Goalkeeper: Close-In Interface Protection

Stephen D. Wolthusen
Fraunhofer-IGD, Germany
wolt@igd.fhg.de

Abstract

This paper discusses a potential security issue in common operating system and application environments regarding dynamically attached devices and device interfaces.
A set of countermeasures for the identified threats is described along with the integration of countermeasures into a policy-based security infrastructure; finally, an implementation of the countermeasure in the form of a policy enforcement module integrated into the kernel of the Microsoft Windows 2000/XP family of operating systems is described.

1 Introduction

The security models underlying most operating systems assume that certain resources such as memory and, to a more limited extent, storage are under the exclusive control of the operating system. It is typically only in network communications that the need for identification and authentication of remote entities and the protection of the communication channel against e.g. eavesdropping and manipulation is explicitly considered.
Even there, though, such controls are not always employed consistently — particularly in using wireless communication this can pose a severe risk in itself.
However, a number of interface types on general-purpose computers have emerged in recent years that share the dynamic properties formerly associated only with network interfaces; it is those interfaces not immediately or not at all associated with network protocols that this paper is concerned with.
The fact that these interfaces require physical contact (such as in the case of USB or IEEE 1394) or are essentially line-of-sight (e.g. infrared and Bluetooth) should not distract from their inherent threat potential.
We contend that the device interfaces and dynamically attached devices can provide direct or indirect access to information or mechanisms for modifying the behavior of the systems to which they are attached that is not dealt with adequately or with requisite flexibility by the security mechanisms in place at either the operating system or the applications typically dealing with such devices.

One example of such an operation is the use of data synchronization mechanisms for PDAs or similar devices; it is trivial to initiate a synchronization (i.e. transfer) of otherwise protected data from a workstation to an untrusted device.

Such an operation can, depending on the interface used, be performed clandestinely within seconds while the legitimate user of the workstation does not notice or is absent from the workstation's console (which can e.g. occur in a populated area such as an airport lounge). Moreover, the fact that such a data transfer has occurred may not be readily apparent and visible only in audit data that is likely to be used infrequently if at all.

Manually disabling and re-enabling of application programs or interfaces altogether can protect against such attacks, but is highly inconvenient to the user, and above all likely to be omitted in situations other priorities interfere even if the user is sensitive to the risks associated with such interfaces. Similar problems arise with other coarse-grained separation mechanisms.

Section 2 discusses several threats to the confidentiality, integrity, and availability of systems and their data emanating from the use of dynamic interfaces and dynamically attached devices in current COTS operating systems.

Section 3 then describes a set of countermeasures to mitigate the risks identified within the context of such COTS systems — the subject of the Goalkeeper component proper — and discusses the integration of such mechanisms within a larger policy-based architecture, while section 4 covers a reference implementation based on modular kernel extensions to the Microsoft Windows 2000/XP/2003 family of operating systems[1]; section 5 discusses a selection of related work. Finally, section 6 provides brief conclusions and an outlook on related and future work.

[1] In the interest of readability, the term Microsoft Windows in the following refers to the entire Windows 2000/XP/2003 family.

2 Threats

Regardless of the operating system type used, several broad threat categories can be identified, some of which depend on the type of interface and level of dynamism in the respective operating system.

2.1 Application control

The model used for allocating various devices attached via interfaces such as serial (RS-232C), parallel (IEEE 1284), USB, FireWire (IEEE 1394), and Bluetooth in most currently dominant COTS operating systems including various Unix derivatives, Linux, and the Microsoft Windows NT family of operating systems permits the association of application programs with said devices and interfaces.
This model is generally limited to access control mechanisms operating on the device interface as the sole entity (e.g. access control to a device object under Microsoft Windows or device special files under Unix derivatives).
As a result, the enforcement of any security policy (e.g. access controls) typically falls in the area of responsibility of the application program or the user of the application program. Thus, if an operating system otherwise enforces security policies with regard to other external interfaces such as network interfaces and storage intefcaes including ones for removable media, this leaves a gap in the enforcement mechanism suite that could be exploited by both malicious users and external threats.
The example scenario discussed in section 1 reflects such a threat, here an application (the synchronization software) implicitly assumes that any device present is authorized to receive and transmit data. Similar problems arise with other storage and communication devices such as modem interfaces that are also fully under the control of an application.

