A Context-Sensitive, Secure File System

Britton Dennis, Tyler Travis, Clark Wood

Abstract

In this paper we present a context-sensitive, secure file system. Our system was designed to offer block level security as well as present access to a file under multiple varying contexts in diverse fashions. Some of our related works illustrate different ways to use the different context / different view approach, but these prior works raise similar security concerns, and consequently we choose to do ours by limiting the information that gets presented to the viewer. This is because of a fundamental capacity for sensitive data leakage in these systems, owing to the presence of naming collisions. This becomes important when trying to design a secure file system which adheres to such security schemes and Biba or BLP[H, I], which restrict reading or writing to objects of a higher or lower security level. We currently implement a version of Bell LaPadula, which enforces a general “no read up” policy to protect data confidentiality.

Under the most privileged access, wherein a user or application has complete access to all file blocks, the file will appear as normal, but as the security level drops due to monitored contextual changes which our system deems as putting the system in a less secure state, less and less information will be presented. We left the choice of what gets displayed under each context to the user, since this is a policy decision, where users with different security requirements would have different views as to what constitutes important information.

I Introduction

Context is all around us. The difference between a genuine compliment and a snide affront is subtle, minute, and determined within a split second. Computers, like humans, experience a variety of contexts at different times. The difference between visiting the real Facebook website and a clever imposter may be the difference between seeing HTTP and HTTPS in one’s browser. Until now, security context was determined in real-time by users by picking up various context clues and modifying their own behavior accordingly. The disadvantage of this manual, case-by-case decision is that humans make mistakes. A less technology-savvy person may not even notice that what appears to be their banking website has an invalid SSL certificate. The burden of determining how secure of a context a given system is in cannot be placed solely upon end-users.

In order to mitigate these concerns, researchers are developing context-aware software. Michael Mitchell explained environmental context as differing from the conventional process context switching. Instead, environmental context is derived from many different sources that each explain a small piece of where the machine is and how it is being used. For instance, sensors like GPS, processes running within the machine, and its user’s internet activity can be used in unison to determine the context in which the machine is being operated [G].

There already exist frameworks for creating context-sensitive files, like quFiles [F], but, for various reasons, these projects do not achieve the type and level of security some users require. Context-sensitive security can mean many different things, so for the scope of this paper context-sensitive security will be defined as protection of a resource’s confidentiality and integrity as determined by the current nature of its environment. There is a natural understanding of varying levels of security, from a completely insecure environment, perhaps an unencrypted hard drive on a system with many conspicuous processes running which is attached to and sending sensitive data over an unencrypted wireless network, to a

highly secure, air gapped, guarded SCIF which can only be accessed by individuals with a particular security clearance [J].

Time-tested security models, like Biba and BLP [H ,I], provide general ways to implement security for integrity and confidentiality respectively. Confidentiality gives access to information only to authorized users, while integrity prohibits the changing of this information by unauthorized users [K]. Biba protects integrity by disallowing read down or write up, and inversely Bell Lapadula protects confidentiality by disallowing read up or write down. Both of these properties, along with availability, are valuable to users, and must be taken into consideration.

An application which provides context-sensitive security must safeguard confidentiality, integrity, and availability by moving to and from more and less secure states depending upon the environmental context. This context exists as an aggregation of the aforementioned factors, and our project detects these changes in context and responds appropriately by moving between security levels. Our project exists at the filesystem levels, and consists of three major parts: i) a context monitor, which runs a suite of tests to determine the current security context, ii) a custom file system based on the ext3 file system, with the ability to handle multiple versions of the same file, which co-exist at a low level, but which may be invisible to a user and applications, depending upon the system's security level, and iii) A set of IOCTL calls which allow the context manager to interact with the file system.

II Motivation

Context-based security in the file system follows in the footsteps of such research as ext3cow, an offshoot of the ext3 file system, which was developed to allow for easier file versioning and auditing which was beginning to be required by legislation [O]. As a file system, ext3cow resides in a sweet spot, where no modification to the kernel is required, and user land applications can be oblivious to the custom file system. This is the primary reason for the placement of context sensitivity in the file system, because costs to the user are minimized. No custom hardware setups or purchases are required, so adopters of this file system incur relatively few extra purchasing costs. No kernel modification is required, which greatly reduces complexity and reduces time costs to users. Lastly, user land applications require no modification, because the changes from within the file system are obscured from users and user land processes.

