



TECHNISCHE
UNIVERSITÄT
WIEN

VIENNA
UNIVERSITY OF
TECHNOLOGY

D I P L O M A R B E I T

Extending .NET Security to Native Code

ausgeführt am Institut für
Rechnergestützte Automation
der Technischen Universität Wien

unter Anleitung von Ao. Univ-Prof. Dr. Wolfgang Kastner
und
Univ.Ass. Dr. Christopher Kruegel
und
Univ.Ass. Dr. Engin Kirda

durch
Patrick Klinkoff

Blümelgasse 1/23
1060 Wien

Datum

Unterschrift

Abstract

Increasingly, applications are downloaded from the Internet by unsuspecting users, unaware of the intentions of the program author. To facilitate and secure this growing class of mobile code, Microsoft has introduced a new development and runtime environment called .NET framework. The framework combines a virtual machine, strong typing, and a configurable rule-set to subject the application to user-defined permissions. However, the framework also allows for access to native libraries to include legacy code or invoke the Windows API. Unfortunately, this allows an attacker to completely circumvent the framework's security mechanism.

This project extends the .NET framework Code Access Security (CAS) to native code. We investigate and compare static and dynamic analysis methods and present a system for preventing an attacker from executing malicious code not permitted by CAS. We continue by examining possible methods for an attacker to subvert the proposed system and explain how we counter them.

Kurzfassung

Mit der wachsenden Popularität des Internet werden zunehmend Programme aus dem Internet geladen, ohne den Autor oder seine Motive zu kennen. Um die Verwendung von mobilem Code sicherer zu machen, entwickelte Microsoft eine neue Entwicklungs- und Laufzeitumgebung namens .NET Framework. Dieses Entwicklungsrahmenwerk kombiniert eine virtuelle Maschine, starke Typisierung und ein konfigurierbares Regelwerk um die Sicherheitseinstellungen für Applikationen festzulegen. Zusätzlich erlaubt .NET die Einbindung von bestehenden *native* Bibliotheken wie der Windows API.

Leider erlaubt diese Verwendung von *native code* einem Angreifer die vollständige Umgehung des .NET Sicherheitsmodells. Um diese Sicherheitslücke zu schliessen, erweitern wir in der vorliegenden Arbeit das .NET Code Access Security (CAS) Sicherheitsregelwerk auf *native code*. Wir untersuchen und vergleichen sowohl statische als auch dynamische Analysemethoden und stellen ein System vor, welches verhindert, dass ein Angreifer potenziell schädliche Aktionen ausführt. Weiters untersuchen wir Möglichkeiten eines Angreifers unser System zu umgehen und erklären, wie wir diese verhindern.

Contents

| | | |
|--|---|-----------|
| 1 | Introduction | 6 |
| 2 | Dynamic Analysis | 8 |
| 2.1 | Windows Architecture | 8 |
| 2.2 | Windows API Hooking | 11 |
| 2.2.1 | Import Address Rewriting | 11 |
| 2.2.2 | Detours | 12 |
| 2.3 | Windows System Call Hooking | 14 |
| 2.3.1 | Anatomy of a System Call | 14 |
| 2.3.2 | Hooking Windows System Calls | 17 |
| 3 | Static Analysis | 19 |
| 3.1 | Control Flow Graph | 19 |
| 3.2 | Reaching Definitions Analysis | 22 |
| 3.3 | Constant Propagation | 24 |
| 3.4 | Range Analysis | 25 |
| 3.5 | Limitations – | |
| Correctness vs. Completeness | | 25 |
| 4 | .NET | 27 |
| 4.1 | Architecture | 27 |
| 4.2 | .NET Security Model | 29 |
| 4.3 | The Platform Invoke Service | 32 |
| 4.4 | .NET Remoting | 34 |
| 5 | System Design | 35 |

| | | |
|----------|--|-----------|
| 6 | Implementation | 41 |
| 6.1 | Code Access Security | 41 |
| 6.2 | Kernel Driver | 41 |
| 6.3 | .NET Remoting | 43 |
| 6.4 | Remoting Server | 44 |
| 6.5 | Page Protection | 45 |
| 6.6 | Patching the .NET Application | 47 |
| 7 | Evaluation | 48 |
| 7.1 | Functionality | 49 |
| 7.2 | Security | 50 |
| 7.3 | Performance | 51 |
| 7.4 | Limitations | 52 |
| 8 | Future Work | 54 |
| 9 | Conclusions | 56 |
| A | Popular Windows Disassemblers | 60 |
| B | Lattices | 61 |
| C | Code Listings | 63 |
| C.1 | Kernel Driver | 63 |
| C.2 | Remoting Server | 65 |
| C.3 | Remoting Proxies Sample Source | 68 |

List of Figures

| | | |
|-----|---|----|
| 2.1 | Windows Architecture | 9 |
| 2.2 | Hooking with Detours Including Trampoline | 13 |
| 2.3 | Example Schematic Control Flow | 17 |
| 3.1 | Example Control Flow Graph | 21 |
| 4.1 | .NET Architectural Overview | 28 |
| 4.2 | The Policy Resolution Process | 31 |
| 5.1 | Two Step Authentication | 38 |
| 5.2 | System Architecture | 40 |
| 6.1 | Remote Parameter Passing | 44 |
| 6.2 | Remoting Server Startup Operations | 45 |
| 6.3 | Security Layer Operations | 46 |
| B.1 | Hasse Diagram | 62 |

Chapter 1

Introduction

With the rising popularity of the Internet, applications are increasingly downloaded from remote sources. This mobile code can be a full fledged program or a small embedded application in the web-browser. Often, little or no knowledge exists about the author or his intentions. Therefore, users are susceptible to executing potentially malicious programs on their home computers. Malicious programs contain code that executes in any unauthorized or undesirable way. Different techniques have been developed to detect malicious behavior. We distinguish between systems that specify forbidden actions, such as signature-based systems, and those specifying permitted actions, called specification-based systems.

Signature-based systems use signatures of known malicious code to identify attacks. Current anti-virus software is usually signature-based. These systems require frequent updates to the signature database to remain up-to-date. This implies that previously unknown attacks cannot be identified and prevented. Also, various tricks have been developed and are employed by attackers to remain undetected. Therefore, these systems are always one step behind in the arms race with potential attackers. The advantage is that the number of false positives remains relatively low. A false positive is an attack warning generated by valid program execution – a false alarm. In contrast, a false negative is an undetected attack.

Specification-based systems use a model to specify exactly what behavior is permitted. These systems infer the existence of malicious code by comparing the program behavior to a model of *normal* behavior. One possibility of specifying the model is with rules. Generating a rule-set to minimize the number of false negatives can, however, be a tedious and complicated

task. This method provides the advantage of detecting attacks previously unknown.

We distinguish between two approaches of validating the program against the model. Static program analysis is a method to detect potentially malicious code by analyzing source or binary code of a program before execution. Primarily developed for compiler development, various tools exist to statically identify the behavior of a program. This behavior can be tested against the model to identify malicious code. In contrast, execution monitoring systems identify unauthorized activity by comparing the runtime behavior of an application with the model.

Microsoft recently introduced a new development framework, intended as the unifying model for Microsoft Windows development. The framework will be used for all Microsoft-based development, from web applications, database applications, to Windows-based user applications. The framework, called .NET framework, is modeled after Sun Microsystem's Java system. Similar to Java, it also consists of byte code executed in a virtual machine. To increase the usage of .NET in mobile code, Microsoft introduced a specification-based security model called Code Access Security (CAS). CAS tests the execution of code in the virtual machine against a set of permitted actions generated by a user defined rule-set. However, the framework also allows for calls into native code, not covered by CAS. This provides an attacker with full access to the machine and thus allows the execution of arbitrary code.

This project extends the .NET framework CAS to native code. We investigate and compare static and dynamic analysis methods and present a system for preventing an attacker from executing malicious code not permitted by CAS. We continue by investigating possible methods for an attacker to subvert the proposed system and explain how we counter them.

Chapter 2

Dynamic Analysis

Dynamic analysis is a method for monitoring the execution of a program. Contrary to static analysis, dynamic analysis is performed during execution of the monitored application. This chapter concentrates on dynamic analysis of programs running on the Windows operating system. However, the presented concepts can also be applied to programs running under different operating systems. We begin with a brief description of the Windows operating system architecture, followed by two methods used to monitor the interaction of programs with the underlying Windows operating system.

2.1 Windows Architecture

The Windows NT operating system was designed as a replacement for previous 16-bit Windows operating systems. Important requirements for understanding the architecture are:

- Robustness for recovery from errors
- Portability across different hardware architectures
- Compatibility for supporting older versions of Windows, as well as UNIX and OS/2
- Extensibility for organic growth of the operating system

Most modern operating systems, including Windows NT, separate the operating system from the applications running on it. To this end, the operating system runs in a privileged processor mode called kernel space, whereas

applications run in a non-privileged mode called user space. Privileges enjoyed in kernel space include direct access to physical devices, access to the entire physical memory space, as well as execution of certain instructions not allowed from user space¹. The separation between user and kernel mode is enforced by the CPU with the help of a context flag in a status register indicating the current mode. User space code cannot access code or data in kernel space. This ensures that user space code cannot modify or corrupt the operating system with malicious or errant applications. This feature is important for achieving the robustness goal.

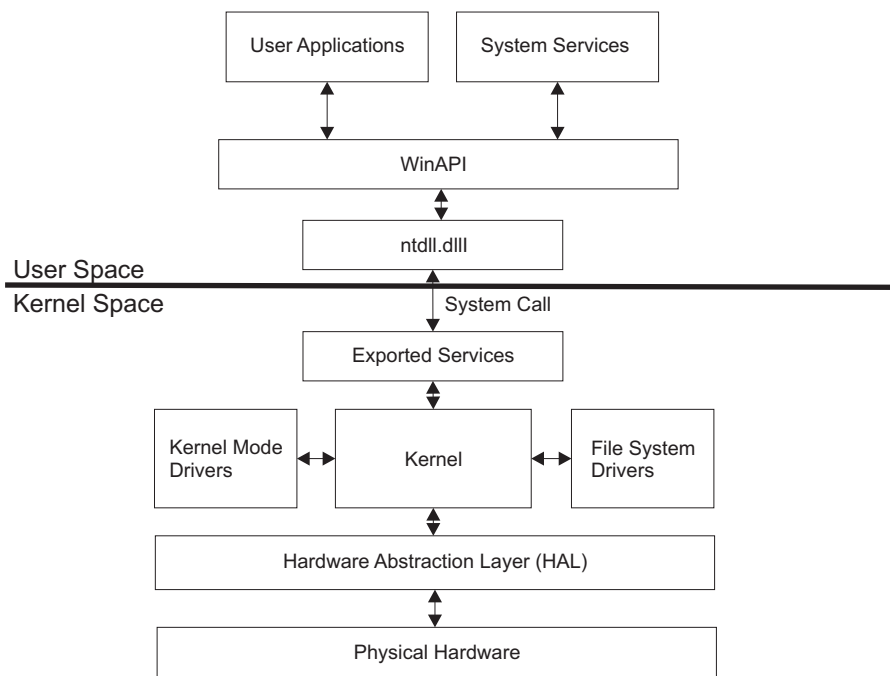


Figure 2.1: Windows Architecture

The top half of Figure 2.1 represents user space. User space components include Windows services and user applications. These processes reside above the Microsoft Windows Application Programming Interface (WinAPI). The WinAPI is a set of exported functions for use by programmers targeting the Windows platform. Besides offering building blocks for faster application development, the WinAPI gives programmers large flexibility in developing

¹See the Intel Pentium Reference for details [9]

applications. Even low-level interactions with the Windows operating system are supported. Even though new approaches are being developed to further abstract the underlying operating system, notably WinFX², the WinAPI represents the most important access to Windows specific functionality. Its maturity also means that the WinAPI is very stable, even across different versions of the Windows operating system. The WinAPI consists broadly of six components:

- Base Services - Groups low-level access to the operating system. Important functions served by the Base Services are file system access, memory, registry access, processes, threads and error handling.
- User Interface - Provides basic controls (widgets) and management for windows.
- Common Control Library - Provides applications with advanced controls (widgets), such as status- and tool bars.
- Graphics Device Interface - Gives access to the underlying graphics functionality of Windows
- Network Services - Access to various networking subsystems.
- Windows Shell - Provides the Windows operating system shell and enhances it.