2.2 Identification and authentication

Both common operating systems and application programs typically do not identify and authenticate devices and application programs (or users), regardless of whether the device is configured statically or dynamically (see section 2.3). In many cases the underlying devices do not provide for such mechanisms themselves (e.g. in case of human interface devices attached via radio or infrared interfaces, these commonly employ only limited disambiguation), which can lead to undesirable interactions at both functional and security levels.
In other cases, the identification and authentication mechanism does not, in addition to potential weaknesses e.g. in the strength of cryptographic mechanisms, establish the identity of the communicating entities at the semantic level required. As an example, the Bluetooth pairing mechanism[2] establishes only the knowledge of the PIN code, not the identity of a device or even of a subject controlling such a device.
In the example scenario discussed in section 1, common PDA software performs an identification verification on initiation of a data synchronization process by the user, but does not authenticate this information. As a result, no additional user intervention (e.g. if a legitimate user is absent and has locked the console) is required, and an attacker can trivially prepare a PDA with the requisite user identity derived heuristically or from unrelated communication.
Similar threats arise from devices in bus or broadcast configurations taking over unauthenticated device identifiers without causing reconfiguration to take place (see section 2.3); while such operations may be detected in case of simultaneous operations of both the legitimate device and the attacker's device, human interface devices are particularly susceptible to this type of attack since the shared medium (bus or wireless broadcast) is rarely contended.
Taking over an unused or temporarily dormant identifier can also be used in an attack, e.g. for eavesdropping on USB bulk data transfers between a host and a legitimate device such as those provided e.g. by the KeyGhost[3] product.

2.3 Dynamic configuration

A crucial feature permitting attacks described above and one that is potentially problematic in its own right is the dynamic and automatic configuration mechanism for new devices and devices instances integrated into operating systems. In most Unix derivatives this is somewhat limited, and although e.g. the Sun Solaris USB framework includes nexus drivers supporting mass storage profiles and hence also has full volume manager support for USB-based mass storage and removable media, most Unix systems and Linux depend on individual device drivers to support one or more dynamically (or Plug-and Play, PnP) configured devices.

The Windows NT family of operating systems, however, includes extensive support for PnP and therefore faces several threats that do not exist in the previously mentioned systems. Here, devices found both during the boot process and identified at runtime are activated. This occurs regardless of the privilege level of the user or users currently logged in.
If, for a given device, no device driver is currently loaded, the PnP manager will attempt to install a driver for such an identified device. It is particularly noteworthy that if the system contains the setup components for the device (which

[2] Establishing a shared secret (a PIN code) for symmetric channel encryption and authentication using an out-of-band mechanism

[3] A family of products by KeyGhost Ltd., Christchurch, NZ, which perform purely hardware-based human interface device action logging, corporate URL: `http://www.keyghost.com`

is frequently the case since the Windows NT family by default includes a repository of device drivers and setup components on installation), even this reconfiguration will occur regardless of the currently active users (see figure 1).
Such device drivers, even if they are not Trojan horses installed by an adversary, may cause undesirable interactions with existing components or permit the attachment and operation of devices along with application programs and system services automatically installed along with the device driver by the setup components (which operate at administrative privileges) that contradict security policies in effect for a given system.
It is therefore possible to induce a demonstrably insecure system state by having a system recognize an additional or new device without requiring the presence and actions of an authorized user or even elevated privileges.