Security is baked into every level, from NX bits at the hardware level, which mark whether a page can be executed or not [M], to HTTPS, an application layer protocol which provides web site authentication and internet traffic encryption [N].

In addition, context itself is becoming more and more important as users become increasingly mobile. By 2012 Q3, over 1 billion smartphones were in use worldwide, and this proportion is predicted to rise over time by Strategy Analytics [Q]. One of the major markers of this meteoric trend in computing is that users are able to access all sorts of data in all sorts of locations. The rise of social media has made it increasingly easy to find personally identifiable information (PII) about users, and social engineering continues to be the easiest route to getting PII or other sensitive information. Users must, in a sense, be protected from themselves, because it incurs an unreasonable cost to continue to re-educate all users as technology continues to change and progress. It is important, in light of these changes, to adapt security to protect users from themselves, since they are often the weakest link in the security chain [R].

A secure, context-sensitive file system is a broad topic, with many possible applications and consequent threat models. In particular, we aim to address confidentiality by enforcing ascending levels of security. If an object is labeled with security level n, then this object should be inaccessible to all users

when the security level is less than or equal to n - 1. We are assuming attackers have no access to the hardware or root access, since either of these scenarios allows attackers access to the kernel and file system data structures, and completely circumvents our protections. Hardware access attacks could be mitigated by encrypting secure files.

III Related work

III.i quFiles

In 2008, the concept of quFiles was introduced. The authors argued that environmental context was becoming more and more important, as could be seen by the rise in applications designed to act differently depending on such context clues as battery life, or the screen size of a smartphone or tablet. Veeraraghavan et al strove to provide a clear framework to capture the unifying abstraction they believe these individual use cases fell into, and created quFiles. Under the quFiles system, what appears to be one file to user land applications may actually consist of several files. The right file is served to the application on demand, so the system is invisible to, and works seamlessly with, userland programs [D].

Under the hood, what appears to be a file to the application is actually a modified directory, which contains a number of versions of the file to be served, depending on context. Usually there are at least two versions, called views, of the file: a default view and a raw view. The default view is served for file systems and applications that do not understand quFiles. The raw view is a binary representation of the data, which can be used by quFiles-aware applications to create new views. Any number of these custom views is supported, and the authors provide a framework for creating new views by defining context-specific policies, making quFiles very extensible [D].

quFiles has several advantages, which we emulate, such as simplifying higher level applications job by working at the file level. quFiles can also realize performance gains, for instance by using excess storage to hold music files in an uncompressed state and providing this uncompressed version to a music player, saving battery power in smartphones. They can also implement a copy-on-write filesystem [D].

However, quFiles are not realizable as a base for our project because of several disadvantages. First and foremost, all quFile views are completely visible to the user. They are implemented by appending a .qufile to the directory in question, and storing various versions of the quFile within this directory [D]. There is nothing stopping a user from changing to this directory and interacting with the contents directly, in fact, this is encouraged to make quFiles policies more easily writable by third-parties. This is unacceptable for our project, because if a secret user can read a top secret file, then we have failed to provide data confidentiality. Furthermore, the simplistic scheme of appending .qufile to the directory allows for collisions, however unlikely. Collisions may allow malicious users to learn some sensitive data in small increments.

III.ii ext3cow

Where quFiles was created to provide a skeleton for generating context policies at the file level of granularity, ext3cow was created to address versioning and auditing concerns raised by legislation like HIPAA and the Sarbanes-Oxley Act of 2002. The goal of ext3cow is to support lightweight methods to snapshot a file system and maintain different versions of files across time. This is done through a time-shifting interface, which allows users to specify a file and a point in time, from which the file system will retrieve the correct version of the file. The emphasis on cheap and easy snapshots makes system recovery after failure, increasing availability, while helping to improve integrity by maintaining a complete timeline of all edits for a particular file [O].

