Security Policy Enforcement at the File System Level in the Windows NT Operating System Family

Stephen D. Wolthusen
Fraunhofer-IGD
Security Technology Department
Rundeturmstr. 6, Darmstadt 64283, Germany
wolt@igd.fhg.de

Abstract

This paper describes the implementation of an enforcement module for file system security implemented as part of a security architecture for distributed systems which enforces a centrally administered security policy under the Windows NT operating system platform. The mechanism provides mandatory access control, encryption, and auditing on an individual file basis across distributed systems while being fully transparent to both users and application programs and functioning regardless of the type of file system or its attachment mechanism.

1. Introduction

Providing security at the file system level is part of the basic security functionality provided by virtually all general-purpose operating systems; however, the limitations of file system protection only via data structures maintained by the operating system have long been obvious. One problem is that such protection mechanisms require the operating system providing the service to be active. This assumption can be violated in non-networked configurations in at least two cases. One case occurs when storage media are accessed at the local node, but with an operating system other than the one enforcing the security mechanisms (possibly just another instance of the same operating system, only configured differently), the other occurs when the storage media are exposed; typically another system will not honor protections set for removable media.

The latter is part of a larger problem one is faced with in a heterogeneous distributed environment, namely the lack of enforced conformance to a consistent, centrally enforced security policy focused on individual data objects instead of node interrelations. While this can be achieved partially using current COTS operating systems, it typically requires a homogeneous network environment and unified administration of all nodes in such a network.

What we therefore consider desirable is an architecture to provide access and use control over data objects at a sufficiently high level of abstraction that can be enforced even if the object (e.g. document) is moved across network node boundaries, combined with a comprehensive audit trail encompassing all operations on these objects. To ensure that the enforcement mechanisms are honored in case of a file system — other areas that must be dealt with are network and general I/O mechanisms — we propose to use object labeling in conjunction with encryption at this level as the tool to achieve this goal.

This enforcement mechanism is part of a larger system that segregates security policy determination and decisions based on such policy rules from enforcement [18, 10, 12], but does so at the level of network nodes, i.e. there exists a set of servers that provide consistent policy information to all systems within the protected set of nodes.

This paper concentrates on a description of the implementation of the file system security enforcement component on the Microsoft Windows NT family of operating systems; this mechanism has been under ongoing development by our group since early 1998. First working prototypes were finished in 1998; the mechanisms and implementation have been evolving constantly since.

The background and basic concepts of the system are discussed in section 2, followed by an overview of the Microsoft Windows NT file system I/O architecture in section 3 and a discussion of the implementation of the enforcement mechanism in section 4.

2. Background

Security architectures and mechanisms reaching beyond addressing problems found in individual areas such as network security or file system security have mainly been pur-

sued as implementations of entire operating environments with security as their focus [13]. While this is desirable for a number of reasons, particularly a high degree of assurance that can be achieved with such an architecture such as the Flask/Fluke [16] architecture, such an approach is limited in its immediate appeal since a migration of hosts and especially application programs to the new environment would be required. It is, however, imperative that security mechanims are implemented at the operating system level to be meaningful [5].

In the design of the architecture described here we have therefore attempted to ensure that the required security mechanisms can be retrofitted to existing operating systems. To ensure interoperability with existing COTS and custom applications, these modifications and security enhancements must be kept invisible to both applications and users — at least while they are operating within the limits set by the security policy; due to typically less than robust error management one must also ensure that failures closely mimic behavior in case of failures that such applications expect.

Since the security architecture must function in a heterogeneous, networked environment, it is necessary to protect a number of aspects of the system, namely the file system, network, and general I/O mechanisms. One must assume that each individual node is exposed to a potentially hostile environment and peer nodes which are not necessarily equipped with the same security enhancements. At the same time it is necessary to enforce a consistent and concurrent view of a security policy that is identical across all nodes, principals and objects under the nominal control of such a policy.

The architecture described here deals with these issues in a two-pronged approach. The first guiding principle is the separation of policy from enforcement; consistent enforcement is assured by performing this separation within the network: nodes called ERMs (externally controlled reference monitor) provide consistent policy information while all other nodes enforce this policy at the operating system level based on decisions either related to the enforcement subsystem directly from a node distributing policy data and decisions or as a result of a delegated derived security policy.

