AN EXTENDED ACCESS CONTROL SYSTEM FOR WINDOWS XP

AUTHORS

Roberto Battistoni[1], Emanuele Gabrielli[2], Luigi Vincenzo Mancini[2]

[1] Secure Edge s.r.l.
Via Palmiro Togliatti 1601, 00155 Rome, Italy
r.battistoni@computer.org

[2] Dipartimento di Informatica
Università di Roma "La Sapienza"
Via Salaria 113, 00198 Rome, Italy
gabrielli@dsi.uniroma1.it
lv.mancini@dsi.uniroma1.it

ABSTRACT

We propose an access control systems, called WHIPS that controls the invocation of all the system calls critical for the security of Windows OS. WHIPS is implemented as a kernel driver, also called kernel module, using undocumented structure of the Windows kernel, where it is integrate without requiring changes in the kernel data structures and algorithms. WHIPS is also transparent to the application processes that continue to work correctly with no need of source code changes or re-compilation. A working prototype has been implemented as a kernel extension of Windows XP.

Keywords. Access Control. Privileged Processes. Critical System calls. Native API. Windows services. Buffer overflow. Exploit. Windows Operating System. Intrusion Detection System. Intrusion Prevention System.

1 INTRODUCTION

Attacks to the security of network clients and servers are often based on the exploitation of flaws present in a specific application process. By means of widely known techniques [Al96, Co00], a malicious user may corrupt one or more memory buffers in such a way that on return from a function call, a different piece of code, "injected" by the attacker, is executed by the flawed application process. Obviously the buggy application process maintains its special privileges (if any). As a consequence, if the attack is successful against a privileged process the attacker may gain full control of the entire system. For example, the malicious code could execute a shell (shell or cmd.exe) in the privileged application context and allow the attacker to become an administrator of the system. An example of recently exploit using Buffer Overflow is

the slammer worm [MPSW03] that attacks the MS-SQL server for Windows 2000/XP to gain high privileges and saturates the network bandwidth causing a denial of service attacks.

This paper presents the design and implementation of an extended access control system for Windows XP that, by monitoring the system calls made by the application processes, allows immediate detection of security rules violations. The proposed prototype employs interposition at the system call interface to implement the access control functionality and requires no change to the kernel code and to the syntax and semantics of existing system calls. Basically, the system call execution is allowed just in case the invoking process and the value of the arguments comply with the rules kept in an Access Control Database (ACD) within the kernel. The proposed access control system intends to protect against any technique that allows an attacker to hijack the control of a privileged process.

The REMUS system [BGM02] has shown that immediate detection of security rules violations can be achieved by monitoring the system calls made by processes in Linux. Here we try to apply a similar technique to the Windows XP OS. The access control system proposed here is called WHIPS, Windows-nt family Host Intrusion Prevention System. Indeed, *Intrusion Prevention Systems* (IPSs) strive to stop an intrusion attempt using a preventive action on hosts to protect the systems under attacks. The WHIPS prototype runs under Windows NT, Windows 2000 and Windows XP. In the following, by the word *Windows* we refer Windows XP, but the consideration and the prototype design are applicable to all the *Windows NT family OS,* born in 1993 with the first version Windows NT 3.51.

The rest of the paper is organized as follows. Section 2 characterizes the privileged and dangerous processes and defines when a system call is critical and dangerous for a Windows system, showing how the Windows system call are invoked by the user processes. Section 3 proposes the WHIPS prototype, showing the implementation and the performance analyses of the prototype.

2 SECURITY PROBLEM: PRIVILEGED PROCESSES AND CRITICAL SYSTEM CALLS

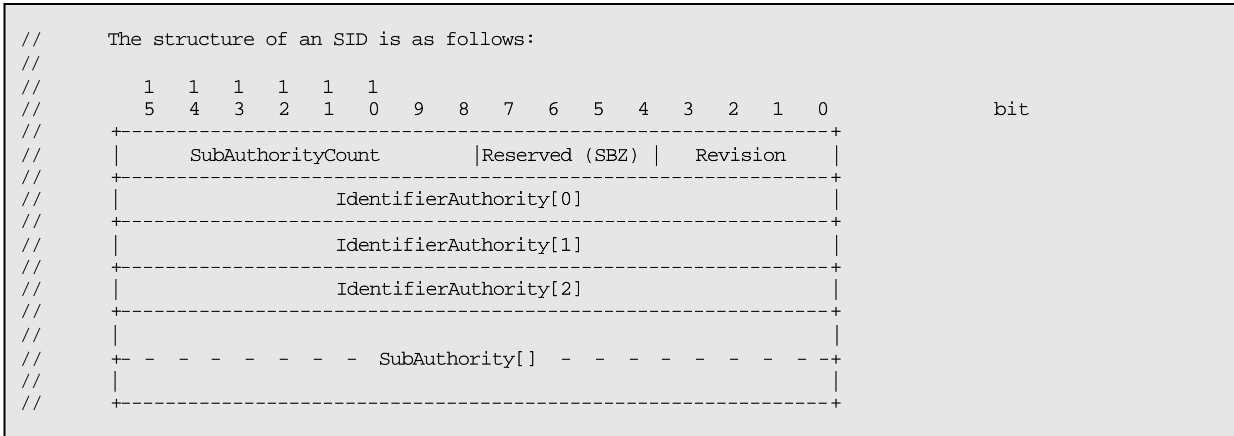
In order to gain control of an OS, an attacker has to locate a target process that run with high privileges in the system. For example, if the OS belong to *Linux family*, the privileged processes include *daemons* and *setuid* processes that execute their code with the effective user *root* (EUID=0). In the following, first we introduce the Windows processes security context and then we characterize when a process is *privileged* or *dangerous* and when a system call is *critical* or *dangerous* in Windows.

2.1 Process Security Context

This section examines the *Security Identity Descriptor* (SID), and the *Access Token* (AT), which are the components of a process structure that represents its security context.

2.1.1 Security Identity Descriptor

Security Identity Descriptors (SIDs) identify the entities that execute the operations in a Windows system and may represent a user, a group, a machine or a domain. If *G* is the groups set, *U* is the users set, *M* is the machine set, and *D* is the domain set, every element in *G*, *U*, *M* and *D* has a corresponding SID.



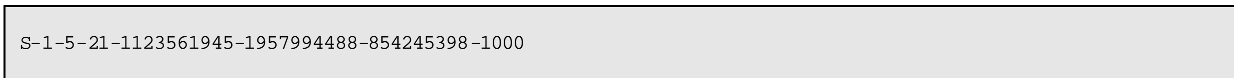
```
//      The structure of an SID is as follows:
//
//         1   1   1   1   1   1
//         5   4   3   2   1   0   9   8   7   6   5   4   3   2   1   0              bit
//      +---------------------------------------------------------------+
//      |      SubAuthorityCount        |Reserved (SBZ) |   Revision    |
//      +---------------------------------------------------------------+
//      |                   IdentifierAuthority[0]                      |
//      +---------------------------------------------------------------+
//      |                   IdentifierAuthority[1]                      |
//      +---------------------------------------------------------------+
//      |                   IdentifierAuthority[2]                      |
//      +---------------------------------------------------------------+
//      |                                                               |
//      +- -  -  -  -  -  -  -  -  SubAuthority[]  -  -  -  -  -  -  - -+
//      |                                                               |
//      +---------------------------------------------------------------+
```

Figure 1 – SID structure.

