A Mutually Suspicious File System

Charles Perkins
Chief Cypherpunk and Brewmaster, Kuracali Sake and Beer Brewery

Nov 08 2013

Traditional computing systems delegate the protection of stored data to a trusted, priveleged party (e.g. the kernel, the supervisor, the administrator, the sysadmin, etc.) This trust is not always warranted. The Mutually Suspicious File System demonstrates a method by which the user-level process may retain the authority and responsibility for access control of stored[1] data.

This paper presents an implementation of `msfs` in the `go`[2] programming language. The program source code is open source and freely redistributable under the MIT license:

The program text is presented in the body of this paper like so:

about this code

```
source code statements
```

This LaTeX type-set paper *[A Mutually Suspicous...]* is drawn from the comments in the `go` source code via a `bash`-executable but `go`- and `godoc`-invisible preamble, unifying the program and the documentation in one text and thereby forming a minimally capable literate programming environment.[3]

[1] Protecting data and computations in memory from the kernel and from other users is also critical but is beyond the scope of this paper.

[2] Team, Go. The Go programming language specification. Technical Report. http://golang.org/ref/spec (retrieved Oct 2013.)

[3] Donald E. Knuth, Literate Programming, Stanford, California: 1992, CLSI Lecture Notes, no. 27.

2 Paper Overview

A successful program design may be approached in stages with each stage buidling on concepts and capabilities introduced in an earlier stage. An approcahable paper likewise may sequentially introduce and expand upon concepts and assertions. This paper is structured in several sections:

1. Introduction
2. Paper Overview
3. Theory of Operation
4. Subsystems
5. Program Flow
6. File System Usage
7. Literate Coding Conventions

By the end of this paper the reader should have a good understanding of how a mutually suspicious file system may be implemented.

3 Theory of Operation

`msfs` is based on the observation that public key cryptography, symmetric key ciphers and hash algorithms may combine to allow mutually suspicous use of a shared resource:

- RSA/DSA certificates provide user identity and introduction
- Symmetric-key ciphers provide efficiency and security
- Hash algorithms provide object reference and retrieval

Hard drive sectors are the shared resource in the case of the `msfs` file system and a serialization scheme allows multiple readers and writers of the underlying block store.

The theory of operations can be further broken down as follows:

1. File System Operations
2. `msfs` Notional Structures
3. Reading
4. Writing
5. Access Control
6. Notional to Block Mapping
7. Serializing Access

3.1 File System Operations

`msfs` should appear to be a conventional file system from the point of view of the user and the user's executing programs. FUSE (File System in User Space) is a protocol specification and operating system interface supported on Linux and OSX that allows user-level programs to appear to other user-level programs as a file system. The `msfs` program implements this interface in order to implement file-system functionality for other programs.

To implement the FUSE interface the program responds to the following requests:

- (FS) Root() – return the root directory
- (Dir) Attr() – return directory attributes
- (Dir) Lookup( ) – return a file entry
- (Dir) ReadDir( ) – return an array of directory entries
- (File) Attr() – return the attributes of a file
- (File) ReadAll( ) – return the contents of a file

3.2 Notional Structures

The `msfs` user-level client will maintain several notional structures that will cache information and assist in the translation of file i/o operations to block-level operations.

3.3 Reading

Reading from a file or a directory involves several steps.

3.4 Writing

Creating a file and writing to it involves several steps.

3.5 Access Control

Changes in access control are made by writing certificates and distributing keys.

3.6 Notional to Block Mapping

The notional structures maintained by the client are mapped to blocks-in-storage.

3.7 Serializing Access

Block-level hash storage is mediated by a goroutine and accessed over TCP/IP.

4 Subsystems

The `msfs` program includes several packages, each of which ecapsulates a portion of the program's functionaltiy, each in a separate file:

1. The File System Interface (in `msfsfiles.go`)
2. Hashing and Encryption (in `msfshashes.go`)
3. Block Storage (in `msfscas.go`)

The above files are imported by the main program executable, described in Section 5, Program Flow:

4. Program Sequencing, UI and Internetworking (in `msfs.go`)

4.1 The File System Interface

Conventional programs expect a conventional file system interface. The /textttmsfsfs/msfsfiles.go file implements the file system functionality of `msfs`:

msfsfiles (in msfsfiles.go) implements the msfs fuse interface

```
package msfsfiles
```

In addition to the standard `go` library, `fs.go` relies on functionality provided by the `fuse` package:

dependencies

```
import (
        "fmt"
        "os"

        "rputbl.com/msfs/msfshashes"
        "rputbl.com/msfs/msfscas"

        "bazil.org/fuse"
        "bazil.org/fuse/fs"
)
```

4.1.1 File System Structures and Methods

The `FS` structure holds the reference to the root directory and global information about the state of the file system.