3 Countermeasures

While for some of the interface types discussed here, disabling of devices or device types at the level of the operating system can be adequate (e.g. enforcing access controls to device objects or device special files), this assumes that a given user or all users for a given system will have no legitimate use for the interface and all devices that may be attached to such an interface.
Particularly in case of bus-type interfaces that may also have required (e.g. human interface) devices attached, this is clearly inadequate.
Another possible countermeasure is the selective granting of elevated privileges for accessing devices to certain applications or processes; an example of this approach is the device allocation mechanism commonly found in MLS systems (e.g. Sun Microsystems' Trusted Solaris[4]) for removable media.
While such approaches permit e.g. the handling of removable media, dynamically configured bus systems are subject to similar constraints as in the previously described countermeasure.
Given the limitations of such static protection mechanisms, it appears that a security mechanism for dynamic devices and device interfaces should also be itself dynamic and adaptive and be able to enforce any security policy an individual or an organization might have with regard to the admissibility and use of such devices.

3.1 Dynamic Device Interface Security

The requisite mechanism can be divided into two component classes. The first component class consists of enforcement mechanisms for the security controls; since, as noted above, these are generally not present in current COTS operating systems, these must be retrofitted into the operating system of concern.
The security policies to be enforced may require several types of intervention, all of which must be supported by instrumenting the components. The simplest intervention type is the enforcement of access controls, followed by the granting or continuation of control and data flows; all of which have in common that they are reactive and result from actions taken or attempted by either users or devices.
Security policies which can take trace-based properties into account can further restrict undesirable behavior. This generally places such policies outside the Execution Monitoring class identified in [17], imposing severe limits on the efficient expressibility of such policies; however, in the application area relevant to this discussion, terminating traces or even partial traces that are determined as unacceptable frequently are sufficient to enforce security policies (e.g. if protocol elements are detected, terminating the protocol is frequently an adequate response).
Additionally, the instrumentation may also be instructed to perform certain operations (using the same control flows as in the case of reactive behavior) or to collect data for use by the security policy enforcement mechanisms.
Since in many cases a number of operating system components including device drivers are involved, this implies that the enforcement instrumentation must be fitted in each of the relevant loci. At the same time, the modifications introduced by this instrumentation must not affect the overall behavior of the operating system both from the perspective of individual device drivers and from application programs and users.
The former is a pragmatic requirement since providing individual instrumentation for each possible device driver would be highly inefficient; it is therefore necessary to provide the requisite instrumentation at a level that does not require such modifications, which is facilitated by the abstraction mechanisms for device classes found in most operating systems.

The second component class provides the requisite dynamic control via policy-based mechanisms. Given enforcement components as described above which intercept control flows inside the configuration management components and device interfaces at well-defined instrumentation points, decisions on the admissibility of such operations can be made based on security policies either configured locally (as is the case in the standalone Goalkeeper system) or from remote systems in externally controlled reference monitors (ECRM) [23].
In case of permitted configuration changes and operations the intercepted control flow continues unimpeded, while a negative decision on the part of the ECRM must result in an error condition that can be handled properly by the remain-

[4] `http://wwws.sun.com/software/solaris/trustedsolaris/`

ing system components since it is frequently not possible to transparently extend the range of error conditions supported by unmodified operating system components and third party device drivers.
The information contained at the device level is frequently limited (e.g. in identifying higher level abstractions such as which subjects and objects are involved in a transaction); by instrumenting the operating system at several abstraction layers as described in [21], it is, however, possible to correlate the requisite information at the ECRM.
This permits the determination of compliance with security policies at higher levels (which typically implies that operations that would have to be denied at a lower abstraction level can be granted and may also include protocol-specific actions such as the embedding of additional information on subject and object identity) and handing over of enforcement to the semantically appropriate layer.
However, this paper does not cover the details of the layering mechanism; these are described in [21] while [23] describes the underlying policy mechanism.