Figure 1 shows the structure of a SID, which is a variable length numeric structure with the following fields:

- **Reserved, Revision:** reserved bits and revision number, 8 bits;
- **SubAuthorityCount:** number of subauthorities, 8 bits;
- **IdentifierAuthority [0..2]:** a maximum of three Identifier Authority, 16 bits each one;
- **SubAuthority [0..N]:** an array of subauthorities and its *Relative ID* (RID).

A RID (relative number) is a number that distinguishes two SIDs otherwise equal in a Windows system. This is an example of a SID number:



```
S-1-5-21-1123561945-1957994488-854245398-1000
```

Figure 2 – SID example.

S is only a fixed character for SID number, *1* is the revision number, *5* and *21* are two Identifier Number and *1123561945-1957994488-854245398* are three SubAuthority number, finally *1000* is RID number. Every Windows system has a lot of SIDs; some of them identify particular users or groups and are called Well-Known SIDs [Mi02a].

2.1.2 Access Token

The SRM (Security Reference Monitor) is a Windows kernel component that uses a structure called *Access Token* to identify a thread or a process security context [RuS01]. A *security context* is a set of privileges, users and groups associated to a process or a thread. During the log-on procedure, *Winlogon* (one of the Windows component performing users authentication) builds an initial token that represents the user access,

and links this token to the users shell process. All the processes created by the user inherit a copy of the initial access token. Figure 3 examines a description of the Access token fields:

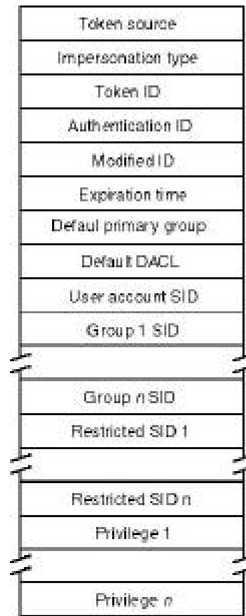


Figure 3 – Access Token structure.

- **Token source**: short description of the entity creators of the token.
- **Impersonation type**: impersonation type applied to the Access token.
- **Token ID**: token unique identificator.
- **Authentication ID**: identificator assigned by the token creator.
- **Expiration time**: token expiration time, actually not used (tokens do not expire).
- **Default primary group and default DACL**: predefined groups (only POSIX) and predefined DACL for objects created by the process or thread, owner of the Access Token.
- **User account SID**: user SID that has activated the process or thread.
- **Group 1..N SID**: groups SIDs the user belongs to.
- **Restricted 1..N SID**: restricted SIDs restrict the use of token.
- **Privilege 1..N**: privileges list assigned to the token (ex: *SeBackup*, *SeDebug*, *SeShutdown*, *SeTakeOwnership*, etc.) [RuS01].

We can have two types of Access Token:

- **Primary token**: is a token that normally is assigned to a process to identify its security context.
- **Impersonation token**: is a token that normally is assigned to a thread when an *impersonation* activity occurs.

Every process has an Access Token called *primary token*. A process in Windows is passive, in the sense that every process has associated a primary thread and a variable number of secondary threads that executes the process operations. The primary thread inherits a copy of the primary token, whereas a secondary thread may

inherit a copy of the primary token too, or may obtain a restricted copy of the primary token by the *impersonation* mechanism.

2.1.3 Impersonation

Impersonation is a mechanism that allows a security context of a process or a thread to migrate in another security context. For example, an impersonation occurs when a server accesses its resources on behalf of a client. In this case, the impersonation mechanism allows the server process to use the security context of the client that requested that particular operation [RuS01]. The possible impersonation types are:

- **ImpersonateNamedPipeClient**: a server communicates with a client through a named pipe.
- **DdeImpersonateClient**: a server communicates with the client through Dynamic Data Exchange (DDE).
- **RpcImpersonateClient**: a server communicates with the client through Remote Procedure Call (RPC).
- **ImpersonateSelf**: a thread can create an impersonation token that is simply a copy of its process token. The thread can then alter its impersonation token, to disable SIDs or privileges.
- **ImpersonateSecurityContext**: a Security Support Provider Interface (SSPI) package can impersonate its clients. SSPIs implement a network security model such as LAN Manager 2 or Kerberos.

To avoid an improper use, Windows does not permit a server to impersonate a client process without client consensus. Some impersonation levels follow:

- **SecurityAnonymous**: is the most restrictive level of impersonation, the server cannot impersonate or identify the client.
- **SecurityIdentification**: lets the server obtain the identity (the SIDs) of the client and the client privileges, but the server cannot impersonate the client.
- **SecurityImpersonation** : lets the server identify and impersonate the client on the local system.
- **SecurityDelegation**: is the most permissive level of impersonation. It lets the server impersonate the client on local and remote systems.

If a client does not choose an impersonation level, *SecurityImpersonation* is the default.

2.1.4 Windows Privileges

A *privilege* in Windows is the right to operate on a particular aspect of the entire system, so a privilege acts on the entire system, whereas a right acts on an object of the system [Scm01]. A privilege may be assigned to a user or a group in Windows. When a user logs on a Windows system, a process will be created and assigned to the user. Then the privileges assigned to the user or the group will be added in the Access Token privileges list of the user process.

There are many privileges in Windows each allowing a particular action on the system, but not every privilege is dangerous for the system security. Only a subset of the entire set of Windows privileges contains dangerous privileges that can be exploited by a malicious user. Below is an example of dangerous privileges [HLB01]:

- **SeBackupPrivilege**: permits to a user to perform a backup of some data even if the user has no access rights to the data.
- **SeTcbPrivilege**: with this privilege a user or a group is a trusted component of the OS (predefined account *LocalSystem* is the only one that has this privilege in Windows by default).
- **SeDebugPrivilege**: a user with this privilege can debug any process viewing and modifying the process memory.
- **SeAssignPrimaryTokenPrivilege**: replace a process-level token.
- **SeIncreaseQuotaPrivilege**: adjust memory quotas for a process.

2.2 Privileged and dangerous processes

As discussed above, some privileges are dangerous in Windows OS. If we want to know if a process is privileged, we must look to the process Access Token of the user that run the process.

A malicious user can attack these processes, and when the attacker runs a malicious code in the process context, the attacker gains all the process privileges. If some privilege is dangerous, the process becomes dangerous. Simply to identify the dangerous processes, one might look for dangerous privileges into the process Access Token. Now assume that P is a set of processes, so $P = \{p_1, p_2, \ldots, p_{n1}\}$, where $p_1, p_2 \ldots p_{n1}$ are the processes in a Windows system, D is the set of dangerous processes, so $D = \{d_1, d_2, \ldots, d_{n2}\}$; D is a subset of P, $D \subseteq P$. Assume that K is the set of privileged processes $K = \{k_1, k_2, \ldots, k_{n3}\}$, K is a subset of P, $K \subseteq P$. The set $D \cap K$ is the set of privileged and dangerous processes, since in general $D \subseteq K$ the set of privileged and dangerous process is equal to D.

- **Definition:** a *privileged process* is a process with some Windows privilege.
- **Definition:** a *dangerous process* is a privileged process that has some dangerous privilege.

We can characterize the set D analysing the processes Access Token. If in the Access Token privileges list there are one or more dangerous privileges, the process, owner of the Access Token, belongs to D. In the following, we discuss a particular set of privileged processes: the *Services*.

2.2.1 Services

Almost every OS has a mechanism to start processes at system start up time that provide services not tied to an interactive user. In Windows, such processes are called *services*. Services are similar to UNIX daemon processes and often implement the server side of client/server applications.

On Windows, many services log-on to the system with a predefined account: *System* account (called *LocalSystem* too). This account belongs to group *Administrators* and is very powerful because it has many dangerous privileges. This is a critical point for the Windows security.