We focus primarily on the Base Services category as it provides basic services for running applications. For more detailed information on the categories, see [26].

The bottom half of Figure 2.1 depicts kernel space, which is also organized into multiple components. The kernel itself consists of the low-level operating system implementations. These implementations are exported as system calls. System calls provide the basic operations used to implement higher level constructs. The Hardware Abstraction Layer (HAL) is used for abstracting the hardware to achieve the requirement for portability. Device drivers include hardware-, file system- and network drivers. These drivers are extensions to the operating system that enable specific functionality. For more detailed information on the Windows NT architecture, see [22].

²WinFX is a new .NET-based Programming Interface introduced in Windows Vista

We will continue by introducing two methods for monitoring the interaction of applications with the operating system. The first method will focus on user space and will target the WinAPI. The second method will target kernel space and focus on the exported kernel services.

2.2 Windows API Hooking

Hooking is a term in programming that usually refers to inserting functionality into predefined exported interfaces. We refer to the inserted functionality as the hook. That is, if we imagine the execution flow, we see that we inserted a detour from the exported interface to the hook function. After leaving the hook function we usually return to the point after the hook call and proceed as before. For example, an email program could provide hooks to allow for encryption routines to be inserted by third party developers. So, before sending an email, the program would transfer control to the hook function and perform the encryption.

The ability to create hooks in functions of our choosing can be useful in a variety of contexts. A major need arises when changes or additions need to be performed to applications without possessing the source code. Creating hooks would allow to inspect and change data or call external functions at will. Another example is hooking for performance profiling. Inserting timer hooks allows for making detailed measurements inside applications without access to source code. Finally, an interesting use for hooks arises in software security. An application of hooks in this area will be demonstrated in Chapter 5.

Unfortunately, the Windows API does not provide us with the comfort of predefined hooks. For this reason, we must find a way to force hooks into Windows API functions. In the following subsections, we will explain two techniques for hooking functions and explain their drawbacks in the following discussion.

2.2.1 Import Address Rewriting

Windows applications share a common file format, which is called Win32 Portable Executable File Format (PE). A PE file consists of multiple sections. These sections logically contain one aspect of the program, e.g., code or data.³

³For more details on the PE Format see [16] and [17]

One interesting part is the import section. The import section contains external references to libraries. These are wrapped in the Import Address Table (IAT). More precisely, the IAT stores function pointers to the external functions. All references to an external function point to the corresponding entry in the IAT. At program startup, all required libraries are loaded and the IAT is patched with the real address of the functions [15]. This mechanism allows for increased efficiency when binding dynamic libraries, as not every call to the library must be patched.

The approach to hooking library functions is to alter the function pointers in the IAT. By overwriting the IAT with the address of our function, we would redirect all calls to our own function. The most obvious limitation of this approach is that only imported library functions can be hooked. Another, equally important drawback is late binding. Applications can also load external libraries on demand or query additional functions of a loaded library.

2.2.2 Detours

This section will introduce a method for hooking arbitrary Win32 functions on x86 machines. The method was introduced and developed by Microsoft's research department and is available as a general purpose library, called *Detours* [8]. This library allows Win32 functions to be redirected to a user supplied detour function while preserving access to the original function through a trampoline. This functionality allows programmers to either place code between functions or completely replace functions without having access to the original source code.

Detours operates by directly overwriting native code in memory. Because functions are located in executable memory pages and executable pages are marked as read-only, detours changes the memory protection for the affected pages. It continues by first replacing the first five bytes of instructions of the target function with an unconditional jump statement to the detour function. This transfers control immediately to the detour function on entry. The replaced bytes are copied into extra allocated memory and suffixed with an unconditional jump back to the remainder of the original function. This constitutes the trampoline. However, because x86 instructions vary in length and in number of operands, detours must identify the valid sequence of instructions and determine the start of the next instruction. This means that, possibly, the instruction boundary is located more than five bytes after the

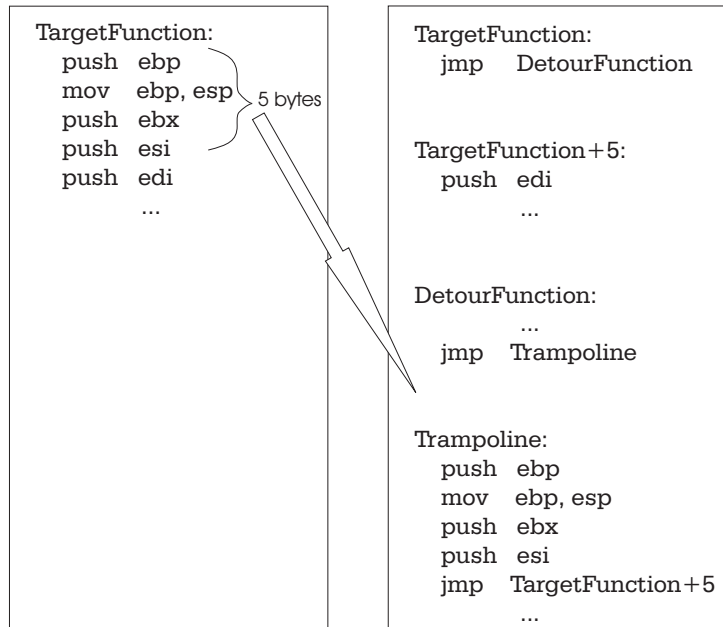


Figure 2.2: Hooking with Detours Including Trampoline

function start. Detours cannot hook functions that are less than five bytes long. Figure 2.2 shows the process detours uses for hooking.

In previous versions of Windows, hooks in the Windows API affected all programs loading the respective DLL. This is not desirable, as we would like to control for which applications hooks are set. Fortunately, Windows technology allows us to do just that. In Windows NT a new loading process was introduced called "Copy on Write". This concept allows multiple programs to map the virtual address space of a loaded library to the same physical pages, allowing these to be shared. If a process attempts to write to one of these pages marked as "Copy on Write" the pages are copied to a new physical page. Once the virtual memory tables are updated, the application can write to its private copy of the pages. This functionality allows detours to hook WinAPI functions on an application-specific level, by setting the hooks in a private copy of the library.

2.3 Windows System Call Hooking

The following dynamic analysis method concentrates on the lower half of Figure 2.1 – the kernel space. As all operating system functionality is located in the kernel, important operations must always transfer to kernel space.

We saw in the Windows architecture description that kernel space and user space are separated. However, we also saw lines connecting user space to kernel space and, clearly, we need some form of communication between the two realms. In particular, user space code must be able to instruct the kernel to perform operations that require operating system functionality. Software interrupts provide a way to request operating system services by jumping from user space to kernel space. If a user space application invokes an interrupt, control is passed to kernel space. Specifically, control is passed to the interrupt handler, which then decides what operations to perform based on the interrupt invoked.

The following section provides a closer look at the steps involved when invoking operating system functionality.

2.3.1 Anatomy of a System Call

To explain the steps involved in making a system call, we use a simple example. We will assume a programmer wants to create a file called `c:\test.txt` and is using the C programming language. The following code would achieve our goal:

```
#include <stdio.h>

void main(void)
{
    FILE * f = fopen("c:\\test.txt", "w");
    fclose(f);
}
```

After compiling, we can inspect the generated executable with a suitable Windows disassembler⁴. Let us look at the first part of the `_main` function.

```
.text:00401000    push    ebp
```

⁴See Appendix A for a list of Windows disassemblers.

```

.text:00401001    mov     ebp, esp
.text:00401003    push   ecx
.text:00401004    push   offset aCw ; "w"
.text:00401009    push   offset aCTest_txt ; "C:\\test.txt"
.text:0040100E    call   _fopen

```

After pushing the two arguments onto the stack in the last two push operations, the function invokes the `_fopen` call in the final instruction. This function calls, `__fsopen`, which in turn calls `__openfile`, which calls `__sopen`, until we finally arrive at the following section in the `__sopen` call:

```

.text:004035A3 loc_4035A3:
.text:004035A3    push   0
.text:004035A5    push   esi
.text:004035A6    push   [ebp+dwCreationDisposition]
.text:004035A9    lea   eax, [ebp+SecurityAttributes]
.text:004035AC    push   eax
.text:004035AD    push   [ebp+dwShareMode]
.text:004035B0    push   [ebp+dwDesiredAccess]
.text:004035B3    push   [ebp+lpFileName]
.text:004035B6    call   dword ptr ds:CreateFileA

```

After pushing a number of arguments, the final statement calls the function `CreateFileA` from the `.idata` section. This function, which is the ANSI string version of `CreateFile`, is imported from `kernel32.dll`. The `CreateFileA` call, performs some sanity checks on the parameters and then calls `NtCreateFile` exported by `ntdll.dll`. Let us look at the disassembly of `NtCreateFile`:

```

.text:7C91D682    mov     eax, 25h           ; NtCreateFile
.text:7C91D687    lea   edx, [esp+4]
.text:7C91D68C    int   2Eh
.text:7C91D68E    retn  2Ch

```

Surprisingly, this is quite short. First, the magic number `0x25` is loaded into the `EAX` register. This value indicates that we wish to perform an `NtCreateFile` system call. Next, a pointer to the stack is loaded into the `EDX` register. The target address is actually the current stack pointer `ESP` plus four, therefore pointing right behind the return address, exactly where

the arguments are passed to the `NtCreateFile` call. Finally, the interrupt `0x2E` is invoked to transit into kernel space. Quite straightforward.

As mentioned previously, we now enter the interrupt handler. The `0x2E` interrupt handler is exported as label `KeSystemService`. The value in the `EAX` register is now used to make a lookup in the `SYSTEM_SERVICE_DESCRIPTOR` table (SDT). Let us look at its definition, taken from [23].

```
typedef NTSTATUS (NTAPI *NTPROC) ( ) ;
typedef NTPROC *PNTPROC;
#define NTPROC_ sizeof (NTPROC)

typedef struct _SYSTEM_SERVICE_TABLE
{
    PNTPROC ServiceTable; // array of entry points
    PDWORD CounterTable; // array of usage counters
    DWORD ServiceLimit ; // number of table entries
    PBYTE ArgumentTable; // array of byte counts
}
SYSTEM_SERVICE_TABLE ,
* PSYSTEM_SERVICE_TABLE ,
* * PPSYSTEM_SERVICE_TABLE ;

typedef struct _SERVICE_DESCRIPTOR_TABLE
{
    SYSTEM_SERVICE_TABLE ntoskrnl ; // ntoskrnl.exe (native api)
    SYSTEM_SERVICE_TABLE win32k; // win32k.sys (gdi/user support)
    SYSTEM_SERVICE_TABLE Table3; // not used
    SYSTEM_SERVICE_TABLE Table4; // not used
}
SERVICE_DESCRIPTOR_TABLE ,
* PSERVICE_DESCRIPTOR_TABLE,
* * PPSERVICE_DESCRIPTOR_TABLE ;
```

The SDT consists of four `SYSTEM_SERVICE_TABLE` (SST) entries. Each entry is a structure containing an array of function pointers, called entry points. The interrupt handler looks up the function pointer in the entry point array of the first SST in the SDT. This pointer points to the system call `NtCreateFile` and is invoked to perform the call. Figure 2.3 schematically shows the control flow for the above example.

