NAME

safe_is_path_trusted_fork – check the trust of a path

SYNOPSIS

#include "safe_is_path_trusted.h"

int safe_is_path_trusted_fork(const char **pathname***, struct safe_id_range_list ****trusted_uids***, struct safe_id_range_list ****trusted_gids***);**

DESCRIPTION

safe_is_path_trusted_r(3) should be used in most cases instead of **safe_is_path_trusted_fork**.

safe_is_path_trusted_fork checks if the path name given by *pathname* can only be modified by the list of trusted users and groups. The type of modifications include the file system object itself, and which file system object the path refers. The check is done by forking a process, the child process calling **safe_is_path_trusted**(3), and returning the result to the parent. This function is called internally by **safe_is_path_trusted_r**(3) as needed, and the results are identical.

The function returns the type of trust that the path provides based on the trusted users and groups. The specific types of trust for a path are detailed in the return value section.

An error of **EINVAL** is returned if any of the parameters are **NULL**.

Parameters

pathname
: The path name to verify.

trusted_uids
: The set of user ids that are considered trusted. The *root* (0) user account is implicitly trusted.

trusted_gids
: The set of group ids that are considered trusted. No groups are implicitly trusted.

Algorithm

safe_is_path_trusted_fork works by walking the path from the first directory entry in the path to the last. Each directory entry is checked to verified to make sure that only the trusted users and groups and modify (write) the file system object referred to by the directory entry.

If a directory entry is not trusted **SAFE_PATH_UNTRUSTED** is immediately returned. If an error occurs **SAFE_PATH_ERROR** is returned.

The trust of the entire path is the trust level of the last component of the path if all the prior directory components are trusted and without error.

Relative paths, symbolic links, directories with the sticky bit set and the determination of a path being confidential are described below.

Relative Paths

If the *pathname* is a relative path, the trust of all the directories from the current working directory up to the root directory are checked to verify that each directory is trusted. If any of the directories are not trusted **SAFE_PATH_UNTRUSTED** is returned.

This is done because the process must use a set of paths to arrive at the current working directory. If one of the directories from the current working directory to the root is not trusted, then their is no path the process could use that is trusted to get to the current working directory, so any path relative to the current working directory cannot be trusted.

Sticky Bit Directories

Sticky bit directories (see **chmod**(2)) are treated specially. If a directory is not otherwise trusted, but has the sticky bit set, has an owner that is in the set of trusted users, then the directory is sticky dir trusted. The value **SAFE_PATH_TRUSTED_STICKY_DIR** is returned if the last component is sticky dir trusted.

If the parent directory of a directory entry is sticky dir trusted, and the directory entry type is not a directory, then the path is not trusted. This includes symbolic links and regular files in a sticky dir trusted directory. This is because an untrusted user can create a hard link in the sticky bit directory to make a file appear as if a trusted user created the file.

Symbolic Links

If a symbolic link is encountered while checking the directory components, the referent of the symbolic link is read and the trust of the referent is recursively checked. If the depth of a recursion exceeds **SYMLOOP_MAX** (or 32 if not defined), **ELOOP** is returned as do all system calls that take a path parameter.

Confidential Trust

The value **SAFE_PATH_TRUSTED_CONFIDENTIAL** is returned if in addition to the path being trusted, the final directory entry has permissions such that only the trusted users and groups can read the file system object. For directories, both the ability to read the directory entries (read permission) and to access the directory entries (execute permission) are allowed by only trusted users and groups.

Path Exceeds PATH_MAX

This function can handle paths of arbitrary length directly.

Thread Safety

This function is thread safe.

RETURN VALUE

The function returns the type of trust for the given path. The types of trust form a hierarchy. If a type of trust is needed, then a path with a higher level of trust will also be suitable for the same purpose, while any path with a lower trust will not be suitable.

The constant for each trust level is given with the numeric value given in parenthesis.

SAFE_PATH_ERROR (-1)

An error has occurred while checking the path. *errno* will be set to the type of error.

SAFE_PATH_UNTRUSTED (0)

The path can be modified by a user or group that is not in the set of trusted users and groups.

SAFE_PATH_TRUSTED_STICKY_DIR (1)

The path is trusted up to the final component of the path. The final component of the path is directory owned by a trusted user, allows untrusted user and groups to create entries in the directory, and has the stick bit set. This type of directory is typically used for the /tmp directory.

This type of directory is safe to use for two types of operations: (1) create a secure directory, or (2) create a temporary file (one that has secure permissions and is only accessed through the file descriptor returned from the create function).

SAFE_PATH_TRUSTED (2)

The path can only be modified (written) by trusted users or groups.

SAFE_PATH_TRUSTED_CONFIDENTIAL (3)

The path can only be modified (written) and accessed (read) by trusted users and groups. The confidentiality property is only based on the last component of the path. An alternative would be to treat anything contained in a directory that is confidential to be confidential, but hard links could subvert this, so the conservative approach was chosen.