Often a careful analysis of services could restricts services privileges. This can be done with a new account defined for the specific service, where this account has less privileges then *System* account [HLB01]. To

avoid this security problem, in Windows XP there are two new accounts for services: *local service* and *network service*. These two new XP accounts have minimum privileges necessary to the execution of some services, typically Internet and network services. So an attack to these services is less powerful than an attack to a service that log-on with *System* account [HLB01]. Tab. 1 presents the privileges of *LocalSystem*, *LocalService* and *NetworkService* predefined account. How you can see, *LocalSystem* has almost all the Windows privileges whereas *LocalService* and *NetworkService* have a little bit of these privileges.

Privilege	Local System	Local Service	Network Service
SeCreateTokenPrivilege	X		
SeAssignPrimaryTokenPrivilege	X		
SeLockMemoryPrivilege	X		
SeIncreaseQuotaPrivilege	X		
SeMachineAccountPrivilege			
SeTcbPrivilege	X		
SeSecurityPrivilege	X		
SeTakeOwnershipPrivilege	X		
SeLoadDriverPrivilege	X		
SeSystemProfilePrivilege			
SeSystemtimePrivilege	X	X	X
SeProfileSingleProcessPrivilege	X		
SeIncreaseBasePriorityPrivilege	X		
SeCreatePagefilePrivilege	X		
SeCreatePermanentPrivilege	X		
SeBackupPrivilege	X		
SeRestorePrivilege	X		
SeShutdownPrivilege	X		
SeDebugPrivilege	X		
SeAuditPrivilege	X	X	X
SeSystemEnvironmentPrivilege	X		
SeChangeNotifyPrivilege	X	X	X
SeRemoteShutdownPrivilege			
SeUndockPrivilege	X	X	X
SeSyncAgentPrivilege			
SeEnableDelegationPrivilege			

Table 1 – Local System, Local Service and Network Service privileges.

2.2.2 Services Identification

To identify a service we must check the SIDs in the process Access Token, precisely the so-called *Well Known SIDs*. We have two possibility: if the service logs on to the system with *LocalSystem* account, the user account SID in the Access Token, is equal to string *"S-1-5-18"*, Local System SID. Otherwise, we must look in groups SIDs; the process is a service if there is *Well-Known SID Service* represented by the string *"S-1-5-6"*. But it is not simple to know exactly when a process is a service. We need some implications that help us.

- **Process is a Service ⇒ Access Token User SID is equal to Local System SID, or in the Access Token Group SIDs is present Service SID:** if we consider a service process then its Access Token contains the LocalSystem or Service Well known SID.
- **Access Token Group SIDs has Service SID⇒ process is a Service**: if Service Well known SID appears in the group section of the process Access Token, process is securely a service.

- **Access Token User SID is equal to LocalSystem SID ⇒ process IS NOT NECESSARILY a service***:* if user SID is LocalSystem the process, owner of the Access Token, is not necessarily a service, it could be a system process too.

If we consider only the first implication, we will find a set of processes that contains securely set of services, but is not necessarily equals to this set. Below we present a simply pseudo-code test to determine if a process is a service or a system process:

```
If (USER-SID=="Local System SID") OR (GROUP-SID includes "Service SID")
                (process_type=SERVICE) OR (process_type=SYSTEM_PROCESS)
         else
                (process_type!=SERVICE) AND (Process_type!=SYSTEM_PROCESS)
```

Figure 4 – Test to determine if a process is a service or a system process.

2.3 Critical and dangerous system calls

In this section, we introduce the definition of *system calls* in Windows and then we characterize when a system call is a *critical* system call.

2.3.1 Native APIs: Windows system calls

APIs (Application Programming Interfaces) are programming functions held in dynamic library, and run in user-mode space and kernel-mode space. We call *native APIs* [Ne00] the APIs in kernel-mode that represent the system call of Windows. We call simply APIs the APIs in user-mode space.

Four dynamic libraries export APIs of the Win32 Subsystem:

- **User32.dll**: interface APIs.
- **Gdi32.dll**: graphic interface APIs.
- **Kernel32.dll**: system management APIs.
- **Advapi32.dll**: advanced system management APIs (registry, lsass, etc.).

The APIs in *user32.dll* and *gdi32.dll* invoke the native APIs implemented in kernel mode by *win32k.sys* module, which is the kernel mode of the Win32 subsystem. The APIs exported by *kernel32.dll* (system APIs) use a particular library *Ntdll.dll* that invokes native APIs in the kernel. Native APIs invoked by ntdll.dll, are the Windows system calls.

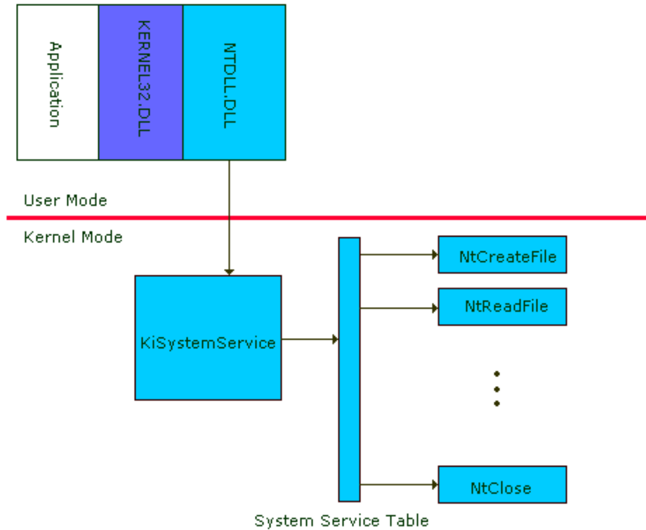


Figure 5 – System Service Table (SST).

Figure 5 shows that when an API of *Kernel32.dll* is called by an application, this API recalls one or more functions present in ntdll.dll. This library represents a bridge between user-mode and kernel-mode space [Ne00, Osr03]. The user-mode library Ntdll.dll is the front-end of the native APIs, which are implemented in the Windows kernel, *ntoskrnl.exe*.

Ntdll.dll exports all the native APIs with two type of function name prefix: *Nt* and *Zw*. True native APIs (in the kernel) have the same name of APIs exported by Ntdll.dll, but they are not the same functions.

Figure 6 shows an example of the native API *NtCreateFile(),* obtained disassembling *ntdll.dll*.

```
NtCreateFile:

mov    eax,0x0000001A
lea    edx,[esp+04]
int    0x2E
ret    0x2C
```

Figure 6 - Assembly code of NtCreateFile in NTDLL.DLL.

Function *NtCreateFile* loads registry EAX with the index *0x1A* of the native API in a particular table called *System Service Table* (KiServiceTable, fig. 8), then EDX registry points to the user-mode stack, *ESP+04*, where there are the parameters of Native API, and finally raises interrupt *0x2E* that executes the *System Service Dispatcher* of Windows (defined in Section 2.3.1.1). System Service Dispatcher is the kernel routine that invokes the true native API in the kernel [Ne00, Osr03].

Not all the native API exported by Ntdll.dll are exported by ntoskrnl.exe. This probably, is to prevent unauthorized use of particular and dangerous native API by a driver [Scr01, Ne00].

Disassembling the library ntdll.dll, we can observe that every Nt native API and its corresponding Zw native API have the same assembly code represented in fig. 6. If we disassemble ntoskrnl.exe, the true native APIs with the Nt prefix contain the true code of native API, and the native APIs with the Zw prefix have the representation in fig. 6, see also [Ne00, Osr03].