This separation of concern between end nodes enforcing security policy and nodes controlling policy data can be performed using externally controlled reference monitors (ECRM). Using a proper balance between centralized decisions and locally delegated security policies, the overall network load — which is limited by latency, not bandwidth — can be kept at an acceptable level. Details on the ECRM mechanism can be found in [18].

The second element of the architectural approach is to use a layering of abstraction levels when dealing with specifying a security policy. To permit a basic protection mechanism that can be specified easily, an outer layer of coarse granularity is necessary that surrounds the entire node. This implies all inputs and outputs that are handled by a node, particularly all network traffic and file systems regardless of their type or physical location. Refinement of the lower granularity controls can then occur by analyzing, modifying, and controlling activities at higher semantical levels such as sequence analyses of application and user behavior (see figure 1). This has several pragmatic benefits, the most important of which is the reduction in the required minimum complexity of the security policy and the default-deny stance that can be implemented easily. Other approaches [2] require a detailed specification of permitted behavior of principals; this can become an issue once reasonably complex application programs (e.g. COTS word processors) must be fitted into a policy. Details on the layering approach as well as on networking aspects of this architecture can be found in [19].

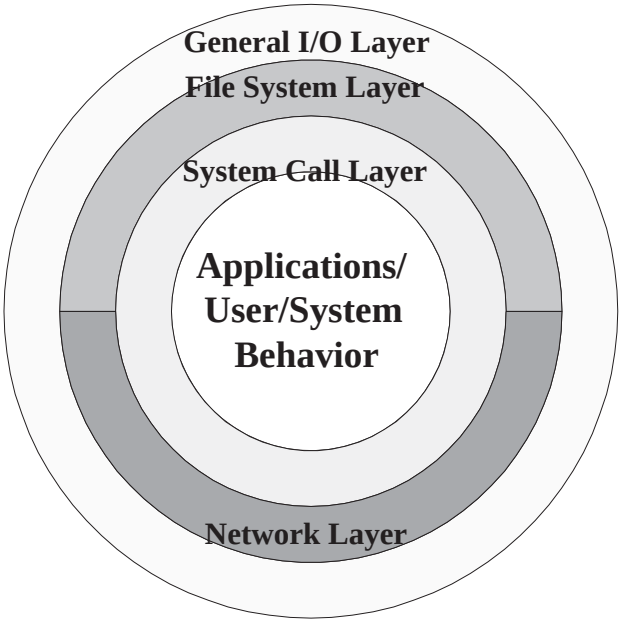

Figure 1. Layering of Protection Elements

In case of the system component described here, the security architecture must detect the use of file systems as opposed to direct access to lower-layer interfaces or other access semantics and permit file system mechanisms to cross the general I/O protection layer (and if necessary, as is e.g. in the case of file servers or network attached storage, the network layer) since policy control in such cases is best enforced with the semantic information at the file system layer unless an even higher level abstraction layer can be identified for an operation.

2.1. File System Protection Mechanisms

In the architecture described here, files (as well as other objects such as network streams) are affixed with a label that is handled by the security subsystem and is transparent to the remainder of the system (i.e. both lower and upper layer drivers of the host operating system as well as application programs and users are oblivious to the labeling mechanism). These labels are protected against manipulation by being tied to the content (i.e. unique identifying characteristics) of the object; the actual policy information is contained in either the ERM node providing policy information or is temporarily delegated to the (protected) security subsystem, the ECRM. To ensure enforcement even if the security subsystem is inoperative, automatic encryption can and should be used on the contents of thus labeled objects. The encryption mechanism also necessitates the provision of policy information in case an object is transferred to another node.

There is one category of file objects to which labels must not be affixed; this category consists mainly of files which are required for bootstrapping the entire system and which are accessed prior to the enforcement system being loaded. Even if such files were only labeled and not encrypted, this could lead to unpredictable results.

As a general rule, however, each file object must carry an object label even if only to identify the object. Based on the identity of the file object, the ECRM can then determine what – if any – actions must be applied to a given file based on the security policy or security policies applicable to either the object or to an operation.