Peterson and Randal accomplished this by modifying the existing inode structures for directories in ext3 to also contain so-called birth and death data. These birth and death stats, each UNIX epoch

timestamps, give a range of when all files within the directory were last modified. This allows for easy auditing by constructing a timeline for a given file or set of files and the writes to them. They developed a snapshot linux utility, which takes the file or directory to be snapshotted as an optional argument, defaulting to the current working directory. Snapshot will create a backup of all files in the format [filename]@[epoch]. Ext3cow is implemented with no changes to the kernel and no changes to VFS. Only on-disk and in-memory metadata structures have been modified. In particular, an epoch counter, in the form of a 32-bit unsigned integer, was added to the superblock. This counter is appended to snapshotted files, and updated atomically before any writes to data or metadata. There were also changes to the ext3 inode, re-purposing fields for HURD OS and for disk sector alignment to instead contain a 32-bit epoch counter and a pointer to the inode with the next epoch's version of the file. Each version of a file has its own inode, so there might be zero or one inode per file for a given epoch, depending upon whether the file has changed since the previous epoch or not. This all occurs at the file system level, which is a major advantage of ext3cow, as no kernel modification is required, merely the loading of a new kernel module [O].

This snapshotting is valuable for backups and restoring the system to a known good state, and also allows for easy system auditing, but falls short of some of our design goals in similar ways to quFiles. The major concern is that, once again, while different versions of files are transparent to applications because the original file name is preserved when a snapshot file is made, users can easily see these snapshots and interact with them. In addition, the naming convention, which involves appending '@epoch' to the file, is a valid file name itself, and consequently allows for collisions, just as in quFiles.

III.iii NCryptfs

The filesystem level has been leveraged not only for context and file versioning, but also encryption, another necessary component of security. NCryptfs, based on Cryptfs, is an encrypted file system, designed to be both secure and convenient to users. The designers of NCryptfs believed that the reason for the lack of widespread adoption of encryption was because it was cumbersome to use and easy to set up incorrectly. They designed NCryptfs to attach to a directory in any Linux filesystem. The directory and its contents are treated as ciphertext, which NCryptfs will, when presented with the correct key, decrypt into plaintext for the user [E].

This functionality is particularly useful for situations where eavesdroppers might be present, such as networked file systems, which may transmit files and directories in plaintext. It also protects against unauthorized users of the machine, who will see NCryptfs-mounted directories and files as random ciphertext. In order to protect against any information leakage, the authors had to modify the directory cache and the inode cache to remove any possible plaintext, since decrypted data might be left in the cache when data is re-encrypted for one reason or another. They also had to modify Linux process management so that processes don't have sensitive data like keys or plaintext in lingering data structures like the process table [E]. This is of particular relevance to our project since, without encryption, a sophisticated attacker can access files that we try to hide, and also because the same problems involving caches and persistent data structures exist for our system.

IV File System

IV.i File System Design

We looked into several different implementation approaches. These design choices were driven by the needs for portability, security, and performance in that order. Earlier approaches used a data structure within a file system object to mark information. This re-purposing of existing file system structures had the speed advantage, since minimal additional overhead from added data structures is introduced, but also is harder to extent because of the use of borrowed space. Later approaches tend to move farther and farther away from using the leftover bits of existing file system structures, and more

towards adding additional levels to the file system and modifying existing pointers to point to this intermediate layer. This approach is slower, but allows for greater flexibility and extensibility.

The first approach explored was the re-purposing of a bitmap within the inode or extended attributes. This bitmap would contain the security level of each block, and for a security context level i = 0,1,2... file read access will be restricted to all blocks of level i or lower. Writing within the same context updates the bitmap, while writing within a different context requires allocating a new block and insertion into the bitmap, along with page mapping. This approach has a very light memory footprint because all contextual information is stored and updated in a bitmap. Read performance improves as we come closer to the most secure level because it approaches a normal file read. However, as security level diminishes, reads slow down because more and more blocks have to be skipped over. This could be a source of extra information for an attacker, who can purposefully read a file, perform actions which lower their security context, and re-read the file to get a feel for the inner workings of the system. It is also hard to implement this because multi-level bitmaps are complex. In addition, the default ext3 inode structure would have to be modified, and since there is extremely limited extra space in the ext3 inode the functionality of the file system would be limited using this approach. Fortunately, portability with vfs is not affected because the vfs inode is contained within ext3_inode_info, and can remain untouched. However, reducing the available space of the ext3 inode or the extended attributes does reduce portability, and this coupled with the difficulty of implementation make using bitmaps an untenable choice. The option of a hashmap was also explored, which would be easier to implement, with similar read performance. However, a hashmap introduces a much higher memory footprint and would be stored within either the inode itself of the extended attributes.

