Using the Tcl VFS for Encryption

By

Phil Brooks - Mentor Graphics Corporation

&

Arman Hunanyan - Mentor Graphics Corporation

Presented at the 22nd annual Tcl/Tk conference, Manassas Virginia, October 2015

Mentor Graphics Corporation
8005 Boeckman Road
Wilsonville, Oregon
97070
phil_brooks@mentor.com

arman_hunanyan@mentor.com

Abstract: In today's world of constant hacker and security threats, the Tcl VFS Interface provides a mechanism whereby Tcl scripted applications that are previously unaware of security and encryption concerns can transparently access both data files and Tcl modules from an encrypted container file without modifying the application specific file accesses. This mechanism allows data residing on disc to be strongly encrypted while it can be transparently accessed from a Tcl application that has the appropriate decryption keys to access the data transparently through normal Tcl file accesses. We examine CalVFS, an implementation that allows users of Mentor Graphic's Calibre electronic circuit verification tool to encrypt sensitive user information in such a way that it can be transparently accessed by the Calibre application.

Motivation

Mentor Graphics’ Calibre product line has a large number of customers and suppliers that provide extensions to our solutions that are specific to various integrated circuit foundries and design technologies. These groups are often providing proprietary extensions in the form of design files, data models, Tcl scripts, and Calibre rules files that their customers use in conjunction with our tools to build complete design solutions. The CalVFS system serves as a mechanism that these intermediate suppliers can use to securely encapsulate their design solutions and deliver them through to end users. Using this system, they build an encapsulated .calzip file that contains encrypted data files and Tcl scripts, deliver that file to end users who can then run the full design solution.   The advantage of the CalVFS system and its implementation on top of Tcl's VFS filesystem support is that scripts that were developed independently of any knowledge of the .calzip file or how to get at its contents can seamlessly access data and run Tcl scripts using the system. The .calzip files provide transparent read-only access to their contents within the Calibre runtime process through the Tcl VFS filesystem.

There is not a lot of documentation of the Tcl VFS or examples of its usage. As a result, we are providing this paper as a summary of our experiences developing an interface that uses the VFS in a manner that may be compelling to other developers.

Tcl VFS Overview

The Tcl Virtual File System (VFS) provides a low level C api through which all regular Tcl file mechanisms (things like **open, read,** and **seek** as well as higher level commands like **source** and **glob**) are routed. This abstraction allows a simple point of access that allows you to make up your own mechanism for supplying the data that any Tcl script can simply view as a bunch of regular file accesses.

Tcl_FS* Functions

The basic implementation of the Tcl VFS is through the Tcl_Filesystem structure and a group of functions that all start with Tcl_FS. The Tcl_Filesystem structure aggregates a group of functions that provide the underlying functionality of a particular filesystem implementation, and the Tcl_FS* functions allow installation, management and destruction of filesystem implementations as well as safe 'C' level access to VFS filesystem functionality that would be short circuited by direct use of the 'C' filesystem routines like stat(), access(), read() etc. For example, using the stat() function call directly from 'C' will not reflect any VFS filesystems installed, while use of TclFSStat will. The Tcl_FS* functions also provide consistent behavior across platforms (to the extent possible).

 The Tcl_FS functions provide a wide array of file related functionality. Highlights of the ‘C’ level function are provided below, a full list of the functions (from the Tcl FileSystem man page) is provided in the appendix:

File Access:
Direct access and manipulation of a file and its contents is provided by a number of functions like Tcl_FSOpenFileChannel, Tcl_FSStat and Tcl_FSAccess.

Directory Manipulation
Directories,their contents, symbolic links, etc. can be accessed and manipulated using functions like Tcl_FSMatchInDirectory, Tcl_FSGetCwd, Tcl_FSChdir, Tcl_FSLink, Tcl_FSCreateDirectory and Tcl_FSUtime.

File Path manipulation and construction
Platform independent construction and manipulation of file paths are provided by functions like Tcl_FSPathSeparator, Tcl_FSNormalizePath, Tcl_FSJoinPath, and Tcl_FSSplitPath.

VFS construction, manipulation and maintenance
The VFS filesystem implementations themselves use functions like Tcl_FSRegister, TclFSUnregister, Tcl_FSData and TclFSMountsChanged to install and maintain VFS Filesystems.

