(see the previously defined destructor of the ClientInfo class for more information). The default trust level assigned by the XML-RPC server to unknown clients is also overwritten with a non-sense value.

4.2.9 Known Uses

Secure XML-RPC Server Library

Other uses of this pattern probably exist. Find them.

4.3 Secure Directory

4.3.1 Intent

The intent of the Secure Directory pattern is to ensure that an attacker cannot manipulate the files used by a program during the execution of the program. See “FIO15-C. Ensure that file operations are performed in a secure directory” in *The CERT C Secure Coding Standard* [Seacord 2008] for additional information regarding this issue.

4.3.2 Motivation

A program may depend on a file for some length of time during program execution. The program developers usually assume that the files used by the program will not be manipulated by outside users during the execution of the program. However, if this assumption is false, a file may be modified by multiple users, which means that a malicious user may modify or delete the file during a critical time when the program relies on the file remaining unmodified, causing a race condition in the program.

The Secure Directory pattern ensures that the directories in which the files used by the program are stored can only be written (and possibly read) by the user of the program.

4.3.3 Applicability

The Secure Directory pattern is applicable for use in a program if

- The program will be run in an insecure environment; that is, an environment where malicious users could gain access to the file system used by the program.
- The program reads and/or writes files.
- Program execution could be negatively affected if the files read or written by the program were modified by an outside user while the program was running.

4.3.4 Structure

Programmatically, the structure of the Secure Directory pattern is fairly simple. Prior to opening a file for reading or writing, the Secure Directory pattern states that the program must

1. find the canonical pathname of the directory of the file (see Section 4.4, “Pathname Canonicalization”)
2. check to see if the directory, as referenced by the canonical pathname, is secure

The structure of the secure directory is such that the directory has write permissions limited to the user and the superuser. No other users may modify files in the secure directory. Furthermore, all directories that appear before the directory of interest must prevent other users from renaming or deleting the secure directory.

4.3.5 Participants

The participants in the Secure Directory pattern are

- the program reading and writing the file
- the file system

4.3.6 Consequences

- Secure Directory reduces the possibility of race conditions occurring between programs controlled by different users. Race conditions involving a secure directory may be produced only by multiple programs under the control of the user.
- The program speed will be degraded due to the canonicalization of pathnames and the checking for secure directories. To reduce the overhead of checking for secure directories, it is possible to cache the result of checking the security of a particular directory. Note that the caching of secure directory results assumes that the permissions of directories used by the program are not changed during program execution.

4.3.7 Implementation

Unless a program is run with root privileges, it does not have the ability to create secure directories. Therefore, the program should check that a directory offered to it is secure, and refuse to use it otherwise. As discussed in the Structure section, the basic implementation of the Secure Directory pattern involves the following steps:

1. Find the canonical pathname of the directory of the file to be read or written. (See Section 4.4, "Pathname Canonicalization.")
2. Check to see if the directory, as referenced by the canonical pathname, is secure.
   - If the directory is secure, read or write the file.
   - If the directory is not secure, issue an error and do not read or write the file.

4.3.8 Sample Code

The sample code provided in this section was taken directly from "FIO15-C. Ensure that file operations are performed in a secure directory" in *The CERT C Secure Coding Standard* [Seacord 2008].