In a simple implementation this policy could consist only of enforcing access controls (e.g. discretionary access control) that are independent of the host operating system and enforced consistently for identified objects across an entire distributed system. An application program or other process attempting to access such a file would be confronted with an error code matching regular error codes for the operating system.

Security models requiring classification can be implemented by specifying a security which requires rewriting or creation of object labels that assign an object label identity belonging to an identity class hierarchy reflecting the classification level. Other models and mechanisms such as RBAC and domain and type enforcement are also subsumed by this mechanism.

However, if enforcing information flow controls is of concern, mere access control is, as described in section 1, insufficient even when coupled with mandatory labeling and classification.

Instead, encryption must be used under such circumstances. This can be done transparently within the file system protection mechanism, ensuring that only after passing through a trusted subsystem information is made available to a process (subject) also under the control of the security policies being enforced.

For this purpose, as with other decisions regarding operations, the ECRM can – depending on what the overall security policy dictates – either query an authoritative ERM for a decision, possibly including key material, or it can resort to consulting a local cache of policy decisions and rules for which an authoritative ERM has specified a lifetime.

The file system layer has access to other important information. This information correlates users and files they are using. We need to distinguish three types of files. The first type of file is the executable file as seen by the operating system. Such executables, which usually consist of several parts (a main file and a number of dynamically loaded shared objects or dynamically linked libraries), can be identified and matched against security policy rules containing approved applications. The second type of file is harder to identify when located at the file (operating) system layer and involves all scripting languages, i.e. mechanisms that involve files classified as non-executable by the operating system but executed by an intermediate application program. This class of applications includes macro languages found in many applications and has been the source of a large number of successful attacks. Here only heuristics and elaborate checks can attempt to identify and protect against malicious code. The third type of file consists of plain data objects. This information can be combined with other information collected at different layers. In particular, the integration of the file system layer permits the dynamic "sandboxing" of applications.

One example of such sandboxing in case of a MLS-like policy is the dynamic restriction of a process from making certain network connections once it has accessed a data object whose classification label does not match with the classification of a given network peer. The same mechanism obviously also is applicable to operations within the node local file system and can be used to implement a purely local MLS configuration. In most cases, however, caching and common networked file systems will require coordination of policy across node boundaries.

3. File System I/O Structure in Microsoft Windows NT

The Microsoft Windows NT family of operating systems [14, 15] exposes several APIs via environmental subsystems. While these APIs are largely procedural in nature, the internal processing is asynchronous and packet-based in nature. In this regard, it shares more with OpenVMS [3] than with Unix [4], although one major difference to OpenVMS is that, like Unix System V Release 4 and later derivatives,

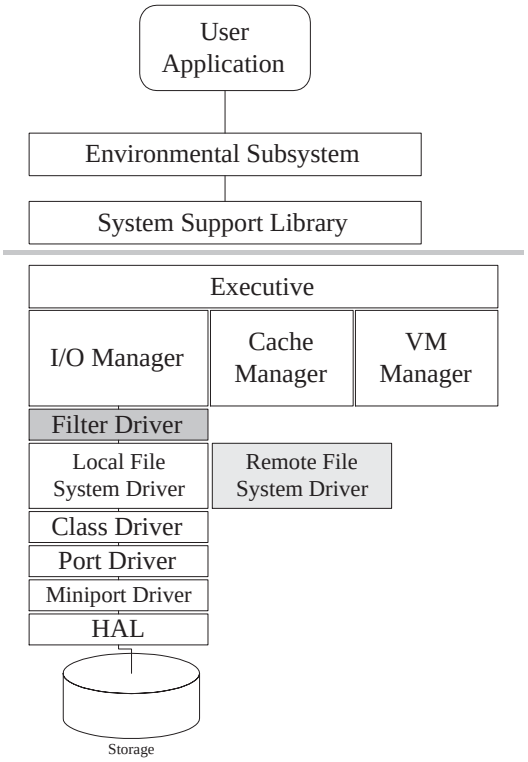

Figure 2. Components involved in file system I/O

it has a unified file system cache and virtual memory architecture.