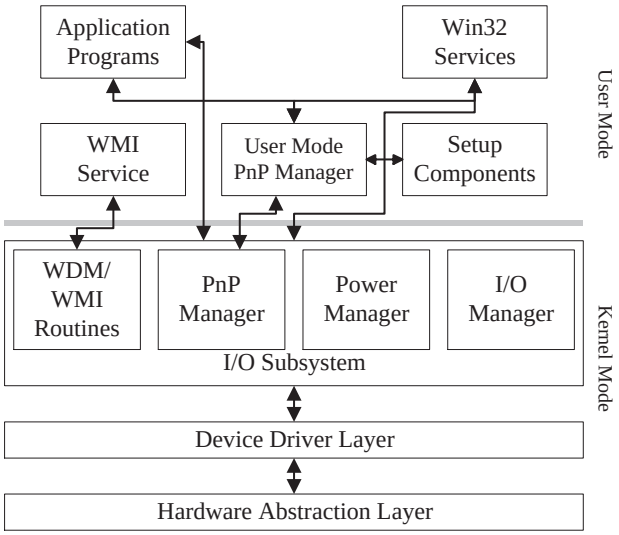

Figure 1. Windows 2000 I/O Architecture

4 Implementation

While the general concepts outlined above are applicable to other COTS systems, the following concentrates on the Microsoft Windows NT [18] family of operating systems and specifically on Windows 2000 [19] and later since these introduced the dynamic reconfiguration mechanisms discussed above and hence also the problems associated with such features.
Even though the interfaces exposed to application programs (both the environmental subsystems such as Win32 and POSIX and the Native API) for device handling resemble e.g. Unix derivatives in that file objects are used for representing and communicating with device drivers and devices, the underlying operating system itself is asynchronous and packet-based. Figure 1 omits the procedural interface layer and shows the component interactions involved in device I/O relevant to this discussion.
In this model, I/O requests originating from environmental subsystems are routed through the native system services API and the kernel-level I/O manager. The I/O manager dispatches I/O request packets (IRP) to device drivers that are registered with it[5]; ultimately the device drivers interface with the hardware abstraction layer (HAL) which provides access to the physical ports and memory areas required for interfacing with the devices proper.
While monolithic drivers exist, the device driver layer is typically subdivided into several devices. General I/O processing for a class of devices is provided by class drivers while processing for types of ports such as USB are frequently handled by port drivers. Specific device instances within a type of port are then handled by miniport drivers relying on the support of port drivers.
Moreover, the IRP-based I/O structure permits the insertion of a type of driver called filter drivers into arbitrary positions of this device driver stack and to require processing both in the control flow direction towards the hardware as well as in the reverse direction.
This enables both pre- and postprocessing steps for each IRP; the latter are also possible within individual drivers in the form of I/O completion routines that are called by the I/O manager after the processing of an earlier step.

Conversely, if a device raises an interrupt, the appropriate device driver registered for the given interrupt enters an interrupt service routine (ISR) in which the operations required for servicing the device are either performed directly in trivial cases or, more frequently, transformed into a deferred procedure call (DPC), which then completes the required operations at a lower interrupt privilege level, avoiding blocking the remainder of the system. All of these processing steps must be handled by filter drives as well.
As can be seen from the above discussion, inserting filter drivers into the processing stack permits the insertion of instrumentation points for monitoring and auditing as well as for access and other behavior-based controls for existing, loaded device drivers; the typically modular structure of the device drivers (class, port, and miniport drivers) permits the efficient interception of several generic device interfaces without requiring knowledge of device drivers for individual device models.

[5] While there are other types of device drivers involving specialized treatment, the types of devices relevant here are covered by the above discussion.

Additional device driver functionality that can also be made subject to interception includes I/O cancellation routines that are called whenever an I/O operation is canceled either explicitly or by termination of the thread that caused the original IRP to be issued; since such operations may not only result in cleaning up of data structures but also involve operations on physical devices, they must be intercepted and made subject to security policies as well.

4.1 Device Initialization

For all devices to be subjected to security policy control, the interception mechanism must be set up individually. While some devices are configured at boot time and could therefore be statically instrumented, supporting dynamic configuration controls as described in section 2.3 requires support for the PnP mechanism of Microsoft Windows.
Two configuration control paths must be distinguished. One is explicit configuration based on registry database entries, the other is called device enumeration and results from issuing commands to bus-type devices (referred to as physical device objects) to enumerate each device attached to the device (the basic mechanism also applies to devices added to bus configurations later, in this case an interrupt is generated by the bus device).