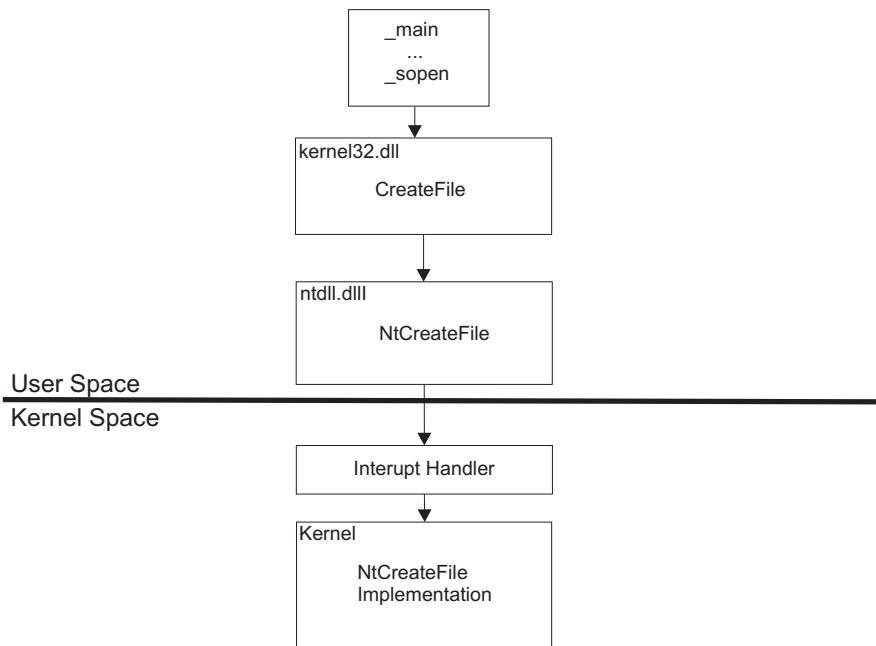


Figure 2.3: Example Schematic Control Flow

2.3.2 Hooking Windows System Calls

Hooking a Windows system call means redirecting the system call to an arbitrary function that we provide. This can be used to monitor system call invocations of processes as in [20]. One method to hook system calls is to rewrite the function pointer in the SDT. This approach was introduced in [21]. The problem with this approach, however, is that the SDT is located in kernel space. That is, we can not modify the SDT from a user space application. Because we cannot modify the Windows kernel and we lack access to source code, we require another method to enter the kernel. For this, we will use kernel drivers.

A kernel driver is located in kernel space and has full access to the Windows kernel. Because of this, we can now easily modify the pointers in the SDT to make the necessary changes. All we need to do is locate the correct array entry of the function we want to hook and change the pointer to our own provided function. The provided function must of course match the signature of the system call function. However, Microsoft does not publish documentation for the the system call interface. Some information can be

deducted from the `ntddk.h` include file, but a more complete reference can be found here [13].

This chapter armed us with the tools necessary to hook functions in user and kernel space. This allows us to place hooks into arbitrary functions, including system calls, necessary for in-depth runtime monitoring. We, therefore, successfully laid the foundation for using dynamic analysis to investigate program behavior during runtime.

Chapter 3

Static Analysis

In this chapter, we will investigate methods of static code analysis. Static code analysis is a method to extract a model of program behavior without execution. These models can either be constructed from source code or from binary code. Static analysis is a familiar concept in compiler development and is becoming an important component of security research. Static analysis offers two compelling advantages over dynamic analysis. First, static code analysis can provide provable statements on the behavior of a program. Also, static code analysis does not suffer from runtime performance penalties incurred by dynamic analysis methods. However, general static analysis problems are undecidable [10]. We will consider the limitations of static analysis in Section 3.5.

From the security point of view, our interest is focused on interactions of a program with the underlying operating system. We are, therefore, interested in statically extracting information about system call invocations. The following quote [6] serves as motivation.

”A compromised application cannot cause much harm unless it interacts with the underlying operating system [...]”

3.1 Control Flow Graph

Our first goal is to determine the set of system calls a program can possibly invoke. More specifically, we are interested in finding the set of possible system calls that can be reached from any entry point in the application. To achieve this, we need to determine all possible execution flows of our

program (An execution flow is a possible, valid path that a program can take). For this purpose, we construct a *control flow graph* (CFG). A CFG is a static program representation that separates the program into basic blocks, which are represented as nodes in the CFG. A basic block is a sequence of instructions. These instructions contain no control transfer instructions, such as jumps or branches, anywhere but at the end of the block. In addition, no targets of any control transfer instructions point anywhere into a block except to the start. To conclude, a jump target is always located at the start of a basic block, while a jump is always located at the end. An edge from block A to block B denotes that control flow can reach block B from block A via a control flow instruction. The basic blocks reachable by the jump at the end of a basic block B are called its block successors, denoted by $succ(B)$. Basic blocks that reach a basic block B with their jumps are called block predecessors, denoted by $pred(B)$.

Formally, a CFG is a directed graph

$$CFG = (B, E)$$

with B the set of basic block nodes and $E = B \times B$ the set of edges. The set of edges is defined as

$$(b1, b2) \in E \Leftrightarrow b1 \in pred(b2)$$

Let us consider the following code fragment as example

```
x=y+1;
while (x>1)
{
    y++;
    x--;
}
y=y+2;
```

The flow graph depicted in Figure 3.1 shows a CFG representation of the code fragment above.

A basic block $b1$ is reachable from basic block $b2$, if and only if there exists a path through the CFG connecting $b1$ to $b2$. All code reachable from a basic block b is represented by the set of all basic blocks for which a path exists from b .

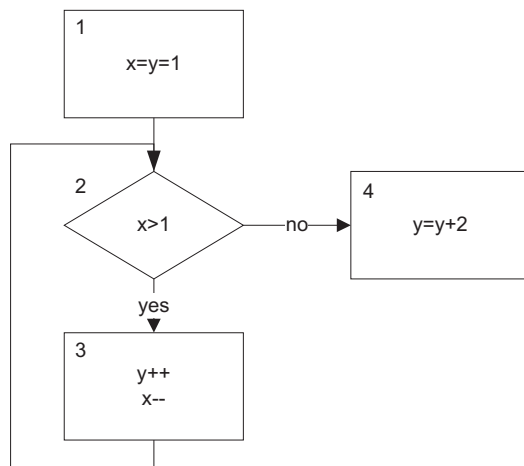


Figure 3.1: Example Control Flow Graph

Therefore, if we start from a program entry point, construct a CFG, and follow all paths from the entry basic block, we attain all code possibly reachable from that entry point. If we now further investigate the system calls invoked in every reachable basic block, we have the set of system calls possibly invoked from the application entry point.

Unfortunately, constructing CFGs is not always trivial. One problem arising in the construction of CFGs results from the use of indirect jumps. An indirect jump is a jump through a pointer variable. That is, we jump to the location indicated by the pointer. This means, we cannot statically infer the jump target without knowing the value of the pointer variable. Thus, data flow needs to be taken into consideration to statically identify the pointer variable.

The CFG can be used in various ways to extract security-relevant information. For example, Wagner [25] uses automata theory to construct models of valid system call sequences to determine abnormal behavior. We, however, are more interested in the individual system call invocation. For example, the set of invoked system calls can already help us identify potentially undesired actions. Consider a rule-set that forbids an application to open or create any files. If the set of system calls derived from the CFG contains an `NtCreateFile` call, we know the application possibly violates the rule-set. However, this limits the flexibility of defining rules. Assume a user wants to limit access only to one folder. This would require the analysis of the pa-

rameters to the `NtCreateFile` function. We therefore need further analysis; in particular, we need to evaluate the possible function parameters involved in the system calls.

If we were able to statically identify the *exact* value of each variable at every point in the program, we could identify system call parameters and allow for the more fine grained access limitation described above. Also, by knowing the value of the variable used for any indirect jumps, we can easily construct the CFG. Unfortunately, this will most likely not be the case. Instead, in the best case, we can derive a set of possible values a variable *may* take at any point. These possible values can be determined by data flow analysis, which is introduced below.

3.2 Reaching Definitions Analysis

In a first step, we would like to identify variable definitions, i.e., assignments, and their existence throughout the program. This analysis is called *reaching definitions analysis*. We introduce one approach to generating the reaching definitions. The reader is referred to [1] and [14] for more advanced methods and solutions.

Our starting point is the CFG. First, we define the following two functions:

$$\begin{aligned} gen &: Block \rightarrow \{Variable\ Definitions\} \\ kill &: Block \rightarrow \{Variable\ Destructions\} \end{aligned}$$

The function *gen* takes a block and generates the set of variables defined in that block. The function *kill*, correspondingly, takes a block and generates the set of variables destroyed in that block. A variable definition is destroyed if the variable is assigned a new value. Note that assigning a variable a new value destroys all previous assignments throughout the entire program. Table 3.1 shows the sets generated by *gen* and *kill* for the example program of Section 3.1. For example, the third row of Table 3.1 shows that variable *x* and *y* receive new values in the loop body. This kills all possible assignments to *x* and *y*, including the assignment to *y* in block 4.

One method to generate the reaching definitions is with a set of reaching equations for each basic block. Solving the sets of equations will give us information on the definitions of variables before (*in*) and after (*out*) every

| Block (l) | $gen(l)$ | $kill(l)$ |
|---------------|----------------------|--|
| 1 | $\{(x, 1), (y, 1)\}$ | $\{(x, 1), (x, 3), (y, 1), (y, 3), (y, 4)\}$ |
| 2 | \emptyset | \emptyset |
| 3 | $\{(x, 3), (y, 3)\}$ | $\{(x, 1), (y, 1), (y, 4)\}$ |
| 4 | $\{(y, 4)\}$ | $\{(y, 1), (y, 3)\}$ |

Table 3.1: Variable Definitions

basic block B is executed. The equations are given by

$$REACH_{in}(B) = \bigcup_{b \in pred(B)} REACH_{out}(b)$$

$$REACH_{out}(B) = gen(B) \cup (REACH_{in}(B) - kill(B))$$

$REACH_{in}$ represents the set of variable definitions at the start of a basic block and $REACH_{out}$ represents the set at the end of the basic block. Obviously, the value of $REACH_{in}(b)$ is the union of $REACH_{out}$ for all predecessors of b . The value of $REACH_{out}$ is given by all generations within the block plus the definitions at the start of the block minus the definitions destroyed. If there is no predecessor, we will define $REACH_{in} = \emptyset$. Table 3.2 shows the reaching definition equations for the above example

$$\begin{aligned}
REACH_{in}(1) &= \emptyset \\
REACH_{in}(2) &= REACH_{out}(1) \cup REACH_{out}(3) \\
REACH_{in}(3) &= REACH_{out}(2) \\
REACH_{in}(4) &= REACH_{out}(2) \\
\\
REACH_{out}(1) &= gen(1) \cup (REACH_{in}(1) - kill(1)) \\
REACH_{out}(2) &= gen(2) \cup (REACH_{in}(2) - kill(2)) \\
&= gen(2) \cup (REACH_{out}(1) \cup REACH_{out}(3) - kill(2)) \\
&= gen(2) \cup (REACH_{out}(1) \cup gen(3) \cup \\
&\quad (REACH_{out}(2) - kill(3)) - kill(2)) \\
REACH_{out}(3) &= gen(3) \cup (REACH_{in}(3) - kill(3)) \\
REACH_{out}(4) &= gen(4) \cup (REACH_{in}(4) - kill(4))
\end{aligned}$$

Table 3.2: Reach Definition Equations

Note that the loop from the CFG in the example results in a recursion at block 2 in the reaching equations. $REACH_{out}(2)$ on the left side, in turn,

depends on $REACH_{out}(2)$ on the right side. Nonetheless, the set of equations can be solved by finding a fixed-point, for example, with iterative methods such as chaotic iteration [3].

So far, we have avoided procedure calls in our reaching definitions analysis. Even though inter-procedural reaching definition analysis is considerably more complex, it is an extension to the here introduced system. Nielson [14] provides extensive coverage of intra- and inter-procedural reaching definitions analysis.

We want to continue by extending reaching definitions to also take variable values into account.

3.3 Constant Propagation

The first technique to take variable values into account for analyzing possible system call parameters is *constant propagation* [14]. Although this approach does not help us determine arbitrary system call parameters, it helps us identify constant parameters. Specifically, constant propagation propagates constant variable values throughout the program. In this way, system call parameters that are constant can easily be identified.

Consider a complete lattice¹(see Appendix B). The elements of the lattice are possible single values for a variable. For simplicity, we will consider only single integer values. The least element \perp represents no integer, whereas the greatest element \top represents every possible value. \top obviously does not expand our knowledge.