2.3.1.1 System Service dispatcher

Dispatcher of interrupt *0x2E* is the *System Service Dispatcher* routine. It is implemented in the executive layer of the Windows kernel, trough the kernel function *KiSystemService()*. Figure 7 shows that the APIs in *gdi32.dll* and *user32.dll* call directly the dispatcher *KiSystemService()*, and after the dispatcher invokes functions in *win32k.sys* module. The APIs in *kernel32.dll* invoke the functions exported by *ntdll.dll* and then that functions call the native APIs in Windows kernel.

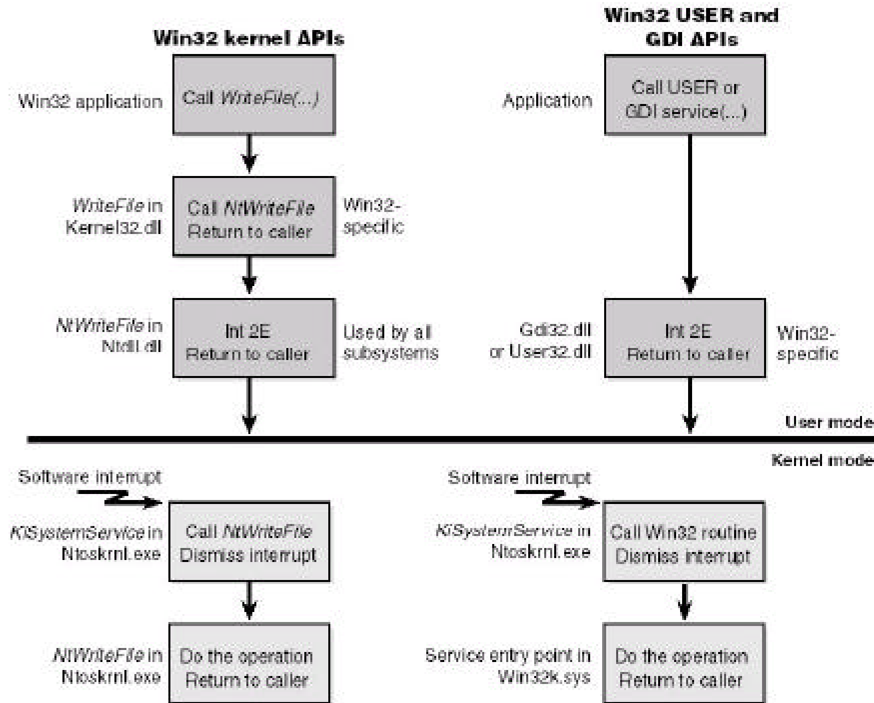


Figure 7 - Dispatching of native APIs and USER & GDI APIs.

When *KiSystemService()* is invoked, the dispatcher runs a series of control. First it controls the validity of index passed in EAX register, then it controls if the space expected for the native API parameters is correct and finally executes the native API in the kernel or APIs in *win32k.sys*.

KiSystemService() uses a structure called *System Service Descriptor Table* (*SDT*). SDT is represented by the *KeServiceDescriptorTable* structure (fig. 8), when *KiSystemService* manages native APIs, but when it manages *win32k.sys APIs*, *SDT* is represented by another structure called *KeServiceDescriptorTableShadow* [Scr01, HLB01].

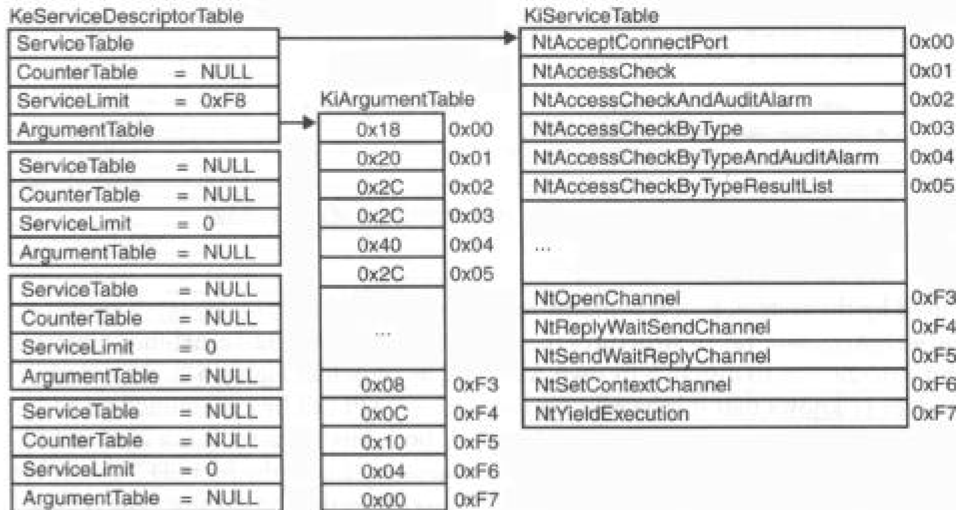


Figure 8 – KeServiceDescriptorTable.

Figure 8 shows that *KeServiceDescriptorTable* has two table pointers: *KiServiceTable* (*System Service Table, SST*) and *KiArgumentTable*. First table contains an index for every native API. This index is used by native API code in *ntdll.dll* to invoke the corresponding native API in the kernel (fig. 6). Second table contains, for every native API, the allocation space for native API parameters. This space is used for the kernel-stack memory allocation.

2.3.2 Critical and Dangerous native APIs

We have defined a native API as a Windows system call, but when is a system call in Windows a critical system call?

A native API is a generic kernel function; it has a function name, a series of parameters and a return value. If we consider a native API by itself, it is not critical, but it becomes critical when dangerous parameters are passed to it.

Consider a simple example: the native API *NtOpenFile()*. Typically this native API opens an handle to a file on the File System. Its only parameter is a pointer to a string that represents the file name (with path) that will be opened. If the file name is *readme.txt*, this native API is not critical for the system. But, if the file to be opened is equal to *c:\winnt\cmd.exe*, the Windows shell, the native API NtOpenFile with this particular parameter is critical. So we define a critical system call as follows:

- ❖ **Definition:** a *critical system call* is a native API that could be invoked with *dangerous parameters*.

And we define a dangerous system call:

- ❖ **Definition:** a *dangerous system call* is a critical system call invoked by a *dangerous process* (see Section 2.2).
- ❖ **Definition:** a *critical system call* is dangerous for the system only if the invocating process is a *dangerous process*.

A dangerous process that calls a native API with dangerous parameters may represents an attack of a malicious user.

2.3.3 Native API Classification

Native APIs in Windows 2000 and XP are about 250, and only 25 of them are documented by Microsoft with DDK (Driver Development Kit). All others native APIs are not documented. Fortunately Microsoft gives us a utility, *depends.exe* that displays all the APIs exported by *ntdll.dll* and all the functions exported by *ntoskrnl.exe.*

The problem is that we can view only the names of native APIs and not its parameters or its return value. Gary Nebbet helps us with a "bible" of native API: *"Windows NT/2000: Native API reference"* [Ne00].