This model is also supported for so-called legacy drivers that do not support PnP operations; the system initialization itself starts with a virtual bus-type device[6] on which all legacy and bus devices are enumerated in a device tree; for device drivers not initialized at boot-time, the PnP manager (see figure 1) loads device drivers for detected devices along this tree.
However, if a device is encountered for which no driver is available in the system, the standard Microsoft Windows system calls on the user-mode PnP manager for installing device drivers, possibly installing new devices from existing driver cabinet files. While such device driver installation is not performed if user interaction is required and either the current user has no administrative privileges or no user is logged into the system, such installation can proceed completely automatically and unchecked.
By intercepting the calls to the user-mode PnP manager from the kernel-mode PnP manager and determining the admissibility of such operations based on the device instance ID, this threat can be addressed. While replacing the kernel-mode PnP manager directly with an instance containing redirected entry would be a possible implementation, an alternative is preferable in the interest of portability and installability.

This alternative consists of using a pseudo-device driver (with no physical device associated with it) that is loaded early in the boot section prior to enumerating any devices relevant to the discussion here which dynamically alters the entry points of the kernel-mode PnP manager at runtime to point to its interception mechanism (which in turn return control flow back to the PnP manager) [10].
If a device driver's presence is now detected during enumeration, the interception mechanism for device operation must be set up. To this end, filter drivers must be registered for each device that is to be controlled; the Microsoft Windows systems here distinguishes between driver objects (for device drivers which may support multiple devices) and device objects.

Functions to be intercepted are controlled by the driver object, although latter are also of interest for the discussion here since they must be used to distinguish between instances during interception.
During this loading process, filter drivers must be attached firstly to all bus-type devices (bus filter drivers) to control (e.g. by causing cancellation of IRPs) the further configuration changes of a given bus.
Second, the PnP manager must then be dynamically instructed (using each devices' registry class key) to load filter drivers for the actual function drivers; this process is repeated along the device tree as well as for each change in device configuration as monitored by the abovementioned pseudo-device driver.
While it is generally possible to employ both lower-level filter drivers (inserted between the bus driver and the function driver) and upper-level filter drivers (inserted between the I/O manager and the functional device driver), only the latter are covered by the implementation described here.
By redirecting the control flow of the kernel PnP manager, monitoring for the occurrence of the operations described above, and dynamically performing actions directly (such as denying the continuation of a device driver installation) or by inserting the requisite data structures into the registry to ensure that all required filter drivers are loaded, the threats described in section 2.3 can thus be countered effectively.

4.2 I/O Processing

For each filter driver (i.e. both bus filter drivers and upper-level filter drivers) inserted dynamically as described in section 4.1, the drivers must support the dispatch, I/O, interrupt service, and deferred procedure call service routines as well as the completion and cancellation routines described in section 4.
In most cases, however, it is sufficient to merely pass through the interrupt request packets (performing audit instrumentation steps as necessary) without modifying the filtered device behavior (exceptions to this include the use of

[6] On PC98 and later systems this is generally an ACPI enumerator

IOCTL calls for performing device operations; these must be dealt with at a device-specific level).

Only the dispatch routines for the device objects themselves (e.g. for opening, reading from and writing to devices) must be handled separately since the instrumentation must support all operations specified by the security policies as outlined in section 3.1.

While the Microsoft Windows operating system employs a reference monitor mechanism, this mechanism does not meet the completeness property of [1] since not all operations are or can be subjected to the scrutiny of the system's own reference monitor.

For processing IRPs, two strategies must be employed. For simple IRPs, an IRP stack location entry can be used to associate pending processing with an IRP; otherwise associated IRPs must be created which link back to the original IRP to be processed.

In either case the instrumentation points for each of the dispatch point are accessed and control flow is transferred to the externally controlled reference monitor for auditing and a decision on the admissibility of further processing. In case this is denied, the IRP must be canceled or return with an error code depending on the circumstance under which the denial occurs.