Regardless of which environmental subsystem is used, the I/O operation eventually results in a call to the system service dispatcher in kernel mode. This dispatcher handles the distribution of the operations into the various kernel components[1]. For the discussion here, only some components are of interest. Besides the I/O Manager, the Windows Management Instrumentation (WMI), Plug and Play (PnP) Manager, and the Power Manager components (these appear only with Microsoft Windows 2000 and later revisions) are also relevant for device level operations.

The central component, however, is the I/O manager. It creates I/O request packets (IRP) from incoming requests[2] and ensures that all drivers for which an IRP is relevant are called with the IRP in the proper sequence. Each IRP sent to a kernel-mode driver represents a pending I/O request to that driver. An IRP will continue to be outstanding until the recipient of the IRP invokes the `IoCompleteRequest()` service routine for that particular IRP. Invoking `IoCompleteRequest()` on an IRP results in that I/O operation being marked as completed, and the I/O Manager then triggers any post-completion processing that was awaiting completion of the I/O request. Each request must be completed exactly once.

This mechanism lends itself to a layered processing approach in which IRPs are cascaded across several driver layers (possibly with additional IRPs created along the way at lower levels). As a side effect of this architecture, one can alter the functionality of the operating system by interposing additional layers in the driver stack. One example of such an interposition is shown in figure 2.

The placement of the filtering layer in figure 2 has the advantage of such a module being able to intercept and operate on generic (file-system independent) operations from upper operating system layers; this type of filter is called a file system filter driver. Most importantly, the depicted interposition layer allows operations on the file level. Common disk encryption mechanisms typically work by adding special disk drivers or lower level filter drivers; as a result they are dependent on specific hardware or are not able to work on individual files; in addition, they do not support remote file systems. While handling files individually entails a significantly higher complexity, it is necessary to support the semantics found in the system described here. Interposition at this level is also largely[3] oblivious to the type of file system.

Microsoft Windows NT does not fully adhere to the packet-based I/O model for all types of drivers, though. A special case exists in case of file systems, therefore also for file system filter drivers. This exception is the Fast I/O mechanism; here the I/O Manager, Cache Manager, and the various file system implementations (if they support this mechanism) interact by means of explicit cross-module calls instead of creating IRPs. This performance enhancement adds considerable complexity to the design of any file system filter drivers since additional communication paths must be handled. While it is possible for a driver (particularly a filter driver – this has the result that lower-level drivers are also not confronted with Fast I/O for a given call) to signal that Fast I/O is not supported with the result that an equivalent request is created in the form of an IRP and sent again by the system service dispatcher, the double performance penalty thus incurred is not justifiable.

Microsoft Windows NT uses the filter driver mechanism — not only for file systems — itself to support additional functionality that is optional or can be made available for different file system types with a single driver; one example in Microsoft Windows 2000 is the Single Instance Store

[1] Since Microsoft Windows NT 4.0, graphics interfaces are part of the kernel mode and bypass this mechanism. The DirectX family of APIs even allows bypassing of normal operating system protection (memory and devices) by user mode appliations.

[2] With the exception of Fast I/O which bypasses this step, loosely patterned after the OpenVMS concept by the same name