FS encapsulates whole-file-system features

```
type FS struct{
        storeversion, execversion,serverloc string
        Hi *msfshashes.HSC
        Cas     *msfscas.CASC
        root *Dir
        Own *Dir
}
```

The file system as a whole is initialized when the system starts.

Prepare() sets up the file system for use.

```
func (msfs *FS) Prepare( hi *msfshashes.HSC, cas *msfscas.CASC) fuse.Error {
        msfs.execversion = "msfs.0.0.1"
        msfs.storeversion = "n/a"
        msfs.serverloc = "some server"
        msfs.Hi = hi
        msfs.Cas = cas

        msfs.root = new(Dir)
        msfs.Own = new(Dir)

        msfs.root.Prepare( []fuse.Dirent{
                {Inode: 2, Name: ".own", Type: fuse.DT_Dir},
                {Inode: 3, Name: ".status", Type: fuse.DT_File},
                {Inode: 8, Name: "testfile", Type: fuse.DT_File}, },msfs)

        msfs.Own.Prepare( []fuse.Dirent{
                {Inode: 4, Name: "privateKey", Type: fuse.DT_File},
                {Inode: 5, Name: "publicKey", Type: fuse.DT_File}, },msfs)



        fmt.Fprintf(os.Stderr, "Status: %s\n",msfs.Cas.Status)

        return  nil
}
```

Root returns the root directory to the FUSE interface.

```
func (msfs *FS) Root() (fs.Node, fuse.Error) {
        return msfs.root, nil
}
```

4.1.2 Directory Structures and Methods

`msfs` maps file system directory operations on hashes of the names of the contents of the directories.

Dir implements both Node and Handle for the root directory

```
type Dir struct{
        DirList  []fuse.Dirent
        Fs *FS
}

func (msd *Dir) Prepare(contents []fuse.Dirent, Fs *FS)  fuse.Error {
        msd.DirList = contents
        msd.Fs = Fs
        return nil
}


func (Dir) Attr() fuse.Attr {
```

```
        return fuse.Attr{Inode: 1, Mode: os.ModeDir | 0555}
}

func (msd *Dir) ReadDir(intr fs.Intr) ([]fuse.Dirent, fuse.Error) {
        return msd.DirList, nil
}

func (msd *Dir) Lookup(name string, intr fs.Intr) (fs.Node, fuse.Error) {
        if name == ".own" {
                return msd.Fs.Own, nil
        }else{
                for _,v:= range msd.DirList{
                        if name == v.Name {
                                f := new(File)
                                f.Prepare( v.Inode, v.Name, msd )
                                return f, nil
                        }
                }
        }

        return nil, fuse.ENOENT
}
```

4.1.3 File Structures and Methods

File implements both Node and Handle for the hello file

```
type File struct{
        Inode uint64
        Name string
        Msd *Dir
}

func (fi *File) Prepare(i uint64, n string, Msd *Dir)  fuse.Error {
        fi.Inode = i
        fi.Name = n
        fi.Msd = Msd
        return nil
}

func (File) Attr() fuse.Attr {
        return fuse.Attr{Mode: 0444}
}

func (fi File) ReadAll(intr fs.Intr) ([]byte, fuse.Error) {
        if fi.Inode == 3{
                return []byte(fmt.Sprintf("CAS Client State: %s\nCAS Client Mode: %s\n",fi.Msd.Fs.C
        }
        if fi.Inode == 5{
                return []byte(fi.Msd.Fs.Hi.PubCert), nil
        }
        return []byte(fmt.Sprintf("Test File Contents for file %s!\n",fi.Name)), nil

}
```

4.2 Hashing and Encryption

The file system structure maintains information needed to map file system operations (exposed by FUSE to the `msfs` executable) onto assertions and queries made via TCP/IP to a local or remote hash store.

The /textttmsfsfs/msfshashes.go file implements the hashing functionality of `msfs`:

msfshashes (in msfshashes.go) implements msfs hash functionality

```
package msfshashes
```

In addition to the standard `go` library, `fs.go` relies on functionality provided by the `fuse` package:

dependencies

```
import (
        "bazil.org/fuse"
        "encoding/base64"
        "errors"
        "fmt"
        "io"
        "io/ioutil"
        "os"
        "crypto"
        "crypto/rand"
        "crypto/rsa"
        _ "crypto/sha1"
        "crypto/sha256"
        "crypto/x509"
        "encoding/pem"
)
```

4.2.1 Hash Store Client Structures and Methods

HSC implements the hash store client

```
type HSC struct{
        PrivKey *rsa.PrivateKey
        PubCert []byte
}

func (hsc *HSC) Prepare(rsaName string)  error {
        var err error
        hsc.PrivKey, hsc.PubCert, err  = GetPKI( rsaName )
        return err
}
```