Constant propagation uses the CFG as starting point. Each edge of the graph is associated with a transfer function. The transfer function describes the modification that traversing the edge of the graph performs on the variables and values. Therefore, a transfer function maps a set of lattice elements to another set of lattice elements. Because transfer functions depend on previous values of variables, we iteratively traverse the graph until no changes occur. This indicates that we have found a fixed-point. If a variable is mapped to a single lattice element, we can deduce that this variable is a constant throughout the program. Unfortunately, parameter functions are seldom constant and we therefore require methods to handle these cases as well.

¹Note that lattice theory can be applied to all data flow analysis problems. See the respective references for details.

3.4 Range Analysis

We now model the possible values that variables, such as system call parameters, can take. To this end, we will relax the requirement of variables to hold only constant values and also allow sets or ranges of values. First, we will investigate the possible range of values integer variables can take. These ranges are represented as pairs of integers and can be partially ordered by inclusion. The partially ordered set of integer ranges is a complete lattice with the \top element representing the complete range of integers possible, and therefore no knowledge, and \perp no element. Each lattice element represents a particular range.

Ranges must be expanded when multiple paths join in the CFG. This occurs, for example, after an if-then clause or at the beginning of loops. Again, consider our example from Section 3.1. Because of the loop at block 2, we must merge the two paths from block 1 and block 3. This merge would expand the range of possible values for x and for y by 1. Because of the merge, we must map the lattice elements of the branches to their least upper bound element. However, special attention must be given to loops. Loops must be solved by either determining closed forms or by finding fixed-points that do not change the integer range. Generally, range analysis suffers from frequent degeneration that produces no knowledge about the actual values, because of rapid convergence to the \top element.

In [2] range analysis was extended to cover string expressions. The authors use regular grammars as nodes in the lattice and can, therefore, model string operations such as concatenation. String expression analysis would be useful for determining string parameters to system calls. Unfortunately, this analysis suffers the same precision problems as the range analysis it extends.

3.5 Limitations — Correctness vs. Completeness

As mentioned in the introduction of this chapter, static analysis problems are generally undecidable [10]. This fact and the following quote of Rice's theorem [18] bring about an important ramification for static analysis:

"[...] there exists no automatic method that decides with generality non-trivial questions on the black-box behavior of computer

programs.”

Because we use static analysis to answer non-trivial questions, we must always settle for approximations. Our goal is soundness, i.e., all solutions must correspond to correct information about the program and we will never see false negatives. Therefore, to accommodate soundness we have to always make conservative approximations. For example, this means that reaching definitions analysis will always give us a superset of the actual definitions occurring during runtime. Also, the CFG will always represent a superset of the actual paths used during run-time. This implies that, in our context, we potentially deny a benign program from being executed.

Chapter 4

.NET

The .NET framework is a new development framework from Microsoft targeting the Windows operating system. .NET framework is expected to be the unifying component for future Windows development, ranging from web applications, database programs, to Windows applications. A main goal of the framework is to allow for easier, faster and more secure development. This is achieved with a range of measures, such as removing manual memory management by providing a central memory manager, strong typing of all languages, and complete object-oriented development. Programs for the .NET framework can be developed in a variety of languages. Much effort has been invested by Microsoft to convert third party developers to its new system and it can be expected that .NET will become even more important in the future.

4.1 Architecture

The .NET framework consists of multiple components working together to construct the Common Language Infrastructure (CLI)[4]. The CLI is a specification describing the various components used in the .NET framework.

All .NET code requires a runtime environment to execute in. This environment is provided by the Common Language Runtime (CLR). The CLR manages, among other things, memory, threads and security enforcement on behalf of the applications. For this reason, all .NET code is called managed code. Code executing outside of the runtime is called unmanaged code. .NET code also requires the .NET Framework Class Library (FCL). This library

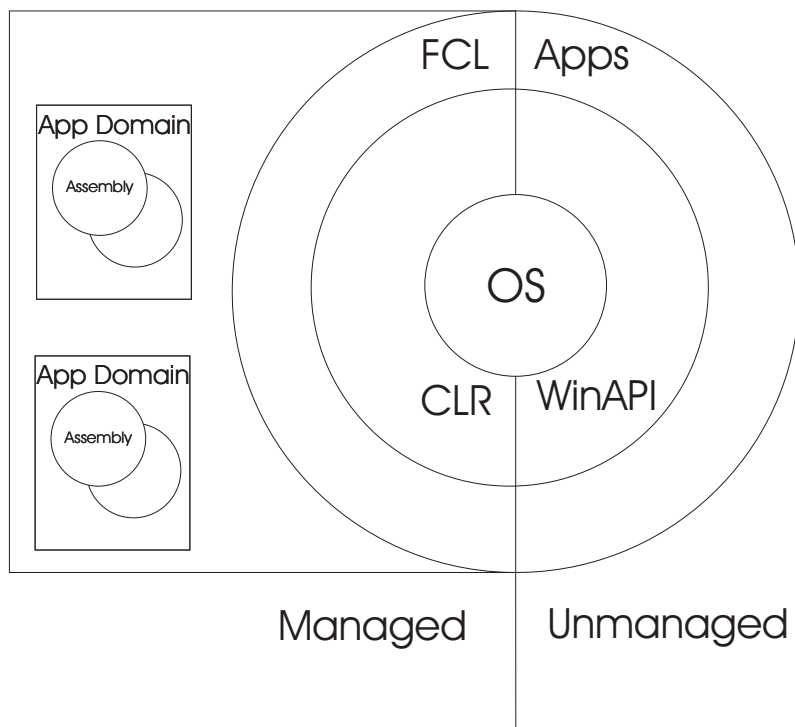


Figure 4.1: .NET Architectural Overview

contains the basic types, classes and interfaces required for running managed code. Figure 4.1 gives an overview of the .NET architecture.

All .NET-based programming languages compile their code into a common bytecode format called Microsoft Intermediate Language (MSIL). MSIL code is grouped together to a fundamental unit called an assembly. An assembly groups a set of types and resources for deployment.

Before one can run an assembly, it needs to be loaded into an application domain. An application domain is a lightweight operating system process. Historically, programs were isolated from each other by running in separate processes. This ensured that one program could not corrupt the memory of another. In .NET, this isolation can be achieved with a more lightweight concept. The CLR can run multiple application domains in one process, allowing for less overhead while providing the same level of isolation. This isolation can be guaranteed because .NET code is type safe. This means that all variable access is performed in only well-defined ways. A well-defined way is defined by the access the type of the variable permits on its structure. The C programming language is not type safe. It is possible to address and manipulate memory directly, independent of the data structure represented by the memory. In .NET, access to variables is checked to ensure that only valid operations are performed.

Running an assembly is performed in a multi-step process. The CLR is first bootstrapped by a CLR host. This host can be a web server, a browser, a shell or any other user program. A bytecode verifier then checks that the loaded code is type safe. The CLR performs just in-time (JIT) compilation of the MSIL to native code, which it then executes.

4.2 .NET Security Model

The current status quo of application security is based mostly on user credentials. Applications are generally allowed to run if the user possesses the required privileges. However, in today's networked world, lots of code is downloaded from the Internet or other non-trustworthy places. Users are generally not able to distinguish if a piece of software is benevolent or not. It is therefore desirable to grant access to code based on properties of the code itself. In this way, the same user can run different code with distinct permissions.

The .NET security model consists of multiple layers including role-based

security. For our purposes, we concentrate on the layer called *Code Access Security* (CAS)¹. CAS is a layer that determines if and how a software program is allowed to run based on evidence provided by the program itself. Evidence exists in two forms. The first is inherent evidence provided by the software. This includes the assembly content, publisher certificate, strong name², and others. The other type is runtime-based evidence, such as the location or URL the assembly is loaded from. Together these two types form an evidence set.

The process of generating a set of permitted actions based on the provided evidence is called policy resolution. Policies exist at the level of the enterprise, machine, user, and application domain and are configured by the user. Each policy level consists of three elements: permission sets, trusted assemblies, and code groups. Code groups map security permissions to applications based on evidence. Policy resolution traverses the hierarchy of policies, evaluates the code groups based on the evidence, and generates the final permission set. Figure 4.2 shows a flow chart of the policy resolution process.

The .NET class library provides a set of permission classes that represent possible security relevant actions. An example would be a file access or a network socket access. The permission classes allow a fine grained representation of specific security relevant actions that can be permitted.

A code group is a mapping of evidence to permission sets. For example, take a code group located in the machine policy called `TuCodeGroup`. The code group has a membership condition specifying that any assembly loaded from the URL `http://www.tuwien.ac.at` belongs to it. The code group further specifies that each assembly belonging to the code group receives the named permission set `ALLFILEACCESS`, granting full access to the file system. Code groups themselves can be arranged into hierarchies of code groups specifying child code groups. To expand on the previous example we could specify a child code group called `TuTrustedCodeGroup` that allows all assemblies loaded from `http://www.trusted.tuwien.ac.at` the additional privilege of accessing the windows registry.

To conclude, policy resolution is the process that assigns a given assembly a set of permissions based on the evidence the assembly provides to the framework.

¹For more details on the .NET security framework see [5]

²A strong name is a cryptographic signature authenticating the publisher.

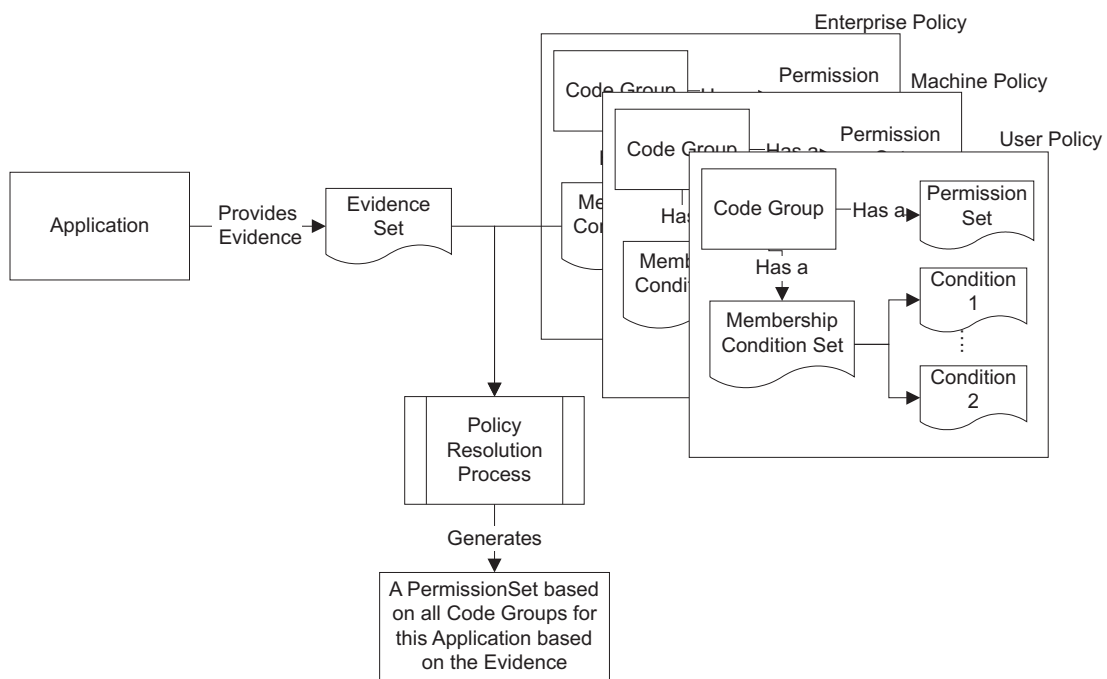


Figure 4.2: The Policy Resolution Process

One specific permission that is relevant for this work is the `SecurityPermission` residing in the `System.Security.Permissions` name space. For calling into unmanaged code a `SecurityPermission` with the `UnmanagedCode` flag set must be available. A quote from the class description in MSDN [11] states:

Since unmanaged code potentially allows other permissions to be bypassed, this is a dangerous permission that should only be granted to highly trusted code. It is used for such applications as calling native code using `PInvoke` or using COM interop.

