A Host Intrusion Prevention System for Windows Operating Systems

Roberto Battistoni, Emanuele Gabrielli, and Luigi V. Mancini

Dipartimento di Informatica , Università di Roma "La Sapienza"
Via Salaria 113, 00198 Roma
r.battistoni@computer.org, {gabrielli,lv.mancini}@di.uniroma1.it

Abstract. We propose an intrusion prevention system called WHIPS that controls, entirely in kernel mode, the invocation of the critical system calls for the Windows OS security. WHIPS is implemented as a kernel driver, also called kernel module, by using kernel structures of the Windows OS. It is integrated without requiring changes to either the kernel data structures or to the kernel algorithms. WHIPS is also transparent to the application processes that continue to work correctly without source code changes or recompilation. A working prototype has been implemented as a kernel extension and it is applicable to all the Windows NT family OS, e.g. Windows 2000/XP/2003. The WHIPS first contribution is to apply the system call interposition technique to the Windows OS, which is not open source. It is not straightforward to apply this technique to Windows OS, also because Windows kernel structures are hidden from the developer, and furthermore, its kernel documentation is poor.

1 Introduction

Attacks on the security of network clients and servers are often based on the exploitation of flaws that are present in a specific application process. By means of well known techniques [Al96,Co00], a malicious user may corrupt one or more memory buffers so that while returning from a function call, a different piece of code, which is injected by the attacker, is executed by the flawed application process. Of course, 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 in the privileged application context and allow the attacker to become a system administrator. An example of a recent exploit using buffer overflow is the slammer worm [MPSW03] that attacks the MS-SQL server for Windows 2000/XP to gain high privileges and then saturates the network bandwidth causing a denial of service attacks.

This paper presents the design and implementation of a Host Intrusion Prevention System (HIPS) for Windows OS that immediately detects security rules violations by monitoring the system calls made by the application processes. The proposed prototype, working entirely in kernel mode employs interposition at the

P. Samarati et al. (Eds.): ESORICS 2004, LNCS 3193, pp. 352–368, 2004.

system call interface to implement the access control functionality, and requires no change to the kernel code or to the syntax and semantics of existing system calls. Basically, the system call execution is only allowed when the invoking process and the value of the system call arguments comply with the rules kept in an Access Control Database (ACD) within the kernel. The task of the proposed HIPS is to protect the Windows OS against any technique that would 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 propose to apply a similar technique to the Windows 2000/XP/2003 family.

The HIPS being proposed here is called WHIPS, Windows-NT family Host Intrusion Prevention System. Indeed, Intrusion Prevention Systems (IPSs) strive to stop an intrusion attempt by using a preventive action on hosts to protect the systems under attack. Our WHIPS prototype runs under Windows 2000/XP and Windows 2003. Herein, by the term Windows we refer to Windows XP, but the consideration and the prototype design are applicable to all the Windows NT family OS. WHIPS' first contribution is to apply the system call interposition technique to the Windows OS. Though this technique is generally known, we could not find any related works that follow similar ideas as for Windows OS which is not open source. This is mainly because it is hard to design and to implement solutions in the Windows OS kernel structures which are hidden from the developer. On the contrary, there are many IPS that are implemented as wrapper executed in user mode such as [Ep00]. Moreover, many implementations of the system call interposition technique are well known on Linux OS. A related study 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 defined as a critical system call invoked by a privileged processes.

This 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 calls are invoked by the user processes. Section 3 proposes the WHIPS prototype, showing the implementation, a brief example of its effectiveness, and the performance analyzes of the prototype.

2 Privileged Processes and Critical System Calls

In order to gain control of an OS, an attacker has to locate a target process that runs with high privileges in the system. For example, if the OS belongs 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, then we characterize when a process is *privileged* or *dangerous* and when a system call is *critical* or *dangerous* in Windows.

2.1 Windows Processes Security Context and Privileges

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

Security Identity Descriptor: SIDs identify the entities that execute operations in a Windows system and may represent users, groups, machines or domains. A SID contains a so-called *RID* (relative number) field that distinguishes two SIDs otherwise equal in a Windows system. Every Windows system has a lot of SIDs; some of them identify particular users or groups and are called Well-Known SIDs [Mi02a].