2.3.3.1 Native API Category

Below a first classification of native APIs by category, suggested by Russinovich [RuS01]. There are 21 categories in tab. 2 [Ru98]:

Index	Category	Description
1	**Special Files**	These APIs are used to create files that have custom characteristics.
2	**Drivers**	These functions are used by NT to load and unload device driver images from system memory.
3	**Processor and Bus**	Processor registers and components can be controlled via these functions.
4	**Debugging and Profiling**	The profiling APIs provide a mechanism for sample-based profiling of kernel-mode execution. The debug control function is used by WinDbg for obtaining internal kernel information and controlling thread and process execution.
5	**Channels**	These functions were introduced in NT 4.0 and are present in NT 5.0 Beta 1. However, they are all stubs that return STATUS_NOT_IMPLEMENTED. Their names imply that they were intended to provide access to a communications mechanism.
6	**Power**	There is only one Native API for power management in NT 4.0. Interestingly, this API was introduced in NT 4.0, but was a stub that returned STATUS_NOT_IMPLEMENTED. NT 5.0 (2000) fleshes out the API and adds more commands.
7	**Plug-and-Play**	Like the Power API, some of these were introduced in NT 4.0 as unimplemented functions. NT 5.0 fleshes them out and adds more.
8	**Objects**	Object manager namespace objects are created and manipulated with these routines. A couple of these, like NtClose, are general in that they are used with any object type.
9	**Registry**	Win32 Registry functions basically map directly to these APIs, and many of them are documented in the DDK.
10	**Local Procedure Call**	LPC is NT core interprocess communications mechanism. If you use RPC between processes on the same computer you are using LPC indirectly
11	**Security**	The Native security APIs are mapped almost directly by Win32 security APIs.
12	**Processes and Threads**	These functions control processes and threads. Many have direct Win32 equivalents.
13	**Atoms**	Atoms allow for the efficient storage and referencing of character strings.
14	**Error Handling**	Device drivers and debuggers rely on these error handling routines.
15	**Execution Environment**	These functions are related to general execution environment.
16	**Timers and System Time**	Virtually all these routines have functionality accessible via Win32 APIs.
17	**Synchronization**	Most synchronization objects have Win32 APIs, with the notable exception of event pairs. Event pairs are used for high-performance interprocess synchronization by the LPC facility.
18	**Memory**	Most of NT virtual memory APIs are accessible via Win32.
19	**File and General I/O**	File I/O is the best documented of the native APIs since many device drivers must make use of it.
20	**Miscellaneous**	These functions do not fall neatly into other categories.
21	**Jobs**	These functions implement Job objects, which are new to NT 5.0. They are essentially a group of associated processes that can be controlled as a single unit and that share job-execution time restrictions.

Table 2 – Native API categories

Table 2 shows that in Windows we have many system calls, for every type of work. Linux give us many information with its source code on its system calls, while Windows does not give us any information on its system call.

For every native API category we could analyse what are the Windows critical system calls, but to do this we must implement a general-purpose monitor for these system call. Now this is not in our work objectives.

3 THE WHIPS PROTOTYPE

WHIPS is a prototype for the detection and the prevention of the invocation of dangerous system calls in Windows. This prototype is based on the initial idea related to the REMUS Project [BGM02]. REMUS is a Reference Monitor (RM) for Linux OS and, in its first version, it was implemented like a RM embedded in the Linux kernel: a patch to kernel source code and a recompiling process. The new version of REMUS is implemented with a dynamic loadable module of Linux kernel.

WHIPS is implemented as a kernel driver, also called kernel module, using undocumented structure of Windows kernel and also the routines typically employed for drivers development [BDP99]. The WHIPS prototype can be seen as a system call RM for Windows.

3.1 Reference Monitor for Windows XP

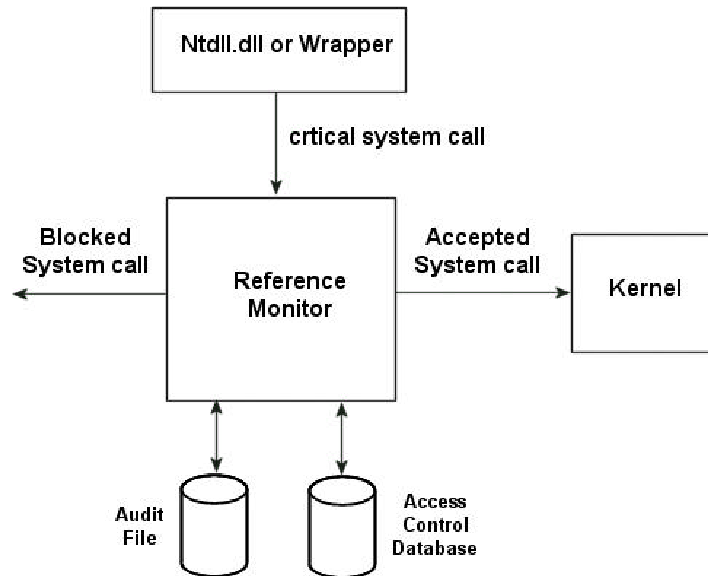


Figure 9 – WHIPS Reference Monitor.

A *Reference Monitor* is a black box that filters every critical system call invoked by a process and establishes if the critical system call is dangerous, as defined in Section 2.3.2. If the system call is not dangerous it will be passed to the kernel for the execution, otherwise it will be stopped and not executed. RM control policies are established by a small database called *Access Control Database* (ACD).

ACD includes rule for a subset of every critical system call and every dangerous process (Cartesian product). For every critical system call, if a rule exists in the ACD matching system call name, parameters and invoking process, the system call is executed, otherwise it is stopped. **The ACD defines the allowed actions for RM.** Now we examine the RM implemented by WHIPS.

Every time a process invokes a critical system call through *ntdll.dll* or a wrapper (a code that rise int 0x2E), the dangerousness of the process is checked by WHIPS RM; the RM checks the match through dangerous process, critical system call and its parameters with the ACD rules. If a rule exists that satisfies this match, the native API is executed otherwise is not executed because it is a dangerous system call.

The technique we have used in WHIPS, suggested by *Russinovich, Dabak et al.* [Cru97, BDP99], replaces the native APIs pointers in the System Service Table (fig. 10), with pointers to new native APIs supplied by us. New native APIs are wrappers to original native APIs and implements RM concepts.

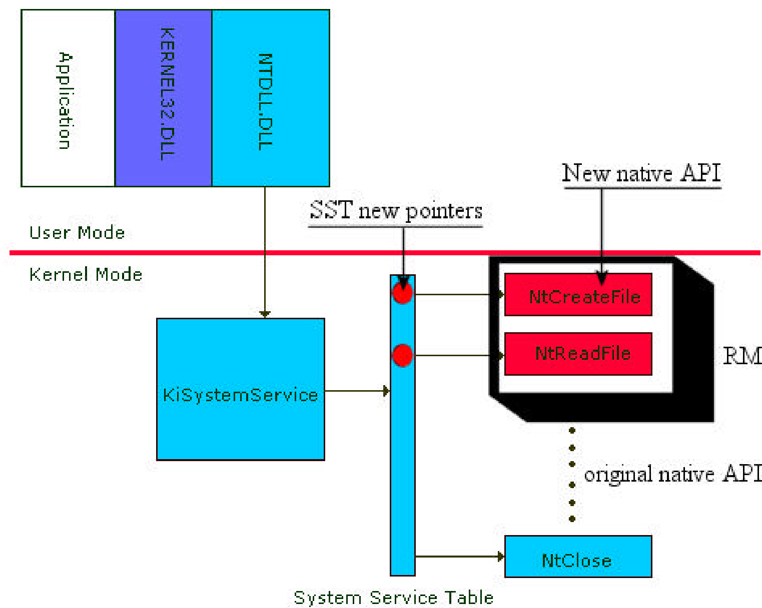


Figure 10 – WHIPS implementation architecture.