4.2.2 Hash Store Server Structures and Methods

HSS implements the hash store server

```
type HSS struct{}
```

```
func (hss *HSS) Prepare(destination string)  fuse.Error {
        return nil
}
```

4.2.3 Miscellaneous Structures and Methods

Sha224base64 performs a sha224 hash on a byte array and then perfroms a base64 encoding on the result.

```
func Sha224base64(item []byte) (string, []byte) {

        phash := sha256.New224()
        io.WriteString(phash, string(item))
        phashbytes := phash.Sum(nil)
        return base64.StdEncoding.EncodeToString(phashbytes), phashbytes
}
```

Un64 decodes a base-64 encoded hash string and returns a byte array or an error.

```
func Un64( hash64val string) ([]byte, error){
        return base64.StdEncoding.DecodeString(hash64val)
}
```

Sign64 signs a byte array with a private key.

```
func Sign64(rsakey *rsa.PrivateKey, item []byte) (string, []byte) {

        hashFunc := crypto.SHA1
        h := hashFunc.New()
        h.Write(item)
        digest := h.Sum(nil)
        signresult, _ := rsa.SignPKCS1v15(rand.Reader, rsakey, hashFunc, digest)
        return base64.StdEncoding.EncodeToString(signresult), signresult
}
```

GetPKI retrieves 'rputn' RSA public and private key values

```
func GetPKI( rsaName string ) (*rsa.PrivateKey, []byte, error) {

        rsa_file := fmt.Sprintf("%s/.ssh/%s", os.Getenv("HOME"),rsaName)
        rsapub_file := fmt.Sprintf("%s/.ssh/%s.pub", os.Getenv("HOME"),rsaName)

        _, err := os.Stat(rsa_file)
        if err == nil {
                _, err = os.Stat(rsapub_file)
        }
        if err != nil {
                estr:="Please generate a reputation public/private key pair\n" +
                "(ssh-keygen -t rsa -C \"<user>@<host>\" -f \/.ssh/<filename>)\n"+
                "rsa_file: "+rsa_file+"\n"+
                "rsapub_file: "+rsapub_file+"\n"
                return nil, nil, errors.New(estr)
        }

        rputn_rsa, _ := ioutil.ReadFile(rsa_file)
```

```
        rputn_rsa_pub, _ := ioutil.ReadFile(rsapub_file)
        block, _ := pem.Decode(rputn_rsa)
        rsakey, _ := x509.ParsePKCS1PrivateKey(block.Bytes)

        return rsakey, rputn_rsa_pub, nil
}
```

4.3 Content Aware Storage

The file system structure maintains information needed to map file system operations (exposed by FUSE to the `msfs` executable) onto assertions and queries made via TCP/IP to a local or remote content-aware hash store.

The /textttmsfsfs/msfsblocks.go file implements the content aware storage functionality of `msfs`:

msfscas (in msfscas.go) implements msfs content-aware persistent storage

```
package msfscas
```

In addition to the standard `go` library, `fs.go` relies on functionality provided by the `fuse` package:

dependencies

```
import (
        "fmt"
        "os"
        "io"
        "syscall"

        "github.com/ncw/directio"
)
```

4.3.1 CAS Client Structures and Methods

CASC implements the block store client

```
type CASC struct{
        Cass CASS
        Status string
        Mode string
}
```

Status may be disconnected, unintroduced, or connected, mode may be direct, local, or remote.

Prepare() initializes the Content Aware Storage Client

```
func (casc *CASC) Prepare(cass CASS)  error {
        casc.Cass=cass
        casc.Status="disconnected"
        casc.Mode="direct"
        return nil
```

```
}

type CasReq struct {
        Request      string
}
type CasAns struct {
        Answer       string
}
```

4.3.2 CAS Server Structures and Methods

CASS implements the block store server

```
type CASS struct{
        Requests chan *CasReq
        Answers chan *CasAns
        BlockDevice string
        BlockHost string
        BlockPort string
        BlockFormat string
        BlockSize int64
        BlockMagic string
        BlockStat syscall.Stat_t
        Status string
}
```

