How to Open a File and Not Get Hacked

James A. Kupsch Barton P. Miller
Computer Sciences Department
University of Wisconsin
Madison, WI 53706-1685 USA
{kupsch,bart}@cs.wisc.edu

Abstract

Careless attention to opening files, often caused by problems with path traversal or shared directories, can expose applications to attacks on the file names that they use. In this paper we present criteria to determine if a path is safe from attack and how previous algorithms are not sufficient to protect against such attacks. We then describe an algorithm to safely open a file when in the presence of an attack (and how to detect the presence of such an attack), and provide a new library of file open routines that embodies our algorithm. These routines can be used as one-for-one substitutes for conventional POSIX `open` *and* `fopen` *calls.*

1. Introduction

Common programming idioms allow adversaries to violate security constraints. Some of these programming idioms that allow exploits to occur involve opening, creating and performing other operations on files. In this paper we focus on opening and performing file operations in such way that an adversary will not be able to subvert security.

In particular we are assuming the adversary has local access to the machine running the program, but not as a user the program must trust, such as the *root* account. The access to the untrusted account may be achieved by numerous means, including having a legitimate account, breaking into the account, or by exploiting a service with a network interface. We assume the adversary has the capability to create, remove and otherwise manipulate files and directories anywhere the permissions of an untrusted account allows.

Specific types of attacks that can occur when precaution are not taken include (1) race conditions when the idiom uses a file name multiple times such as between checking for the existence of a file and creating it [5, pages 528–530], (2) race condition between opening or creating the file and checking the ownership and permissions of the file to prevent confidential data from being disclosed as in Globus's gt4 [7], (3) inadvertently following symbolic links allowing the creation or modification of files in unexpected locations as in IBM's DB2 [4] and Xsession [3], and (4) weak file permissions resulting from incorrect use of the API.

These types of problems are especially problematic when the target application is running with elevated privileges such as *root*, and the file used in the attack is in a directory that is writable by the adversary. If the target application is running with elevated privilege, it can modify files or disclose file contents that the adversary would not normally be able to access and compromise the entire system. If the target application is running as an ordinary user, attacks are still possible on data owned by the user running the application. The problem of accessing files in a directory that is writable can be prevented by the application detecting that this situation is occurring and refusing to process files in such locations. If the application must access files where an adversary could modify them, the file needs to be operated on using safe techniques.

This paper is organized into two main parts. The first part (Section 2) shows how to detect if a file name is safe from adversarial attacks; we encapsulate this detection in an algorithm that checks if a file name is a trusted path. We describe how attacks can be performed on such a path and what are the requirements of a trusted path. We present two prior algorithms for determining if a path is trusted and discuss their limitations. We then present a new algorithm that does not have those limitations.

The second part (Section 3) describes functions that are replacements for the standard library functions for opening and creating files. These functions should be used when an application opens a file that is not reached by a trusted path. These replacement functions provide a simple API similar to the original functions, and eliminate the problems of race conditions and symbolic links; they also make the initial permissions when a file is created explicit by making it a required parameter to any of the functions that can create files. We also provide alternative functions for opening and creating a file; these functions give a knowledgeable developer additional features such as atomically replacing a file if it exists, and allowing an `fopen`-style function to open a file for appending and fail if it does not already exist.

These functions also provide a mechanism for an application to detect if file names used with these functions are being manipulated to refer to different file system objects as might occur during an active attack on the file name.

The use of these new routines alone does not guarantee correct security behavior as it is still possible for a design flaw or coding errors in the use of these functions to cause a vulnerability in the program. However the use of our new routines should significantly reduce the risk of many common security exploits.

2. Trusted Paths

A path is a string used to refer to files to perform operations such as opening a file. An example of a path is `/home/user/report.pdf`. A more technical description of a path is a function that maps a string to a traversal of the file system, which produces an object in the file system such as a file or directory. In the above example, the traversal would be `/` $\rightarrow$ `home` $\rightarrow$ `user` $\rightarrow$ `report.pdf`. This traversal is done each time that a path is used in a system call. If an attacker can manipulate this traversal by making changes to directory entries, the same path can refer to different file system objects and can be used to exploit a program by making it think that it is acting on one file when it is really acting on a different one.

This section describes how to assess if a path traversal or contents of a file can be manipulated by a malicious user in a POSIX [10] environment as found in operating systems such as UNIX and Linux. In such an environment, user and group ids are used to control access to resources. Processes have one owner user id and multiple group ids that are used to check access to file system objects. File system objects have one owning user and group id, plus three sets of capabilities (the set chosen is based on the file's owning user id, group id, or neither matching that of the process). Of the capabilities in each set, the paper is concerned with the write flag as that is the flag that allows modifications to the object (other flags allow reading and execution/access).

To meet security requirements, programs often need assurances that the contents of a file could have only been modified by processes that were run with a user id and group ids from a set that the program trusts not to be malicious. The set of ids that the program trusts not to be malicious will be referred to as *trusted ids*. A process that has at least one trusted id for its user or groups will be referred to as a *trusted process*, while a process where none of its user or groups are trusted ids will be referred to as an *untrusted process*. The set of trusted ids depends upon the security requirements of the program and how the user and groups are used in the file system. The *root* user id is trusted, since file access for a process running with the *root* user id is always granted. The trusted ids may also vary in an application from file to file. For instance, if the security requirement is that the configuration file of the program must not be modifiable by the user running the program, then the set of trusted user ids might be (*root, prog-admin*), and the trusted group ids might be (*root, wheel, admin*). The same program may have the requirement that data files should be modifiable by the user and group of the process, so for these types of files the trusted user ids might be (*root, user*), and the trusted group ids might be (*root, admin, user-group*).

A trusted path is one where the permissions are such that only a trusted set of user and group ids can manipulate the traversal to the file system object or contents of the object referred to by the path. Only trusted processes can modify the meaning of a trusted path, and they should not modify the path in a way that violates the security requirements of the program. These properties allow a trusted path to be used in ways that would create security problems if the path were not trusted, such as assuming the contents of a file do not change, or multiple uses of a path always access the same file. A function to determine if a path is trusted is not a standard operating system or library service, and published algorithms are deficient in some respect.

The rest of this section presents definitions and details about trusted paths, the types of exploits possible if a path is not trusted, and a list of properties that an algorithm should have to determine the trust of a path. We then describe two previous algorithms for determining if a path is trusted, and evaluate how well they meet the properties given. Finally we present a new algorithm that meets all the desired properties.

2.1. Definitions

This section provides definitions used in the rest of the discussion on trusted paths. We describe different types of paths and file system objects along with their properties.

File system object - an object that is accessible within the file system, such as a file, directory, or symbolic link. The file system object is identified by a unique device and inode number pair.

Path - a textual representation of a file system object. The path consists of a series of names separated by slashes, where all but the last directory entry are directory or symbolic link objects.

A path is usually limited in length to the constant `PATH_MAX`, and if an attempt is made to use a longer one, the operating system returns the error `ENAMETOOLONG`. It is possible to create a series of deeply nested directories where the absolute path length exceeds `PATH_MAX`. In this case the file system objects may be accessed by setting the current working directory of the process to a directory that is a prefix of the absolute path and then using a relative path that is shorter than PATH_MAX to reach the directory entry desired.

Absolute path - a path that begins with a slash, where the initial directory for the traversal begins at the process's root directory.

Relative path - a path that does not begin with a slash, where the initial directory for the traversal begins at the process's current working directory.