The processing outlined above also applies to device-initiated (interrupt) communication; any such processing must – because of the delays inevitably introduced by calling on the externally controlled reference monitor and because the ECRM may not be able to perform its processing at the device IRQ level – occur inline by tagging the resulting data for processing at a more permissive IRQ level later on.

Figure 2 shows a schematic diagram of the interception mechanism and the interposition of the filter drivers on each of the functional components of the device driver depending on the device object selected by the IRPs ultimately issued in response to API calls at the application levels or interrupt processing.

In the case of a standalone Goalkeeper system, the ECRM module does not rely on a formal model for security policy decisions but rather is configured statically for a given system and device configuration.

In most cases, protocol analysis must take into account information that is obtained in the dispatch points for opening, reading from, and writing to a device object. Any processing must be applied to transport-decoded streams (but may also be applied to raw data streams; however, this leads to a significant performance penalty).

Filtering steps can be achieved by a set of timed general Büchi automata $\mathbf{A} = \{(Z^i, \mapsto^i, I^i, Z^i_0, \mathcal{F}^i)\}$ where for each i, $(Z^i, \mapsto^i, I^i)$ is a transition system with a fixed alphabet, $Z^i_0 \subseteq Z^i$ the set of initial states and $\mathcal{F}^i \subseteq \mathbb{P}(Z^i)$ the set of sets of

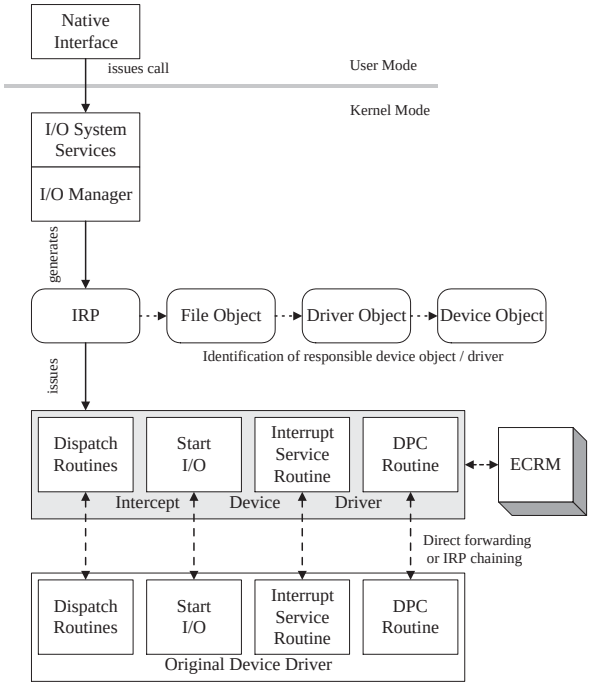

Figure 2. Interception of Device I/O

accepting states [20]. Büchi automata are desirable since they are well understood and can model infinite accepting states, a situation that can arise frequently in the type of protocol interception required here.

Each automaton may undergo several transitions prior to establishing either an accepting state or rejecting the data stream; the outbound stream can only be forwarded if no automaton is still in an intermediate state. For most protocols used with the devices discussed here (e.g. SyncML and earlier protocols), the required information can be established quickly.

For some devices, memory mapping and direct access via the caching subsystem (called Fast I/O) must also be considered, in this case, however, the individual protocol steps cannot be followed efficiently since protocol data units may be exchanged via mapped memory segments without involving the I/O manager or IRP processing.

As noted in section 3, it is frequently desirable to perform security policy decisions at appropriate semantic levels to retain maximum expressiveness and hence limit the types of filtering explicitly occurring at the device stage while referring policy decisions to security policy enforcement mechanisms situated at higher abstraction layers.

As an example of this processing, consider the attachment of a storage device such as popular USB memory sticks. The device filter driver loaded on obtaining the device instance ID can forward this information to the ECRM, which

can determine whether a file system filter driver [22] enforcing the security policy for file-level semantics is present and active for the newly attached device (e.g. by transparently encrypting and decrypting data stored on the memory stick).
If this is the case, then the ECRM can perform the requisite policies through the file system filter driver while instructing the device filter driver to let all communication to the file system driver stack unimpeded.