ERRORS

The value of *errno* can be set to any values that the following functions set: **open**(2), **close**(2), **pipe**(2), **read**(2), **write**(2), **fstat**(2), **lstat**(2), **fchdir**(2), **readlink**(2), **strdup**(3), **malloc**(3), **sigfillset**(3), **sigprocmask**(2), **sigaddset**(3), **fork**(2), and **waitpid**(2).

In addition *errno* can be set to the following errors:

EAGAIN

If when processing a symbolic link along the path, the symbolic link is repeatedly replaced with a new symbolic link whose referents are longer than the previous referent. This should be a very rare event and can only occur if a trusted user or group is constantly replacing a symbolic link with a new symbolic link (if a symbolic link can be replaced by an untrusted user or group the function will return that the path is untrusted before reading the referent of the symbolic link). Retrying the call may succeed.

EINVAL
One of the parameters was NULL.

EXAMPLES

The following example provides a command line interface to **safe_is_path_trusted_fork**. The typical usage pattern of **safe_is_path_trusted_fork** would be to use **>=** to test if a path is trusted enough for an operation or to use **<** to test is a path is not trusted enough.

```
#include "safe_id_range_list.h"
#include "safe_is_path_trusted.h"
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    char *path;
    int status;
    struct safe_id_range_list safe_uids;
    struct safe_id_range_list safe_gids;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <path>\n", argv[0]);
        exit(1);
    }

    /* init id list structs */
    if (safe_init_id_range_list(&safe_uids) == -1) {
        perror("safe_init_id_range_list(&safe_uids): ");
        exit(2);
    }
    if (safe_init_id_range_list(&safe_gids) == -1) {
        perror("safe_init_id_range_list(&safe_gids): ");
        exit(3);
    }

    /* add current user's uid and gid to safe lists */
    if (safe_add_id_to_list(&safe_uids, getuid()) == -1) {
        perror("safe_add_id_to_list(&safe_uids, getuid()): ");
        exit(4);
    }
    if (safe_add_id_to_list(&safe_gids, getgid()) == -1) {
        perror("safe_add_id_to_list(&safe_gids, getgid()): ");
        exit(5);
    }

    path = argv[1];
    status = safe_is_path_trusted_fork(path, &safe_uids, &safe_gids);

    safe_destroy_id_range_list(&safe_uids);
    safe_destroy_id_range_list(&safe_gids);

    /* print exact type of trust */
```

```
    switch (status) {
      case SAFE_PATH_ERROR:
        perror("ERROR: safe_is_path_trusted_fork: ");
        exit(6);
      case SAFE_PATH_UNTRUSTED:
        printf("%s is not trusted.\n", path);
        exit(7);
      case SAFE_PATH_TRUSTED_STICKY_DIR:
        printf("%s is a trusted sticky-bit directory.\n", path);
        break;
      case SAFE_PATH_TRUSTED:
        printf("%s is trusted.\n", path);
        break;
      case SAFE_PATH_TRUSTED_CONFIDENTIAL:
        printf("%s is trusted and confidential.\n", path);
        break;
      default:
        printf("ERROR: unknown status: %d\n", status);
        exit(8);
    }

    printf("\nPermissable operations are:\n");
    if (status >= SAFE_PATH_TRUSTED_STICKY_DIR) {
        printf("\n");
        printf(" - Create a trusted directory in this directory.\n");
        printf("\n");
        printf(" - Create a true temporary storage file, using\n");
        printf("   mkstemp() followed by unlink() (the path can\n");
        printf("   only be used by these two functions).\n");
    }
    if (status >= SAFE_PATH_TRUSTED) {
        printf("\n");
        printf(" - Assume the file can not be modified, or directory\n");
        printf("   entries can not be be created by untrusted\n");
        printf("   users or groups.\n");
        printf("\n");
        printf(" - Create files (or non-directory objects) in this\n");
        printf("   directory, that can be trusted.\n");
        printf("\n");
        printf(" - Use the path repeatedly without the possibility\n");
        printf("   of an untrusted users or groups creating an\n");
        printf("   exploitable race condition.\n");
    }
    if (status >= SAFE_PATH_TRUSTED_CONFIDENTIAL) {
        printf("\n");
        printf(" - Assume the file's contents or directory's entries\n");
        printf("   can not be read by untrusted users or groups.\n");
    }

    return 0;
}
```

AUTHOR

James A. Kupsch

SEE ALSO

safe_is_path_trusted_r(3) **safe_is_path_trusted**(3) **safe_init_id_range_list**(3) **safe_destroy_id_range_list**(3) **safe_parse_uid_list**(3) **safe_parse_gid_list**(3)

"How to Open a File and Not Get Hacked", *2008 Third International Conference on Availability, Reliability and* Security (ARES)," Barcelona, Spain, March 2008, http://research.cs.wisc.edu/mist/safefile