Under a POSIX-compliant OS, a function to check a directory to see if it is secure may be implemented as follows:

```
#include <stdlib.h>
#include <unistd.h>
#include <limits.h>
#include <libgen.h>
#include <sys/stat.h>
#include <string.h>

/* Returns nonzero if directory is secure, zero otherwise */
int secure_dir(const char *fullpath) {
  char *path_copy = NULL;
  char *dirname_res = NULL;
  char **dirs = NULL;
  int num_of_dirs = 0;
  int secure = 1;
  int i;
  struct stat buf;
  uid_t my_uid = geteuid();

  if (!(path_copy = strdup(fullpath))) {
    /* Handle error */
  }

  dirname_res = path_copy;
  /* Figure out how far it is to the root */
  while (1) {
    dirname_res = dirname(dirname_res);

    num_of_dirs++;

    if ((strcmp(dirname_res, "/") == 0) ||
   (strcmp(dirname_res, "//") == 0)) {
      break;
    }
  }
  free(path_copy);
  path_copy = NULL;

  /* Now allocate and fill the dirs array */
  if (!(dirs = (char **)malloc(num_of_dirs*sizeof(*dirs)))) {
    /* Handle error */
  }
  if (!(dirs[num_of_dirs - 1] = strdup(fullpath))) {
    /* Handle error */
  }

  if (!(path_copy = strdup(fullpath))) {
    /* Handle error */
  }

  dirname_res = path_copy;
  for (i = 1; i < num_of_dirs; i++) {
    dirname_res = dirname(dirname_res);

    dirs[num_of_dirs - i - 1] = strdup(dirname_res);

  }
  free(path_copy);
  path_copy = NULL;

  /* Traverse from the root to the leaf, checking
   * permissions along the way */
  for (i = 0; i < num_of_dirs; i++) {
    if (stat(dirs[i], &buf) != 0) {
       /* Handle error */
    }
    if ((buf.st_uid != my_uid) && (buf.st_uid != 0)) {
      /* Directory is owned by someone besides user or root */
      secure = 0;
    } else if ((buf.st_mode & (S_IWGRP | S_IWOTH))
      && ((i == num_of_dirs - 1) || !(buf.st_mode & S_ISVTX))) {
        /* Others have permissions to the leaf directory
         * or are able to delete or rename files along the way */
        secure = 0;
     }

    free(dirs[i]);
    dirs[i] = NULL;
  }

  free(dirs);
  dirs = NULL;

  return secure;
}
```

Given the `secure_dir()` function, the Secure Directory pattern may be implemented in C as follows:

```
char *dir_name;
char *canonical_dir_name;
const char *file_name = "passwd"; /* filename within the secure directory */
FILE *fp;

/* initialize dir_name */

canonical_dir_name = realpath(dir_name, NULL);
if (canonical_dir_name == NULL) {
  /* Handle error */
}

if (!secure_dir(canonical_dir_name)) {
  /* Handle error */
}

if (chdir(canonical_dir_name) == -1) {
  /* Handle error */
}

fp = fopen(file_name, "w");
if (fp == NULL) {
  /* Handle error */
}

/*... Process file ...*/

if (fclose(fp) != 0) {
  /* Handle error */
}

if (remove(file_name) != 0) {
  /* Handle error */
}
```

4.4 Pathname Canonicalization

4.4.1 Intent

The intent of the Pathname Canonicalization pattern is to ensure that all files read or written by a program are referred to by a valid path that does not contain any symbolic links or shortcuts, that is, a canonical path.

4.4.2 Motivation

Because of symbolic links and other file system features, a file may not actually reside in the directory indicated by a path. Therefore, performing string-based validation on the pathname may yield false results. Having the true, canonical pathname is particularly important when checking a directory to see if it is secure.

4.4.3 Applicability

The use of the Pathname Canonicalization pattern is applicable if all of the following conditions are true:

- the program accepts pathnames from untrusted sources
- an attacker could provide a pathname to the system that non-obviously refers to a directory or file to which the attacker should not have access
- the program runs in an environment where each file has a unique canonical pathname

4.4.4 Structure

Programmatically, the structure of the Pathname Canonicalization pattern involves calling an OS-specific pathname canonicalization function on the given pathname prior to opening the file. The canonicalized pathname is used when operating on the file.

The canonicalized pathname itself has a structure such that every element of the canonicalized path, except the last, is the genuine directory, and not a link or shortcut. The last element is the genuine filename, and not a link or shortcut.

4.4.5 Participants

The participants in the Pathname Canonicalization pattern are

- the program opening file(s)
- the file system (potentially, depending on the implementation of the OS-specific canonicalization function)