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* builds an initial token that represents the user rights, and links this token to the users shell process. All the processes created by the user inherit a copy of the initial AT. We have two types of AT: *primary token* and *impersonation token*. Every process has an AT called *primary token*. Every process in Windows 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, or may obtain a restricted copy of the primary token by the *impersonation* mechanism.

Impersonation: It 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]. To avoid an improper use, Windows does not permit to a server to impersonate a client process without the client consensus. Some impersonation levels follow: *SecurityAnonymous, SecurityIdentification, SecurityImpersonation, SecurityDelegation*. If a client does not choose an impersonation level, *SecurityImpersonation* is the default.

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 AT 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.

Definition 1. *A dangerous privilege is a Windows privilege that can be used by a malicious user to compromise the availability, the confidentiality and the integrity of the system.*

Examples of some dangerous privileges reported in [HLB01] are: *SeBackupPrivilege*; *SeTcbPrivilege*; *SeDebugPrivilege*; *SeAssignPrimaryTokenPrivilege*; *SeIncreaseQuotaPrivilege*.

2.2 Privileged and Dangerous Processes

As discussed above, some privileges are dangerous in Windows OS and if we want to know if a process is dangerous, we can look to the process AT of the user that activates this process. A malicious user can attack a dangerous process executing malicious code in its security context and gaining all the process privileges. We can say that if a process privilege is dangerous, then the process is dangerous too. To identify a dangerous process we can look for dangerous privileges into the process AT. Now we introduce some definitions to summarize the concepts:

Definition 2. *A privileged process is a process with some Windows privilege.*

Definition 3. *A dangerous process is a privileged process that has some dangerous privilege.*

If in the AT privileges list there are one or more dangerous privileges, the process, owner of the AT, belongs to the set of dangerous process. In the following, we discuss a particular set of privileged processes: the *Windows Services*.

Services Identification. 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*). This account belongs to group *Administrators* and it is very powerful because it has many dangerous privileges. This is a critical account for the Windows security.

Often a careful analysis of services by the administrator could restrict the services privileges. This could be done with a new account created for the specific service, where this account has less privileges then the *System* account. Following this idea, in Windows XP/2003 there are two new accounts for services: *local service* and *network service.* These new accounts have the 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 the *System* account [HLB01]. *LocalSystem* has almost all the Windows privileges whereas each of *LocalService* and *NetworkService* has a subset of the *LocalSystem* privileges. In general, the SIDs in the process AT identify the services, precisely the so-called *Well Known SIDs*. We have two possibilities:

if the service logs onto the system with *LocalSystem* account, the user account SID in the AT is equal to string *S-1-5-18*, Local System SID. Otherwise, we must look in the AT group SIDs; the process is a service if there is the *Well-Known SID Service* represented by the string *S-1-5-6*. Summarizing, the following rules help us to know exactly when a process is a service:

Proposition 1. *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.*

Proposition 2. *Access Token Group SIDs contains Service SID ⇒ process is a Service.*

Proposition 3. *Access Token User SID is equal to LocalSystem SID ⇒ process is NOT necessarily a service.*

Note that if user SID is *LocalSystem* the process owner of the AT is not necessarily a service, it could be a system process too. If we consider only the first rule, we will find a set of processes that contains the set of services, but is not necessarily equals to this set.

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 *critical*.

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 simply call APIs, the APIs in user-mode space.

Four dynamic libraries export APIs of the Win32 subsystem: *user32.dll, gdi32.dll, kernel32.dll, advapi32.dll*. The APIs in *user32.dll* and *gdi32.dll* invoke the APIs implemented in kernel mode by *win32k.sys* module, which is the kernel module of the Win32 subsystem. The APIs exported by *kernel32.dll* (system APIs) use a particular library named *Ntdll.dll* that invokes native APIs in the kernel. Native APIs invoked by ntdll.dll are the Windows system calls.

Figure 1 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*.