[3] There are some differences in behavior for remote file system redirectors

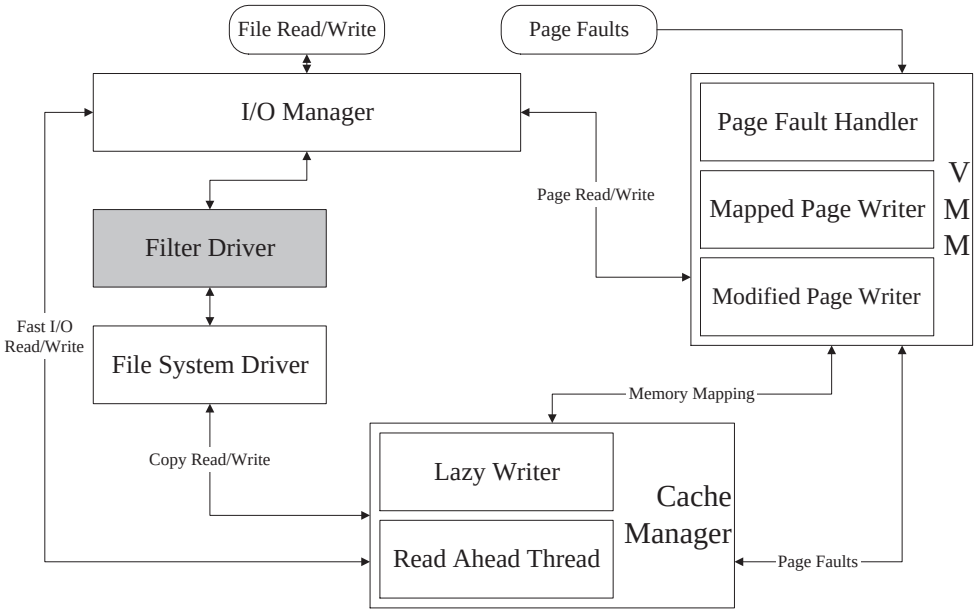

Figure 3. Interactions between file system components

(SIS) file system filter driver that conserves disk space by removing multiple copies of a file and replacing them with links to a single shared copy in a common directory. Another application example of a file system filter drivers is a virus scanner; again, this type of application requires access to file system semantics.

Some care needs to be taken in case a file system filter driver modifies data on underlying file systems since the unified cache and VM architecture results in Fast I/O requests bypassing the regular file system (filter) and accessing the cache directly as shown in figure 3. Failing to update all relevant access paths could thus lead to inconsistencies.

Some additional details can be found in [8, 9], although regrettably most of the internal interfaces that need to be supported are largely undocumented. It should, however, be noted that apart from issues arising from defects removed and occasionally introduced and some minor additions, the internal file system APIs of this platform have remained relatively stable despite major changes in the portions of the system visible to most users and developers.

4. Implementation

This section discusses several aspects of implementing the transparent file system security mechanism by means of inserting a file system filter driver into the operating systems of the Microsoft Windows NT platform.

4.1. Structure and Bootstrapping

Each kernel mode driver must provide a `DriverEntry()` function that is called by the I/O manager on driver load. This function performs initialization operations of the driver such as reading parameters from registry settings, allocating data structures, `DeviceObject`, and `SymbolicLink` objects, and initializing the call table (`MajorFunction` table).

The `MajorFunction` table is a list of dispatch points supported by the driver. Each I/O request is packed into an IRP by the I/O manager and contains all information describing the request including the desired operation (the `MajorFunction`). The driver may set an entry in this table for each `MajorFunction` it wishes to process with its own dispatch functions.

The enforcement driver architecture consists of two parts, namely a filter driver dealing with file-system specific parts and another kernel module or driver (i.e the ECRM – which is realized as a pseudo device driver, but is in fact accessed by direct kernel mode calls instead of using an IRP-based mechanism) which contols the actual policy operations such as deciding on access rights or encryption and decryption. This latter driver is called on by the file system filter driver for all intercepted calls on the file system and encapsulates all necessary operations such as communication with a cryptographic coprocessor which may house the

actual ECRM mechanism.

As a result, no cryptographic operations or other operations related to policy are visible outside of the ECRM. If such operations are delegated to a tamper-resistant cryptographic coprocessor, this ensures that the policy decisions and key material used for enforcing such decisions cannot be modified unless considerable effort is expended.

The filter driver mechanism must be logically located on top of the driver modules whose behavior it wishes to change, augment, or replace. It is also possible for such a filter driver to create new calls to (among others) such lower layer drivers as may be the case if some more elaborate information or modification to the file system are required. The filter driver described here is located above the file system and is therefore able to work on and identify individual files and does not deal with individual device types. File system filter drivers operating below the file system driver level (but above storage class drivers) are somewhat misnamed as they are only capable of working on amorphous data blocks without file system semantics in the calls reaching such layers as file operations are broken up into block--based operations by the respective file system drivers.

The ability to operate on files allows the implementation to gather information on the entities wishing to perform the respective operation and forward this information to the ECRM for further processing. If necessary, each individual file can therefore be treated differently if possesses an identifying feature and the security policy or security policies to be applied in such a case dictate this behavior.