Symbolic link - a file system object that contains a path. This is a pointer to another location in the file system that is evaluated when the operating system encounters the symbolic link while processing a path. If the symbolic link is an absolute path, the current directory of the traversal is reset to the root directory; otherwise, the current directory of the traversal is left as is (the referent of the symbolic link is relative to the directory containing the symbolic link). Once the symbolic link referent is completely processed, the remainder of the path is then processed. Note that the referent of a symbolic link may contain a symbolic link that must be processed recursively. The symbolic link path may form a loop that can only be easily detected at path processing time. To detect this, operating systems limit the depth of the recursion to some small number such as 10 or 16, and if the depth exceeds this, they return the error ELOOP regardless of whether there is an actual loop.

Most system calls completely resolve all symbolic links encountered in a path. Some system calls such as readlink and lstat operate on the symbolic link itself instead of the referent, but even these only operate on the symbolic link if the symbolic link is the last component of the path. For example, using lstat on the path /dir/symlink stats the symbolic link itself, but using the path /dir/symlink/. stats the directory to which /dir/symlink refers.

Static path - an absolute path to a file system object, where no component in the path is a symbolic link. All the information to determine the directory entries needed to process this path are contained in the path itself.

Dynamic path - any path that contains a symbolic link. If a component in the path contains a symbolic link, the actual file system object referred to can only be determined by knowing the path, plus the referents of all symbolic link directory entries in the path.

Canonical path - a static path, where no components refer to the current directory (represented by '.' or the empty value), or to a parent directory (represented by '..'). The canonical path forms a unique representation for the directory entry referred to by the path (this is not necessarily a unique name for the file system object as there may be multiple directory entries hard linked to the same file system object).

A static path can be transformed into a canonical path using only textual manipulations of the path, since without symbolic links, all the directory entries are known and semantics of the unwanted components are also known.

1. current directory components imply that the traversal stays in the same directory, so they can be deleted. For example, //home//./user/. $\Longrightarrow$ /home/user.
2. parent directory components imply that the traversal should be in the parent directory of the prior directory to the "..", so the parent directory and the ".." can be deleted (prefix/dir/../remainder $\Longrightarrow$ prefix/remainder). The parent directory, represented by dir above, must not be one of the unwanted components (they must be eliminated first). In this process the parent directory of the root directory, /, is itself, so a parent directory component immediately after the root directory becomes the root directory, so /../home $\Longrightarrow$ /home. For example, /../var/./../home $\Longrightarrow$ /home.

On the other hand, dynamic paths cannot be turned into a canonical path using the same algorithm because after processing a symbolic link, the parent of the current directory in the traversal can be anywhere.

Sticky bit directory - a directory file system object that has the sticky or saved text bit set in its mode flags. A directory of this type has different permission semantics when operations are called that remove directory entries such as unlink, rmdir, or rename. The /tmp directory is commonly created as a sticky bit directory, because it designates a world writable directory where only the owner of the file system object in the directory can remove the file. The semantics of removing a directory entry in a sticky bit directory are such that it will fail unless one of the following is true:

1. the uid of the process is *root* (0)
2. the uid of the process owns the directory
3. the uid of the process owns the entry

In most current operating systems, only the *root* user can change the ownership of files and directories, but in some old environments, if _POSIX_CHOWN_RESTRICTED is not defined, any user can change ownership of files they own to any other user. In this case a sticky bit directory must be treated as an ordinary directory, because a directory created by an untrusted process can appear to have been created by a trusted process. This happens when an untrusted process creates a directory and then changes ownership of the created directory to a trusted user. In an environment where _POSIX_CHOWN_RESTRICTED is not defined, the only secure use a sticky bit directory is to create a file and never use the path name again, except to remove the file.

Figure 1. `TrustEntry` algorithm. Returns the trust of a directory entry given the trust of the parent and `stat` buffer of the entry.

```
function TrustEntry(pTrust, curStat, u, g)
    — u is the trusted user list
    — g is the trusted group list
  if pTrust is UNTRUSTED
          or (pTrust is STICKYTRUST
              and curStat type is not directory) then
      return UNTRUSTED
  if curStat type is symbolic link then
      return TRUSTED
  if curStat owner is not in trustUsers
          or (curStat group is not in trustGroups
              and curStat group perms allows writing)
          or curStat other perms allows writing then
      if curStat type is sticky bit dir then
          return STICKYTRUST
      else
          return UNTRUSTED
  return TRUSTED
```

2.2. Checking the trust of a path

This section explains how to determine if a path is trusted. First, we show how to assess if a directory entry is trusted. Then, we show how to use a directory entry being trusted to assess if a whole path is trusted. Finally, we discuss how to tell if a file system object is trusted, which means that there is a path to the file system object that is a trusted path. There are three possible outcomes for trust: *trusted*, *sticky dir trusted* and *untrusted*. Trusted means that an untrusted process cannot modify the contents of the file system object or to which file system object the path refers. Untrusted means that an untrusted process can do those things. Finally, sticky dir trusted meets some of the properties of a trusted directory but not all, as explained below.

2.2.1. Checking the trust of a directory entry. The trust of a directory entry can be determined by knowing the trust of the parent directory and by the properties of the file system object to which the directory entry refers; these properties are obtained by using the `lstat` system call. The relevant properties consist of the type of the object, its owning user and group ids, its permissions and the sticky bit. Figure 1 shows an algorithm, `TrustEntry`, to compute the trust of an entry. The trust of a directory entry is determined by one of the three cases described below:

1. If the directory entry is a symbolic link, then it is trusted if the parent directory is trusted (not just sticky dir trusted).

   Since the permissions of a symbolic link are not used, checking for writability by untrusted user and group ids is not required. Besides normal directory entry manipulations such as delete, rename, stat, and create a hard link, there are only two operations that can be performed on a symbolic link: create a symbolic link with a given referent (path name) and get the referent. Because of these restrictions, a symbolic link object is immutable and the only attacks possible are by manipulating the containing directory.

   If the directory is a sticky bit directory, there are two cases to consider based on the owning user id of the symbolic link. If the owning user id is not a trusted user id, then the owner can delete and recreate the symbolic link, and therefore this symbolic link is untrusted.

   If the owning user id of the symbolic link in the sticky bit directory is a trusted user id, then an attack is still possible; by using a hard link an untrusted process can create a directory entry in the sticky bit directory that is owned by a trusted user. Strictly speaking, POSIX forbids creating hard links to symbolic links, so a symbolic link with a trusted owner could be considered safe. In pactice many implementations allow creating hard links to symbolic links, so the conservative approach is to treat them as untrusted.

   A hard link is a directory entry to an existing file system object, and is created using the `link` system call. The new directory entry is indistinguishable from the original file system object as both directory entries point to the exact same file system object, and therefore share all the attributes including the contents, owning user, owning group and permissions.

   POSIX places few restrictions on the use `link` system call: the source must not be a directory, the calling process must be able to create a file system object at the destination path (the same as `open` with `O_CREAT|O_EXCL`), the calling process must be able to access the source path's metadata (have search path permissions for all the directories in the source path), and the source and destination must be within the same physical file system. The process calling `link` is not required to have the same user and group ids as the owner of the source file, nor does it need to have read access to the contents of the source file.

2. For all other types of directory entries, the entry is trusted if all the following are true of the file system object to which the directory entry refers:

   (a) the object's user id is a trusted user id;

   (b) the permissions allow writing only by the object's group if the group is a trusted group id;

   (c) the permissions do not allow writing by others; and

   (d) if the parent directory is a sticky bit directory, then the type of the object is a directory.

   The owner of the file system object needs to be trusted because the owner of the object can always change the permissions to an arbitrary value. So even if the permissions are currently such that they prevent an untrusted process from modifying the object, the untrusted process could later change permissions to allow modifications. Obviously, if the permissions allow untrusted group members or others to modify the object, then the object is also untrusted. For the same reasons given in item 1 above, any directory entry that can be the source of a hard link (everything except directories) is untrusted if the parent directory is sticky dir trusted.

