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 can 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 against opening 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 [4, 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 [5], (3) inadvertently following symbolic links allowing the creation or modification of files in unexpected locations as in IBM's DB2 [3] and Xsession [2], and (4) weak file permissions resulting from incorrect use of the API.

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. The second part (Section 3) describes functions that are safe replacements for the standard library functions for opening and creating files. 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`. Each time a path is presented to the operating system, it traverses the file system directory by directory to find the file system object referred to by the path. 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 [7] environment as found in operating systems such as UNIX and Linux. 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, and those of the process). Of the capabilities in each set, the paper is

This research funded in part by National Science Foundation under subcontract with San Diego Supercomputer Center and NATO CLG-983049.

concerned with the write flag as it controls modifications to the object.

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*. The *root* user id is trusted, since file access for a process running with the *root* user id is always granted. Other trusted ids are determined by the applications requirements, and may also vary from file to file. For instance, an application may use a trusted id list of *root* for its configuration file, and *root* and *app-user* for its data files.

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 the types of exploits possible if a path is not trusted, how to check if a path is 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 satisfies all the desired properties.

2.1. 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 - if the permissions on the file itself are untrusted

Denial of service - remove a directory entry in an untrusted directory along the path traversal to make the file system object inaccessible through this path.

Replace directory entry - remove or rename a directory entry in an untrusted directory along the path traversal, and insert a new directory or file in its place. This allows an attack to control the permissions and contents of file system objects even though the attacker might not be able to modify the original object referred to by the path.

Symbolic link manipulation - create a symbolic link in an untrusted directory along the path traversal to make the path refer to any arbitrary file system object to be opened or possibly created.

Hard link attack - is an attack where it appears 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 is indistinguishable from the original file and any changes made to one are 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.

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

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

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.

The trust of a directory entry can be determined recursively using the trust of the parent directory and by the properties of the file system object to which the directory entry points. The relevant properties consists of the object's type, owning user and group ids, permissions and sticky bit (relevant for directories). If the sticky bit is set on a directory, only the directory's owner, or the directory entry's owner can remove entries in such a directory.

The trust of a directory entry is determined by the first matching condition: (1) if the parent directory is not trusted, then the directory entry is untrusted as an attacker can manipulate the untrusted ancestor, (2) if the parent directory is sticky dir trusted and the directory entry is not a directory, then the directory entry is untrusted (due to hard link attacks), (3) if the type is a directory, the owner is trusted, and the sticky bit is set, then the directory entry is sticky dir trusted, (4) the directory is trusted, so if the permissions on the directory entry prevent untrusted processes from modifying its contents, the directory entry is trusted, otherwise (5) the directory entry is untrusted.

A *trusted path* is a path where all the components of the path and components of symbolic link referents (a path name pointer to another location in the file system) 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 (the direct path from the root directory to the given 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.

2.3. Properties

An algorithm to check if a path is trusted should have the following properties: (1) supports multiple trusted user and groups ids, (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 and detects symbolic link loops, (5) works properly with sticky bit directories, (6) 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), (7) is concurrent execution safe, and (8) 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 refer to the same file system object), (b) modify the object's contents, and (c) an untrusted process cannot create, rename or delete a directory entry owned by a trusted user and group id.

2.4. Prior Work

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

2.4.1. `safe_dir` algorithm. In the book *Building Secure Software* [8, pages 222–225], John Viega and Gary McGraw present an algorithm, `safe_dir`, to check if a directory is trusted. This algorithm only checks directories, and only allows a single trusted user id.

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* or supplied trusted user id, and the group and other are not allowed to write to the directory). Next, the algorithm performs the same test on each directory from the supplied directory to the root directory, "`/`". It works its way up the directory tree 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, "`/`".

This algorithm would be correct if the path was a canonical path. Unfortunately, `safe_dir` only checks if the last component of the path is a symbolic link.

There is a TOCTOU race condition in this function, not in the algorithm itself but in the interface provided. 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 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 path.

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 limitations of this algorithm include (1) not checking the path, but the canonical path to the directory, (2) only supporting a single trusted user id and *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,

If the path is not a canonical path, this algorithm satisfies none of the desired properties of Section 2.3.

2.4.2. `trustfile` algorithm. Matt Bishop [1, pages 300–307] presented an algorithm, `trustfile`, that checks 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 supplied user id for each directory encountered in the path. While the approach of computing the trusted group ids is correct, it is both inefficient, and it increases the operational overhead. Adding a new user to the system may require updating the applications list of trusted user ids.

Bishop's algorithm works by first textually manipulating the path into a path without components referring to the current directory, "`.`" (just removed), and the parent directory, "`..`" (remove it and the preceding directory component). This transformation appears correct but the transformation can result in a path that is not equivalent in the case of a symbolic link preceding a parent directory such as `/tmp/symlink/../file`. The algorithm would test `/tmp/file`, but the actual file could be anywhere. If the path contains a symbolic link the operating system and algorithm will access different file system objects.