Figure 2 shows an example of the native API *NtCreateFile,* obtained disassembling *ntdll.dll*. Function *NtCreateFile* loads registry EAX with the index $0x1A$ of the native API in a particular table called *System Service Table* (KiServiceTable), then EDX registry points to the user-mode stack, *ESP+04*, where

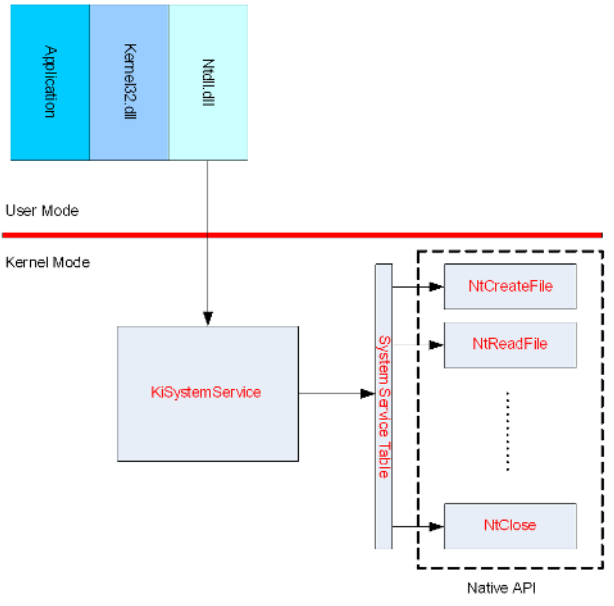

Fig. 1. System Service Table (SST)

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

Fig. 2. NTCreateFile assembly code

there are the parameters of native API, and finally raises interrupt $0x2E$ that executes the *System Service Dispatcher* of Windows. System Service Dispatcher is the kernel routine that invokes the true native API in the kernel. Not all the native API exported by *Ntdll.dll* are exported by *ntoskrnl.exe*. This seems to prevent the unauthorized use of particular and dangerous native APIs within any module implemented as a kernel driver. Disassembling the library *ntdll.dll*, we can observe that every *Nt* native API and its corresponding *Zw* native API have the same assembly code, fig. 2. 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 the example of figure 2, see also [Ne00,Osr03,Scr01] for details.

System Service Dispatcher. Dispatcher of interrupt $0x2E$ is the *System Service Dispatcher* routine. It is implemented in the executive layer of the Windows kernel, through the kernel function *KiSystemService*. 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 these exported functions call the native APIs in Windows kernel. When *KiSystemService* is invoked, the dispatcher runs a series of checks. First, it controls the validity of index passed in EAX register, then it controls if the argument space expected for the native API parameters is correct and finally executes the native API in the kernel. When *KiSystemService* invokes the native APIs, it uses a structure called *System Service Descriptor Table* (*SDT*), represented by the *KeServiceDescriptorTable* structure [Scr01,HLB01]. *KeServiceDescriptorTable* has two table pointers: *KiServiceTable* (*System Service Table, SST*) and *KiArgumentTable*. The first table contains an index for every native API, used by native API code in *ntdll.dll*, to invoke the corresponding native API in the kernel. The second table contains, for every native API, the allocation space for this API parameters. This space is used for the kernel-stack memory allocation.

Critical and Dangerous Native APIs. A native API can be considered 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 it has dangerous parameters. Consider a simple example: the native API *NtOpenFile*. Typically this native API opens a 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 open is equal to *c:\winnt\cmd.exe*, the Windows shell, *NtOpenFile* with this particular parameter is critical because it could be used to open a system administrative session. So we introduce some definition:

Definition 4. *A parameter of a Native API is dangerous if it can be used by a malicious user to compromise the availability, the confidentiality and the integrity of the system.*

Definition 5. *A critical system call is a native API that could be invoked with dangerous parameters.*

Definition 6. *A dangerous system call is a critical system call invoked by a dangerous process.*

Note that a *critical system call* is dangerous for the system only if the invoking process is a *dangerous process*. A dangerous process that calls a native API with dangerous parameters may represents an attack of a malicious user.

Native API Classification. Native APIs in Windows 2000 and XP are about 250, and only 25 of them are documented by Microsoft within the Driver Development Kit (DDK). All others native APIs are not documented. A good support for native API documentation is *Windows NT/2000: Native API reference* [Ne00] which is not an official Microsoft documentation of Windows OS.