4.3 The Platform Invoke Service

The Platform Invoke Service (`PInvoke`) allows .NET managed code to call unmanaged code. An example for this is a possible call into the native Win32 API. Because of the robustness and maturity of the Win32 API in comparison to the new .NET framework class library, programmers must often revert back to the native API. Also, much existing and tested legacy code exists in the form of Windows DLLs or COM objects. To extend this functionality into the .NET realm, `PInvoke` provides the ideal mechanism.

`PInvoke` performs the following sequence of actions to call into native code:

1. Locate the DLL containing the required function
2. Load the DLL into memory
3. Locate the address of the function in memory
4. Push the arguments (marshaling data if required) onto the stack
5. Transfer control to the located function

Observe that the unmanaged DLL is loaded directly into the address space of the managed code. This point will become important later on.

Parameter passing is an important step. The parameters must be moved from managed memory to unmanaged memory to make them available to native code. However, various obstacles make this difficult. Some objects

contain references to other objects. When passing such a parameter to unmanaged memory, the entire reference graph needs to be moved. For example, if the parameter is a linked list, the reference graph consists of all objects in the list. Otherwise the parameter object would be corrupted. Another problem arises when a particular managed type has no corresponding unmanaged type or vice-versa. For example, some .NET structures have different field orderings than their unmanaged relatives. The process of correctly moving objects between managed and unmanaged code is called marshaling.

When data is moved from managed to unmanaged code, data must be marshaled appropriately. .NET has two families of types: value types and reference types. A value type is not allocated on the heap and is not accessed via a pointer. We can imagine it as a basic type, such as an integer. A reference type is always allocated on the heap and is accessed via a pointer. A class is an example of a reference type.

In addition to the type of the parameter, the marshaling also depends on the way that this parameter is passed. Parameter can be either passed by-value or by-reference (see [19] for details on .NET types).

When a value type is passed by-value, the value type data is copied from the managed stack to the unmanaged stack. However, to make things more complicated, value types can also be passed by reference. This requires the value type to be transformed into a reference type first, by a process called boxing. Boxing allocates memory from the heap and transfers the value of the value type into the allocated memory. When we pass a value type by reference, the runtime boxes the value type. The marshaler allocates memory on the unmanaged heap and copies the contents of the boxed value type on the managed heap into the allocated memory on the unmanaged heap. Finally, a pointer to the allocated memory is pushed onto the unmanaged stack. When the call returns, the unmanaged memory is copied back into the managed heap and freed. Finally, unboxing returns the data on the heap into a value type.

Reference types are types that are always allocated on the managed heap and always referenced through their pointers. Reference types include classes, arrays, and strings. When reference types are passed by value, there are two possibilities, depending on the reference type. The first option is that the reference type is copied from the unmanaged heap to the managed heap and a pointer is pushed on the stack. This is similar to the process of boxing described previously. The other option is that the object remains in the managed heap, but the corresponding heap memory is pinned. Pinning

means locking the object to its memory location by instructing the garbage collector to refrain from moving the object around in memory. Pinning has the advantage that we do not have to copy the data, but can instead just push a pointer to the pinned memory onto the unmanaged stack.

Finally, a reference type can be passed by reference. Again, memory is allocated on the unmanaged heap and the data is copied from the managed heap to the newly allocated space. The pointer is pushed on the unmanaged stack. After the call returns, the memory is copied back from the unmanaged heap to the managed heap.

The standard marshaling routines can be modified or extended with custom marshalers. These marshalers have access to both managed and unmanaged memory and allow a programmer to implement his own marshaling implementations.

4.4 .NET Remoting

.NET remoting is a framework that allows for invocation of methods on remote objects. This functionality is useful for distributed programs. A server can store and provide remote objects. Clients can connect to the remote objects and invoke methods. This allows for easier distribution and centralization of code on the server.

An object can be made into a remote object by inheriting from the `MarshalByRefObject` class. A remoting server offers the remote object to clients over a network socket. A client can connect to the network socket and receives an object reference (a stub). This stub refers to the remote object on the server. All methods of the stub are proxies for transporting the request to the remoting server and invoking the method there. When the proxy method is invoked, all parameters are serialized and are transported to the remoting server, where they are deserialized and used for the call. This process is similar to marshaling described before.

The remoting framework provides additional services for error handling and management of remote objects.

Chapter 5

System Design

.NET Code Access Security (CAS) allows users to model permitted actions of an application on a fine-grained level. However, due to legacy code and the mature interface of the WinAPI, we expect a considerable number of .NET programs to require access to native libraries. Unfortunately, CAS does not extend to native libraries. Therefore, an attacker can use native libraries to circumvent the rules specified by the user. Our goal is, therefore, to bring native code invoked with P/Invoke from .NET under the control of the CAS rule-set. This chapter describes our contribution and details the proposed system.

Microsoft's Windows security model, though extensive, is significantly different than the .NET security model. Windows security is based on user and role-based credentials. CAS on the other hand, is based on the identity of the code, via its evidence, optionally combined with user credentials. A comparable concept of evidence does not exist in Windows security. For example, it is not possible to define Windows security based on the URL the program was downloaded from. Further, Windows security permissions also differ from .NET permissions. For example, whereas one can finely restrict network access in CAS to particular hosts, this feature is not possible with Windows security.

To extend the .NET framework security model to native code invoked with P/Invoke, a security layer has to be inserted. This layer checks requests from the native code against the rules defined by CAS. The fundamental interface used by native code to interact with the operating system is the WinAPI. We can therefore expect all security relevant operations, such as file

access, networking, memory, etc.¹, to pass through the WinAPI. In addition, .NET security permissions map well to WinAPI calls. If we evaluate the parameters of the WinAPI calls, we can create corresponding .NET permission objects. For example, we can evaluate the parameters of the `CreateFile`² WinAPI call and create a corresponding .NET permission object representing the filename and the requested action (create or open). Thus, in a first step, we place the security layer in the WinAPI.

Using the techniques explained in Section 2.2.2, we create hooks in security relevant functions of the WinAPI. The hook functions evaluate the parameters and create corresponding .NET permission objects. These permissions are then tested against the permission set granted to the application. If the requested action represented by the security permission is not permitted, a `SecurityException` is thrown. A valid request is passed on to the original WinAPI call to perform the requested operation.

Native code is very flexible and, unfortunately, allows an attacker a range of possible methods to overcome the security layer. For example, the attacker could attempt to bypass the security layer by avoiding the WinAPI altogether. Let us look at two possible methods how this could be achieved. First, the attacker could target the `ntdll.dll`, which is a wrapper for kernel level system calls. The `ntdll.dll` is located in user space just below the WinAPI. A second method for bypassing the WinAPI would be to call kernel level system calls directly. This could be achieved by pushing all necessary parameters onto the stack, writing the relevant system call number into the `EAX` register, and invoking the `0x2E` interrupt. Both methods completely bypass our envisioned security layer.

Another attack vector that needs to be mitigated is overwriting parts of the .NET framework. More precisely, because native code has complete access to the address space, an attacker could overwrite the variables holding the granted permission set to elevate his own permissions. He could also modify executable parts necessary for security enforcement. Thus, the security layer and the .NET runtime need to be protected from modification by a potential attacker.

To prevent an attacker from bypassing the WinAPI, we introduce a second layer. This layer consists of a driver loaded directly into kernel space. The driver hooks operating system calls, as described in [21], and monitors

¹For details on the Windows API see [26]

²The name of this call is slightly misleading as it is also used to open files.

the invocation of these calls from the native code. In particular, it enforces that system call invocation must first pass through our security layer in the WinAPI. To this end, the functions in the WinAPI are modified so that a subsequent system call is authenticated. When the native code attempts to bypass the checks in the security layer in the WinAPI, the second layer identifies the invocation as an unauthenticated call. The kernel driver is the only trusted component in the system. Therefore, security enforcement is ultimately handled by the driver. Authentication is performed by communication between the security layer and the driver. If the attacker circumvents the WinAPI, the invoked system call is not authenticated and is therefore blocked by the driver.

Of course, the attacker could attempt to bypass the parameter evaluation in the security layer and jump directly to the instructions that authenticate the system call. We prevent this with a two step authentication process. The check routine in the security layer immediately authenticates the system call. The parameters are then evaluated and, if any check fails, the security layer revokes the authentication. Thus, to authenticate a system call, the attacker must therefore always jump before the actual argument check routines and run through the entire process. Figure 5.1 shows the two step authentication process.

An attacker cannot modify the driver because it is located in kernel space. We can therefore use the driver also as a trusted storage for important data. To mitigate the danger exhibited by an attacker modifying the granted permission set, we safely store it in the trusted storage. When we start the application, we serialize the permission set and store it in the driver. When checking a requested action, the security layer does not check against the (possibly modified) permission set residing in .NET. Instead, the security layer first retrieves the trusted permission set from the driver and checks against this set.

Executable sections of the .NET framework, for example, class methods or Windows DLL code, are marked as execute-only in memory. To modify these sections, the attacker must first change the protection of the memory pages. To prevent this, the driver hooks the system call for modifying page protection. Thus, the native code is effectively prevented from writing to executable pages in memory.

At this point, the CAS protection is successfully extended to native code. Unfortunately, system call identification becomes very difficult because valid system calls can also originate from managed code. These system calls do

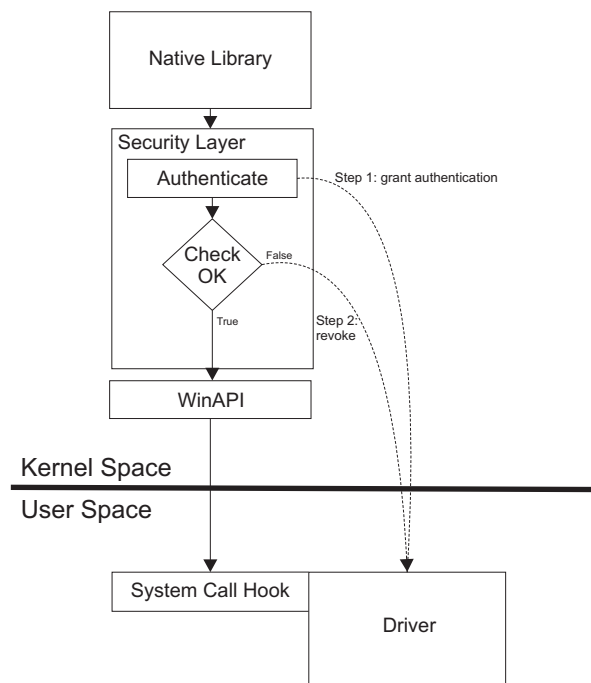


Figure 5.1: Two Step Authentication

not necessarily pass through the WinAPI, but are still validated by CAS. Blocking these calls would prevent managed classes from functioning correctly. To make tracking of these calls easier, we isolated the native code from the managed code that invokes it. To this end, we created a process boundary between the managed code and the native code. In particular, we replaced the P/Invoke calls with .NET remoting calls, similar to RPC, that carry the P/Invoke call across the process boundary to a separate process³. A remoting server resides in the second process and accepts remoting calls through a remote object interface. The remote object invokes the requested native code call and transports the results back to the managed code. The security layer and the driver monitor the remoting server process because the native code calls now originate from there. Unfortunately, the remoting server also invokes unauthenticated system calls to support the remoting operations. For this reason, we recorded invoked system calls generated by the remoting environment. The minimal set of required system calls are directly authenticated in the driver. Possible limitations of this approach are discussed in Section 7.4.

Figure 5.2 gives a schematic overview of the system described in this section.