The algorithm correctly processes the components in the path one-by-one to check if it trusted using the criteria of Section 2.2, 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 (exactly the troublesome case described previously). 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 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.

If the path is not canonical, then this algorithm only satisfies only properties 1, 2 and 7 of Section 2.3, i.e. supports multiple trusted user ids, works on all types of file system objects, and is concurrent safe.

2.5. `safe_is_path_trusted_r` algorithm

Figure 1 presents our algorithm for checking if a path is trusted. Figure 2 shows the internal state of the algorithm while processing a path. It satisfies all the properties of Section 2.3 for all paths.

The algorithm works by using the techniques of Section 2.2. If the path is a relative path, then if first checks the trust of the current working directory. Next it processes the components of the path one-by-one until they are all consumed. A path is formed using the functions RemoveNextComponent and PathRelative. RemoveNextComponent removes and returns the next component of the path to process (if the path was absolute, it returns "/" for the initial call). Similarly PathRelativeTo returns the concatenation of the two paths (if the new path is "/", "/" is returned).

Each directory entry is tested for trust using the definition of directory entry trust in Section 2.2. If an untrusted directory entry is encountered, untrusted is immediately returned. Otherwise the trust value when the entire path is consumed is returned.

If a symbolic link is encountered the current path is pushed on a stack and the referent is then processed. Loops are detected by the depth of the stack exceeding `SYMLOOP_MAX`.

This algorithm satisfies all the properties of Section 2.3 except 3, i.e. failure due to path length limitations. This is caused by the algorithm creating paths that may become to large due to the current working directory being too deep, or the contents of symbolic links causing the path to become too large. 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 we show how to perform these mutually exclusive properties so all the properties are met.

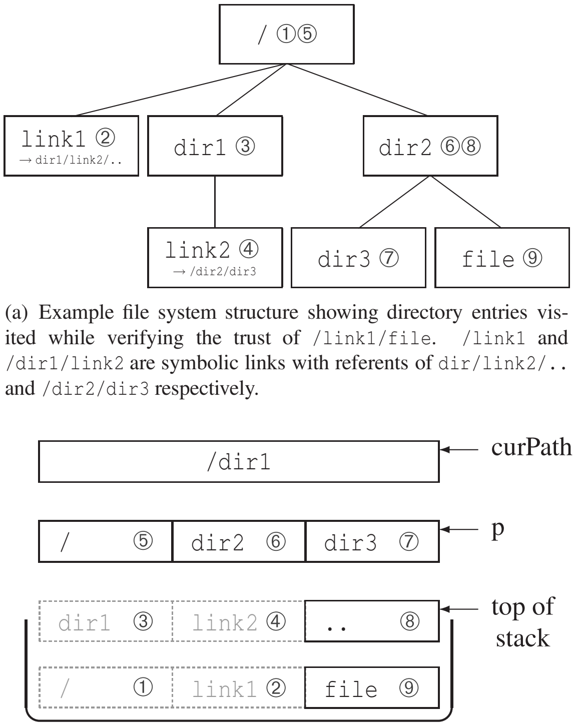
`safe_is_path_trusted_r` can detect when the path becomes too long and can call `safe_is_path_trusted_fork`. This function sets up a communication channel to a forked process that can safely change the working directory with without a concurrency problem.

This new algorithm, `safe_is_path_trusted`, is similar to the `safe_is_path_trusted_r`, except it changes directories during processing instead of concatenating a path name together, so the name returned by `RemoveNextComponent` is always in the current working directory (or is the root directory).

Figure 1. `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 safe_path_is_trusted_fork(path, u, g)
     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(prevPath, nextName)
   if curPath > PATH_MAX then
     return safe_path_is_trusted_fork(path, u, g)
   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
```

Figure 2. 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 1 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.

Combined these algorithms satisfy all the requirements and only have the overhead of a fork when needed.

3. Safe Open

Opening and creating files in a POSIX environment is a common cause of security problems, caused by interfaces that are easy to use incorrectly and, in some cases, have semantics that are difficult to use securely for 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.

We first describe common types of problems when using the standard system calls to open and create files: `open`, `creat`, and `fopen`. We then present a set of replacement functions for these standard system call 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. `creat` exists for historical reasons and can be replaced with a call to `open`, and will not be discussed further. The standard C [6] function `fopen` 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, or needs to process files in an untrusted user's home directory. Without precautions, an untrusted process can manipulate components of the path to get the application to create or open a file at an arbitrary location as described in Section 2.1.

Symbolic links in the directory portion of the file name can be avoided by changing the current working directory to the directory portion of the file name, and verifying that the current working directory satisfies the security requirements. Detecting a symbolic link in the last component of the file name 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 is 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. 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 combine these two techniques to check if the last component is 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 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 variadic function; the initial file permissions is not required by the compiler, but the function uses the value when creating a file. 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 [8] 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

Replacement functions for open and fopen should have the following properties to make them use both secure and easy to use:

(1) O_CREAT should never be used without O_EXCL, to prevent a file being created in an arbitrary location if the last component is a symbolic link, (2) when an existing file is opened, the call should by default fail if the last component is a symbolic link to avoid opening a file in an arbitrary location, (3) the initial file permissions should be required and explicit whenever a function can create a file, (4) the replacement functions easily should be substituted for existing calls to open and fopen, and (5) replacement functions should operate as if they were atomic, just like the original calls.

3.3. Direct safe open replacements

We provide two direct replacement functions for open, safe_open_wrapper and safe_open_wrapper_follow.

Figure 3. 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
      $f \leftarrow$ safe_create_fail_if_exists(fn, flags, perms)
    else
      $f \leftarrow$ safe_create_keep_if_exists(fn, flags, perms)
  else
    $f \leftarrow$ safe_open_no_create(fn, flags)
  return f
```