Because of the disadvantages involved with a bitmap or hashmap, the idea of using multiple inodes was discussed. In this implementation, the block pointers held by the inode would instead point to a series of other inodes. Calls would use the inodes normally until data is needed, at which time the call would be passed to the sub-inode for the particular level. Sub-inodes will each point to the same blocks instead of duplicating data. Since there are no duplicate blocks, when making a change at the lowest security level, the other inodes will be updated to point to any added blocks. When a higher level makes a change then we must check to see if the block is pointed to by any other inodes, and if this is the case we must write to a new block and propagate the pointer to all higher levels.

This scheme allows for very efficient reads for the lowest levels. Also, it resolves the possibility of namespace collisions by working at the block level and having an additional layer of indirection in the form of additional inodes. Also, no structures themselves are modified, merely how the structures are used, specifically what inode pointers point to. However, the reads grow much less efficient as the security level raises because the blocks to be read are in more and more disparate address spaces. Performance will be significantly worse than the earlier bitmap/hashmap-based approaches. It's also hard to implement because le32 block pointer integers must instead point to struct inodes (in memory) or inode numbers (in order to persist). In additional, all calls that read data from an inode must be modified to instead read data from the appropriate sub-inode. The approach also pollutes inode space with an additional, intermediate level of inodes, and breaks portability since code outside the file system is modified. Finally, unless the implementation uses indirect pointers, only 15 different security levels will be supported. For now this is accepted because only 2 different security levels have been implemented, but if this is to be expanded to support multi-lateral security[U], then an arbitrarily large, or at least much larger, state-space would be desired to allow for a more detailed security label lattice. Another, related option is to have the dentry point to multiple inodes. This is essentially the same as the above, except instead of a directory entry holding one inode, it would hold the same number of inodes as there were security levels. Similarly, under this scheme there will be no namespace collisions, similar read performance, and without the additional layer of indirection there will be one less inode used. Modifying dentry comes with the same disadvantages too, however, and in addition either vfs will need to be modified or a wrapper will need to be implemented, along with modifying all calls requiring the d_inode field of dentry (dentry->d_inode), since there will be multiple inodes per file in the dentry.

Keeping in mind the advantages and disadvantages of the above solutions, as well as the limited amount of time to develop a prototype, we decided on using a dummy file to stand in place of any file, and a number of regular hidden files to represent the file at each unique security level. These files all reside in the same directory, with the hidden files following a pattern of [dummy filename]:n, where n = 0,1,2... is the security level. The hidden files are not dot files, but rather they are scrubbed from the read directory operation, so that a user cannot find these files with conventional means. The merits of this approach are that files are an easy to understand data structure, and no kernel data structures have to be modified or re-purposed. Everything is simply a file. Currently, to use the files and still be compliant with VFS, we open and close files within kernel space, which is bad practice. It is hoped subsequent versions of the file system can avoid this. However, since VFS remains un-modified, this is highly portable. However, in order to assure against naming collisions we have to recursively check the names of user created files. For instance, there must be safeguards in place to safely catch a user-created ‘foo:1’ when the security level is 0 and there already exists a ‘foo:1’. The naming scheme for must not be a source of data leakage.

IV.ii File System Implementation

The filesystem implements several basic file operations as well as adding a couple of ioctl operations. Open stores the calling process in a list of processes to kill when the context level changes. Close (flush) doesn’t need to do anything different than the default ext3 implementation. Read (aio_read because sync_read will call it) works by checking to see if the file passed in is a dummy file in which case it opens the correct file, calls read again, and then closes the file, otherwise it performs like the default ext3 implementation. Read directory iterates through the items in the directory and if it finds one that is not a dummy file, it skips over it. Create uses the inode directory and the dentry struct containing the dummy inode that would be created in normal instances and in addition to creating the dummy inode, a new dentry structure is made for each new file by using the dentry->parent name, the dummy name, the separator string, and the security level. Create also propagates the settings of the dummy inode to that of the new inodes created. The first ioctl added is a way to change security levels. If a change of level is detected, the file system will close all opened files and then kill the processes that opened them as well as changing the global information so that opens, reads, writes, etc. will know which file to open. The killing of the processes is done by taking the pids of opened files from a list and sending them to a recursive calling function that keeps recursing on the children list. After getting to the bottom, it climbs back up the function stack sending sigkill signals. The second ioctl added passes in the mount point of the filesystem. This is mostly done as a hack because we couldn’t figure out how to find that information and didn’t want to waste too much time on something with such little importance.