5 Related Work

Most existing work on protection at the device level has, with the exception of device allocation mechanisms already found in the Multics system prior to project Guardian [4, 11] has concentrated on the virtualization of system instances.
Some virtual machine monitors, such as the SDC KVM/370 [5] and the DEC VAX VMM Security Kernel [7] have been used to separate mandatory security classes.
The drawback of such approaches is that the flexibility in separating the individual virtual machines (and in extreme cases individual application programs) is not very high, although this is presumably adequate for a pure MLS system; for other security policies, however, the availability of more information to base security policy decisions permits finer granularity security policies.
As noted by [12], the virtual machine-based approach is further made undesirable by the limitations of both the underlying hardware found in COTS hardware based on the iAPX 86 architecture and the lack of trustworthy software (operating system) platform to base such a system on.
The issue of trustworthiness in such underlying systems has also been the subject of research particularly with regard to microkernel implementations such as the Mach-based DTOS [14, 13, 15] and its successor Flask/Fluke (which in fact is based on recursive virtual machines as its underlying organizing principle [3, 16].
Both of these systems also provide a fine-grained access control to the device objects. However, such systems are confronted with a relative dearth of application programs available.
Other related approaches include the general mechanisms for interposing security enforcing code between application programs and the system kernel as in the case of SPIN [2, 6] and the kernel hypervisor mechanism [9]. Capability-based systems such as [8] can also provide adequate security mechanisms by enforcing security as properties of relations between applications.

6 Conclusion and Outlook

This paper has described a class of threats emanating from device interfaces and the dynamic configuration and usage of such devices with insufficient support from operating system security mechanisms and application programs.
A set of countermeasures for defining and enforcing security policies over such interfaces has been described along with a reference implementation based on the Microsoft Windows family of operating systems that effectively countered the threats identified at the outset.
Beyond the reference implementation described here, other platforms require different interception mechanisms depending on their I/O architecture.
For systems such as OpenVMS, the packet-processing approach is equally applicable (although the lack of PnP support implies that several of the threats discussed in this paper are not relevant for this platform; also, since OpenVMS supports device and volume allocation, the application area is largely limited to mechanisms such as using synchronization mechanisms for PDAs); therefore the need for the mechanisms described here appears limited.

For Unix derivatives, however, the increasing support for PnP mechanisms would indicate the contrary. Here, preliminary results indicate that interception mechanisms are on one hand simpler to implement (given the device special file architecture, only a very limited set of functions must be instrumented, and this can occur by dynamically replacing the device special calls), and on the other hand significantly more difficult to maintain for a larger set of devices.
This is due to the heavy reliance on IOCTL data structures, which are per-device specific and at the same time due to the lack of a generalization and abstraction mechanism such as the one found in the VFS file system architecture first introduced in Sun Solaris and now widely adopted for Unix derivatives.
The mechanisms described in this paper can be extended significantly and must in fact be tailored to specific interfaces and even application programs if the security policies to be supported call for a level of control that requires the semantic analysis of such protocols.
However, as such support is frequently not required at an elaborate level (e.g. identifying whether a given protocol or protocol element is initiated is sufficient for enforcing a policy simply prohibiting the completion of the protocol element), the burden on implementers appears manageable.

References

[1] J. P. Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, Air Force Electronic Systems Division (AFSC), L. G. Hanscom Field, Bedford, MA, USA, Oct. 1972. AD-758 206, ESD/AFSC. (Also available as Vol. I, DITCAD-758206. Vol. II, DITCAD-772806).

[2] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensi-

bility, Safety and Performance in the SPIN Operating System. *ACM Operating Systems Review*, 29(5):267–284, Dec. 1995. Proceedings of the 15th Symposium on Operating Systems Principles (15th SOSP'95).

[3] B. Ford, M. Hibler, J. Lepreau, R. McGrath, and P. Tullmann. Interface and Execution Models in the Fluke Kernel. In USENIX, editor, *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 101–116, New Orleans, LA, USA, Feb. 1999. ACM Press. Published as an ACM Operating Systems Review Special Issue.