Tcl_Filesystem structure
The Tcl_Filesystem structure contains pointers to functions that implement underlying VFS filesystem operations. Many of them correspond directly to Tcl_FS* functions listed above. These functions are called by the underlying Tcl when an installed VFS path is accessed through Tcl commands or the Tcl_FS* interfaces. One important function in the structure is the Tcl_FSPathInFilesystemProc. This function is the key mechanism by which a VFS filesystem identifies files that belong to it. The remaining functions can similarly be grouped according to function. The full structure is presented in the appendix.

File Access:
Tcl_FSOpenFileChannelProc, TclFSStatProc, and Tcl_FSAccessProc. Tcl_FSOpenFileChannelProc is important to us in that the Tcl channel interface is used to return decrypted data to the end user.

Directory Manipulation:
TclFSMatchInDirectoryProc, Tcl_FSGetCwdProc, Tcl_FSChdirProc, Tcl_FSLinkProc, Tcl_FSCreateDirectoryProc, and Tcl_FSUtimeProc.

File Path manipulation and construction
Tcl_FSPathSeparatorProc, Tcl_FSNormalizePathProc.

Tcl_FSJoinPath and TclFSSplitPath are algorithmically implemented and does not interact with an installed VFS.

vfs:: Active Tcl package
Provided with ActiveTcl, or available from SourceForge, the vfs:: package provides a number of vfs implementations.   It consists of three parts:

1. vfs – the vfs::filesystem commands – allow mount, unmounts and manipulation of vfs filesystems.
2. vfs-filesystems: Several useful filesystems: zip, mk4, tar, ftp, ns, webdav, http, urltype
3. vfs-fsapi – the interface for creating a filesystem. This allows you to write a command ensemble that supports access to a vfs filesystem

Tcl Encryption Overview
Many encryption packages are available for Tcl. The following are available with Tcllib:

- blowfish: http://tcllib.sourceforge.net/doc/blowfish.html
- aes: http://tcllib.sourceforge.net/doc/aes.html
- des (including triple des): http://tcllib.sourceforge.net/doc/des.html
- rc4: http://tcllib.sourceforge.net/doc/rc4.htmltcl

Here is a des example:

```
package require des

set key "12345678";  # Must be 8 bytes long
set msg "abcde"

##### ENCRYPTION
set encryptedMsg [DES::des -dir encrypt -key $key $msg]
# $encryptedMsg is a bunch of bytes; you'll want to send
this around...

##### DECRYPTION
set decryptedMsg [DES::des -dir decrypt -key $key
$encryptedMsg]
puts "I got '$decryptedMsg'"
```

Also more complicated, but with a lot of different components and interfaces:

Cryptkit - www.patthoyts.tk/programming/**Cryptkit**.pdf

The encryption packages can be coupled with a Tcl Channel. This allows us to build a channel that does decryption and writes to a channel as it reads from an encrypted input channel. For example, the aes, blowfish, DES and rc4 packages all support –in <channel> and –out <channel> arguments when the channel object is created.

Combining VFS and Encryption

By building a decryption channel described above in the Tcl_FSOpenFileChannelProc function that is part of the Tcl_Filesystem api, a transparent read capability is created. As a result, Tcl code can transparently access objects in the .calzip filesystem:

```
set   fd [ open “foo/bar.calzip/text_file.txt” ]
set buf [ read $fd ]
source foo/bar.calzip/datafile.tcl
```

Calibre VFS

Limitations of our implementation

Since the target usage scenario for CalVFS is a supplier creates an archive, ships it to end users who then have read only access to the archive – or write once, read many delivery, the implementation only implements a read-only file system. Also, it does not support symbolic links. While the Tcl_Filesystem interface calls for implementation of more of the filesystem interfaces, we found that a perfectly workable read-only filesystem can be implemented with only the following interfaces:

pathInFilesystemProc – identifies files that are part of the VFS

statProc – provides information similar to stat()

accessProc - provides information similar to access()

openFileChannelProc – implements the read channel for files in the filesystem

Mounts Anyone?

The vfs:: package requires an explicit mount with a directory to enable a filesystem. In our use model, we don't know beforehand where the .calzip files will be. The user simply specifies a path name that may or may not include a .calzip archive. The Virual filesystem API doesn't specifically require a mount point, however. Instead, the CalVFS function that implements Tcl_FSPathInFilesystemProc (part of the Tcl_Filesystem structure) responds to a query signifying whether a particular path name is part of 'the CalVFS filesystem' or not. The files are identified by examining the filepath and looking for .calzip archives along the path. Once a .calzip archive is identified, it is opened by the CalVFS filesystem implementation and its contents are transparently accessed.