³For a description of issues and complexities in parameter passing, see 6.3 and 6.6

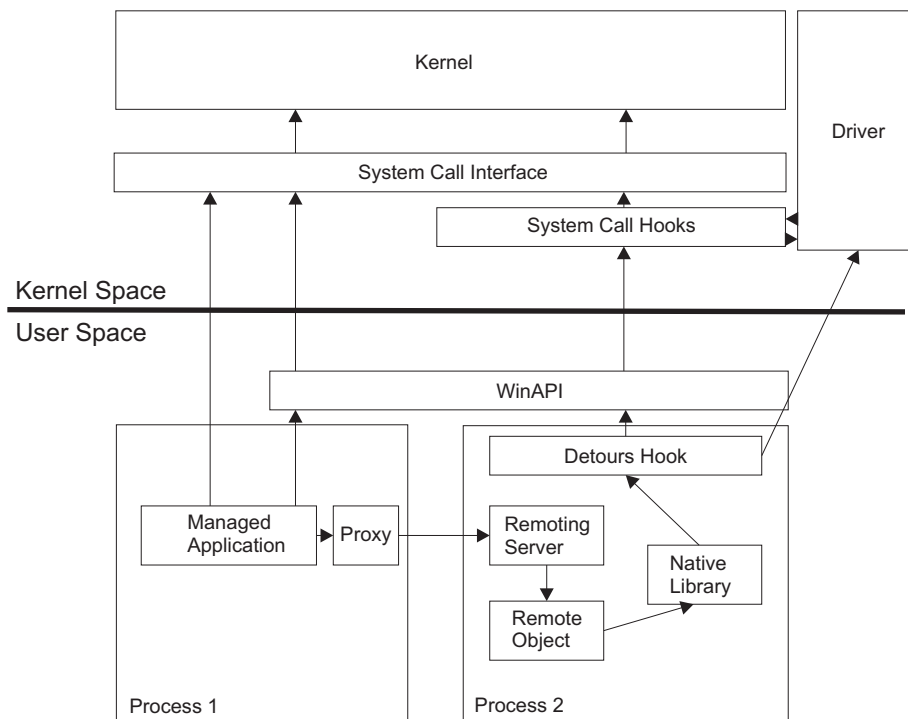


Figure 5.2: System Architecture

Chapter 6

Implementation

This chapter discusses the implementation of the design presented in Chapter 5.

6.1 Code Access Security

The current implementation allows the CAS rule-set to be configured with the .NET configuration wizard or the on-board `caspol` command-line tool. The user simply configures the desired policy and assigns it to the remoting server assembly. The remoting server then applies the rule-set to all native code executed in its process space. This allows for the use of comfortable configuration utilities, easing the rule-set configuration process for the user.

6.2 Kernel Driver

The kernel driver is an important component of the system. Besides providing a trusted storage location in kernel space, the driver hooks system calls and provides the routines to prevent unauthorized system call invocations. To achieve this, the driver provides a set of functions. These functions represent actions that can be invoked from user space. On installing the driver, the driver registers routines for handling specific I/O Request Packets (IRP). IRPs are the basic communication quantum used by the kernel to communicate with the driver. The most important, for our purposes, is the function handling `IOCTL` requests. `IOCTL` requests allow user space applications to communicate with the driver over the WinAPI. Actions are indicated with

control codes and data communication is performed via buffers passed between user and kernel space. The control codes are shared with the user space side via a common include file ¹

Once the driver is loaded and receives an `IOCTL` request, the registered function inspects the control code to identify the requested action. Possible actions are:

- Hook or unhook a specified system call - System call hooks are inserted on request. The user space application sends an appropriate control code to hook the desired system call. Unhooking is performed in a similar manner.
- Monitor a specific process - Process monitoring is done by saving the `Process ID` of the process to be monitored in the driver. The system call hooks check the calling `Process ID` against the saved value to see if the invocation originates from the monitored process.
- Authenticate the next system call number - When a request is received to authenticate a system call, the system call number is stored in the driver. Once the system call is invoked by the monitored process, the system call hook tests the system call number against the stored value. Only if the numbers match, is the system call invoked.
- Get and set the trusted buffer - Getting and setting the trusted buffer is done by copying the `IOCTL` buffer to a buffer in the driver and vice versa. The trusted buffer is stored in a private structure in the driver.
- Prevent code from elevating memory protection - Hooks the specific system call for page protection and evaluates the parameter for the desired protection. If the call attempts to modify the protection of execute-only code pages, the call is returned with an error.

Communication with the driver from user space is initiated by invoking a `CreateFile` call with the driver name. Unless this permission is specifically granted to the native library, the attacker cannot communicate with the driver. Attacker communication could otherwise easily unhook system calls or change the monitored application and is therefore prohibited.

¹see Appendix C.4 for include file listing

The kernel driver was developed with the Microsoft Driver Development Kit (DDK), which provides the compiler and include files for driver development. For installing the driver, we wrote a small utility program that uses the WinAPI to load and unload the provided driver.

Interesting parts of the kernel source code are listed in Appendix C.1.

6.3 .NET Remoting

.NET remoting stubs are necessary as proxies between the managed and the native code. These proxies were automatically generated from the managed assembly. We used a combination of .NET reflection and an analysis of disassembled MSIL code to obtain the necessary information, such as the number of parameters and their types. This information was used to generate code in C++ .NET Managed Extensions along with a `Makefile` for compilation. After compilation, two libraries are generated.

The first library acts as an interceptor that is a replacement for the original library in the original process. This library contains method stubs for each function of the native code that the managed code invokes. These stubs use .NET remoting to invoke a method in the second library.

The second library, called remote object, exposes a remote method for each function that the managed code uses in native libraries. Each remote method corresponds to one method stub in the interceptor library. The remote method performs the actual invocation of the native library in the remote process.

Because of our remoting infrastructure, special care must be given to parameters passed by-reference from managed code to the interceptor stub. Variables cannot be passed across the process boundary with remoting by-reference. This is because pointer values have no meaning outside the process address space. Therefore, remoting parameters are always passed by-value. However, `P/Invoke` allows for by-ref parameter passing and we must take this into account. The interceptor stub passes the parameter variables by-value to the remote object. Once the call into the native library is completed by the remote object, the interceptor stub requests the parameter variables back. The reference parameters need to be copied back into the original process, because changes in the remote process must be reflected in the original object. That is, we simulate the by-ref parameter passing by copying the variables back and forth by-value. Figure 6.1 shows the process of simulating by-ref

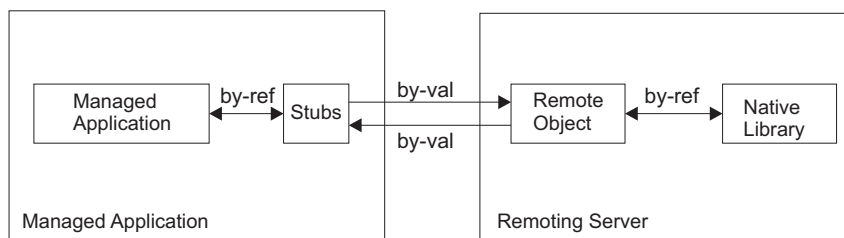


Figure 6.1: Remote Parameter Passing

parameter passing.

A sample of two remoting stubs can be found in Appendix C.3.

6.4 Remoting Server

The remoting server hosts both the remote object and the native libraries that should be confined. Before the native code can be executed, the remoting server has to perform initialization tasks (shown in Figure 6.2), such as setting up driver communication, setting the WinAPI hooks, and saving the permission set in the trusted storage. To this end, the remoting server establishes communication with the driver and registers his own process ID for monitoring with the driver. The .NET security manager is used to generate the granted permission set given the assembly evidence. This permission set is then serialized to an XML format and sent to the trusted storage. After registering the remote object, the security layer WinAPI hooks are set. From that point onward, the remoting server process cannot perform unauthorized system calls.

The security layer jumps into action as soon as a remote method of the remote object library is invoked on the remoting server. Figure 6.3 details the steps. After the remote object method is invoked, the remote object invokes the native library function requested. If the native code invokes a WinAPI function that is hooked, control passes to the security layer hook function. This function immediately grants the system call. Next, the function queries the driver for the trusted permission set, which is regenerated from the XML representation. The trusted permission set is intersected with the desired permission for the WinAPI call. If the intersected set is empty, the system call grant is revoked and a `SecurityException` is thrown. Otherwise, the call passes transparently to the actual WinAPI function and continues to the

system call.

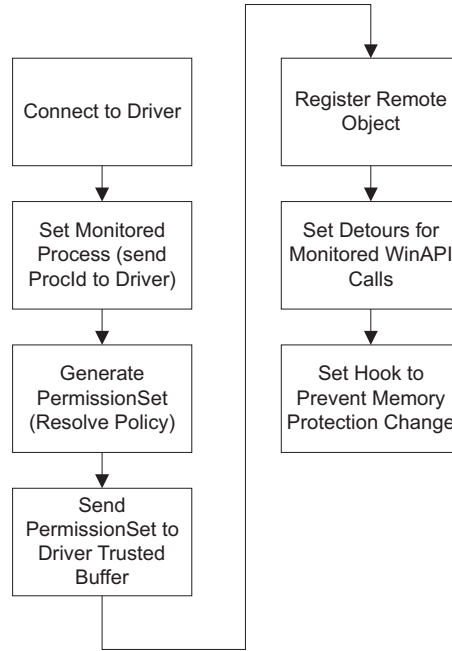


Figure 6.2: Remoting Server Startup Operations

6.5 Page Protection

An attacker capable of modifying important .NET runtime functionality or Windows DLL code can easily circumvent the security layer. To prevent an attacker from modifying these functions, executable code is protected using memory page protection. This protection is achieved by hooking the system call responsible for page protection as described above (See Appendix C.3 for sample source code of hooking system calls).

Pages containing executable code are generally marked as only `PAGE_EXECUTE`. This prevents reading or writing to any memory location in the page. To modify the functions, an attacker would have to change the page protection to allow for write access. To prevent this, we deny write modifications to any `PAGE_EXECUTE` pages. More precisely, we query the desired page protection before modification and do not allow elevation to write access for any page that has the execute flag set.

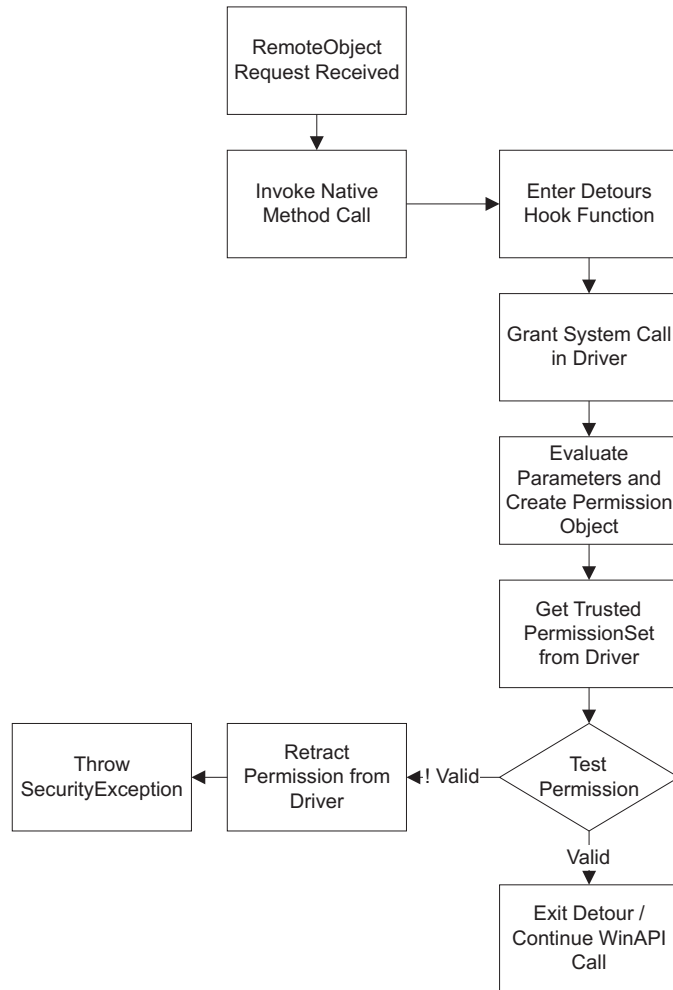


Figure 6.3: Security Layer Operations

This approach prevents an attacker from modifying executable code, but still allows for dynamic library loading. When a library is loaded dynamically, for example with the `LoadLibrary` call, memory is allocated with `PAGE_READWRITE` protection [15]. Only after the library is loaded, the protection is changed to `PAGE_EXECUTE`. Our page protection mechanism, therefore, allows dynamic library loading.