3. If the above two cases do not make the directory entry trusted, the directory entry is a sticky bit directory, and the directory's owning user id is a trusted user id, then it is sticky dir trusted. This is a weaker form of trust, in that an untrusted process can create directory entries in the directory, but they cannot delete directory entries owned by other users including all trusted users (which is prevented by the owning user id of the sticky bit directory being a trusted user id).

   The sticky dir trusted concept is worth considering because of the wide use of the `/tmp` directory, which has this property. This type of directory can be used in a safe and trusted manner in two cases:

   (a) Create a temporary file, so the file is trusted as described in item 2 above, using `mkstemp` or something similar, and only access the file through the returned file descriptor. The path to the file must never be used again to open the file, as the path is not trusted. It is not trusted as there is no guarantee that a trusted process, such a temporary file cleaner process, did not remove the file. An untrusted process could then create a directory entry with the same name that is a hard link to a file that looks like it was created by a trusted process as discussed previously. Even if the directory entry is removed, the file descriptor will still refer to the file system object returned by `open`. In fact `unlink` should be called immediately after opening a temporary file so the storage is freed on closing the file descriptor (which happens automatically on process exit), and access through the path is prevented.

   (b) Create a directory with trusted permissions, and use this as a trusted directory for all temporary directory needs. The created directory is safe to use as a path, because without the sticky bit set this directory is trusted. The created directory is trusted because a hard link to a directory is not allowed (or can only be created by the root user in older operating system versions).

2.2.2. Checking the trust of a complete path. A *trusted path* is a path where all the components of the path and components of symbolic link referents are comprised solely of trusted directory entries.

The directory entries of the path are checked left to right. If a directory entry is found that is untrusted, processing stops and untrusted is returned. If a symbolic link is found then the referent of it must be assessed recursively before the remainder of the current path is checked.

If the path is relative, then it is also a requirement that all the directories from the process's current working directory to the root directory are also trusted. This requirement is necessary because the process or its parent used a path or several in succession to set the current working directory. If the canonical path is not trusted, then no path to the current working directory can be trusted, and rerunning the executable could result in a different current working directory, therefore the current working directory should not be trusted. This does not mean that all the paths used to get to the current working directory are trusted, but it is assumed that the path(s) used to set the current working directory are trusted if you wish to trust relative paths.

A *trusted file system object* is a file system object where there exists a trusted canonical path to the file system object (there can be more than one due to hard links).

If there is a trusted path to a file system object, this implies that the file system object itself is trusted. This can be seen because the canonical path is the path that starts at the root directory and proceeds to the directory entry of the leaf. Since the only allowed traversals from one directory to another is either up to the parent or down to a direct child, each directory entry in the canonical path must be visited at least once in any path to the file system object. Since the definition of a trusted path is one where all the components of the path and of the symbolic link referents are trusted, all the components of the canonical path must be trusted, therefore the canonical path is a trusted path.

The converse of this property is not true and is easily seen through the counter example where the file system object `/trusted` is trusted, but the path `/untrusted/dir/../../trusted` is not.

2.3. Possible Attacks

These are different types of exploits that an untrusted process could accomplish if a path is untrusted and would be prevented if the path is trusted.

Modify contents of a file - this requires that the permissions on the file are untrusted, or that they were untrusted sometime during the lifetime of the file.

Denial of service - if one of the directories in the path traversal is an untrusted directory, then the directory entries contained in the untrusted directory can be modified through renaming or deletion such that the ultimate desired file system object will become unaccessible through this path.

Replace directory entry - after deleting or renaming a directory entry, a new one can be inserted in its place to make the program use the attacker's directory or file.

Hard link attack - is an attack where it will appear that a trusted process created a file with a particular path, when it did not. The attack is possible in any directory in which an untrusted process can create a file, such as a sticky bit directory like `/tmp`. The attack is based on the fact that any process can create a hard link to any file, regardless of the file's ownership and permissions. This link can be created anywhere the untrusted process could create an ordinary file. The newly created hard link will be indistinguishable from the original file and any changes made to one will be seen by the other including ownership, permissions and contents. If an application uses `/tmp` directly for storing files and incorrectly assumes any file with the correct ownership and permissions are trusted files, then an attacker can create additional files with these properties at a given path by using a hard link to an existing file. The source of the hard link can even be a file that the trusted process has removed if the attacker made a hard link before the source was removed and then uses this link as the source when needed for an attack.

If the trusted process ever created a file with permissions allowing the attacker to write to the file, or if the attacker can get the trusted process to write the desired contents to a file, then they can also control the contents of the file.

For this reason, any file system object that allows a hard link (everything except directories) should never be trusted in a sticky bit directory.

This type of attack is analogous to a protocol replay attack [6] in cryptography. A protocol replay attack is where a protocol has messages that are cryptographically protected from forgery, but an attacker can capture the protected message and replay messages at a later time, even though the attacker cannot create a new valid message. The named file corresponds to the message and the attacker cannot directly create a valid file, but they can provide a previously seen file to the trusted process.

Symbolic link manipulation - if an attacker can write to some directory that the path traverses, then the attacker can control the file system object to which the path resolves. This attack can cause the program to open and possible create files in arbitrary locations.

Race conditions - if the program uses the same path with multiple system calls, combinations of the previously described attacks can be used to make the program access one file system object in one call and a different file system object in a subsequent call. If information from the first system call, conditionally determines the use of a subsequent call using the same path, this is known as a time of check, time of use (TOCTOU) attack.

2.4. Trusted path algorithm properties

An algorithm to check if a path is trusted should have the following properties:

1. Supports multiple trusted user and groups ids instead of just one trusted user and group id; modern practices use many trusted user and groups ids to compartmentalize programs and services.

2. Works on all file system object types, including files and directories.

3. Only fails to produce a result if the operating system would also fail when presented with the path, so constructing paths and calls such as `getcwd` cannot be used because they can fail in deeply nested directories.

4. Properly checks symbolic link referents.

5. Properly detects symbolic link loops.

6. Works properly with sticky bit directories.

7. Is efficient in the number of system calls, directory scans, and inode accesses. Operations on file descriptors should be preferred over those using a path.

8. Is concurrent execution safe. Multiple instances of the algorithm should be able to be called concurrently from threads and signal handlers, and it should not change shared state in the process, such as the process's current working directory.

9. If the algorithm returns the path is trusted, an untrusted process cannot

   (a) modify the object referred to by the path; multiple uses of the path should refer to the same file system object,

   (b) modify the object's contents,

   (c) create, rename or delete a directory entry owned by a trusted user and group id.

Figure 2. Viega and McGraw's `safe_dir` algorithm.

```
function safe_dir(dir, owner_id)
  savedDir ← open(".", O_RDONLY)
  lstat(dir, lstatBuf)
  repeat
      chdir(dir)
      curDirFd ← open(".", O_RDONLY)
      fstat(curDirFd, statBuf)
      close(curDirFd);
      if statBuf's and lstatBuf's mode, inode
              and device do not match then
          return UNTRUSTED
      if statBuf's mode allows writing by group
              or other or statBuf's owner
              is not in {owner_id, root} then
          return UNTRUSTED
      dir ← ".."
      lstat(dir, lstatBuf)
      getcwd(newDirPath)
  while newDirPath is not "/"
  fchdir(savedDir)
  close(savedDir)
  return TRUSTED
```

2.5. Prior Work

This section describes two prior algorithms for checking if a directory or a path is trusted. We describe the algorithms, the properties they satisfy, and the complexity of the algorithms.