For every original critical native API we have introduced a new native API, where it has the same function name with a prefix *New* (ex. NewNtCreateFile). This new API analyses its invoking process and its parameters: if invoking process is dangerous and API is critical we know that the native API is dangerous, so original API is not called and not executed, otherwise it is called and executed.

3.1.1 Access Control Database

In the WHIPS prototype the *Access Control Database*, ACD, is implemented by a simple text file (protected by the system ACL and accessible only to the Administrator). The structure of a generic rule in the ACD is:

Rule Type	Process name	Native API Name	ParamAPI [1] … ParamAPI [n]
Defines if the rule is an effective rule or a logging rule.	Dangerous invoking process Name.	Critical native API name (with prefix Nt).	Parameters of the native API.

Table 3 –ACD Rule schema

- *Rule Type*: can be *debug* or *rule*. When the type is *rule*, this means that the rule filters the execution of system call. When the type is *debug*, the execution of a critical system call will be traced and not filtered.

- *Process Name*: is the name of the executable image that has activated the dangerous process. This name is a string that identifies only the name and not the complete path. WHIPS prototype works entirely in kernel-mode and it has not access to process block to retrieve the complete path of executable image. This information is accessible only in user-mode.

- *Native API*: is the name, with prefix *Nt*, of the critical native API invoked by *Process*.

- *Param [1..N]*: are the actual parameters of the critical system call.

These ACD rules state that the process *Process Name* can execute the specific *Native API* with the specific *Param[1..N]*. Now we examine some functions of WHIPS implementation.

3.1.2 System Service Table patch implementation

WHIPS is a kernel module, also called a driver in Windows. Now we examine a C-like representation of the source code that realize the patch to the System Service Table (SST) of Windows.

Figure 11 shows the main function of the WHIPS prototype. This is the common main function of all the drivers in Windows OS. This function does not driver any peripheral of the System and the only work that it does is calling the *HookServices* function at system start-up.

```
DriverEntry(DriverObject)
{
        ProcessNameOffset=GetProcessNameOffset();
        IoCreateDevice(DriverObject,…,&deviceName,,…,&deviceObject);

        if (CreateDevice_success) {
                IoCreateSymbolicLink (&deviceLink,&deviceName);
                HookServices();
                DriverObject->MajorFunction[IRP_MJ_CREATE]                  =
                DriverObject->MajorFunction[IRP_MJ_CLOSE]                   =
                DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL]          = DriverDispatch;
                DriverObject->DriverUnload                                  = DriverUnload;
                return;
        }
      return;
}
```

Figure 11 – WHIPS main function.

Figure 12 shows the C-like representation of the System Service Table (SST) patch. The first operation that it does is to load the ACD database in the kernel memory with *LoadDB* function, and then it patches the

SST. With macro *SYSTEMSERVICE* it first saves the old reference to the native API in *OldNtApiName*, then substitutes the old references in the SST with the new references to the new native APIs supplied by us.

```
HookServices()
{
        LoadDB("RmDB.rbt", &ruleArrayRM, &numruleRM);
        OldNtCreateFile         =SYSTEMSERVICE(ZwCreateFile);
        OldNtOpenFile           =SYSTEMSERVICE(ZwOpenFile);
        OldNtDeleteFile         =SYSTEMSERVICE(ZwDeleteFile);
        OldNtOpenProcess        =SYSTEMSERVICE(ZwOpenProcess);
        OldNtUnloadDriver       =SYSTEMSERVICE(ZwUnloadDriver);
        OldNtLoadDriver         =SYSTEMSERVICE(ZwLoadDriver);
        OldNtClose              =SYSTEMSERVICE(ZwClose);

        Disable_Interrupt;
        SYSTEMSERVICE(ZwCreateFile)     =NewNtCreateFile;
        SYSTEMSERVICE(ZwOpenFile)       =NewNtOpenFile;
        SYSTEMSERVICE(ZwDeleteFile)     =NewNtDeleteFile;
        SYSTEMSERVICE(ZwOpenProcess)    =NewNtOpenProcess;
        SYSTEMSERVICE(ZwUnloadDriver)   =NewNtUnloadDriver;
        SYSTEMSERVICE(ZwLoadDriver)     =NewNtLoadDriver;
        SYSTEMSERVICE(ZwClose)          =NewNtClose;
        Able_Interrupt;
        return;
}
```

Figure 12 –System Service Table patch.

3.1.3 New native API Implementation

Figure 13 shows the representation in C-like of the *NewNtOpenProcess*. When a process calls the NtOpenProcess, the new native API NewNtOpenProcess analyse its parameters. In this case it detects the process name of its invoking process, and the process name that will be opened by the native API.

After it stores this data in a temporary rule called *rule*. This rule has a structure much similar to a rule in the ACD, and we have cerate it to simplify our work.

Next thing to do is evaluate if the invoking process is a dangerous process. We know that the process is critical because we have done a patch to this native API. The function is*ProcessDangerous()* analyses the Access Token of the invoking process and return *true* if the process is dangerous,*false* otherwise.

The function *VerifyDebugNativeAPI()* analyses if the native API must be traced in the debug environment, while *VerifyNativeApi()* looks in to the ACD database to find a rule that satisfy the temporary *rule*. Only if this function return true, original native API is called with the invocation of *OldNtOpenProcess()* saved in the HookServices function.

```
NewNtOpenProcess(phProcess,…,pClientId)
{
        startTimeO=KeQueryPerformanceCounter(&frequency);
        GetProcess(currProc);
        GetProcessByProcessID(pClient,pClientId);

        rule.processName=currProc;
        rule.api="ntopenprocess";
        rule.numparam=1;
        rule.api_param[0]=pClient;
        CurrentProcessIsDangerous=isProcessDangerous();
        if (VerifyDebugNativeAPI(rule,CurrentProcessIsDangerous))
                        Insert_Debug_Information;

        if (VerifyNativeAPI(rule,CurrentProcessIsDangerous)) {
                endTimeO=KeQueryPerformanceCounter(&frequency);
                Show_Overhead_Information;
                OldNtOpenProcess(phProcess,…,pClientId);
        }
        else {
                endTimeO=KeQueryPerformanceCounter(&frequency);
                Show_Overhead_Information;
        }
        sendToDebug(rule,CurrentProcessIsDangerous);
        return;
}
```

Figure 13 – An example of NewNativeAPI implementation.

3.2 Performance Evaluation

The actual impact of WHIPS on the global system performance is negligible for all practical purposes. This is mainly because the number of critical system call invocations is small with respect to the total number of instructions executed by a process. However, in order to evaluate even the minimal overhead introduced by the WHIPS prototype, we have devised further experiments based on micro benchmark. In particular, the kernel function KeQueryPerformanceCounter() exported by the kernel is used. The header function is shown in fig. 14:

```
LARGE_INTEGER KeQueryPerformanceCounter(PLARGE_INTEGER PerformanceFrequency);
```

Figure 14 – KeQueryPerformanceCounter function.

KeQueryPerformanceCounter() returns the clock ticks counter (*#tick*) from system boot, while the clock tick counter per second (*#tick/sec*) is expressed by the function *PerformanceFrequency.*

If we would calculate the execution time of a piece of code, we may consider two invocations of function *KeQueryPerformanceCounter()*. These two invocations determines respectively $\#tick_1$ and $\#tick_2$.