6.6 Patching the .NET Application

Patching means replacing or adding sections to an original program. The need for patching the original .NET assembly results from various factors explained below.

Self defined classes or structures that are passed to native code must be marked as `Serializable` for cross-process communication. For this reason, self defined classes that are not serializable must be marked as such in the MSIL code.

Another need for patching are system DLLs. The normal search order for loading DLLs is to first look in the current directory, then in user defined paths, and finally in the system directories. However, system DLLs, such as `kernel32.dll`, are immediately loaded from the system directory. This means, we must redirect calls to `kernel32.dll` to the generated proxy library by means of patching.

Patching can be performed broadly in three ways. First, bytecode can be manipulated directly on the hard drive. As this approach is somewhat complicated due to the requirement of parsing byte code, other approaches are to be preferred. A comfortable alternative is to use the Microsoft .NET assembler and disassembler included in the framework. The disassembler can generate readable IL Code from a given assembly. This code can be manipulated and transformed back into an assembly by the assembler. The final and most simple approach is to use a bytecode manipulation library such as MBEL [24]. However, only modifications provided by the library can be performed with this approach. Both the assembler/disassembler and the MBEL library were used in this project.

Chapter 7

Evaluation

To evaluate the proposed approach, we developed a prototype, proof-of-concept implementation of the proposed system (see Chapter 5). This prototype extends CAS to the following areas

- Creating and opening files
- Opening registry keys
- Reading environment variables

In particular, we created WinAPI, and their corresponding system call hooks for the following functions

| WinAPI Function | System Call Function |
|------------------------|-----------------------------|
| CreateFile | NtCreateFile |
| RegOpenKey | NtOpenKey |
| GetEnvironmentVariable | |

Table 7.1: Secured WinAPI and System Call Functions

The remaining chapter reports on the evaluation results. We investigated whether the current prototype achieves our stated goal of extending .NET CAS to native libraries. Further, we report on results of our simulations of the attack methods discussed in Chapter 5. We continue by shedding light onto the performance penalty incurred by the design and conclude with a discussion of the limitations of the design and the implementation.

7.1 Functionality

Functionality testing is directly linked to our stated goal. We would like to ensure that native code cannot perform actions that are restricted by the Code Access Security rule-set. For this purpose, we first constructed a CAS rule-set that permitted unrestricted access. We tested our prototype on the three WinAPI calls from above. As expected, all calls were permitted. The following driver log trace shows a successful `CreateFile` WinAPI call for creating `test.txt`

```
IOCTL_NEXT_SYSCALL
next valid syscall is 37
IOCTL_GET_PERMISSIONBUFFER
NtCreateFile \??\C:\test.txt
This system call is authenticated
```

This output corresponds to actions taken in the security layer. First, the IOCTL message grants access to the `NtCreateFile` system call (system call number 37). Then, the security layer has to query the driver for the permission set buffer, which is stored in trusted kernel space. It checks for permission to access `test.txt`, which is allowed by the unrestricted permission set obtained from the kernel, and therefore grants the system call.

Next, we modified the CAS rule-set to deny every call. The following sample is a trace of the driver log for a `CreateFile` call for `test.txt` with a rule-set denying this access.

```
IOCTL_NEXT_SYSCALL
next valid syscall is 37
IOCTL_GET_PERMISSIONBUFFER
IOCTL_NEXT_SYSCALL
next valid syscall is 0
NtCreateFile \??\C:\test.txt
This system call is not authenticated
```

Because the permission set does not include the requested action, the security layer revokes the previous grant by setting the next valid system call to 0.

The functionality tests demonstrate that our prototype successfully extends the CAS rule-set to all monitored functions.

7.2 Security

After evaluating the basic functionality, we tested the security solution of our prototype in a variety of ways. This section provides an overview of the results obtained.

We tested two approaches for circumventing our security layer. Our first approach attempts to dynamically load the `NtCreateFile` from `ntdll.dll` with `LoadLibrary` and `GetProcAddress`. The `ntdll.dll` is located just above kernel space and below the WinAPI, thereby bypassing our security layer. We invoked the call and, as expected, our security layer denied the call as it was not authenticated from the WinAPI security layer.

In the second approach, we decided to avoid using libraries altogether. We used inline assembly code to invoke system calls directly. For this, we adopted the code provided by `ntdll.dll` functions. We thereby completely avoided any user space DLLs and, of course, our security layer. Again, our kernel driver prevented the system call invocation.

Our next attack attempts to circumvent the security layer checks. An attacker has multiple routes to achieve this, all ultimately resulting in a jump beyond the check routines in the security layer or directly to the original WinAPI function. However, this attack is futile as skipping the security layer also skips system call authentication required to execute system calls. The following trace demonstrates this.

```
NtCreateFile \??\C:\test.txt  
This system call is not authenticated
```

The log shows that the security layer was never invoked. Thus, the required `NtCreateFile` system call was not authenticated and is, therefore, not executed

Next, we simulated an attacker's attempt to subvert the runtime or the security layer. An attacker could, for example, inject a jump statement in the check routines to prevent the security layer from revoking a previously authenticated call, thereby completely defeating the security layer. To modify an executable function, the attacker must change the memory page protection. We simulated this attack by attempting to modify an executable function. First, we attempted to change the page protection to allow for write access. The following trace shows the driver's actions.

```
NtProtectVirtualMemory: 72cf684
```

```
Protection: 4
Old Protection: 260
Blocking this page protection modification
```

Protection value 4 indicates that the desired protection is `PAGE_READWRITE` and 260 indicates that the currently held protection is `PAGE_EXECUTE`. This page protection modification is not allowed and is thus not granted.

The results obtained from our first attacks indicate the soundness of our described method. We successfully showed that all system call invocations must pass through the security layer, and the checks therein, first.

We did, however, identify one possible way to defeat the security system. This attack requires unrestricted registry access. The Windows registry provides a key for specifying DLLs that are loaded on every application start. If an attacker can load a DLL into every program, he is able to use the library's `DLLMain` routine to execute arbitrary code before program execution [7]. All user space applications, including of course the remoting server, would come under the control of the attacker. Therefore, care must be taken in securing the relevant registry keys.

7.3 Performance

Performance analysis will give us an idea of the overhead incurred by the security layer and, in particular, the remoting infrastructure. To this end, we conducted a series of performance tests.

We expect the .NET remoting infrastructure to incur the largest performance penalty. To measure this penalty, we isolated the remoting infrastructure from the remaining system. For this, we modified our remoting server to not instantiate the security layer and to not interact with the driver. Our test library function takes no parameters and returns no variables. The test function solely invokes the `CreateFile` function from `kernel32.dll` to create a file. The remoting server is hosted on the same machine, preventing network effects to skew results. Table 7.2 compares the average running time over 10 calls of a direct `P/Invoke` call to a call redirected over .NET remoting. As we suspected, the .NET remoting mechanism creates a considerable performance penalty. This penalty arises from the remoting mechanism, because parameters must be converted, serialized and transported across the process boundary.

| Direct Call (P/Invoke) (ms) | Remoting Call (ms) |
|------------------------------------|---------------------------|
| 15 | 234 |

Table 7.2: P/Invoke vs. Remoting (no Security)

In our next test, we use the remoting server as described in Chapter 5. The security layer is in place and interacts with the driver. Our test function is the same as above, i.e., it takes no parameters and returns no value. The following table, again, shows the average running time over 10 calls.

| Direct Call (P/Invoke) (ms) | Secured Call (Remoting) (ms) |
|------------------------------------|-------------------------------------|
| 15 | 234 |

Table 7.3: P/Invoke vs. Remoting (Security layer in place)

Results indicate that our security layer introduces no measurable, i.e., less than 1 ms, performance penalty.

Finally, we want to investigate how parameter passing affects performance. Our next test compares the overhead produced by parameters in the .NET remoting call. The `CreateFile` call has 7 parameters, including one string, and one return parameter. For this test, we included the 7 parameters including return value in the remoting call.

| Direct Call (P/Invoke) (ms) | Secured Call (Remoting) (ms) |
|------------------------------------|-------------------------------------|
| 15 | 286 |

Table 7.4: P/Invoke vs. Remoting (Direct `CreateFile` Call)

Table 7.4 indicates that the parameter passing exacerbates the performance penalty incurred. As more information needs to be serialized and transported, the result is not surprising.

7.4 Limitations

This section will discuss some limitations that underly the current design and implementation. We will briefly discuss the severity and, if possible, explain how to avoid the limitation.

First, we will discuss a shortcoming of the current implementation. Windows registry access is often performed on relative, not absolute, paths. In particular, the `RegOpenKeyEx` function uses previously obtained handles to query for sub-keys. Five standard keys are used to obtain the initial handles. The current implementation can only test registry key access on sub-keys of these five standard keys. Future versions will extend the system to handle relative registry key queries. For this, the security layer must track previous registry queries and create a mapping of handles to queries. This would allow each relative query to be mapped to its absolute path.

The following limitation is a design limitation as the .NET remoting infrastructure created problems for the kernel driver. The remoting infrastructure invoked system calls that were not authenticated, and which were subsequently denied. As described in Chapter 5, we permitted these calls directly in the driver. As an example, the remoting server invoked some `NtCreateFile` system calls to load the library containing the remoting stubs and to load .NET class libraries. Additionally, some `NtOpenKey` requests (which is a system call for registry access) were issued to find the .NET framework directory. We resolved the problem by checking the arguments to the system calls and permitting those that were absolutely required for remoting. This means, however, that an attacker could also issue these system calls independently of the security settings. Due to the rare number of necessary calls and the limited use to an attacker, we believe this does not pose a significant threat to the overall system.

The need for patching .NET assemblies creates a problem with strongly named assemblies. A strong name is a cryptographic signature for the assembly to prevent tampering. Because the assembly is modified, the signature becomes invalid. However, the assumption that code is issued by an attacker means the signature is of no value to the user anyway.

Chapter 8

Future Work

This chapter outlines future work that we currently consider most interesting. The main goals of future enhancements are the increase in scope of our system and further extensions of possible restrictions of operations in native code.

Primarily, we want to extend our system to the complete WinAPI. Our current implementation has focused mainly on WinAPI functions that bear close proximity to kernel system calls. This means that mapping WinAPI functions to system calls was relatively easy and could be done manually. Furthermore, these WinAPI functions generally invoke only one system call. However, functions providing higher-level constructs can invoke multiple system calls. An example is the `LoadLibrary` WinAPI function, which calls a number of system calls, such as `NtCreateFile`, `NtAllocateVirtualMemory`, `NtProtectVirtualMemory`, etc. Therefore, we would like to automatically generate a mapping of WinAPI functions to their respective system calls. To this end, the control flow graph of each WinAPI function could be statically analyzed to determine the set of system calls ultimately reachable. This would extend the proposed system to higher construct WinAPI functions invoking multiple system calls.

Static data flow analysis methods as outlined in Chapter 3 could also be applied to complement our dynamic analysis. Static analysis, though complex, could guarantee that certain native code parts cannot perform any malicious action. In this case, no runtime monitoring would be necessary. This would reduce the need for our remoting infrastructure and thus, the incurred performance penalty.

Finally, we would like to further increase the restrictions on native code possible. Native code allows for actions not possible in managed code. These

actions, such as memory management, can currently not be controlled with on-board CAS permissions. Fortunately, the .NET framework allows for the definition of novel permissions to cover additional resources. We plan on creating custom CAS permissions to cover further actions that we want to restrict in native code. For example, we envision configurable permissions that control memory allocation. These, and other, customizable permissions can then be configured by the user to further restrict native code if desirable.

Chapter 9

Conclusions

.NET code access security (CAS) succeeds in restricting undesired actions of managed code. The permission to invoke unmanaged code, however, gives a potential attacker complete freedom to circumvent all restrictions.

