created in the `/tmp` directory, which is world writable and readable. For example, the GNU C compiler creates temporary files when it is compiling programs, and later passes them to the linker. Many of the functions to create temporary files that are found in the C standard library are insecure. The reason is that they do not return a file descriptor, but rather a file name. An adversary that is able to

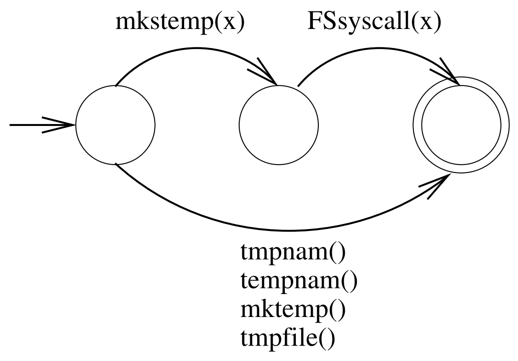


Figure 3. An FSA to detect insecure uses of temporary files.

predict the filename can thus create the file before the application has a chance to open or create it. This attack can give the adversary ownership of the temporary file, which is undesirable[4].

We identified the set of insecure functions: `mktemp`, `tmpnam`, `tempnam`, and `tmpfile`. These functions should never be used. There is one function that can be secure, depending on how it is used: `mkstemp`. Security requires that the filename retrieved from `mkstemp` is never subsequently used in another system call: `mkstemp` returns both a file descriptor and a filename, but a secure program should not use the filename. Figure 3 illustrates our automaton.

Below we show a representative example of a program that violates the clause of our property that finds reuses of the parameter passed to `mkstemp`. Not only is this the most complicated example presented thus far, but it shows how the whole-program inter-procedural analysis was effective. The code will be presented in several fragments as they occur temporally while executing the program. The example comes from the program *yacc* from the package `byacc-1.9-25`.

```
static void open_files() {
  int fd;
  create_file_names();
  if (input_file == 0) {
    input_file = fopen(input_file_name, "r");
    if (input_file == 0)
       open_error(input_file_name);
    fd = mkstemp(action_file_name);
    if (fd < 0 || (action_file =
      fdopen(fd, "w")) == NULL) {
      if (fd >= 0)
        close(fd);
      open_error(action_file_name);
    }
}
```

Before the above program fragment executes, there is some setup code that sets the value of the variable `action_file_name`. Specifically, it is a string whose first component is a pathname to a temporary directory (by default, it chooses `/tmp`, but this can be changed by defining an environment variable), and whose second component is a temporary file template[5]. The above code alone should be of concern to us. Recall that `mkstemp` returns a file descriptor that can be safely used, but the template passed to `mkstemp` is not safe to re-use. In this case, we see the template being passed to another function called `open_error`:

```
void open_error(char *filename) {
  warnx("f - cannot open \"%s\"", filename);
  done(2);
}
```

From the above fragment it looks like the function `warnx` may be a good candidate for inspection because it is the recipient of the filename we are interested in tracking. Strangely enough, MOPS directs us to the function `done`:

```
void done(int k) {
  if (action_file)
    fclose(action_file);
  if (action_file_name[0])
    unlink(action_file_name);
```

Here we find the bug. The variable `action_file_name`, which is the template passed to `mkstemp`, is re-used as an argument to the `unlink` system call. This is unsafe. By the time we call `unlink`, the filename may no longer point to the location we think it does. Recall that the directory in which the file is being created may be world writable. An attacker that has write access to the file can change it to a symbolic link, and cause the program to unlink other unexpected files on the system. Unfortunately there does not appear to be a good resolution to the problem.

3.4 Attacks Against `strncpy`

There are several common attacks against programs that misuse the standard library function `strncpy`. `strncpy(d,s,n)` copies a string of characters pointed to by `s` into the memory region designated by `d`. If `s` contains more than `n` characters, `strncpy` only copies the first `n` characters. If `s` contains fewer than `n` characters, `strncpy` copies all the characters in *s* and then fills *d* with null characters until the length of `d` reaches `n`.