To ensure that the security policy is enforced uniformly, all file systems on a node must be intercepted and brought under the control of the security system. This is achieved by registering a callback function with the I/O manager which is called whenever a file system is loaded. This ensures that the filter driver can attach itself to all file systems, even those that are loaded dynamically after booting. Dynamic loading of file systems can, for example, occur when removable media are loaded. An implication of this is that the filter driver must be loaded prior to all file systems. This can be achieved by assigning it either to the “Filter” driver group or associating the necessary tag value with it in the registry settings for the driver load sequence.

The only file system for which such a filter driver is not notified of a load event is the Raw file system (permitting access to the raw device without any file system interpretation). In this case the filter driver must attach itself explicitly to this file system. Another exception from the notification mechanism that must be dealt with explicitly is the LAN Manager redirector used for accessing network shares. This, however, appears to have been an oversight by the developers of Windows NT since this behavior is no longer observed under Microsoft Windows 2000.

Once the notification callback is called, the filter driver can attach to the file system or file system recognizer, respectively, and is then able to intercept the file system control requests (with the minor functions `LoadFS` and `MountVolume`) and attach itself to mounted volumes. Once it is attached to a mounted volume, the filter driver can intercept all necessary I/O requests.

4.2. Considerations for modified read and write behavior

As a consequence of the unified virtual memory and file system architecture in the Microsoft Windows NT operating system family, it is not sufficient to modify only the behavior of read (`IRP_MJ_READ`) and write (`IRP_MJ_WRITE`) operations. Doing so would lead to a partially encrypted (or otherwise modified by the filter driver) cache since in addition to simple read and write requests, memory mapping operations (e.g. used for mapping executable files into memory) would go a different route from ordinary operations. In case of smart read-ahead by the file system or of explicit mapping, there would also arise a possible inconsistency. Paging requests therefore must also be handled.

4.3. File size considerations

The actual file sizes on the file system that is intercepted by the filter driver and the file sizes reported to the upper levels of the system and eventually to the user-mode API may differ from one another.

There are two reasons for this. One is that it is necessary to maintain an in-line object label identifying the data object (file) for administrative purposes. This could be handled more elegantly for file systems supporting multiple data streams such as NTFS, but since removable media (e.g. FAT floppy disks) and network file systems (e.g. NFS file servers) do not necessarily offer this feature, the more cumbersome in-line mechanism must be used. The second reason is that in case an encryption mechanism is employed for a data object, the algorithms used may dictate padding to a certain multiple of bytes; the end of the padding must also be stored in-line.

Neither the object label nor the padding data size change may be exposed to the upper levels of the driver and executive architecture.

As a result, the file system filter driver must adjust the file size reported by the underlying file systems in IRPs such as `IRP_MJ_QUERY_INFORMATION` possibly used for obtaining size as well as `IRP_MJ_SET_INFORMATION`, which could be used to adjust file length information, and the similar `IRP_MJ_DIRECTORY_CONTROL` and `IRP_MJ_QUERY_DIRECTORY` requests.

The padding and object label information is stored at the end of the file stored on the lower-level file system. This

avoids complications with memory-mapped files and continuous offset adjustments that would be necessary if the information were to be stored as a header since otherwise page-sized requests from upper levels of the system would in fact straddle page boundaries and thus incur a significant performance penalty. To the file systems below the filter driver, however, the data looks like the contents of an ordinary file.

Another issue the filter driver must deal with is the locking of files by applications since it may have to modify the file as a result of actions by upper layers that would not require file modification without the presence of the security mechanisms. To be able to access such files, a so-called locking key must be known. Therefore, `IRP_MJ_LOCK_CONTROL` requests also must be intercepted.