The automatic mounting mechanism also allows .calzip archives to access other external archives through relative paths. For example:

If we have the following files:

- archive .calzip archive foo/demo1.calzip which contains a directory dir1/dir2
- archive foo/demo2.calzip which contains the file dir3/dir4/file
- a regular file foo/bar/file that is outside of any archive

Then automatic mounting will allow to access through arbitrary paths like these:

foo/demo1.calzip/dir1/dir2/../../../demo2.calzip/dir3/dir4/file

foo/demo1.calzip/dir1/dir2/../../../demo2.calzip/dir3/dir4/../../../bar/file

In both cases path access will force automatic mount of both the demo1.calzip and demo2.calzip archives. This will allow inclusion of source Tcl files or data files files from outside VFS.

Nesting

CalVFS archives can also be contained inside of one another. ACLs and permissions on contained archives can be different from one another. So the above case can also similarly work when the two archives are nested one inside the other:

- archive .calzip archive foo/demo1.calzip which contains a directory dir1/dir2
- archive demo2.calzip which contains the file dir3/dir4/file is contained inside dir1 of demo1.calzip
- a regular file foo/bar/file that is outside of any archive

Then automatic mounting will allow to access through arbitrary paths like these:

foo/demo1.calzip/dir1/demo2.calzip/dir3/dir4/file

foo/demo1.calzip/dir1/dir2/../demo2.calzip/dir3/dir4/../../../../../bar/file

In these cases path access will also force automatic mount of both the demo1.calzip and demo2.calzip archives.

Write/Encode Process

Files are encoded in a simple serial archival process similar to what Zip or Tar would use. The files are encrypted as they are written into the archive. Contained .calzip files are simply copied into a containing archive.

Read/Decode Process

Files are decoded in 1064 byte chunks and their decode state is maintained in conjunction with the channel that is currently reading from them. Offsets and intermediate decryption information are also maintained inside of the channel object.

Seek

Seek functionality is limited and can seeking can have performance implications since the decryption state must be maintained from a known starting point. When seek moves the current position of the channel away from its current point, access must start at a known starting point and continue forward to the seek position. Optimizations are possible if multiple starting points are possible for the encrypted file as in some block based encoding schemes.

Access Control

As described thus far, any Tcl interpreter running in the calibre process has full access to all of the files in the .calzip archive. One goal of the system is to provide at least some level of protection between the .calzip contents and prying eyes of the end user. (Note that complete protection cannot be provided due to the 'DVD problem' – i.e. if you have a well encrypted file and a box that can decode that file and a sufficiently sophisticated user with sufficient access to the insides of the box, the user can discover how to decrypt the file without using the box.) To that end, the CalVFS system uses a system of Access Control Lists to vary what type of access it will allow to which interpreters in the system. That way, a specific vendor's .calzip file can get its own dedicated interpreter that has access to the .calzip contents while other interpreters in the same running process are denied access to the same .calzip file.

Development Mode

Since the filesystem functionality of the .calzip archive is limited and different from normal file accesses in a regular filesystem, there is a potential debugging issue for developers that intent to deliver .calzip archives:

> If an application depends upon functionality not available in the CalVFS while they are developing the application using the regular filesystem, they will not discover that dependency until they go to the final stage

of creating the .calzip archive and then testing their application in that environment.

Examples of functionality that they might depend upon includes erroneously expecting write access, or expecting meaningful symlinks inside of a .calzip archive, there would be no issue with doing that as long as the user was working in the regular filesystem, but when they create the .calzip archive, then the problem would become apparent. In order to facilitate simple development, testing, debugging and deployment, a development mode for the CalVFS filesystem allows the user to tag a development directory as being part of a CalVFS filesystem without actually creating the archive. Then accesses within the directory will carry the same limitations as they would if that directory were actually inside of a .calzip archive. In the final delivery state, the .calzip archive is, by default, very opaque. As a result, there are a number of other debugging features that allow a developer to control for more or less transparency into the contents of the .calzip archive during runtime.

Conclusions

The Tcl VFS system provides a highly functional interface that allows support for new filesystem functionality in applications that are using regular Tcl file accesses. Encryption libraries present in Tcl can be combined with the VFS filesystem access to provide transparent access to encrypted directories of files by combining elements of a zip or tar archive with encryption and the VFS.

Acknowledgements

Special thanks to Fedor Pikus for helping develop the underlying filesystem semantics.