2.5.1. `safe_dir` algorithm. In the book *Building Secure Software* [11, pages 222–225], John Viega and Gary McGraw present an algorithm, shown in Figure 2, to check if a directory is trusted.

This algorithm requires the path to resolve to a directory, and only allows a single trusted user id (all group ids are considered untrusted). The algorithm works by changing the process's current working directory to the path argument, and checking if the directory is trusted (the directory is owned by the root user or the owner_id user id, and the group and other are not allowed to write to the directory). Next, the algorithm works its way up the directory chain by changing directory to the parent directory, "`..`", and repeats this loop until the root directory is found by checking if `getcwd` returns the root directory, "`/`".

Other than the initial change directory to the path, this algorithm does not use the path. `safe_dir` ends up checking that the canonical path or the directory given is trusted. An attempt is made to detect symbolic links by matching the result of `lstat` with `fstat` after changing directories, but this check ignores symbolic links that are not at the terminal part of the path.

There is a race condition in this function, not in the algorithm itself but in the interface provided. A TOCTOU attack is possible since the design of the interface and the authors' stated use for this algorithm is to call this function with a path to check if it is trusted, and if so, use the path a second time to change directories to the checked path. The race condition occurs because the algorithm does not check the path, but instead checks the trust of all the directories from the argument to the root directory. This is equivalent to the directories of the canonical path. A trusted canonical path does not imply that all paths to the same directory are trusted, as the path given could traverse untrusted directories or symbolic links. Since the argument is not verified to be a canonical path, the two uses of the path can result in different directories (the first being a directory with a trusted canonical path, and the second an untrusted directory after an entry in an untrusted directory along the path being modified). This could be fixed by having the current working directory set to the path checked on exit, or to eliminate the path argument and to check the current working directory.

Given these problems, this algorithm is only safe to use if the path given contains no symbolic links and no parent directory components. There is one useful case where this is true and that is in checking the current working directory, "`.`", which is guaranteed not to be a symbolic link.

The run time complexity of this algorithm is $O(n^2)$, where n is the depth of the directory (it would be $O(n)$ if `getcwd` were not used). The space complexity of this algorithm is a small constant.

The limitations of this algorithm include (1) not checking the path, but the canonical path to the directory, (2) only supporting a single trusted user, `owner_id`, or *root* (3) not handling the unique properties of sticky bit directories, (4) failing if the canonical path gets too long, and (5) not being concurrent-execution safe as it changes the current working directory.

In summary, if the path is not a canonical path, this algorithm satisfies none of the desired properties of Section 2.4. If the path is a canonical path, then it satisfies property 9, i.e. untrusted processes will not be able to change the directory referred to by the path, or create any directory entries in the path.

2.5.2. `trustfile` algorithm. Matt Bishop [2, pages 300–307] presented an algorithm, shown in Figure 3, that will check if a path is trusted.

This algorithm works on paths to arbitrary file system objects, and takes a list of trusted and untrusted user ids. The trust of the group id is computed from the list of trusted user ids (a group is trusted only if its membership consists solely of trusted user ids). While the approach of computing the trusted group ids is correct, it increases the administrative overhead. If a new user id is added as a member to a trusted group, the administrator needs to reconfigure the

Figure 3. Bishop's `trustfile` algorithm.

```
function trustfile(path, okUsers, badUsers)
  if path is a relative path then
      getcwd(curWorkingDir)
      path ← "curWorkingDir/path"

  change path to eliminate "", ".", ".." comp-
          onents using the textual manipulation
          algorithm to convert a static path to
          a canonical path (see the definition
          of a canonical path in Section 2.1).

  curPath ← "/"
  remove leading "/" from path
  loop forever
      if curPath is a symbolic link then
          readlink(referent)
          if referent is absolute then
              p ← referent
          else
              p ← "curPath/../referent"
          linkTrust ← trustpath(p, okUsers, badUsers)
          if linkTrust is not trusted then
              return UNTRUSTED
      else
          isEntryTrusted ← compute trust of curPath
                      as in TrustEntry of Figure 1,
                      except always trust sticky bit
                      directories as a trusted directory
          if isEntryTrusted is not TRUSTED then
              return UNTRUSTED
      if path is empty then
          return TRUSTED
      nextComp ← remove next component of path
      path ← "path/nextComp"
```

program to include the new user id as a trusted user id. This change is required because a group is trusted only if all of its members are trusted, so without the user id being trusted the group is not trusted. Also the computation of a group being trusted is inefficient as it entails scanning all or part of the password and group files for each directory entry visited.

Bishop's algorithm works by converting the path into a canonical path using the textual manipulation algorithm described in the canonical path definition of Section 2.1. There are two major problems with this approach. First, the textual manipulation algorithm can only be used on static paths, but the path could be a dynamic path and therefore produce the wrong results by checking a path different than what the operating system would check. Second, if the transformation were correct, the algorithm is then checking if the canonical path is trusted, but the canonical path being a trusted path has no implications on other paths to the same file system object, and therefore the algorithm might not be verifying the correct path.

The algorithm then pulls components off the path one-by-one, appends the component to the path string being created, and examines each path formed to check if it is trusted. Trust is determined as in the `TrustEntry` algorithm of Figure 1, except non-directory entries in a sticky bit directory are trusted.

If a symbolic link type is found while processing the path, a new path is formed based on the referent and `trustfile` is recursively called. There is no check to limit the amount of recursion in the event of a symbolic link loop and the algorithm goes into an infinite loop. In the case of a relative path referent, the new path is formed by concatenation of the path to the symbolic link, "`/../`", and the referent (the symbolic link name and `..` are removed with the textual manipulation performed in `trustfile`). An absolute path referent is used as-is for the new path. This technique could result in a path that exceeds the maximum length allowed for a path.

The best case run time complexity of this algorithm is $O(m\bar{n}2^s)$, where m is the number of components processed in the traversal, $\bar{n}$ is the average number of components of the path in the traversal, and s is the maximum number of symbolic links directly in the path or symbolic link referents encountered. The space complexity of this algorithm is $O(dl)$, where d is the maximum depth of recursive symbolic links encountered, and l is the maximum length of a symbolic link referent. l is limited in size to `PATH_MAX`.

The limitations of this algorithm include (1) creating and checking the canonical path instead of the actual path, which can test the wrong file system object if given a dynamic path, or can miss untrusted directories given a static path, (2) trusting paths that resolve to a non-directory file system object in a sticky bit directory, (3) not detecting symbolic link loops, and (4) possibly creating a path that is too large from an initial path of appropriate size.

This algorithm correctly processes the components one-by-one, but its use of textual manipulation causes it to be insecure.

In summary, if the path is not canonical, then this algorithm only satisfies only properties 1, 2 and 8 of Section 2.4, i.e. supports multiple trusted user and group ids, works on all types of file system objects, and is concurrent safe. If the path is canonical, then the textual manipulations have no effect and there are no symbolic links, then the algorithm additionally satisfies properties 9, 10 and 11, i.e. untrusted processes will not be able to change the directory referred to by the path, or create any directory entries in the path.

2.6. `safe_is_path_trusted_r` algorithm

Figure 4 presents our algorithm for checking if a path is trusted. It satisfies all the properties of Section 2.4 for all paths.