4.4. Information gathering and processing

The processing of `IRP_MJ_CREATE` requests is of critical importance. This IRP is issued when a file is accessed for the first time (not just for file creation) by an upper level function, so it is possible to perform a number of bookkeeping tasks at the same time as the opening. For this purpose, the filter driver must itself issue IRPs to the subordinate file systems, as well as communicate with other layers and modules within the security system. Subordinate requests are usually necessary to read in the object label (if present) at the end of the file. Based on the object label, the object is classified within the internal bookkeeping mechanisms together with other relevant information regarding the file (e.g. files or memory maps shared with other processes) and further processing can be determined based on the security policy at the associated driver providing policy information (e.g. whether transparent en-/decryption on subsequent read/write requests is required). The information on the file stored in this step is referenced in any further processing of the file since gathering the necessary information would – even disregarding performance issues – not be possible due to restrictions on subordinate IRPs especially during paging operations.

Another request that must be dealt with is `IRP_MJ_CLEANUP`. This request typically precedes the closing of a file. In case the object label itself must be changed or if the size of a file has changed, the new object label gets written during the processing of this IRP.

The `IRP_MJ_CLOSE` request must also be processed since the internal bookkeeping data structures associated with the file must be released.

This information gathering mechanism can be triggered implicitly by opening files and then querying the filter driver from the central policy enforcement mechanism; in addition, explicit functions such as the calculation of signatures over applications can also be triggered at the kernel level without interactions with any user level components.

4.5. Selective use of enforcement mechanisms

Even though the enforcement mechanisms are loaded very early during the boot process, there are some files in Microsoft Windows NT which must be processed prior to the driver becoming active (e.g. `ntldr.exe`, `ntdetect.com`, and `pagefile.sys`). This means that such files must not have an object label and also must not be encrypted. However, other processing, such as signature verification which does not involve intrusive changes to the files is still possible.

Another file class that may not have an object label and generally should not be encrypted is paging files. The latter is mainly due to performance reasons; depending on the threat model it might be acceptable to wipe the paging files on each — orderly — system shutdown.

In addition, the security policy might dictate that certain files, directories, or even entire volumes (volumes are a subcase of directories) are not subjected to labeling and encryption.

For this exclusion mechanism to work, the full names of the files must be kept at hand; since the Microsoft Windows NT stores the device identity separately from the path and in a different format inside the file object, the filter driver must maintain a translation table to minimize overhead during lookup comparisons.

An additional complication arises in conjunction with removable media; simply querying the root directory without a medium being present would result in an error condition. Instead, we resorted to an undocumented function for querying the properties of symbolic links; this works since the “drive letters” in Microsoft Windows NT are implemented as symbolic links.

5. Related Work

Early work on segregating policy decisions from enforcement was performed at UCLA [17] and also in the LOCK project [11]. The DTOS project also dealt with this concept [7] based on a Mach microkernel architecture.

Using an encrypted file system is a transparent mechanism for enforcing a security policy regardless of who has access to the physical file system that does not place an undue burden on users. This has been recognized and both research [1, 6, 20] and commercial (e.g. Microsoft EFS, SoftWinter SeNTry) implementations have resulted. However, the goal pursued by all of these implementations was mainly to provide individual users (or at best small groups of users) a more convenient mechanism for encrypting their personal data and protecting it from other users.

Another alternative, mainly pursued by commercial vendors, is to encrypt entire media regardless of file systems. While this provides protection against theft of storage media, the main drawbacks are limited compatibility and a lack of support for storage accessed via interfaces not intercepted by such mechanisms (e.g. network file systems).

An example of a pass-through filter driver for monitoring file system activity on Microsoft Windows NT 4.0 was described in [9]; some third-party information on the workings of file system drivers and file system filter drivers is found in [8].

6. Conclusions and Future Work

We have presented a mechanism for enforcing security policy at the file system level retrofitted onto the Microsoft Windows NT family of operating systems. It enables the enforcement of a security policy without requiring modifications to applications and only limited changes in user behavior. The mechanism was developed without requiring access to the sources or modifications of the host operating system.

While this mechanism can be used for file system encryption and access control by itself, it becomes fully operational only when tightly integrated with other security policy enforcement components and a mechanism for ensuring consistent enforcement throughout a distributed system.

To enable the use of the system in a heterogeneous environment we are also developing the enforcement mechanisms for other operating system platforms, with Unix System V Release 4 derived platforms (Sun Solaris, SGI IRIX) as the lead system.

Our future work will be directed primarily at tight integration of the individual policy enforcement modules as well as on providing scalability to ensure that the system can be deployed in very large ($>$ 100,000 nodes) networks.