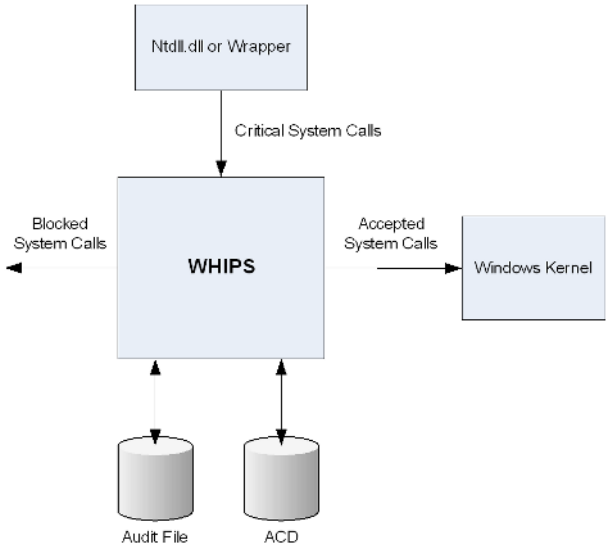

Fig. 3. WHIPS Reference Monitor

Table 1 represents a first classification of native APIs by category (21 categories) which is derived from [RuS01,Ru98], and shows that in Windows we have many system calls. Linux give us more information with its source code on its system calls, whereas Windows does not give us any information on its system call.

3 The WHIPS Prototype

WHIPS is a Reference Monitor (RM) for the detection and the prevention of Windows dangerous system calls invocation. This prototype is based on an initial idea related to the REMUS Project [BGM02]. REMUS is a RM for Linux OS and it 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 the routines typically employed for drivers development [BDP99]. The WHIPS prototype can be seen as a system call RM for Windows and the implementation technique utilized is the system calls interposition. As you can see in Figure 3, WHIPS is a module that filters every critical system call invoked by a Windows service and establishes if the critical system call is dangerous by checking its actual parameters. 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).

The ACD defines the allowed actions on the system by means of a set of rules. Every time a dangerous process invokes a critical system call through *ntdll.dll* or a *wrapper* (a code that rise *int 0x2E*), the call of the process is checked by WHIPS that matches the process name, the critical system call with its parameters, with

Table 1. Native API categories

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.
5	Channels	Provide access to a communications mechanism.
6	Power	Native API for power management.
7	Plug-and-Play	Like the Power API.
8	Objects	Object manager namespace objects are created and manipulated with these routines.
9	Registry	Win32 Registry functions basically map directly to these APIs.
10	LPC	LPC is NT core interprocess communications mechanism.
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 are essentially a group of associated processes that can be controlled as a single unit and that share job-execution time restrictions.

the ACD rules. If a rule exists that satisfies this invocation, the native API is executed otherwise is not executed because the system call invoked is classified as dangerous.

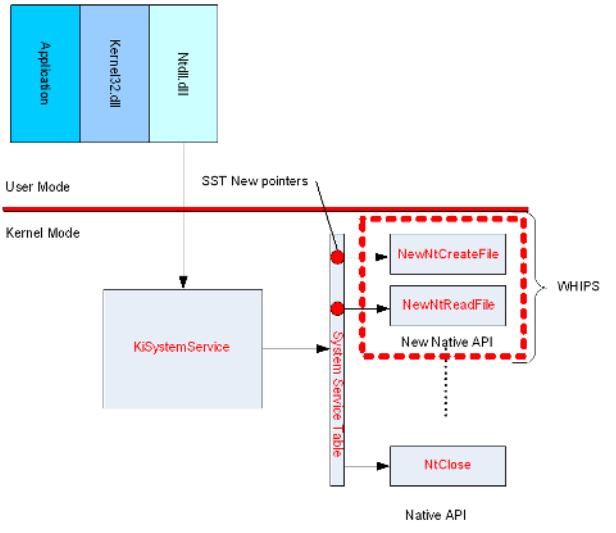

Fig. 4. WHIPS implementation architecture

The implementation technique used by WHIPS, suggested by [CRu97,BDP99], replaces the native APIs pointers in the System Service Table (fig. 4), with pointers to the new native APIs supplied by the prototype. The new native APIs are wrappers to original native APIs and implement the RM checks.

For every original critical native API we have introduced a new native API, which has the same function name with a prefix *New* (ex. *NewNtCreateFile*). This new API analyzes its invoking process and its parameters. If the invoking process is dangerous and the native API is critical, a check is performed on the actual parameters considering the rule in the ACD. If there is not a corresponding rule in the ACD the original native API is not invoked, since the native API may be dangerous. Otherwise it is invoked and executed.

3.1 Access Control Database

The ACD is implemented by a simple text file (protected by the system ACL and accessible only to the Administrator or Administrative groups). ACD is loaded at the driver start-up and it is allocated entirely in main memory; no other accesses are needed to the file and this avoid performance problem. A generic rule in the ACD is represented in table 2.