Assume $T_1 = \frac{\#tick_1}{PerformanceFrequency}$ is the first invocations time and $T_2 = \frac{\#tick_2}{PerformanceFrequency}$ is the second invocations time. We must consider that *KeQueryPerformanceCounter()* introduces an overhead too, and we call this $\Delta T_{OverheadKQuery}$. To estimate $\Delta T_{OverheadKQuery}$, we have measured two consecutive invocations of *KeQueryPerformanceCounter().*

Assume ΔT is the execution time of a generic source code block between two invocations of *KeQueryPerformanceCounter()*, then $\Delta T = (T_2 - T_1) - \Delta T_{OverheadKQuery}$

ΔT cannot be less than $\Delta T_{OverheadKQuery}$, so $\Delta T \geq \Delta T_{OverheadKQuery}$. This is true because if the source code block between two invocations of *KeQueryPerformanceCounter()* is empty then overhead is equal to $\Delta T_{OverheadKQuery}$.

3.2.1 Overhead of new native APIs

In figure 15 we have the *elaboration block* that contains the code to check the parameters of the native API and the level of danger of the invoking process.

To estimate overhead introduced by a generic new native API under the WHIPS prototype, consider $T_1 = \frac{startTimeO}{frequency}$ and $T_2 = \frac{endTimeO}{frequency}$, so the overhead introduced from any new native API is: $\Delta T_{OverheadNewNativeAPI} = (T_2 - T_1) - \Delta T_{OverheadKQuery}$.

```
NewNtApiName(…)
{
        startTimeO=KeQueryPerformanceCounter(&frequency);                      // T1

        [Elaboration_Block]

        if (VerifyNativeAPI(rule,CurrentProcessIsDangerous)) {
        […]
                endTimeO=KeQueryPerformanceCounter(&frequency);                // T2
                OldNtApiName(…);
                Show_Overhead_Information;
        }
        else {
                endTimeO=KeQueryPerformanceCounter(&frequency);                // T2
                Show_Overhead_Information;
        }
        […]
        return;
}
```

Figure 15 – Overhead of a generic NewNtApiName.

We have measured the execution time of the original native APIs in Windows, we call this $\Delta T_{nativeAPI}$. To do this, we have inserted a call to function *KeQueryPerformanceCounter()* before the invocation of the original API, and after the original native API calls completes (fig. 16).

Assume $T_1 = \frac{startTimeA}{frequency}$ and $T_2 = \frac{endTimeA}{frequency}$, so $\Delta T_{nativeAPI} = (T_2 - T_1) - \Delta T_{OverheadKQuery}$.

```
NewNtApiName(…)
{
        […]
                startTimeA=KeQueryPerformanceCounter(&frequency);              // T1
                OldNtApiName(…);
                endTimeA=KeQueryPerformanceCounter(&frequency);                // T2
        […]
        return;
}
```

Figure 16 – overhead of an original NtApiName.

3.2.2 Methodology of Measurement

The system utilized for the measurement is a Personal Computer with AMD Athlon CPU, with clock frequency of 1200 Mhz, 512 Mbytes of RAM and Windows 2000 OS.

For every native APIs intercepted by the WHIPS prototype, we have performed a significant number of experiments (~10.000), and we have elaborated these to obtain average times without spiced values. We have determined the average overhead introduced by every new native API, $\overline{\Delta T}_{OverheadNewNativeAPI}$, and the average time of original native API, $\overline{\Delta T}_{NativeAPI}$.

The average overhead of function *KeQueryPerformanceCounter()*, $\overline{\Delta T}_{OverheadKeQuery}$, is *~0,82 msec* on our Test PC. For every measurement:

- $\Delta T_{OverheadNewNativeAPI} = (T_{2,OverheadNewNativeAPI} - T_{1,OverheadNewNativeAPI}) - \overline{\Delta T}_{OverheadKeQuery}$
- $\Delta T_{NativeAPI} = (T_{2,NativeAPI} - T_{1,NativeAPI}) - \overline{\Delta T}_{OverheadeQuery}$.

API	**Average Execution Time**		**API Incidence**
	API Time $\overline{\Delta T}_{NativeAPI}$ (µsec)	Overhead Time $\overline{\Delta T}_{OverheadNewNativeAPI}$ (µSec)	% Overhead (%)
	A	**O**	**I**: O/A*100
NtClose	7,67886	6,36530	83%
NtCreateFile	246,74359	21,51743	9%
NtOpenFile	53,56300	20,67006	39%
NtOpenProcess	8,49002	23,23371	274%

Table 4 – Comparative table ApiTime and OverheadTime.

Table 4 compares execution time of the original native API (*A*) with the overhead introduced by the corresponding new native API (*O*). As you can see, the overheads (O) are almost the same, except for the *NtClose* native API. In fact the new native API *NewNtClose* performs few operations in the *Elaboration_Block* of fig. 15, since it accepts few parameters. As for *NewNtOpenFile* and *NewNtCreateFile*,

in *Elaboration_Block,* the name of the handle parameter of the native API must be determined, and this cause an overhead of ~20 msec. The last column (*I*) shows the percentage incidence of the new native API overhead on the execution time of original native API.

In *NewNtOpenProcess*, the overhead is bigger than *NewNtClose*, this because in *NewNtOpenProcess* we determine the name of the process parameter of the native API.

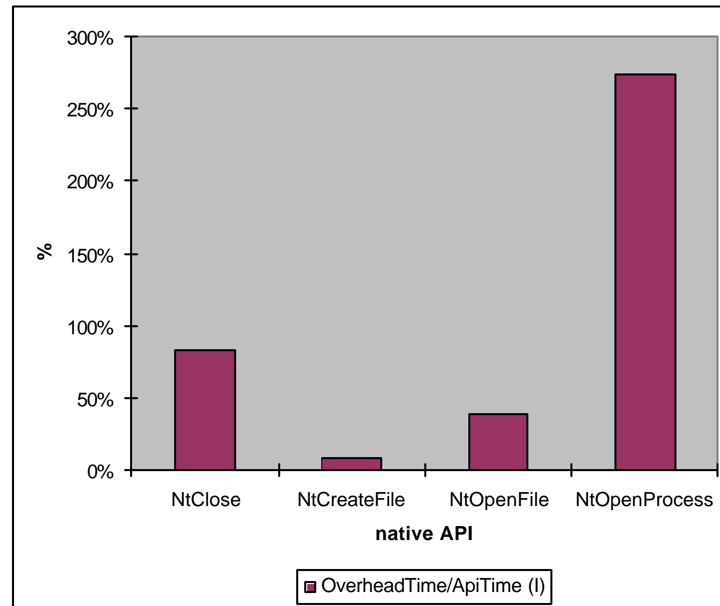


Figure 17 – OverheadTime and ApiTime ratio.

The figure 17 shows a bar chart with percentage incidence (*I*) of every native API intercepted by the WHIPS prototype. As you can see the highest incidence is *NewNtOpenProcess* incidence that introduces an overhead of 274% respect to *NtOpenProcess* execution time. Lowest incidence is of *NewNtCreateFile* that introduces an overhead of 9% respect to *NtCreateFile* execution time.