IV.ii File System Challenges

There are several problems that need addressing with the filesystem. First, writes don’t work correctly. That is, with the exception of a shared first block, writes will work more like that of the traditional ext3 filesystem. Second, and by extension, we lack performance metrics. Once write works, we can start running benchmarks and see how much slower this system performs. At a glance, it seems like it wouldn’t be much slower but then again, all we have working are reads. Third, do to the fact that the filesystem uses files as opposed to something else like inodes, we have to call open/close operations on the files from within kernel context to do operations on them. This is generally a bad idea and should be avoided where possible. If we weren’t concerned with portability with VFS, we could avoid this by changing things closer to when the call first gets issued. For example, if we could modify the function that gets called when the open syscall gets called by the user, we could simply modify the string they pass in. Fourth, another problem with using extra hidden files comes from naming collisions. For example, if a file foo exists then foo, foo:0, and foo:1 will really exist within the filesystem. If then, the user creates, foo:0, the

system will have foo, foo:0, foo:1, foo:0:0, and foo:0:1. However, two different files (foo under a 0 context and foo:0 as a dummy file) will both be using foo:0. Possible fixes are invalidate naming of files with our secure string in them, use a more obscure string, lose portability with VFS and modify the call path similar to the example above, or change structure to not use multiple files. Finally, due to lack of experience in modifying file systems as well as speed of trying to output features, there are probably a number of bugs, memory leaks, and race conditions from lack of proper locking. Since the code base is small, this will probably be easy to fix, but these issues will need to be addressed before any amounts of progress can be safely made.

V Context Monitor

V.i Context Monitor Design

The file system manipulates data depending on the context of the system, but the file system does not determine the context of the file system. This is left to a user space context monitor, which passes information to the file system in the form of IOCTL calls. The monitor accumulates information about the system and uses the information to calculate the context of the system. The monitor consists of three parts: i) a main program that polls every n seconds, running each test in the test suite, and then aggregates the calculations performed by the calculators to determine current security context, ii) a test suite of default and user-defined programs which test one facet of security, returning a 1 if they pass (the system is considered secure) or a 0 if they fail (the system is considered insecure), and iii) a set of calculators, one per test. Currently, calculators do not do anything, and could be thought of as scaling the tests return value by 1. In the future, these calculators' behavior may change.

The monitor was designed to be flexible, modular, and extensible. Originally, it was thought that the monitor would hold all of the security tests in one large program. After exploring the issues with this approach, it was concluded that users may want to add their own security tests to the monitor. This would be difficult to do if all of the tests were wrapped into one large program. Any changes to the monitor could break the whole program. Instead, the monitor was designed to accommodate security modules that would be implemented outside of the monitor itself. Currently, the context monitor and test suite are written in Python, a high level scripting language with powerful libraries that make accessing different parts of the system and network very easy.

Context is determined by running all selected tests in the tests suite, which each return a security metric, and then aggregating the security metric to get a general system security level, after which the monitor passes the information to the file system. The tests in the test suite, along with the algorithm used to determine the system's security level, are user defined, although we have provided defaults. Some examples of tests are the USB test, which polls the mount daemon to check for any mounted devices which match the regex '/dev/sd[^a]', since the first attached USB device is generally mounted to /dev/sdb, followed by /dev/sdc, etc. A plugged-in USB device might suggest a user is attempting to exfiltrate sensitive data from the system, and so the USB test returns a 0 if there are any USB devices attached. If there are no USB devices, the test returns 1. A sample security algorithm might take the minimum of all run tests and pass this via IOCTL as the security level. Thus, the USB test might pass, but another test might fail, and any failing test is deemed enough to put the system in an insecure state.

V.ii Test Suite

The context monitor calls all selected tests in the test suite. What tests will be chosen by a given user, and what custom tests a user might develop, is an open policy question. Currently, the emphasis in completed tests is on quickly pulling together relevant statistics about the network, operating system, and physical machine. The USB test uses the Linux mount daemon to check for any mounted devices