They behave the same as open except they require the initial permissions to be present, and fail if the last component is a symbolic link when creating a file. 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/.../d*n*/f is contained in the directory /d1/.../d*n* if the open succeeds.

These functions are implemented in terms of the advanced safe open replacement functions described in Section 3.5. safe_open_wrapper is implemented by calling the proper advanced replacement function based on the O_CREAT and O_EXCL flags as shown in Figure 3. safe_open_wrapper_follow is similarly written using the _follow version of the advanced replacement functions.

3.4. Path Manipulation Warning Facility

All the safe open and fopen replacement functions support an optional facility to notify the application if they 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 that will be called each time a manipulation of the path is detected. Under normal operations this should not happen because applications should be using unique file names and no other application should be manipulating the path of the file name. If the event occurs often it is likely a sign of an active attack or a misbehaving application.

3.5. Advanced safe open replacements

Figure 4. 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 $\leftarrow$ O_TRUNC is in flags
  if want_trunc then
    remove O_TRUNC from flags
label TRY_AGAIN:
  $f \leftarrow$ open(fn, flags)
  entryStat $\leftarrow$ 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 $f \neq$ -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 $\leftarrow$ fstat(f)
  if entryStat and fdStat refer to different files then
    close(f)
    goto TRY_AGAIN
  if want_trunc and fdStat.size $\neq$ 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
    $f \leftarrow$ safe_create_fail_if_exists(fn, flags, perms)
    if $f \neq$ -1 or errno is not EEXIST then
      return f
    $f \leftarrow$ safe_open_no_create(fn, flags)
    if $f \neq$ -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
    $f \leftarrow$ safe_create_fail_if_exists(fn, flags, perms)
    if $f \neq$ -1 or errno is not EEXIST then
      return f
```

There are six advanced safe replacement functions for open. Their operation depends on the whether the file exists and whether the last component is a symbolic link. They all 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.

The implementation of four of the functions is presented in Figure 4. Some error handling and the path manipulation detection (detected on retries) in these functions has been omitted 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. `O_TRUNC` is handled specially to prevent the wrong file from being truncated, and to work around an undefined behavior with `O_TRUNC` when the file name is a device file.

`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 the `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`, is 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. If the immutable properties (device, inode and type are fixed at creation) of the opened file and the file in the file system match, then the files are the same.

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`. This approach is susceptible to what Kirch calls a cryogenic sleep attack [4]. The attacker stops the process after the `lstat`, but before the `open`, waits for the file to be removed, then waits until a file with the same device and inode can be created, and finally allows the process to 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.

`safe_open_no_create` detects if `open` and the `lstat` accessed different files and tries again until the functions access the same file. 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.

safe_create_fail_if_exists opens the file if it exists and fails if it does not.

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 by alternating between `safe_open_no_create` and `safe_create_fail_if_exists` until one of them succeeds or fails with an error that does not indicate the other function should succeed.

safe_create_replace_if_exists always returns a freshly created file. If the file exists, the file is deleted and then created. The function is implemented by alternating between the `unlink` and `safe_create_fail_if_exists` until the create succeeds or an error occurs that does not indicate the function should succeed. This function is useful when an application needs to create a temporary file in a directory such as `/tmp` and does not care about the previous contents.

safe_open_no_create_follow and **safe_open_keep_if_exists_follow** are equivalent to `safe_open_no_create` and `safe_create_keep_if_exists` respectively except the last component of the file name is allowed to be a symbolic link.

3.6. `fopen` replacements

The replacement functions for `fopen` are the same as those for `open` 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. The behavior of these function is similar to their `open` counterpart.

These functions are implemented by converting the mode string to an appropriate set of `open` flags, calling the corresponding safe `open` replacement function and converting the returned file descriptor to stream using `fdopen`.

The functions have the all the benefits of 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

The use of the functions presented will eliminate common types of attacks, and improve the security of software. An implementation of the functions are available in source form at `http://www.cs.wisc.edu/~kupsch/safefile`.

References

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

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

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

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

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

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

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

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