Tcl_Filesystem and Tcl_FS* functions

Full information is available at the following URL:

https://www.tcl.tk/man/tcl8.6/TclLib/FileSystem.htm

The full list of Tcl_FS* functions available in Tcl 8.6 are:

Tcl_FSRegister, Tcl_FSUnregister, Tcl_FSData, Tcl_FSMountsChanged, Tcl_FSGetFileSystemForPath, Tcl_FSGetPathType, Tcl_FSCopyFile, Tcl_FSCopyDirectory, Tcl_FSCreateDirectory, Tcl_FSDeleteFile, Tcl_FSRemoveDirectory, Tcl_FSRenameFile, Tcl_FSListVolumes, Tcl_FSEvalFile, Tcl_FSEvalFileEx, Tcl_FSLoadFile, Tcl_FSUnloadFile, Tcl_FSMatchInDirectory, Tcl_FSLink, Tcl_FSLstat, Tcl_FSUtime, Tcl_FSFileAttrsGet, Tcl_FSFileAttrsSet, Tcl_FSFileAttrStrings, Tcl_FSStat, Tcl_FSAccess, Tcl_FSOpenFileChannel, Tcl_FSGetCwd, Tcl_FSChdir, Tcl_FSPathSeparator, Tcl_FSJoinPath, Tcl_FSSplitPath, Tcl_FSEqualPaths, Tcl_FSGetNormalizedPath, Tcl_FSJoinToPath, Tcl_FSConvertToPathType, Tcl_FSGetInternalRep, Tcl_FSGetTranslatedPath, Tcl_FSGetTranslatedStringPath, Tcl_FSNewNativePath, Tcl_FSGetNativePath, Tcl_FSFileSystemInfo, Tcl_GetAccessTimeFromStat, Tcl_GetBlockSizeFromStat, Tcl_GetBlocksFromStat, Tcl_GetChangeTimeFromStat, Tcl_GetDeviceTypeFromStat, Tcl_GetFSDeviceFromStat, Tcl_GetFSInodeFromStat, Tcl_GetGroupIdFromStat, Tcl_GetLinkCountFromStat, Tcl_GetModeFromStat, Tcl_GetModificationTimeFromStat, Tcl_GetSizeFromStat, Tcl_GetUserIdFromStat, Tcl_AllocStatBuf

The Tcl_Filesystem structure:

```
typedef struct Tcl_Filesystem {
    const char *typeName;
    int structureLength;
    Tcl_FSVersion version;
    Tcl_FSPathInFilesystemProc *pathInFilesystemProc;
    Tcl_FSDupInternalRepProc *dupInternalRepProc;
    Tcl_FSFreeInternalRepProc *freeInternalRepProc;
    Tcl_FSInternalToNormalizedProc *internalToNormalizedProc;
    Tcl_FSCreateInternalRepProc *createInternalRepProc;
    Tcl_FSNormalizePathProc *normalizePathProc;
    Tcl_FSFilesystemPathTypeProc *filesystemPathTypeProc;
    Tcl_FSFilesystemSeparatorProc *filesystemSeparatorProc;
    Tcl_FSStatProc *statProc;
    Tcl_FSAccessProc *accessProc;
    Tcl_FSOpenFileChannelProc *openFileChannelProc;
    Tcl_FSMatchInDirectoryProc *matchInDirectoryProc;
    Tcl_FSUtimeProc *utimeProc;
    Tcl_FSLinkProc *linkProc;
    Tcl_FSListVolumesProc *listVolumesProc;
    Tcl_FSFileAttrStringsProc *fileAttrStringsProc;
    Tcl_FSFileAttrsGetProc *fileAttrsGetProc;
    Tcl_FSFileAttrsSetProc *fileAttrsSetProc;
    Tcl_FSCreateDirectoryProc *createDirectoryProc;
    Tcl_FSRemoveDirectoryProc *removeDirectoryProc;
    Tcl_FSDeleteFileProc *deleteFileProc;
    Tcl_FSCopyFileProc *copyFileProc;
    Tcl_FSRenameFileProc *renameFileProc;
    Tcl_FSCopyDirectoryProc *copyDirectoryProc;
    Tcl_FSLstatProc *lstatProc;
    Tcl_FSLoadFileProc *loadFileProc;
    Tcl_FSGetCwdProc *getCwdProc;
    Tcl_FSChdirProc *chdirProc;
} Tcl_Filesystem;
```