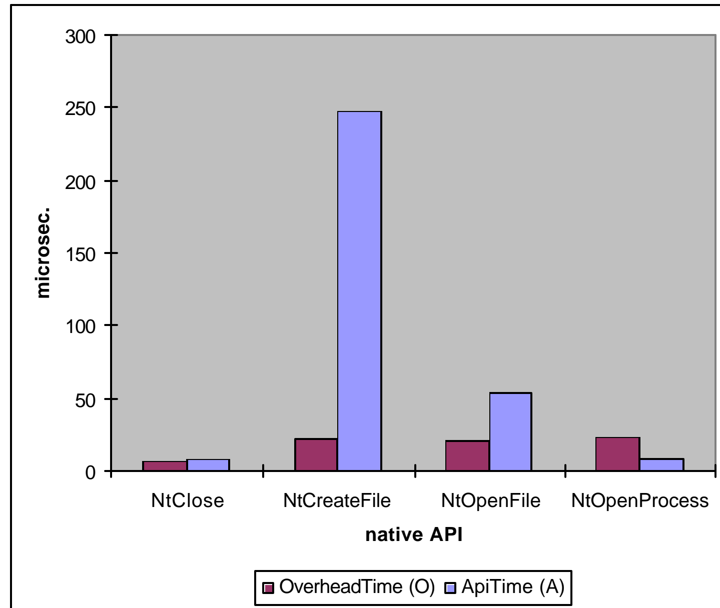


Figure 18 – Comparison between OverheadTime and ApiTime chart.

In figure 18 you can see a comparative bar chart between the overhead introduced by new native APIs (O) and the execution time of original native API (A). Higher is the difference from the overhead and the execution time and higher is the percentage incidence (I).

If we consider for ex. *NtClose*, we have a minimum difference, and this means that the overhead introduced by WHIPS is near to the execution time of original native API. In the case of *NtCreateFile* the overhead is much lesser than the execution time and the incidence is low. In *NtOpenProcess* the overhead is bigger than the original native API execution time and the incidence is high.

RELATED WORKS

As for Windows OS, we could not find related works that follows similar ideas at the OS level, perhaps this is because it is hard to design and to implement solutions in the Windows OS kernel based on undocumented functions.

A related work for the Linux OS is the REMUS Project [BGM02] which implements a reference monitor for the system call invocations in a loadable Linux kernel module. In REMUS, root processes and setuid processes are privileged processes, and a dangerous system call is a critical system call invoked by a dangerous process.

CONCLUDING REMARK

Our work defines privileged processes in Windows OS and proposes a methodology to discover the processes that can be dangerous for the system. We have two ways, one implemented in our prototype, and

the other in study. Defining dangerous processes is the first and much important step in this type of HIPS. Next, we have characterized when a system call is critical in Windows OS. Union of dangerous processes and critical system calls lead to the concept of dangerous system calls (see Section 2.3.2).

We have implemented the WHIPS prototype based on the above concepts. WHIPS stops common exploits that use buffer overflow technique to do *privilege escalation* on a system. If a malicious user wants to execute a *shell* in a context of the exploited process, WHIPS will prevent the attack by stopping the execution of the dangerous system call that invokes the shell. This system call is dangerous because it is critical and a dangerous process invokes it.

Future works include the inspection of all the native API in Windows OS for a full classification of the system calls according to their level of danger for the system security. Another step could be to implement a Web-service like *Windows Update*, named *WHIPS ACD update* that permits to download new sets of rule to configure the ACD automatically. This simplifies the definition of the rule in the ACD database. WHIPS could be more efficient if implemented directly in Windows kernel, but to do this we must access the source code of the Windows kernel, and at the moment this is not allowed by Microsoft license agreement.

ACKNOWLEDGEMENTS

We would like to thank Dejan Maksimovic for useful suggestions on the implementation of WHIPS prototype. The authors gratefully acknowledge the anonymous reference for their helpful comments.

REFERENCES

[Ab95] Abrams et al., "Information Security: An Integrated Collection of Essays", IEEE Computer Society Press, 1995.

[Al96] Aleph One, "Smashing the stack for fun and profit", Phrack Magazine, vol 49, 1996.

[An01] Anonymous, "Maximum Windows 2000 Security", SAMS Publishing, 2001.

[Ba98] Badger, "Information Security: From Reference Monitors to Wrappers", Trusted Information Systems, IEEE AES Systems Magazine, Mar. 1998.

[BGM02] Bernaschi, Gabrielli, Mancini, "REMUS: a security-enhanced operating system", ACM Transactions on Information and System Security, Vol. 5, No. 1, pp. 36-61, Feb. 2002. http://remus.sourceforge.net/.

[BDP99] Borate, Dabak, Phadke, "Undocumented Windows NT", M&T Books, 1999.

[CRu97] Cogswell, Russinovich, "Windows NT System-Call Hooking", Dr. Dobb's Journal, p. 261, 1997.

[Co00] Cowan et al, "Buffer Overflows: attacks and defences for the vulnerability of the decade", Proceedings IEEE DARPA Information Survivability Conference and Expo, Hilton Head, South Carolina, 2000.

[CuR01] Cunnigham, Russel, "Hack Proofing", McGrawHill, 2001.

[Du01] Dutertre, Riemenschneider, Saidi, Stavridou, "Intrusion Tolerant Software Architectures", Proceedings IEEE DARPA Information Survivability Conference and Exposition, Anaheim, California, 2001.

[HLB01] Howard, LeBlanc, "Writing Secure Code", Microsoft Press, 2001.

[ISS98] ISS, "Network- vs. Host-Based Intrusion Detection", Internet Security Systems, Atlanta, 1998, http://documents.iss.net/whitepapers/nvh_ids.pdf.

[MPSW03] Moore, Paxson, Savage, Shannon, Staniford, Weaver, "Inside the slammer worm", IEEE Security&Privacy, pp.33-39, July-August 2003.

[Mi02a] Microsoft, "Well-Known Security Identifiers in Windows 2000", Knowledge Base 243330, 2002, http://support.microsoft.com/default.aspx?scid=KB;EN-US;Q243330&.

[Mi02b] Microsoft, "Always Preemptible and Always Interruptible", Microsoft Software Development Network (MSDN), 2002,

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/kmarch/hh/kmarch/intro_525j.asp.

[Ne00] Nebbet, "Windows NT/2000: Native API reference", Macmillan Technical Publishing (MTP), 2000.

[On99] Oney, "Programming the Microsoft Windows Driver Model", Microsoft Press, 1999.

[Osr03] OSR Open System Resources Inc, "Nt vs. Zw - Clearing Confusion On The Native API", The NT Insider, Vol 10, Issue 4, July-August 2003, Published: 15-Aug-03.

[RuS01] Russinovich, Solomon, "Inside Windows 2000: Third Edition", Microsoft Press, 2001.

[Ru98] Russinovich, "Inside the Native API", Systems Internals, 1998, http://www.sysinternals.com/ntdll.htm.

[Scm01] Schmidt, "Microsoft Windows 2000 Security Handbook", Que, 2001.

[Scr01] Schreiber, "Undocumented Windows 2000 Secrets", Addison Wesley, 2001.

SUMMARY

Authors 1
Abstract 1
1 Introduction 1
2 Security Problem: privileged processes and critical system calls 2
2.1 Process Security Context.. 2
2.1.1 Security Identity Descriptor.. 3
2.1.2 Access Token 3
2.1.3 Impersonation 5
2.1.4 Windows Privileges 5
2.2 Privileged and dangerous processes 6
2.2.1 Services 6
2.2.2 Services Identification.. 7
2.3 Critical and dangerous system calls 8
2.3.1 Native APIs: Windows system calls 8
2.3.1.1 System Service dispatcher 10
2.3.2 Critical and Dangerous native APIs 11
2.3.3 Native API Classification..12
2.3.3.1 Native API Category..12
3 The WHIPS prototype..13
3.1 Reference Monitor for Windows XP 13
3.1.1 Access Control Database..14
3.1.2 System Service Table patch implementation 15
3.1.3 New native API Implementation 16
3.2 Performance Evaluation 17
3.2.1 Overhead of new native APIs 18
3.2.2 Methodology of Measurement 19
Related Works..21
Concluding Remark..21
Acknowledgements 22
References 22
Summary 24