[4] E. L. Glaser. A brief description of privacy measures in the MULTICS operating system. In *Proceedings of the AFIPS Spring Joint Computer Conference (1967 SJCC)*, volume 30, pages 303–304, Atlantic City, NJ, USA, Apr. 1967. AFIPS, AFIPS Press.

[5] B. D. Gold, R. R. Linde, R. J. Peeler, M. Schaefer, J. F. Scheid, and P. D. Ward. A Security Retrofit of VM/370. In *Proceedings of the National Computer Conference*, volume 48, pages 335–344, New York, NY, USA, Nov. 1979. AFIPS, AFIPS Press.

[6] R. Grimm and B. N. Bershad. Providing Policy-Neutral and Transparent Access Control in Extensible Systems. In J. Vitek and C. D. Jensen, editors, *Secure Internet Programming, Security Issues for Mobile and Distributed Objects*, volume 1603 of *Lecture Notes in Computer Science*, pages 317–338, Brussels, Belgium, July 1998. Springer Verlag. Proceedings of MOS'98.

[7] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. A VMM Security Kernel for the VAX Architecture. In *Proceedings of the 1990 IEEE Symposium on Security and Privacy (SOSP '90)*, pages 2–19, Oakland, CA, USA, May 1992. IEEE Computer Society.

[8] C. R. Landau. Security in a Secure Capability-Based System. *ACM Operating Systems Review*, 23(4):2–4, 1989.

[9] T. Mitchem, R. Lu, and R. O'Brian. Using Kernel Hypervisors to Secure Applications. In *Proceedings 13th Annual Computer Security Applications Conference (ACSAC'97)*, pages 175–182, San Diego, CA, USA, Dec. 1997. IEEE Press.

[10] G. Nebbett. *Windows NT/2000 Native API Reference*. Macmillan Technical Publishing, Indianapolis, IN, USA, 2001.

[11] E. I. Organick. *The Multics System: An Examination of Its Structure*. MIT Press, Cambridge, MA, USA, 1972.

[12] J. S. Robin and C. E. Irvine. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In USENIX, editor, *Proceedings of the 9th USENIX UNIX Security Symposium*, pages 129–144, Denver, CO, USA, Aug. 2000. USENIX.

[13] SCC. DTOS Formal Securiry Policy Model (FSPM). Technical report, Secure Computing Corporation, Roseville, MN, USA, Sept. 1996. Contract no. MDA904-93-C-4209, CDRL sequence no. A004.

[14] SCC. DTOS Formal Top-Level Specification (FTLS). Technical report, Secure Computing Corporation, Roseville, MN, USA, Dec. 1996. Contract no. MDA904-93-C-4209, CDRL sequence no. A005.

[15] SCC. DTOS Generalized Security Policy Specification. Technical report, Secure Computing Corporation, Roseville, MN, USA, June 1997. Contract no. MDA904-93-C-4209, CDRL sequence no. A019.

[16] SCC. Assurance in the Fluke Microkernel: Final Report. Technical report, Secure Computing Corporation, Roseville, MN, USA, Apr. 1999. Contract no. MDA904-97-C-3047, CDRL sequence no. A002.

[17] F. B. Schneider. Enforceable Security Policies. *ACM Transactions on Information and System Security*, 3(1):30–50, Feb. 2000.

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

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

[20] W. Thomas. Automata on Infinite Objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 4, pages 133–192. MIT Press, Cambridge, MA, USA, 1990.

[21] 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*, pages 100–108, West Point, NY, USA, June 2001. IEEE Press.

[22] S. Wolthusen. Security Policy Enforcement at the File System Level in the Windows NT Operating System Family. In *Proceedings 17th Annual Computer Security Applications Conference (ACSAC'01)*, pages 55–63, New Orleans, LA, USA, Dec. 2001. IEEE Press.

[23] S. Wolthusen. Access and Use Control using Externally Controlled Reference Monitors. *ACM Operating Systems Review*, 36(1):58–69, 2002.