matching the ‘/dev/sd[^a]’ regular expression. The first USB plugged into a system typically mounts to /dev/sdb, with the second USB mounting to /dev/sdc, and so on. This test returns a 1 if no mount point match the regex, and a 0 otherwise. The sockets test calls ‘netstat -antp’, parsing the output for any unprivileged ports (port number greater than 1023) which have a status of ‘LISTEN’ or ‘ESTABLISHED’. Open sockets on an unprivileged port are a strong indicator of malicious activity, since connect-back malware and reverse shells often set themselves up on high numbered ports on the victim machine. Granted, perfectly legitimate software may open connections on higher numbered ports as well, so this test can return false positives, but this is left as a policy decision. At any rate, system administrators should know what legitimate services will be using higher numbered ports, and can whitelist this software. The wireless test cross references the commands ‘iwconfig wlan0’ and ‘iwlist wlan0 scan’. iwconfig wlan0 returns information about the current wireless AP connection, specifically the ESSID. However, some crucial information, such as the encryption type, is missing. iwlist wlan0 scan lists information such as encryption type and ESSID for all available broadcasting access points, so the wireless test takes the ESSID of the current connection and parses iwlist for the relevant data. Anything less than WPA2 encryption results in a fail, and the wireless test returns 0. If the current wireless connection is WPA2 encryption, then the wireless test returns a 1. Whether or not this policy makes sense for all users is, naturally, an open question.

Future tests might include a process test, which cross references ps with a user-provided whitelist, and drops to an insecure state if processes not on the whitelist appear in the process list. As with any part of this project, attackers with root access, like rootkits, will be able to circumvent this by hooking into the system calls and removing themselves from the process list. Other valuable tests might involve disk/memory analysis or checksums since last known secure state. In the future, it might be valuable from to move from the current polling mechanism in the context manager to asynchronous callbacks, but it is uncertain if this would be possible for all tests, since we allow for user-defined tests.

V Challenges

Throughout development we experienced several notable challenges, which had to be dealt with either by changing direction, working harder, or narrowing the scope of our project. Scoping was the major issue involved with the project. Initially we planned on implementing an entire file system, context monitor, and test suite and doing performance evaluations for typical file system operations. This quickly had to be narrowed because of the experience level of the group, and we have strived to provide a working throwaway prototype within the semester.

The project was first introduced in the context of using FUSE for implementation. FUSE, or Filesystem in Userspace, is a Linux kernel module, which allows file systems to be developed without having to modify code in the kernel [E]. This seemed like a sound idea at first, and trivial working filesystems in FUSE can be very small and manageable, for example, a “Hello world” FUSE filesystem is less than 100 lines of code, but moving beyond small examples proved difficult, and FUSE documentation is sparse.

In place of fuse it was decided to use IOCTLs to communicate between userspace and kernelspace. IOCTL, or input/output control, is a system call which allows userspace programs access to kernelspace functions [T]. The most pressing issue from a security theory standpoint is information leakage resulting from improperly handling system caches and the names associated with files. For examples, assume two security levels, a secure and an insecure level. If a file secure.txt is written while the environmental context puts the system at a secure level, and a memory forensics tool like Volatility is used to capture a complete image of system memory, then an attacker can see secure.txt regardless of the security state. Because of this, in theory, secure files should be encrypted on disk, and caching of secure data must be treated with the utmost care. As another example of the former issue, a file secure.txt, written when in the secure state, is held within the cache when the security context changes

and the file system drops to the insecure level. How, and when, the cache is flushed are very important for insuring against data leakage. As an example of the latter issue, consider a file secret.txt, which has a secure level attached to it. If a user in the same directory as secret.txt, but with an insecure state, were to perform any CRUD operations on a file secret.txt, then the system cannot i) fail disgracefully with an error message explaining the user has no rights on secret.txt, ii) fail silently by ignoring the user's request, or iii) allow the user to perform their operation on secret.txt. In the first two cases this leaks one bit of information to an attacker, namely that there exists a file secret.txt which he or she is incapable of accessing. In the third case the system is completely compromised because an insecure user has accessed a secure file.

