Project Information

Abstract:

This project will focus on providing file integrity checking capabilities to pefs. The file integrity checker will compare cryptographic checksums of files against a static signed checksum list at access time. The files are thought to be immutable and use of securelevel will guarantee that lower filesystems will protect those files. Securelevel will be extended to only permit execution of files with immutable flag set.

Project Description:

1) Kernel Level File Integrity Checking

The concept is simple: check integrity of files that aren't supposed to be altered often (or at all). Most notably system files like configuration files, shared libraries and executables.

1.1) Defining checksums and checksum list

The checksum list is supposed to be static. It will be a file found in root directory of the pefs filesystem, much like the .pefs.db file which is created e.g. with addchain command, if it doesn't already exist. /sbin/pefs will have to be extended to provide another argument that will accept a list of files (in the form a file perhaps) and create .chksum. /sbin/pefs could also create checksums for all files under a given directory.

The checksums themselves will be MAC tags and pefs will use them to verify integrity of each block of data upon access.

Checksum list entries will be in the form of:

<inode, tweak, mac tag list>

The .chksum file will contain an entry for itself and will itself be signed.

1.2) Integrity checking: keys and algorithms used

A HMAC algorithm (e.g. HMAC-SHA256) will be used to create and check a MAC tag for a particular file block. pefs operates on 4096 byte sectors and it's only logical that integrity checking will be performed on 4096 byte sectors as well.

The HMAC algorithm will need a private key that will be used both for creation and checking of MAC tags. This key will be chosen by the user. Use of MAC will make it harder for attacker to create his own mac tags if he has no knowledge of private key or access to execute his own kernel code.

The concept is to have per file unique tweaks which will be combined with the private key and fed to the HMAC algorithm. The tweak will be a combination of file unique tweak plus block offset. This way, two same messages (blocks of data) won't produce the same MAC tag. This mimics what pefs is doing during encryption.

I won't reuse keys/salts that are already used for encryption. Therefore it will be necessary to keep per file unique tweaks in .chksums as there is no space left in filename (already used by the encryption process).

For more thoughts on keys and algorithms used see also parts #2.1 and #2.3

1.3) Make kernel pefs aware of checksum list

The first task would be to get the checksum list into the kernel. During each pefs mount, the respective .chksum file is parsed and a list of those entries will be created into the kernel and referenced by the pefs_mount structure. This way, trust falls on integrity of the .chksum file. If .chksum is signed or at least have mac tags for itself, then during boot the checksum file's integrity will be verified.

1.4) Where/when/how to check integiry of each file

First of all, we have to mark that a particular pf_node needs to be checked for integrity. After pefs mount, the kernel is already aware of all of the files that need to be checked, as well as having their mac tags and tweaks. Files are referenced in checksum list by their unique inode number.

When a new pfnode is created in pefs_node_node(), the in kernel checksum list will be checked to see if there's a entry for that file and if true, the pn_flags of that vnode will be ORed to mark the need for integrity checks. pefs_node will reference the appropriate checksum list entry.

Now, every time a VOP is executed in pefs kernelspace the filesystem knows if it has to check for integrity. We'll check the file's integrity each time it is accessed: e.g. VOPs read, strategy, bmap, getpages, ioctl, open, getattr, readdir, lookup etc.

At this point, we could rely on securelevel mechanism and the lower filesystem to protect the data of that file, especially if immutable flag is turned on. Since all of the files mentioned in .chksum aren't supposed to change, we could also check in pefs level if a VOP tries to do just that. e.g. VOPs write, strategy, putpages, fsync, ioctl, link, rename, remove, open, close, setattr, etc.

If integrity check for a block fails then either return EINVAL and print related information or panic, depending on configuration.

2) Thoughts on this approach

2.1) About checksums, algorithms and keys

The way I proposed we go about verifying the integrity of files may seem a bit excessive and there are certainly performance issues to keep in mind, but:

a) Since checksum list has to be generated at first, we give the option to the user to define the exact list of files that he wishes to be checked for integrity instead of just enabling integrity checking for all files under directory X.

2) The critical immutable files for a server are a) configuration files b) libraries. (we could also add executables to this list, especially those with setuid bit).
Configuration files are only read during boot or application startup, same with libraries although the latter could used more often.

3) It's a design choice and a trade-off. Increased protection for a selected file list vs simpler

algorithm for entire disk. Taking points 1) & 2) into account, the performance hit could be even lighter than a per disk encryption/integrity checking scheme (not taking various code optimizations into account).

Gleb@ mentioned to me that a list of hashes for blocks of file would be an acceptable choice. What I'm considering is perhaps allowing the user to specify if he would like to use a simple hash function instead of the HMACs we provide, thus eliminating the need for an extra private key as well as the extra overhead.

2.2) Why pefs?

There are several reasons why I chose build my integrity checker on top of pefs.

a) Implementing a file integrity checker on top of an existing stackable filesystem would be far easier than coding a separate filesystem from scratch. This way I will also be taking advantage of how pefs operates on blocks of data for each file. I've already described how this will be done to some extent and I believe it's obvious that focusing on the task at hand could provide much better results.