These ACD rules state that *Process* can execute the specific *Native API* with the specific *Param[1..N]*. The specification of the rules to insert into the ACD is possible in a manual or automatic manner. In the first case, the administrator of the system must specify all the Native APIs critical and allowed for the system. This can be done with approximation, step by step, but a period of control of the system is necessary. In fact some omissions could prevent the correct operation of some application.

Table 2. ACD Rule schema

Rule Type	Process name	Native API Name	ParamAPI [1..n]

- *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 but not blocked.
- *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 because 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 legal actual parameters of the critical system call.

The second case is to implement a robot that analyzes all the system calls invoked by the system in a trusted environment and create the correct rules for the ACD.

The performance of the system depends on the number of rules in the ACD and how the matching algorithm is implemented. In this first prototype the ACD is scanned sequentially with a computational cost of $O(n)$, where n is the number of rules in the ACD. In a preliminary study we have estimated that the number of these rules is not more than 1000, so the sequential search seems computational acceptable. Alternatively one could implement other more efficient algorithms to lower the computational cost of the rule search.

3.2 System Service Table Modified

WHIPS is a kernel module, also called a driver in Windows. Now we examine a C-like representation of the source code that implements the patch to the System Service Table (SST) of Windows. The main function of the WHIPS prototype is the common main function of all the drivers in Windows OS, called *DriverEntry*. In WHIPS,this function does not drive any peripheral of the system and the only work that it does is calling the *HookServices* function at driver start-up.

Figure 5 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*, and then it patches the SST. With macro *SYSTEMSERVICE*, the *HookServices* function saves the old references to the native APIs in *OldNtApiName*, and then substitutes the old references in the SST with the new references to the new native APIs supplied by the prototype.

3.3 New Native API Implementation

To explain the implementation of the new native APIs, figure 6 shows the representation in C-like of the *NewNtOpenProcess*. When a process wants to call the

```
HookServices() {
  LoadDB("RmDB.rbt", &ruleArrayRM, &numruleRM);
  OldNtCreateFile=SYSTEMSERVICE(ZwCreateFile);
  .
  .
  OldNtClose=SYSTEMSERVICE(ZwClose);
  Disable_Interrupt;
  SYSTEMSERVICE(ZwCreateFile)=NewNtCreateFile;
  .
  .
  SYSTEMSERVICE(ZwClose)=NewNtClose;
  Enable_Interrupt;
  return;
}
```

Fig. 5. WHIPS Patch function

NtOpenProcess, it really calls the corresponding new native API *NewNtOpenProcess*. The new native API detects its invoking process name and its only parameter: the process name that will be opened by the native API. Then the *NewNtOpenProcess* stores this data in a temporary rule called *rule*. The format of this temporary rule is similar to the format of the rules in the ACD, this is to simplify the check of the rule.

Next the procedure evaluates if the invoking process is a dangerous process. The function *isProcessDangerous* analyzes the AT of the invoking process and return *true* if the process is dangerous, *false* otherwise. Remember that in this version of the prototype dangerous processes are Windows services or Windows system processes (refer to section 2.2). The function *VerifyDebugNativeAPI* analyzes if the native API must be traced in the debug environment, whereas *VerifyNativeApi* looks into the ACD database to find a rule that allows the execution of the invoked API. Only if this function return true, original native API is called with the invocation of *OldNtOpenProcess* saved in *HookServices.*

3.4 Prototype Effectiveness

The WHIPS prototype can be tested with the *Web Server File Request Parsing* vulnerability of *IIS* in Windows NT/2000 up to Service Pack 2. This vulnerability permits to execute shell command in a web browser with an URL not well-formed. If for example, we call the URL:

http://host/scripts/..%255c..%255cwinnt/system32/cmd.exe?/c+dir+c:

We obtain a shell that shows the list of files of the *C:* partition hard drive. But, this vulnerability may become very dangerous if a shell command is passed as an argument to delete or to modify some configuration files not protected by the ACLs. Now we explain how this vulnerability works.

The Web Server IIS, *inetinfo.exe*, executes in its thread the shell command for an incorrect interpretation of the URL format. This thread has *IUSR_HOST*

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

Fig. 6. WHIPS New Native API implementation

privileges (guest privileges). So the Web server erroneously executes code out of the web server directory. In other situations, the attacker could obtain administrator privileges if the threads owned by a process have the same privileges of the administrator.