Prepare must be called before using a Content Aware Storage Server

```
func (cass *CASS) Prepare(bd, bh, bp string, casreq chan *CasReq, casans chan *CasAns)  error {

        var  err error

        cass.Requests = casreq
        cass.Answers = casans
        cass.BlockDevice=bd
        cass.BlockHost=bh
        cass.BlockPort=bp
        cass.BlockFormat = "Unknown"
        cass.Status="disconnected"

        xfi, err := os.Stat(cass.BlockDevice)
        if err != nil {
                return err
        }
        cass.BlockSize = xfi.Size()

        in, err := directio.OpenFile(cass.BlockDevice, os.O_RDONLY, 0666)
        if err != nil {
                return err
        }

        block := directio.AlignedBlock(BlockSize)
        _, err = io.ReadFull(in, block)
        if err != nil {
                return err
        }
```

```
        cass.BlockMagic = fmt.Sprintf("%s ",block[0:10])

        err = syscall.Stat(cass.BlockDevice,&cass.BlockStat)
        if err != nil {
                return err
        }
        fmt.Printf("Size: %d Blocksize: %d\n",cass.BlockStat.Size,cass.BlockStat.Blksize)

        fmt.Printf("OK\n")




        return err
}

func (cass *CASS) Serve() {

        for s:= range cass.Requests{
                fmt.Printf("Block request: %s\n",s.Request)
        }

}

const (
    AlignSize = 512
    BlockSize = 512
)
```

5 Program Flow

The main program begins here::

msfs (in msfs.go) implements a mutually suspicious file system

```
package main
```

In addition to the standard `go` library, `msfs.go` relies on functionality provided by the `fuse` and `directio` packages and includes the packages defined earlier in Section 4:

dependencies

```
import (
        "flag"
        "fmt"
        "log"
        "os"
        "net/http"

        "bazil.org/fuse"
        "bazil.org/fuse/fs"

        "rputbl.com/msfs/msfsfiles"
        "rputbl.com/msfs/msfshashes"
        "rputbl.com/msfs/msfscas"
)
```

The msfs main executable has several sections:

- Startup and Shutdown
- The File System Service Loop
- The Networking Service Loop
- The Block Storage Service Loop

5.1 Startup and Shutdown

In startup, the msfs configuration is assisted by the flag package, which needs a Usage function to instruct the user on options and parameters:

Usage prints invocation options

```
var Usage = func() {
        fmt.Fprintf(os.Stderr, "Usage of %s:\n", os.Args[0])
        a := "mountpoint  blockdevice host port rsafile"
        fmt.Fprintf(os.Stderr, " %s %s\n", os.Args[0],a)
        flag.PrintDefaults()
}
```

The msfs executable checks on startup to see if it has the right number of parameters, initializes the subsystems, then passes execution in parallel to the file system, hash services, communications, and block storage subsystems. Upon return from any of these systems the rest are cleaned up and the program terminates.

main checks parameters and initializes services

```
func main() {
        flag.Usage = Usage
        flag.Parse()

        if flag.NArg() != 5{
                        Usage()
                        os.Exit(2)
        }
        mountpoint := flag.Arg(0)
        blockdevice := flag.Arg(1)
        host := flag.Arg(2)
        port := flag.Arg(3)
        rsafile := flag.Arg(4)

        casreq, casans:= make(chan *msfscas.CasReq), make(chan *msfscas.CasAns)

        blockstoreserver := new(msfscas.CASS)
        if err:=blockstoreserver.Prepare(blockdevice,host,port,casreq,casans); err != nil {
                log.Fatal(err)
        }

        blockstoreclient := new(msfscas.CASC)
        if err:=blockstoreclient.Prepare(*blockstoreserver); err != nil {
                log.Fatal(err)
        }

        hashinterface := new(msfshashes.HSC)
```

```
        if err:=hashinterface.Prepare( rsafile ); err != nil {
                log.Fatal(err)
        }

        filesystem := new(msfsfiles.FS)
        if err:=filesystem.Prepare(hashinterface, blockstoreclient ); err != nil {
                log.Fatal(err)
        }

        go serveFS(mountpoint,filesystem)

        go serveNET(blockdevice,host,port)

        blockstoreserver.Serve()

}
```

5.2 The File System Service Loop

serveFS is a process for each user that converts file system operations into hash operations

```
func serveFS(mountpoint string, filesystem *msfsfiles.FS ) {


        c, err := fuse.Mount(mountpoint)
        if err != nil {
                log.Fatal(err)
        }

        fs.Serve(c, filesystem)
}
```

5.3 The Networking Service Loop

serveNET extends the programs services over the network

```
func serveNET(blockdevice, host, port string) {

        http.HandleFunc("/", handler)
        http.ListenAndServe(":8080", nil)
}

func handler(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintf(w, "Hi there, I love %s!", r.URL.Path[1:])
}
```

6 File System Usage

In ordinary usage a user will orient the `msfs` file system in the `sec` subdirectory of their home directory, e.g. `/home/chuck/sec`.

To unmount the filesystem: `fusermount -u mountdir`

7 Literate Coding Conventions

7.1 Obtaining the `msfs` source code

7.2 Creating the `msfs` executable

7.3 Generating the `msfs` document

7.4 Reporting `msfs` bugs

7.5 Suggested exercises in extending `msfs`