An additional topic of research currently under way is the realization of a policy mechanims of sufficient expressiveness to model even complex security models and specific requirements.

References

[1] M. Blaze. A Cryptographic File System for Unix. Technical report, AT&T Bell Labs, Nov. 1993.

[2] T. Fraser, L. Badger, and M. Feldman. Hardening COTS Software with Generic Software Wrappers. In *Proceedings of the 1999 Conference on Security and Privacy (S & P '99)*, pages 2–16, Los Alamitos, CA, May 9–12 1999. IEEE Press.

[3] R. Goldenberg and S. Saravanan. *Open VMS AXP Internals and Data Structures: Version 1.5*. Digital Press, Maynard, MA, USA, 1994.

[4] B. Goodheart and J. Cox. *The Magic Garden Explained: The Internals of Unix System V Release 4*. Prentice Hall, Englewood Cliffs, NJ, USA, 1994.

[5] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell. The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments. In *Proceedings of the 21st National Information Systems Security Conference, Crystal City, VA*, pages 303–314, Oct. 1998.

[6] E. Mauriello. TCFS: Transparent Cryptographic File System. *Linux Journal*, 40, Aug. 1997.

[7] S. E. Minear. Providing policy control over object operations in a Mach based system. In USENIX, editor, *5th USENIX UNIX Security Symposium, June 5–7, 1995. Salt Lake City, UT*, pages 141–155, Berkeley, CA, USA, June 1995. USENIX.

[8] R. Nagar. *Windows NT File System Internals: A Developer's Guide*. O'Reilly & Associates, Sebastopol, CA, USA, 1997.

[9] M. Russinovich and B. Cogswell. Examining the Windows NT Filesystem. *Dr. Dobb's Journal of Software Tools*, 22(2):42–50, Feb. 1997.

[10] O. S. Saydjari, J. M. Beckman, and J. R. Leaman. Locking Computers Securely. In *Proc. 10th NIST-NCSC National Computer Security Conference*, pages 129–141, 1987.

[11] O. S. Saydjari, J. M. Beckman, and J. R. Leaman. LOCK trek: Navigating uncharted space. In *Proc. IEEE Symposium on Security and Privacy*, pages 167–175, 1989.

[12] O. S. Saydjari, S. J. Turner, D. E. Peele, J. F. Farrell, P. A. Loscocco, W. Kutz, and G. L. Bock. Synergy: A distributed, microkernel-based security architecture. Technical Report version 1.0, National Security Agency, Ft. George G. Meade, MD, Nov. 1993.

[13] E. J. Sebes. Overview of the architecture of Distributed Trusted Mach. In USENIX, editor, *Proceedings of the USENIX Mach Symposium: November 20–22, 1991, Monterey, California, USA*, pages 251–262, Berkeley, CA, USA, 1991. USENIX Association.

[14] D. Solomon. *Inside Windows NT*. Microsoft Press, Bellevue, WA, USA, 2nd edition, 1998.

[15] D. Solomon and M. Russinovich. *Inside Windows 2000*. Microsoft Press, Bellevue, WA, USA, 3rd edition, 2000.

[16] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask security architecture: System support for diverse security policies. In *8th USENIX Security Symposium*, pages 123–139, Washington, D.C., USA, Aug. 1999. USENIX.

[17] B. J. Walker, R. A. Kemmerer, and G. J. Popek. Specification and Verification of the UCLA Unix Security Kernel. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP)*, pages 64–65, 1979.

[18] S. Wolthusen. Enforcing Security Policies using Externally Controlled Reference Monitors. Submitted for publication.

[19] S. Wolthusen. Layered multipoint network defense and security policy enforcement. In *Proceedings from the Second Annual IEEE SMC Information Assurance Workshop, United States Military Academy, West Point, NY*, pages 100–108, June 2001.

[20] E. Zadok, I. Badulescu, and A. Shender. Extending File Systems Using Stackable Templates. In *Proceedings of the 1999 USENIX Annual Technical Conference (USENIX-99)*, pages 57–70, Berkeley, CA, June 6–11 1999. USENIX Association.