[4]In the `gcc` example, an adversary could insert malicious code into a user's program by replacing the temporary file with the desired code.

[5]A template is a partial filename with a number of placeholders denoted by a special character `X` that will be filled in with random numbers by the function creating the temporary filename.

strncpy is easy to misuse for two reasons. First, it encourages off-by-one errors if the programmer is not careful to compute the value of n precisely. Off-by-one errors can often cause the program to write past the end of an array bounds, which can in turn lead to buffer overrun attacks against the program. In particular, consider a case where the string buffer in question is allocated on the runtime stack (as it will be when the buffer is an array local to a function in C), and the user of the program is able to control the contents of the source of s. If the program writes past the end of the buffer, a malicious user may construct a special string s, such that when the program writes past the array bounds, it writes special code into the stack frame that corrupts the program. Secondly, because the function does not automatically null-terminate a string in all cases (for instance, when the size of the source string is larger than n), it is a common mistake for a program to create unterminated strings during its execution.

We have constructed an FSA to try and catch both scenarios as described above. The intuition is that we identify several idioms that are correct ways to null terminate a string, and raise an alarm when one of these idioms is not used. For example, a common idiom is the following code sequence that is safe:

```
buf[sizeof(buf)-1] = '\0';
strncpy(buf, ..., sizeof(buf)-1);
```

In the above case, the buffer will also be terminated. However, the following two cases show a common misuse of strncpy:

-
  ```
  buf[sizeof(buf)-1] = '\0';
  strncpy(buf, ..., sizeof(buf));
  ```

-
  ```
  memset(buf, 0, sizeof(buf)-1);
  strncpy(buf, ..., sizeof(buf)-1);
  ```

In the first unsafe example, the string is null-terminated before the strncpy, and the execution of the function subsequently may overwrite the null-terminating character. In the second unsafe example, memset is used to zero-out the destination buffer; unfortunately, it is misused—the third argument needs to be the size of the entire buffer. Our FSA attempts to detect patterns that appear fishy, and alerts the user to their presence. Pattern variables are used judiciously to make MOPS precisely match the null-terminating code to the strncpy code that uses the same buffers. The property requires more manual inspection than other properties, because we have chosen an approach where we identify correct behavior, then raise an alarm at code that does not match our expectations. Moreover, our property does not attempt to find all strncpy bugs, focusing instead on patterns that are particularly suspicious.

Our strncpy FSA has alerted us to a number of bugs. Below we show one of the most interesting examples. It comes from the program *xloadimage*:

```
void dumpImage(Image *image, char *type,
  char *filename, int verbose) {
  int a;
  char typename[32];
  char *optptr;
  optptr = index(type, ',');
  if (optptr) {
    strncpy(typename, type, optptr - type);
    typename[optptr - type] = '\0';
  ...
```

In the above code fragment, MOPS identifies an idiom that does not appear to be safe. The character buffer typename is declared to be 32 bytes long, but the length passed to strncpy is computed entirely based on the second argument to the function dumpImage. We must verify that this string cannot be constructed in such a way that when optptr - type is computed, the result is longer than 32 bytes. MOPS is able to direct us to the call site of this function, in which we see the following (abbreviated):

```
newopt->info.dump.type= argv[++a];
...
dumpImage(dispimage, dump->info.dump.type,
  dump->info.dump.file, verbose);
```

Shockingly, the contents of the string come from the command line arguments, and can be set entirely by the user. Consequently, a malicious user can supply a carefully crafted argument to the function which causes the function dumpImage to write past the end of an array, causing a buffer overrun.

Unfortunately, this property produced 1378 unique warnings, too many to examine exhaustively by hand. Therefore, we picked a semi-random sample of 16 packages out of the set of 197 packages with one or more warning, yielding a set of 53 warnings. Examining these 53 warnings revealed 11 bugs spread among 6 packages, where in each case a string could be left unterminated or the strncpy() operation could overflow buffer bounds for some input. We did not attempt to assess the security implications of these bugs, though we expect that many of them could be exploited under some circumstances.