If WHIPS prototype is running, the thread created by privileged process *inetinfo.exe* is analyzed. In the ACD database is not present a rule for the native API *NtOpenFile* with shell (*cmd.exe*) parameters, for IIS privileged process (and its threads), so the execution of the native API is stopped and also the attack to the Web server. This experiment shows that WHIPS permits only the allowed Native APIs to operate on the systems, and consequently stops every malicious actions of the dangerous processes.

3.5 Performance Evaluation

The actual impact of WHIPS on the global system performance is negligible for all practical purposes, 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. *KeQueryPerformanceCounter(PerformanceFrequency)* returns the clock ticks counter (*#tick*) from system boot, whereas the clock tick counter per second (*#tick/sec*) is expressed by *PerformanceFrequency*. A generic

invocation time T_i of the kernel function *KeQueryPerformanceCounter* is given in microseconds by:

$$T_i = \frac{\#tick_i}{PerformanceFrequency}. \quad (1)$$

Now assume that ΔT is the execution time of a generic code block between two invocations of *KeQueryPerformanceCounter*. These two invocations determine respectively T_1 and T_2, where T_1 is the first invocation time and T_2 is the second invocations time, and ΔT=T_2-T_1.

We must consider that the execution of the function *KeQueryPerformanceCounter* introduces an overhead too, we call this $\Delta T_{OverheadKeQuery}$. To estimate $\Delta T_{OverheadKeQuery}$, we have measured two consecutive invocations of *KeQueryPerformanceCounter.*

We define the *elaboration block* as the WHIPS code block that implements the control of the native API parameters and the control on the invoking process. The overhead introduced by a generic new native API is:

$$\Delta T_{OverheadNewNAPI} = (T_{2,OverNewNAPI} - T_{1,OverNewNAPI}) - \Delta T_{OverheadKeQuery}, \quad (2)$$

whereas the execution time of the original native APIs in Windows is called $\Delta T_{NativeAPI}$ which is computed measuring a native API execution in a way similar to equation 2.

3.6 Measurements

The system utilized for the measurement is a PC with AMD Athlon CPU, with clock frequency of 1200 Mhz, 512 Mbytes of RAM and Windows 2000 OS. We have measured only four native API and specifically three critical native API: *NewNtOpenFile*, *NewNtCreateFile* and *NewNtOpenProcess*, and one not critical, *NewNtClose*. For every native API, intercepted by the WHIPS prototype, we have done a significant number of measurements (∼10.000), and we have elaborated these to obtain the average times without spiced values. We have determined $\overline{\Delta T}_{OverheadNewNativeAPI}$, the average overhead introduced by each new native API, and $\overline{\Delta T}_{NativeAPI}$, the average time of the original native API. The average overhead of function *KeQueryPerformanceCounter,* called $\overline{\Delta T}_{OverheadKeQuery}$, is $\sim$*0,82 μsec* on our test PC.

Table 3 compares the execution time of the original native API (A) with the overhead introduced by the corresponding new native API (O). This is to measure the impact of the WHIPS prototype implementation respect to the original system. As you can see, all the overheads (O) are almost the same in value, except for the *NtClose* case, because *NewNtClose* performs few operations in the *elaboration block*; in fact these new APIs determine the name of the process or the handle passed to them. The last column (I) shows the percentage incidence of the new native API overhead on the execution time of the original native API.

Table 3. Comparative table ApiTime and OverheadTime

API	Average Execution Time		API Incidence
	API Time (A) $\overline{\Delta T}_{NativeAPI}$	Overhead Time (O) $\overline{\Delta T}_{OverheadNewNativeAPI}$	% Overhead (I) I: O/A*100
NtClose	7,68 μsec	6,37 μsec	83%
NtCreateFile	246,74 μsec	21,52 μsec	9%
NtOpenFile	53,56 μsec	20,67 μsec	39%
NtOpenProcess	8,49 μsec	23,23 μsec	274%