If the path argument is a relative path, the algorithm uses the concept from `safe_dir` of checking the trust of the current working directory by traversing from the current work-

Figure 4. `safe_is_path_trusted_r` algorithm.

```
function safe_is_path_trusted_r(path, u, g)
  — u is the trusted user list
  — g is the trusted group list
 if path is relative then
    curPath ← "."
    curStat ← lstat(curPath)
    curTrust ← TrustEntry(TRUSTED, curStat, u, g)
    repeat
       dirTrust ← TrustEntry(TRUSTED, curStat, u, g)
       if dirTrust is UNTRUSTED then
          return UNTRUSTED
       append(curPath, "/..")
       if length(curPath) > PATH_MAX then
          return ENAMETOOLONG error
       prevStat ← curStat
       curStat ← lstat(curPath)
    until curStat = prevStat   — at root directory
 else
    curTrust ← TRUSTED

 p ← path
 s ← empty stack
 curPath ← ""
 while p is not empty
    nextName ← RemoveNextComponent(p)
    if p is empty then
       if not stack_is_empty(s) then
          p ← pop(s)
    if nextName = "." or nextName is empty then
       restart loop
    prevPath ← curPath
    curPath ← PathRelativeTo(curPath, nextName)
    if curPath > PATH_MAX then
       return ENAMETOOLONG error
    curStat ← lstat(curPath)
    curTrust ← TrustEntry(curTrust, curStat, u, g)
    if curTrust is UNTRUSTED then
       return UNTRUSTED
    if curStat type is symbolic link then
       if num_elements(s) > SYMLOOP_MAX then
          return ELOOP error
       if p is not empty then
          push(s, p)
       p ← readlink(curPath)
       curPath ← prevPath  -
       else if p is not empty then
          if curStat type is not a directory then
             return ENOTDIR error
    return curTrust
```

ing directory to the root directory (the canonical path). It does this by checking the current directory "`.`", and then checks the paths"`./..`", "`./../..`", ... until the root directory is reached. If an untrusted directory is encountered, untrusted is immediately returned as the result. It does this traversal safely by only traversing using non-symbolic link directory entry names.

There are three operations that can be performed on a path: (1) check if the path is empty, (2) remove and return the next component (`RemoveNextComponent`), and (3) create a new path from a name relative to a given static path (`PathRelativeTo`). The `RemoveNextComponent` function behaves as expected, removing the leftmost name from the path (and the next directory separator if any), and returns the directory name. If the path was absolute then the root directory name "`/`", is removed and returned. `PathRelativeTo(baseDir, name)` returns a new path. If `baseDir` is empty or `name` is an absolute path, then the new path is `name`. Otherwise, the new path is the concatenation of `baseDir`, "`/`", and `name`. Since `baseDir` is a static path, some textual manipulations can be applied to simplify the new path.

The rest of `safe_is_path_trusted_r` revolves around four pieces of data: $curTrust$, the trust of the last directory entry; p, a path containing unchecked components of the path being processed; $curPath$, the path name to the current object to check; and s, a stack of paths, used to store the path being processed when a symbolic link is encountered. Initially $curTrust$ is set to the trust of the current working directory if the path is relative, and to `TRUSTED` otherwise, p is set to the path argument to the function, and the other two are empty.

The algorithm then processes directory entries one-by-one until p is entirely consumed. `RemoveNextComponent` is used to remove the next name from p, and name is then used to create the path of the next object to test using `PathRelativeTo` with $curPath$ as the directory. If p is now empty and the stack is not, then the top of the stack is popped into p.

The `lstat` system call is used to get information about the file system object pointed to by $curPath$ such as the type, owner, group, sticky bit and permissions of the file system object. This information then is used to determine the trust of the file system object using the `TrustEntry` algorithm. If it is not trusted, `UNTRUSTED` is immediately returned.

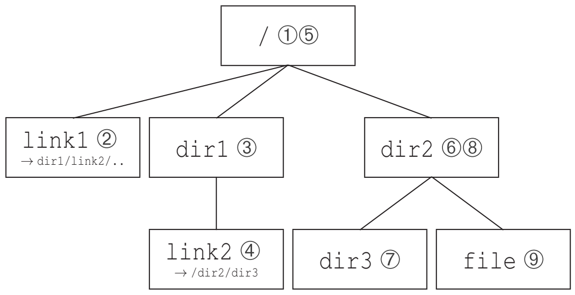
If the object is a symbolic link, then the referent must be processed before the rest of the path in p. If p is not empty, it is pushed on the stack to processed after the referent is read. The referent is assigned to p. If the depth of the stack exceeds some constant, then `ELOOP` is returned to indicate a symbolic link loop as would the operating system.

If an error is encountered, `-1` is returned and `errno` is set to the type of the error.

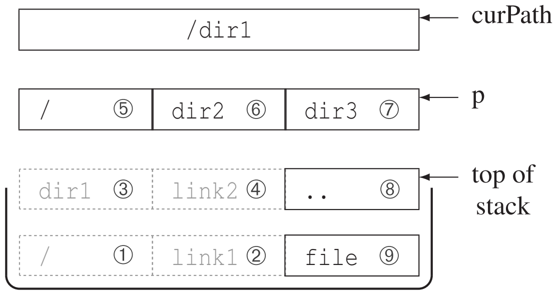
Figure 5 shows the internal data structures of the algorithm in the act of processing a path.

The run time complexity of this algorithm is $O(m\overline{n})$, where m is the number of components processed in the traversal and $\overline{n}$ is the average number of components of the path in the traversal. The $\overline{n}$ arises because the kernel performs a path traversal of each path passed to `lstat`. This cost is $O(n)$, where n is the number of components in the

Figure 5. File system traversal and algorithm operation while processing /link1/file in the directory structure shown. The numbers show the order of the traversal.



(a) Example file system structure showing directory entries visited while verifying the trust of /link1/file. /link1 and /dir1/link2 are symbolic links with referents of dir/link2/.. and /dir2/dir3 respectively.



(b) The state of the variables of the algorithm in Figure 4 immediately after processing link2. The grayed names 1–4 on the stack have already been been removed from the path, and show what each path was originally.

path.

The space complexity of this algorithm is $O(dl)$, where d is the maximum depth of recursive symbolic links encountered, and l is the maximum length of a symbolic link referent. Both of these values are bounded by constant values SYMLOOP_MAX and PATH_MAX.

In summary, this algorithm satisfies all the properties of Section 2.4 except 3, i.e. failure due to path length limitations. This is caused by the algorithm creating paths that may become too large due to the current working directory being too deep, or the contents of symbolic links causing the path to become too large. Both of these cases result in the ENAMETOOLONG error being returned. Without changing directories, it is not possible to satisfy property 3, but if the directory is changed then the concurrency property (8) cannot be satisfied. The next section will show how to perform these mutually exclusive properties so all the properties are met.

2.6.1. Overcoming path length limit. This section describes an algorithm that satisfies all the desired properties except 8, and how to combine this new algorithm with the above to satisfies all the properties desired.

This new algorithm, safe_is_path_trusted, is similar to the previous version, except it changes directories during processing, so the name returned by RemoveNextComponent is always in the current working directory (or is the root directory).

The check of the current working directory ancestry is done by calling open on "..", and then using fstat and fchdir to get the properties and change to the parent directory respectively.

In the main loop *curPath* is eliminated and *name* is used in the lstat and to change directory, which is done at the end of the loop replacing the test of *curStat* not being a directory (chdir will fail if it is not).

This algorithm satisfies all the properties except the concurrency property (8) as it changes the current working directory and depends on the current working directory of the process not changing during its execution. It does have better run time complexity of $O(m)$, where m is the number components processed in the traversal. Each component processed is done in constant time since all paths passed to lstat are either directly in the current working directory or are the root directory.

To satisfy all the properties, a third function was created, safe_is_path_trusted_fork. This algorithm forks a copy of the process (so the current working directory is no longer shared), calls safe_is_path_trusted, and returns the result back to the calling process. This algorithm satisfies all the requirements, but it does incur a time penalty to create the new process.

A solution that has better typical run time, but still satisfies all the properties is to modify safe_is_path_trusted_r to call safe_is_path_trusted_fork only if the error ENAMETOOLONG is generated. Since most paths are typically short, the cost of the fork will only be incurred when the path becomes too long.