By the way, I assume that a stackable filesystem would be the only way to go since sys/admins can already code scripts that perform read only NFS mounts and then use find(8) and md5(8) to perform integrity checking. The goal is to have a in-kernel way to do that without too much hassle.

b) I believe that if we were to have different stackable filesystems for encryption and integrity checking, we would decrease easy of use, making the process a little too complex for the sys/admin.

c) geli(8) already performs integrity checks, as well as being a well round cryptographic filesystem. On the other hand, pefs(8) operates on a per directory basis that is transparent to the filesystems bellow, as opposed to geli's per-disk basis. If integrity checking is turned on for geli, around 12% of disk space is unusable. Since pefs does in place encryption, the only extra space used for encryption is the .pefs.db file. Similarly, integrity checking will require only a .chksums file which won't be as wasteful, unless whole disk is encrypted/checked of integrity in which case perhaps pefs shouldn't be used. Last but not least, pefs implements a key chain hierarchy similar to the Bell-LaPadula security model that could be very useful in some security setups.

2.3) Should integrity checking require filesystem encryption?

A question that's been bugging me is whether the file integrity checker should be able to run without encryption enabled for the filesystem in case of users who just need the integrity checking feature. I haven't made a decision yet and I don't believe I will attempt to separate the two during this GSoC.

However, since the file integrity checker uses salts and creates unique keys for each block, separating the two does not seem like a bad idea at the moment. If the two were never to be separated, we could omit the salts during integrity checking process because two same messages will never be the same after they are encrypted by pefs and therefore, they will wield different MAC tags. If however the file integrity checker used simple hashes, I am not sure whether enabling only integrity protection is a good idea. I will have to talk this through with my mentor.

Timeline of Deliverables:

April 23 – Accepted students proposals announced - community bonding period
April 24 – May 20   Evaluate design choices with mentor and study pefs codebase

May 21 – June 3	Extend sbin/pefs so that it creates checksum file with dummy tags/tweaks which is then read during pefs_mount. pf_nodes are properly marked upon creation.
June 4 – June 17	Extend sbin/pefs so that it accepts secret key from userland and that proper tags/tweaks are created for checksum file. At this point test with one HMAC algorithm.
June 18 – July 1	Integrity checks in Vnode Operations
July 2 – July 12	Excessive testing of new features and review code with mentor. If there's enough time, add extra HMACs or even option for simple hash and test those as well.
July 13	Midterm Evaluations
July 14 – July 30	Implement authentication checks for checksum files during mount
July 31 – August 13	Extension of securelevel and perhaps rtld
August 13 - August 20	More testing, write documentation
August 20	Pencils down

Test Plan:

The underlying filesystem that will be used during the development and testing phases will be standard UFS. dbench was used by gleb@ mostly as a stress tool during development of pefs and it will be used during this GSoC as well. Standard fs regression tools (such as fsx) could be used to operate on files protected by immutable flag, files without immutable flag that should remain intact because they exist in checksum list, as well as normal files with r/w mode. The main issue would be to emulate an attack where attacker is able to bypass securelevel protection and load his own kernel module. I'm thinking about developing my own kernel module and patches that will introduce “holes” in securelevel (maybe as a new scenario for pho's test suite?).

Note:

Because the technical part of my gsoc proposal provided a comparison between pefs and disk level cryptographic systems like geli, John Baldwin requested a comparison between my approach and mac_chkexec, a MAC module developed by csjp@. This is my technical analysis:

Major differences between my approach and mac_chkexec

a) pefs does not rely on a particular lower filesystem like mac_chkexec does

pefs is a stackable filesystem, residing on top of lower filesystems like UFS and oblivious to them. Which means that if pefs is aware of which files should be checked for integrity, it *does not have to* rely solely on lower filesystem to keep them unaltered, even with extended securelevel and immutable flags.

Keep in mind that for this project, files that are listed in .chksum list are thought to be immutable and the checksum list, static.

Therofore, pefs could work with any filesystem that is mountable in FreeBSD. Even without extended securelevel, pefs could intercept calls to VOP_ACCESS or even VOPs that alter lower fs like VOP_WRITE.

e.g. In pefs_write (VOP_WRITE)

[...]

// Marking which vnodes should be checked is described in my proposal

```
if (pf_node->pf_flag & IMMUTABLE) {
 dprintf("some useful info");

return EINVAL;
}
```

mac_chkexec on the other hand relies on extended attributes of UFS2 in order to provide integrity checking.

b) MAC tags with tweaks vs simple hashes

What I propose is that pefs uses a HMAC algorithm . In contrast to geli, in my implementation two messages M1, M2 where M1 == M2 will provide two MAC Tags T1, T2 where T1 != T2. Using MAC with a secret key (key is different from the one used for encryption) will make it even more difficult for an attacker to forge a tag. Also, pefs operates on a per 4096 block basis. This is further explained in my proposal.

On the other hand, mac_chkexec uses simple hash algorithms like md5 and sha1 to create hashes for files. Hashes are created for entire file.

c) protect executables and assorted files vs any kind of file

mac_chkexec is based on the notion of protecting executables, although there is a "dependencies" option. On the other hand, pefs will be used to protect any file that is listed on checksum list, or even every file under directory X. Adding extra protection for executables is only a part of my project proposal, note the entire scope of my proposal.

d) checksum integrity

Something that bugs me about mac_chkexec is its learning mode. In my proposal, checksum list is static and is supposed to be generated at filesystem image generation phase. A serious concern of mine is verifying that checksum list is valid during trusted boot (e.g. Signed key for checksum list, boot from read only media). On the other hand, mac_chksexec simply adds hashes for executables, when they are accessed during learning mode, as extended attributes to the respective file. During boot, there is no way, as far as I can tell, to guarantee that checksums are valid.

On a personal note, I always found FreeBSD MAC policies a little hard to use (or even understand) in practice. By the way, does mac_chkexec run in FreeBSD Current?

I'm currently busy working on my 3d application (port NTFS from Darwin) for GSoC. Let me know if a more thorough analysis is needed and I'll get back at you asap.