This problem, namely that of information-revealing collisions, is inherent in most of the related work, and must be solved at a low level. Several solutions were discussed, most involving inode manipulation. Another ongoing issue is how to determine the current security level of the system. Security context is, by its nature, open to interpretation. It is usually easy enough to say whether a given context suggests more security or less security, but it is hard to assign a hard number to a given context. In addition, some contexts may not be inherently more or less secure. For instance, does moving from an insecure network to no network at all mean that the user can no longer send unencrypted information, and is therefore in a more secure state, or does it hint that the machine has been moved, perhaps without the user's consent, and is consequently in a less secure state? Just how insecure the mounting of a new USB device is is an open question. The amount of security or insecurity introduced by some factor that a test in the test suite analyzes

VI Conclusions

Although there remain significant unsolved questions regarding our system, we have delivered a prototype with a subset of the proposed functionality. We learned much, mostly about system design, and in the future we would spend more time planning our exact threat model and implementation within the file system. The context manager is largely solved, although a case could be made for moving away from polling towards asynchronous callbacks. The problem here is that, in order for the test suite to be easily extended, the amount of expertise involved with developing a test case must be set as low as possible, and requiring an asynchronous hook might involve too much research for users to take advantage of user-defined tests. Perhaps the best solution would be to allow both, and to optimize certain tests, which would prove prohibitively resource expensive, battery draining or slow if performed via polling to a different mechanism, while still allowing for tests to be run every n seconds.

Whether to use IOCTLs, FUSE, or some other method of communicating with the kernel is still under discussion. FUSE lowers the inherent difficulty by removing kernel modification from the development process, but has documentation and performance concerns. In addition, in order for any product developed from this research to be adopted, the Windows operating system must be supported. FUSE appears to have a port to Windows, and Windows maintains a set of IOCTLs in the Win32 API library [T].

Lastly, the file system design decisions are of utmost importance, since this is the most technically complex part of the project. Great care must be taken to avoid information leakage, but also to maintain system consistency and reasonable performance.

Acknowledgements

References

[A] Blaze, Matt. "A cryptographic file system for UNIX." *Proceedings of the 1st ACM conference on Computer and communications security*. ACM, 1993.

[B] Peterson, Zachary, and Randal Burns. "Ext 3 cow: a time-shifting file system for regulatory compliance." *ACM Transactions on Storage* 1.2 (2005): 190-212.

[C] McDonald, Andrew, and Markus Kuhn. "Stegfs: A steganographic file system for linux." *Information Hiding*. Springer Berlin/Heidelberg, 2000.

[D] Veeraraghavan, Kaushik, et al. "quFiles: a unifying abstraction for mobile data management." *Proceedings of the 9th workshop on Mobile computing systems and applications*. ACM, 2008.

[E] Wright, Charles P., Michael Martino, and Erez Zadok. "NCryptfs: A secure and convenient cryptographic file system." *Proceedings of the Annual USENIX Technical Conference*. 2003.

[F] Veeraraghavan, Kaushik, et al. "quFiles: The right file at the right time." *ACM Transactions on Storage (TOS)* 6.3 (2010): 12.

[G] Michael Mitchell Area Survey

[H] Biba, Kenneth J. *Integrity considerations for secure computer systems*. No. MTR-3153-REV-1. MITRE CORP BEDFORD MA, 1977.

[I] Bell, D. Elliott, and Leonard J. LaPadula. *Secure computer systems: Mathematical foundations*. No. MTR-2547-VOL-1. MITRE CORP BEDFORD MA, 1973.

[J] Wilson, George C., et al. "METHOD FOR SECURE ACCESS TO AND SECURE DATA TRANSFER FROM A VIRTUAL SENSITIVE COMPARTMENTED INFORMATION FACILITY (SCIF)." U.S. Patent Application 12/200,223.

[K] Stoneburner, Gary. "SP 800-33. Underlying Technical Models for Information Technology Security." (2001).

[L] http://source.android.com/tech/security/#memory-management-security-enhancements

[M] http://en.wikipedia.org/wiki/Nx_bit

[N] http://en.wikipedia.org/wiki/HTTPS

[O] Peterson, Zachary, and Randal Burns. "Ext3cow: a time-shifting file system for regulatory compliance." *ACM Transactions on Storage (TOS)* 1.2 (2005): 190-212.

[P] http://www.emarketer.com/Article/Smartphones-Continue-Gain-Share-US-Mobile-Usage-Plateaus/1008958

[Q] http://blogs.strategyanalytics.com/WDS/post/2012/10/17/Worldwide-Smartphone-Population-Tops-1-Billion-in-Q3-2012.aspx

[R] Barrett, Neil. "Penetration testing and social engineering: hacking the weakest link." *Information Security Technical Report* 8.4 (2003): 56-64.

[S] http://fuse.sourceforge.net/

[T] http://en.wikipedia.org/wiki/Ioctl

[U] Needham, Roger, and Scott Mcnealy. "Multilateral Security."