2.6.2. Other access control schemes. The safe_is_path_trusted family of functions is designed to check the trust of a path in a POSIX system; it does not support systems that have an alternate permission semantics. These system can allow untrusted access even though from the POSIX mode, owner, and group permissions the file is trusted. Examples of such systems include AFS ACLs [1] and POSIX draft ACLs [9]. If these alternative access control schemes are used, the only likely change to code would be the definition of TrustEntry.

POSIX draft ACLs allow additional users and groups, each with their own set of permissions, but these additional users and groups permissions are only allowed to be as permissive as the standard group mode allows. In this case, if no trusted group ids are passed to the algorithm, the additional users and group ACLs will be effectively ignored, and the algorithm will return the correct result.

3. Safe Open

Opening and creating files in a POSIX environment is a common cause of security problems. This is caused by interfaces that are easy to use incorrectly, and in some cases have semantics that are impossible to use securely when opening files that are not a trusted path. Security problems arise because of the way these functions handle symbolic links and the way that permissions of newly created files are determined.

The rest of this section describes common types of problems when using the standard system calls to open and create files: `open`, `fopen`, and `creat`. We then present a set of replacement functions for these standard system calls that do not have the problems. We also describe a facility provided by these functions that notifies an application when the paths to files they are trying to open are being manipulated by a potential attacker.

Files are created in a POSIX environment using the `open` system call. This call takes a file name, a set of flags that controls the semantics, and an optional permissions value used when a file is created. Multiple flags can be combined by or-ing then together potentially allowing thousands of unique semantics through this one interface. The POSIX system call `creat` is equivalent to a call to `open` with a fixed set of flags consisting of `O_CREAT|O_WRONLY|O_TRUNC`, and exists for historical reasons. The use of `creat` should be replaced by `open`, and is not discussed further. The standard C [8] function `fopen` is also implemented using the `open` function, but is discussed separately as `fopen` cannot directly be replaced by `open`.

3.1. Problems with open and fopen

Some applications need to use an untrusted path to open existing files or to create files. The application may have to open or create files in the `/tmp` directory due to a need to inter-operate with other programs, or the application may be running with elevated privilege that needs to process files at a user specified path such as those files in the untrusted user's home directory. Without precautions, an untrusted process can manipulate components of the file name's path to get the application to create or open a file at an arbitrary location.

The untrusted process can attack the application by moving directories and changing symbolic links. Symbolic links in the directory portion of the file name can be avoided by changing the current working directory (the directory that relative paths are based) to the directory portion of the file name, and verifying that the current working directory is as expected using a function such as `getcwd`. The last component can then be used in the `open` call, but unfortunately detecting a symbolic link in the last component of the file name's path is more difficult.

The result of the `lstat` function is commonly used to determine if the application can safely proceed with the open by verifying properties of the file such as the existence, type (including regular file or symbolic link), owner and permissions. This is not safe because there can be a TOCTOU race condition between the `lstat` and the `open`.

A common approach to avoid this race condition is to open the file and use the `fstat` system call on the file descriptor to assure that a file with expected properties was opened. Unfortunately, using `fstat` cannot detect if the last component of the file name was a symbolic link as the open will always follow the symbolic link and open the referent of the symbolic link. Another problem is that the actions of the `open` call alone can cause security problems before the properties can be checked. The problem is caused by insecure use of two of flags to `open`: `O_CREAT` (create the file if it does not exist), and `O_TRUNC` (truncate the file to zero length).

We show how to combine these two techniques to correctly inspect the last component being a symbolic link while avoiding the race condition in the next section

A call of `open` with the `O_CREAT` flag and without `O_EXCL`, causes a file to be atomically created if the file does not exist or opened if the file does exist. If any of the path components are symbolic links, they are all followed including the final component of the path. If the file does not exist, and the final component is a symbolic link, the file is created at the path specified by the link. The manipulation of the symbolic link can then easily be used as an attack vector, if the process is running with elevated privilege, to create files anywhere the privilege allows.

The use of `O_CREAT` with `O_EXCL` changes the semantics of `open` to create a file if it does not exist, and to fail if the file already exists or if the final file name component is a symbolic link. When used together, the file is always created in the directory that is the file name with the final component removed. `O_CREAT` should always be used with `O_EXCL` as this combination guarantees that an attacker cannot use a symbolic link in the last component of the file name as an attack vector.

A call of `open` with the `O_TRUNC` flag truncates an existing file as part of opening the file. If the file name is an untrusted path, an untrusted process can modify the path to point to an arbitrary file and cause any file that the application's privilege allows to be truncated.

Another problem with `open` is that the function is a vari-

Table 1. Semantics of the two direct replacement functions for `open` based on the flags passed and the existence of the file name.

function	*flags include* O_CREAT	*flags include* O_EXCL	*file existence* exists	*file existence* absent
`int safe_open_wrapper(filename, flags, perms)`	no	-	open (fail if link)	fail
	yes	no	open (fail if link)	create
	yes	yes	fail	create
`int safe_open_wrapper_follow(filename, flags, perms)`	no	-	open (follow link)	fail
	yes	no	open (follow link)	create
	yes	yes	fail	create

adic function; only the file name and the flags are required, and the third parameter is optional and only required if a file is created. Compilers do not produce a warning if this value is missing when the flags contain `O_CREAT`. When a file is created in such a case, the initial permissions of the file are whatever happened to be next on the stack after the flags. This omission may result in too lenient permissions, exposing the contents to an attacker.

`fopen` uses a set of characters in a mode string instead of a set of flags. `fopen` internally calls `open`, but `O_CREAT` is always used without `O_EXCL`, so `fopen` is vulnerable to the symbolic link attacks described above when creating a file. The permissions of a newly created file are implicitly derived from the process's umask value (all the read and write permissions are enabled except those permissions that are included in the process's umask value). If different permissions are needed for different files, then the process's global umask needs to be changed. Modifying this global value can lead to a race condition if the process has multiple threads of control.

Viega and McGraw [11] present symbolic link attacks and show how to detect if the final component of file name is a symbolic link. They also show a safe replacement function, `safe_open_wplus`, for `fopen` with flags of "`wb+`". Their function provides only a direct replacement for two out of the twelve possible mode flags ("`w`" and "`wb+`") of `fopen`. This function is also susceptible to a cryogenic sleep attack (described in the next section), and can return an anomalous error if the file exists and is deleted during the execution of their function.

3.2. Desired Properties

Below is a set of properties that safe replacement functions for `open` and `fopen` should possess to make their use both secure and an easy replacement for the original function:

1. `O_CREAT` should never be used without `O_EXCL`, so a file is never created in an unexpected directory due to the last component being a symbolic link. The semantics of `O_CREAT` without `O_EXCL` is rarely the desired behavior and easily leads to attacks.

2. When an existing file is opened, the call should by default fail if the last component is a symbolic link. This behavior is the safe default, so files outside the expected directory are not opened. In some cases an application may be required to follow symbolic links so this option must also exist in the interface.

3. If the function can create a file, then a valid initial permission parameter is required. The initial permission is then explicit so the developer is more likely to use an appropriate value.

4. The replacement functions easily should be substituted for existing calls to `open` and `fopen`: the types and meaning of parameters, results and errors should match the original where possible, and all the original modes and flags should be supported.

5. The replacement functions should act as an atomic operation, just like original function. Since most of the replacement functions use the file name multiple times, each use can refer to a different file if the file system changes between uses. The function must detect if the results of multiple system calls in its implementation represent different file system objects. When inconsistent results are detected, the operations must be retried. Verifying and retrying inconsistent results of system calls prevents anomalous errors that would never happen with `open`, such as a `EEXIST` (file exists) or `ENOENT` (no such file) error when the flags are `O_CREAT` and `O_RDWR`.