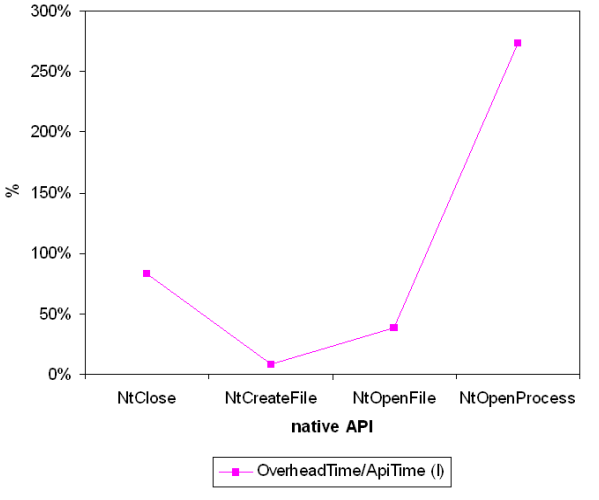

Fig. 7. Overhead and Api Time ratio

The figure 7 shows a line chart with percentage incidence (I) of every native API intercepted by the WHIPS prototype. As you can see the highest is the *NewNtOpenProcess* case 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. In figure 8 you can see a comparative line chart between the overhead introduced by new native APIs (O) and the execution time of the 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 the *NtClose*, we have a minimum difference, and this means that the overhead introduced by WHIPS is close to the execution time of the original native API. In the case of *NtCreateFile* the overhead is less 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.

4 Concluding Remarks

Our work defines privileged processes in Windows OS and proposes a methodology to discover the processes that can be dangerous for the system. Identifying

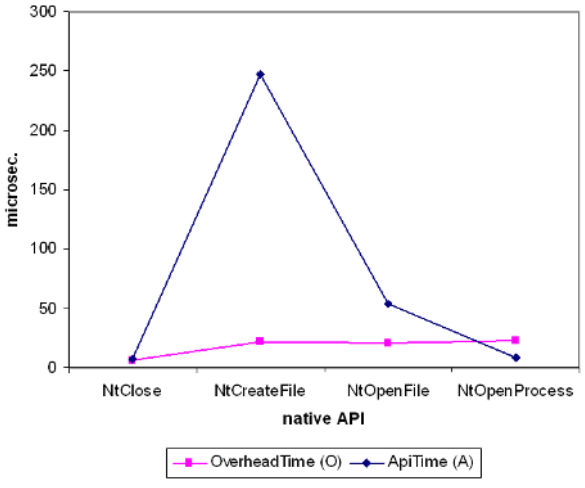

Fig. 8. Overhead and Api Time

dangerous processes is the first and most important step in the design of this type of HIPS. The *Access Token* (AT) privileges list of a process can be used to identify if a process is dangerous. In particular, if in the AT privileges list there are one or more dangerous privileges, the process, owner of the AT, belongs to the set of dangerous process. For simplicity in this paper we have focused on a subset of dangerous process: the *Windows Services*. The relation between the dangerous processes and the critical system calls leads to the concept of dangerous system calls (see section 2.3). The implementation of the WHIPS prototype is based on the above concepts. WHIPS stops common exploits that use the buffer overflow technique to carry out the privilege escalation on a system. If a malicious user wants to execute a shell in a context of the exploited service, WHIPS will prevent the attack by stopping the execution of the dangerous system call that invokes the shell.

Future research will include the inspection of the entire native API in Windows OS, for a full classification of the system calls. Another step could be to implement a Web-service like *Windows Update*, named *WHIPS Update* that allows the user to download new sets of rules in order to configure the ACD automatically. This simplifies the definition of the rule in the ACD database.

WHIPS could be even more efficient if it were implemented directly into the Windows kernel, instead of as a kernel driver, but in order to do so the source code of the Windows kernel must be accessed.

Acknowledgments

The authors gratefully acknowledge dr. R.Bianco for the helpful discussions and the anonymous reference for their helpful comments. This work is funded by the

Italian MIUR under the projects: FIRB WEB-MINDS and PRIN 2003 *WEB-based management and representation of spatial and geographic data.*

References

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

[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*, Proc. IEEE DARPA Information Survivability Conference and Expo, Hilton Head, South Carolina, 2000.

[Ep00] Epstein et al., *Using Operating System Wrappers to Increase the Resiliency of Commercial Firewalls* , Proc. ACM Annual Computer Security Applications Conference, Louisiana, USA, Dec. 2000.

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

[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&.

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

[Osr03] OSR Open System Resources Inc, *Nt vs. Zw - Clearing Confusion On The Native API*, The NT Insider, Vol 10, Issue 4, August 2003.

[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 Publishing, 2001.

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