Based on this limited sample, we suspect that a full manual audit using MOPS would turn up many more strncpy() bugs. Since we saw 11 bugs among 53 warnings, this suggests a false positive rate of about 79%, and a true positive rate of about 21%. If all warnings are equally likely to correspond to real bugs, we might estimate that there are about $1378 \times \frac{11}{53} \approx 286$ bugs, with a 95% confidence interval of between 165 to 468 bugs. Of course, these estimates are fairly rough, but our best prediction is that a full manual audit of all the MOPS warnings would turn up in excess of one hundred `strncpy()` bugs.

4 Related Work

There is a broad and growing array of work on software model checking, and MOPS represents just one of several tools in this area. BLAST [11] and SLAM [3] are dataflow-sensitive model checkers that use adaptive iterative refinement to narrow down the locations of bugs. Both have been used primarily on smaller programs, such as device drivers, but they are able to provide a much more precise analysis. Similar to BLAST is MAGIC [5], a system that abstracts predicates and uses a theorem prover for dataflow analysis. CMC, a model checker for C and C++ programs [13], has been used on large-scale applications like the entire Linux kernel [10]. Metal, a ground-breaking bugfinding tool which is similar in concept to a model checker, has been very successful at finding a broad variety of rule violations in operating systems [2, 9]. Metal has been augmented with Z-ranking, a powerful technique for reducing the number of false positives, and used to find many bugs in large applications [12]. MOPS has previously been used to find bugs in eight security-relevant packages from the Red Hat distribution [6, 7]. However, none of these tools have yet been applied on as large a scale as shown in this paper.

File system race conditions have been extensively studied in the computer security literature. Bishop and Dilger first articulated the vulnerability pattern and developed a syntactic pattern-matching analysis for detecting TOCTTOU vulnerabilities in C code [4]; however, because their analysis is not semantically based, it is unable to find many of the vulnerabilities found in this work. Also, some authors have proposed runtime program analysis methods to detect TOCTTOU bugs by monitoring program executions and preventing their exploitation [8]; our work differs by trying to find TOCTTOU bugs at compile time, rather than at runtime.

5 Conclusion

Our work demonstrates that large-scale model checking is feasible. We showed that it is possible to develop models of incorrect and insecure program behavior that are precise enough to prevent false positives from dwarfing the real bugs; also, as this work showed, many of these properties can be encoded in MOPS without serious loss of soundness. Thanks to the sophisticated error reporting in MOPS, we found that we were able to manually inspect all error traces. Consequently, we were able to find many (108) real exploitable software bugs; in several cases, we have crafted attacks to verify their validity. As a result of this experience, we are convinced that software model checking can easily be integrated into the development process, particularly when using model checkers like MOPS that can be integrated into build processes at the highest level.

References

[1] `mopscode.sourceforge.net`.

[2] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *Proceedings of IEEE Security and Privacy 2002*, 2002.

[3] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL '02: Proceedings of the ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages*, 2002.

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

[5] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *International Conference on Software Engineering*, pages 385–395, May 2003.

[6] H. Chen, D. Dean, and D. Wagner. Model checking one million lines of C code. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, San Diego, CA, Feb. 4–6, 2004.

[7] H. Chen and D. 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, Nov. 18–22, 2002.

[8] C. Cowan, S. Beattie, C. Wright, and G. Kroah-Hartman. RaceGuard: Kernel protection from temporary file race vulnerabilities. In *Proceedings of the Tenth USENIX Security Symposium*, 2001.

[9] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*, 2000.

[10] D. Engler and M. Musuvathi. Model-checking large network protocol implementations. In *Proceedings of the First Symposium on Networked Systems Design and Implementation*, 2004.

[11] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *Proceedings of the 10th SPIN Workshop on Model Checking Software*, 2003.

[12] T. Kremenek and D. Engler. Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations. In *Proceedings of the 2003 Static Analysis Symposium*, 2003.

[13] M. Musuvathi, D. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, Dec. 2002.