3.3. Direct safe `open` replacements

We provide two direct replacement functions for `open`. These functions are `safe_open_wrapper` and `safe_open_wrapper_follow`, with their interface and characteristics shown in Table 1. Both of these functions differ from `open` in that both functions require the initial permissions of a newly created file, and fail if the last component

Table 2. Semantics of the six replacement functions for `open` based on the existence of the file name.

function	*file existence* exists	absent
`int safe_open_no_create(filename, flags)`	open (fail if link)	fail
`int safe_create_fail_if_exists(filename, flags, perms)`	fail	create
`int safe_create_keep_if_exists(filename, flags, perms)`	open (fail if link)	create
`int safe_create_replace_if_exists(filename, flags, perms)`	remove, then create	create
`int safe_open_no_create_follow(filename, flags)`	open (follow link)	fail
`int safe_create_keep_if_exists_follow(filename, flags, perms)`	open (follow link)	create

is a symbolic link when creating a file (the error `EEXIST` is returned instead). `safe_open_wrapper` also fails with the same error if the last component of the file name is a symbolic link.

Selecting between these two functions is application specific. `safe_open_wrapper_follow` changes the semantics the least and should be used in the general case. `safe_open_wrapper` should be used when the directory entry referred to by the file name should never be a symbolic link, i.e. the last component of the filename is not a symbolic link. `safe_open_wrapper` provides the property that the file system object referred to by the file name `/d1/.../dn/f` is contained in the directory `/d1/.../dn` if the open succeeds.

Modifying a program to make use of these functions requires a search for all the calls to `open` and replacing each of with a call to `safe_open_wrapper`. The only other change necessary would be to add the initial file permissions where missing. Making use of these functions eliminates the symbolic link security problems and missing initial file permissions of `open`.

These functions are implemented in terms of the advanced safe `open` replacement functions described in Section 3.5. `safe_open_wrapper` is simply implemented by calling the proper advanced replacement function based on the `O_CREAT` and `O_EXCL` flags as shown in Figure 6. `safe_open_wrapper_follow` is similarly written using the `_follow` version of the advanced replacement functions.

3.4. Path Manipulation Warning Facility

`safe_open_wrapper`, `safe_open_wrapper_follow`, the advanced safe `open` replacement functions (Section 3.5) and the safe `fopen` replacement functions (Section 3.6) all support an optional facility to notify the application in the event that any of the functions detect the file system object to which the file name refers has changed during the course of its operation.

This facility allows the application to register a function callback using the function, `safe_open_register_path_warning_callback`. The callback function is called with the file name once each time a manipulation of the path is detected. Under normal operations these events should be nonexistent because applications should be using unique file names and no other application should be manipulating the path of the file name. In a multiprocess application manipulating a common set of file names, this event may legitimately occur, although rarely. If the event occurs often, it is probably a sign of an active attack or a misbehaving application.

Figure 6. `safe_open_wrapper` algorithm.

```
function safe_open_wrapper(fn, flags, perms)
  if O_CREAT is in flags then
    if O_EXCL is in flags then
      f ← safe_create_fail_if_exists(fn, flags, perms)
    else
      f ← safe_create_keep_if_exists(fn, flags, perms)
  else
    f ← safe_open_no_create(fn, flags)
  return f
```

This facility is implemented in the advanced safe `open` functions. These form the basis for all other functions, so all the other functions inherit this facility. If any of the advanced safe `open` functions described in Section 3.5 have to retry their operation, then this is an indication of a path manipulation and the callback is called if registered.

3.5. Advanced safe `open` replacements

A set of six advanced safe replacement functions for `open` are shown in Table 2. The behavior of these functions varies depending on the existence of the file, and the last component being a symbolic link. All the functions fail if the last component of the file name is a symbolic link and the referent of the symbolic link does not exist. The functions `safe_open_no_create` and `safe_create_keep_if_exists` also fail if the last component is a symbolic link to an existing file. In this case `errno` is set to `EEXIST`. This error was chosen to match what `open` with flags of `O_CREAT` and `O_EXCL` returns when a symbolic link is the last component. A more logical error value would be `ESYMLINK`, but that error does not exist.

The implementation of the first four functions is pre-

Figure 7. Safe `open` replacement functions.

```
function safe_open_no_create(fn, flags)
  if flags contains O_CREAT or O_EXCL then
    return error EINVAL
  want_trunc ← O_TRUNC is in flags
  if want_trunc then
    remove O_TRUNC from flags
label TRY_AGAIN:
  f ← open(fn, flags)
  entryStat ← lstat(fn)
  if lstat failed and open failed then
    return error from lstat
  if lstat failed and open succeeded then
    close(f)
    goto TRY_AGAIN
  if entryStat type is a symbolic link then
    if f ≠ -1 then
      close(f)
    return error EEXIST
  if open failed with ENOENT then
    goto TRY_AGAIN
  if open failed then
    return error from open
  fdStat ← fstat(f)
  if entryStat and fdStat refer to different files then
    close(f)
    goto TRY_AGAIN
  if want_trunc and fdStat.size ≠ 0
          and f is not a tty and f is not a fifo then
    ftruncate(f, 0)
  return f

function safe_create_fail_if_exists(fn, flags, perms)
  add O_CREAT and O_EXCL to flags
  return open(fn, flags, perms)

function safe_create_keep_if_exists(fn, flags, perms)
  remove O_CREAT and O_EXCL from flags
  loop forever
    f ← safe_create_fail_if_exists(fn, flags, perms)
    if f ≠ -1 or errno is not EEXIST then
      return f
    f ← safe_open_no_create(fn, flags)
    if f ≠ -1 or errno is not ENOENT then
      return f

function safe_create_replace_if_exists(fn, flags, perms)
  loop forever
    unlink(fn)
    if unlink failed and errno is not ENOENT then
      return -1
    f ← safe_create_fail_if_exists(fn, flags, perms)
    if f ≠ -1 or errno is not EEXIST then
      return f
```

sented in Figure 7. Some error handling and the path manipulation detection (detected on retries) in these functions has been removed to simplify the presentation.

safe_open_no_create opens an existing file and fails if the file does not exist. An error occurs if `O_CREAT` is included in the flags. There are a few items of note in the implementation. First, if `O_TRUNC` is present in the flags, then `O_TRUNC` is removed and handled after the file is safely opened. The special handling of `O_TRUNC` is done so the file is not irreversibly truncated before the file name is verified to not be a symbolic link. Second, the use of `O_TRUNC` on a file type that is not a regular file is undefined except if the type is a tty or fifo, where POSIX says to ignore `O_TRUNC`. The undefined behavior matters in at least one important case: the use of `/dev/null` as the file name to our `fopen` replacement with a mode of "`w`". On some platforms if the flags are `O_CREAT|O_WRONLY|O_TRUNC`, `/dev/null` opens as expected, but if the flags are `O_WRONLY|O_TRUNC` `open` fails. We compensate for this problem by not performing the truncation unless the current size of the file is non-zero.

Another item of note is how `safe_open_no_create` checks for a symbolic link as the last component of the file name. A symbolic link can only be detected by using `lstat` function. If the `lstat` succeeds and the type of file system object is a symbolic link, then an error indicating the symbolic link, `EEXIST`, can be returned without using the results of the `open`.

The race condition between the `open` and the `lstat` is prevented by verifying that both system calls refer to the same file system object. `safe_open_no_create` checks if file system object of the `open` is the same as that referred to by the `lstat` by (1) opening the file, (2) obtaining the file properties for the file name using `lstat` (returns the file's properties without following a terminal symbolic link component), (3) obtaining the file properties for the opened file using `fstat`, and (4) comparing the immutable properties (the device, inode and type are fixed at file creation). If the immutable properties of the opened file and the file in the file system match, then the files are the same because the device and inode are unique in the file system and cannot be reused while a file is open, even if the file is removed.

The typical idiom for checking if the `lstat` and `open` refer to the same file is to perform the `lstat` first, then the `open` and finally the `fstat`. The purpose of checking the `lstat` result against the `fstat` result is to make sure the file opened was at one point in time at the location opened and that the filename was not a symbolic link. This approach is susceptible to what Kirch calls a cryogenic sleep attack [5]. The attack works by stopping the process after the `lstat`, but before the `open`, removing the file and waiting for a file to be created with the same device and inode, then creating a symbolic link to point to the file to attack and letting the process resume. The process does not detect that the file opened was not at the file name given and that the last component was a symbolic link. To prevent this attack the `open` must be performed first, as the device and inode cannot be reused until the file descriptor is closed, even if the directory entry is removed. The key to the solution is that the order of the `open` and the `lstat` does not matter and that reversing them from the typical order prevents the attack.

safe_open_no_create detects if open and the lstat are accessing different file and, if so tries again until a consistent view of file name is achieved. The function eventually completes as the attacker would have to always be able to replace file name between the open and the lstat for this function to never complete.

If the open fails, an lstat still needs to be performed to check if the file name is a symbolic link. If so, the value of errno is changed to EEXIST instead of the error from open.

safe_create_fail_if_exists fails if the file exists, otherwise it creates and opens the file. This is equivalent to calling open adding both O_CREAT and O_EXCL to the flags.

safe_create_keep_if_exists creates the file if it does not exist and safely opens the file if it does exist. The function is implemented using safe_open_no_create and safe_create_fail_if_exists. The algorithm first tries to create the file using safe_create_fail_if_exists, and returns if this call succeeds or had an error other than EEXIST. If the file exists, the function then tries to open the file using the function safe_open_no_create, and returns if this call succeeds or had an error other than ENOENT. These two functions are alternately tried until one of the conditions is met. The loop is required to prevent an anomalous error as an attacker could be actively creating and delete the file. Unless the attack can keep perfectly synchronized, one of the two functions should succeed.

safe_create_replace_if_exists always returns a freshly created file. If the file exists, the file is deleted and then created. This function also loops to prevent an attacker from causing a spurious failure by continually recreating the file. Unless the attacker can synchronize the file creation to always recreate the file between the unlink and the file creation call, the function eventually returns. This function is useful when an application is required to create a file with a particular file name in an untrusted directory such as the /tmp directory. If an existing file is opened, an attacker could potentially read and write to this existing file due to current or past unsafe ownership or permissions. The only way to guarantee strict permissions is to create the file in a trusted directory or for the process always create the file with proper permissions and to never open an existing file in such a directory.

safe_open_no_create_follow is equivalent to safe_open_no_create except the last component of the file name is allowed to be a symbolic link. The function treats O_TRUNC specially, but otherwise just calls open and returns if open fails. lstat is not called. The behavior of safe_open_no_create_follow is identical to open except for its special treatment of O_TRUNC, and that it will fail if the flags contain O_CREAT.

safe_create_keep_if_exists_follow is equivalent to safe_create_keep_if_exists except the last component of the file name is allowed to be a symbolic link when opening an existing file. The function is written the same way except the _follow version of safe_open_no_create is used.

Table 3. fopen mode to open flags mapping.

fopen mode	open flags
r	O_RDONLY
r+	O_RDWR
a	O_WRONLY | O_CREAT | O_APPEND
a+	O_RDWR | O_CREAT | O_APPEND
w	O_WRONLY | O_CREAT | O_TRUNC
w+	O_RDWR | O_CREAT | O_TRUNC

Modifying a program to make use of these functions, requires a search for all the calls to open and replacing each of with a call to the desired function. Using these advanced replacement function eliminates the same security problems as safe_open_wrapper, but allows the more knowledgeable developer additional control over the semantics of the open. The semantics required can be determined using the algorithm of safe_open_wrapper as shown in Figure 6, or any other function can chosen based on the desired semantics.

If the incorrect function is selected, the program could potentially fail due to the replacement function failing where open would not. The only potentially dangerous replacement would be to use open_create_replace_if_exists when this behavior was not desired as the function deletes and recreates the file, destroying the file's contents in the process.

3.6. fopen replacements

The set of replacement functions for fopen is the same as those functions shown in Table 2, except a "f" appears before "open" and "create" in the name, the flag parameter is replaced with the fopen mode string and a FILE* is returned instead of a file descriptor.

These functions work by calling a function that maps the fopen mode string into a set open of flags. The mapping for the standard mode strings is shown in Table 3 (the "b" flags is not shown and does not affect the mapping). Some platforms define additional mode characters and corresponding open flags. In this case, the mapping function can also map the these mode characters to the correct open flags and no other changes are required to support these platform specific settings.. The corresponding open replacement function is then called, and finally the file descriptor is converted to a FILE* object using the fdopen with the file descriptor and the original mode string. If an error occurs, then the file descriptor is closed if open, and NULL is returned to indicate an error. A safe_fopen_wrapper can also be created in a similar fashion.

The functions have the same benefits as the open replacements. They also require the permissions be passed on each call instead of being determined from the global umask, and thus is more expressive. For instance, a file can now be opened for writing without creating the file if the file does not exist.

4. Conclusion

We presented working solutions that if used will improve the security of software. The first problem solved is that of determining if a path is vulnerable to attack from untrusted processes. This function can test any path that the operating system can process and supports multiple trusted user and group ids.

The second problem solved is a general replacement for the POSIX and Standard C functions `open` and `fopen` that open and create files. The solution prevents common symbolic link attacks using an almost identical interface, so replacement should be a simple matter of find and replace. An alternative set of functions for the knowledgeable developer is also provided that has a more expressive set of behaviors.

A working implementation of these functions in source form is available at `http://www.cs.wisc.edu/~kupsch/safefile`.

5. Acknowledgment

This research funded in part by NATO grant CLG 983049, the National Science Foundation under contract with San Diego Supercomputing Center, and National Science Foundation grants CNS-0627501 and CNS-0716460.

References

[1] *AFS Administration Guide*. IBM, 2000. `http://www.openafs.org/doc/index.htm`.

[2] M. Bishop. How attackers break programs, and how to write programs more securely. `http://nob.cs.ucdavis.edu/bishop/secprog/sans2002.pdf`, 2002.

[3] Cve-2006-5215. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-5215`, 2006. Xsession `/tmp` Symbolic Link Race Condition.

[4] Cve-2007-4270. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-4270`, 2007. DB2 `/tmp` Symbolic Link Race Condition.

[5] M. Dowd, J. McDonald, and J. Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley, 2007.

[6] N. Ferguson and B. Schneier. *Practical Cryptography*. Wiley, 2003.

[7] Globus bugzilla bug 4648. `http://bugzilla.globus.org/globus/show_bug.cgi?id=4648`, 2006.

[8] S. P. Harbison, III and G. L. Steele, Jr. *C: A Reference Manual, 5th edition*. Prentice Hall, 2002.

[9] *IEEE 1003.1e Draft 17: Draft Standard for Information Technology - Portable Operating System Interface (POSIX) - System Application Program Interface*. 1997. `http://xt.pilot.org/publications/posix.1e`.

[10] *The Single UNIX Specification Version 3*. The Open Group, 2004. `http://www.opengroup.org/bookstore/catalog/t041.html`.

[11] J. Viega and G. McGraw. *Building Secure Software*. Addison-Wesley, 2002.