We introduced a system to extend the CAS rule-set to native code. Evaluation of our proposed system shows that our design is viable. In particular, we successfully extended the CAS rule-set to selected WinAPI functions. By forcing a possible attacker to use the WinAPI, we subjected native code to our security layer. Further, we successfully protected our system against various possible attack vectors, such as circumvention of the security layer and memory corruption. Also, we feel that the performance overhead incurred by our system is acceptable for all, but a few, application domains.

Our system is based on dynamic analysis that monitors WinAPI function and kernel system call invocations. Even though static analysis methods have advantages, regarding for example provability, we believe that our dynamic analysis approach is better suited to solving the problem at hand. In particular, the complexity of building a sufficiently robust static analysis approach far outweighs the benefits. Nonetheless, static analysis will become relevant as outlined in future work [Chapter 8].

Previous work has focused on securing native code. Our primary contribution is to combine these approaches and extend them to .NET CAS. Further, while CAS itself has been sufficiently discussed in the security community, native code invocation has been neglected. We believe that .NET will gain further momentum in the future, thus our contribution will increase in relevance.

Bibliography

- [1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.
- [2] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. *Proceedings of 10th International Symposium on Static Analysis*, 2003.
- [3] Patrick Cousot and Radhia Cousot. Automatic synthesis of optimal invariant assertions: Mathematical foundations. In *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*, pages 1–12, 1977.
- [4] ECMA. Ecma 335 - common language infrastructure partitions i to vi, 3rd edition, 2005.
- [5] Adam Freeman and Allen Jones. *Programming .NET Security*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2003.
- [6] I. Goldberg, D. Wagner, R.Thomas, and E. A. Brewer. A secure environment for untrusted helper applications: Confining the wily hacker. *Sixth USENIX Security Symposium Proceedings*, pages 1–12, July 1996.
- [7] Peter Gutmann. An open-source cryptographic coprocessor. *9th USENIX Security Symposium*, 2000.
- [8] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 functions. *Proceedings of the 3rd USENIX Windows NT Symposium*, July 1999.
- [9] Intel pentium 4 - technical documents. <http://developer.intel.com/design/Pentium4/documentation.htm>.

- [10] William Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, 1992.
- [11] .NET framework class library documentation - securitypermissionflag enumeration. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfssystemsecuritypermissionssecuritypermissionflagclasstopic.asp>.
- [12] Steven S. Muchnik. *Advanced Compiler Design and Implementation*. Academic Press, San Diego, CA, USA, 1997.
- [13] Gary Nebbett. *Windows NT/2000 Native API Reference*. New Riders Publishing, Thousand Oaks, CA, USA, 2000.
- [14] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.
- [15] Russ Osterlund. Windows 2000 loader, what goes on inside windows 2000: Solving the mysteries of the loader. *MSDN Magazin*, March 2002.
- [16] Microsoft portable executable and common object file format specification. <http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.msp>.
- [17] Matt Pietrek. An in-depth look into the Win32 portable executable file format. *MSDN Magazin*, February 2002.
- [18] Rice's theorem. http://en.wikipedia.org/wiki/Rices_theorem, November 2005.
- [19] Jeffrey Richter. .NET type fundamentals. *MSDN Magazine*, December 2000.
- [20] Mark Russinovich and Bryce Cogswell. Regmon - registry monitoring utility. <http://www.sysinternals.com/Utilities/Regmon.html>.
- [21] Mark Russinovich and Bryce Cogswell. Windows NT system-call hooking. *Dr. Dobb's Journal*, January 1997.

- [22] Mark Russinovich and David A. Solomon. *Microsoft Windows Internals Fourth Edition*. Microsoft Press, 2005.
- [23] Sven B. Schreiber. *Undocumented Windows 2000 secrets: a programmer's cookbook*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
- [24] Michael Stepp. Mbel: The microsoft bytecode engineering library. <http://www.cs.arizona.edu/mbel>.
- [25] David A. Wagner. *Static analysis and computer security: new techniques for software assurance*. PhD thesis, University of California at Berkeley, 2000. Chair-Eric A. Brewer.
- [26] Platform SDK: Windows API. http://msdn.microsoft.com/library/en-us/winprog/winprog/windows_api_start_page.asp.

Appendix A

Popular Windows Disassemblers

- PEBrowser by Smidgeonsoft
Url: <http://www.smidgeonsoft.prohosting.com/software.html>
- IDA Pro by Data Rescue
Url: <http://datarescue.com/idabase/index.htm>
- Microsoft .NET Framework IL Disassembler
Url: <http://msdn.microsoft.com/netframework/>

Appendix B

Lattices

A set P with a binary relation $\sqsubseteq: P \times P \rightarrow \{true, false\}$ that satisfies

- reflexivity - $\forall a \in P : a \sqsubseteq a$
- transitivity - $\forall a, b, c \in P : a \sqsubseteq b \wedge b \sqsubseteq c \Rightarrow a \sqsubseteq c$
- antisymmetry - $\forall a, b \in P : a \sqsubseteq b \wedge b \sqsubseteq a \Rightarrow a = b$

is called a partially ordered set, denoted by (P, \sqsubseteq) . $l \in Y \subseteq P$ is an upper bound if $\forall l' \in Y : l' \sqsubseteq l$. Similarly, l is a lower bound if $\forall l' \in Y : l' \sqsupseteq l$. The least upper bound, or supremum or join, is the smallest upper bound. The greatest lower bound, or infimum or meet, is the largest lower bound.

A lattice is a partially ordered set where all *nonempty finite* subsets have a least upper bound and a greatest lower bound. A complete lattice is a partially ordered set where all subsets have a least upper bound and a greatest lower bound. \perp is the least element and \top the greatest element of the lattice.

Lattices, and partially ordered sets in general, can be nicely shown with Hasse Diagrams. Figure B.1 shows a Hasse Diagram for the partially ordered set $(\mathcal{P}(\{1, 2, 3\}), \subseteq)$.

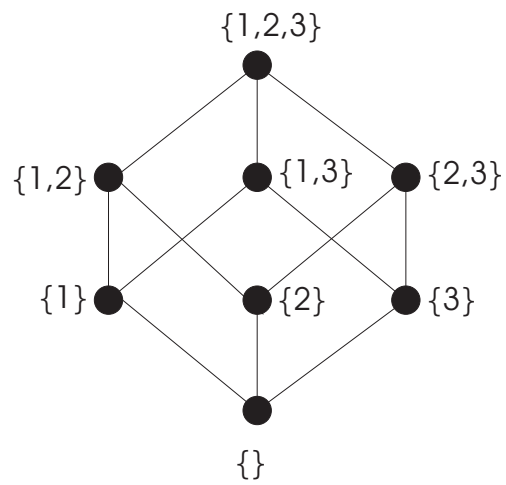


Figure B.1: Hasse Diagram

Appendix C

Code Listings

C.1 Kernel Driver

Listing C.1: Incoming IRP Dispatcher. This function is responsible for identifying the requested action and invoking the respective function.

```
NTSTATUS IRPControl( IN PDEVICEOBJECT DeviceObject, IN PIRP Irp )
{
    NTSTATUS status = STATUS_SUCCESS;
    size_t dataSize = 0;
    PIO_STACK_LOCATION ioStack = IoGetCurrentIrpStackLocation(Irp);

    if (ioStack->Parameters.DeviceIoControl.IoControlCode ==
        IOCTL_MONITOR_PROC)
    {
        DbgPrint("IOCTL_MONITOR_PROC\n");
        DbgPrint(" procId_in_buffer_is %s\n", Irp->AssociatedIrp.
            SystemBuffer);

        RtlCharToInteger((PCSZ) Irp->AssociatedIrp.SystemBuffer, 10, &(
            extension->procId));

        DbgPrint(" Extension_says %d\n", extension->procId);
    }
    else if (ioStack->Parameters.DeviceIoControl.IoControlCode ==
        IOCTL_NEXT_SYSCALL)
    {
        // next syscall
        DbgPrint("IOCTL_NEXT_SYSCALL\n");
        RtlCharToInteger((PCSZ) Irp->AssociatedIrp.SystemBuffer, 10, &(
            extension->nextValidSysCall));
        DbgPrint("next_valid_syscall_is %d\n", extension->
            nextValidSysCall);
    }
    ...
    Irp->IoStatus.Status = status;
}
```

```

    Irp->IoStatus.Information = dataSize;

    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return status;
}

```

Listing C.2: Hooking `NtCreateFile`. This function is invoked by the IRP dispatcher to hook the `NtCreateFile` system call. The SDT entry is replaced with the provided function `ZwCreateFileHook`. The original function is saved to later restore the SDT.

```

void hookNtCreateFile()
{
    if (extension->CreateFileHooked == 1)    // is it hooked already
        return;

    SERVICE_DESCRIPTOR_TABLE * std = ((SERVICE_DESCRIPTOR_TABLE *)
        SDTAddr);

    savedCreateFile = (NTCREATEFILE) std->ntoskrnl.ServiceTable[
        SYSCALLNTCREATEFILE];
    std->ntoskrnl.ServiceTable[SYSCALLNTCREATEFILE] = (NIPROC) &
        ZwCreateFileHook;

    extension->CreateFileHooked = 1;
}

```

Listing C.3: `NtProtectVirtualMemory` Hook Function. This function is the actual SDT replacement function. Prevents the monitored application from elevating page protection rights.

```

NTSTATUS ZwProtectVirtualMemoryHook(
    HANDLE ProcessHandle,
    PVOID *BaseAddress,
    PULONG NumberOfBytesToProtect,
    ULONG NewAccessProtection,
    PULONG OldAccessProtection )
{
    if ((ULONG) PsGetCurrentProcessId() == extension->procId)
        DbgPrint("NtProtectVirtualMemory: 0x%x\nProtection:\n");

    // call original system call
    NTSTATUS ret = (NTSTATUS) savedProtectVirtualMemory(ProcessHandle,
        BaseAddress,
        NumberOfBytesToProtect,
        NewAccessProtection,
        OldAccessProtection);

    if ((ULONG) PsGetCurrentProcessId() == extension->procId)
    {
        if (*OldAccessProtection == PAGE_EXECUTE)
        {

```

```

        // not allowed => revert
        DbgPrint("Blocking this page protection modification");

        savedProtectVirtualMemory(ProcessHandle,
            BaseAddress,
            NumberOfBytesToProtect,
            *OldAccessProtection,
            &NewAccessProtection);
        return STATUS_NOT_COMMITTED;
    }
}
return ret;
}

```

Listing C.4: Sample Control Code for Kernel Communication

```

/** set the trusted permission set buffer */
#define IOCTL_SET_PERMISSIONBUFFER
        CTL_CODE(FILE_DEVICE_UNKNOWN, 0x100, METHOD_BUFFERED, FILE_ANY_ACCESS)

```

C.2 Remoting Server

Listing C.5: Remoting Server Main Routine

```

// local
#include "DriverCommunication.hpp"
#include "DetourFunctionsFileOperations.hpp"

// .NET stuff
using <mscorlib.dll>

using namespace System;
using namespace System::IO;
using namespace System::Security;
using namespace System::Security::Permissions;
using namespace System::Reflection;
using namespace System::Runtime::Remoting;
using namespace System::Runtime::Remoting::Channels;
using namespace System::Runtime::Remoting::Channels::Tcp;
using namespace System::Runtime::InteropServices;
using namespace System::Runtime::Serialization::Formatters::Binary;

#pragma managed
int main(int argc, char ** argv)
{
    connectToDriver();
    setMonitoredApplication();

    // gather permissionSet
    PermissionSet * permSet = SecurityManager::ResolvePolicy(Assembly::
        GetCallingAssembly()->Evidence);
}

```

```

// send the permissionSet to the driver; requires funky marshalling
// from byte[] to --gc string * to char *
setTrustedPermissionSet((char*)(void*)Marshal::StringToHGlobalAnsi(
    permSet->ToString()));

// start remote object
ChannelServices::RegisterChannel(new TcpChannel(8082));
RemotingConfiguration::RegisterWellKnownServiceType(
    typeof(RemoteObject),
    "test1",
    WellKnownObjectMode::Singleton);

// set the detours
setDetourOpenKey();
setDetourCreateFile();
setDetourEnvironmentVars();

// load necessary lib
LoadLibrary("UnmanagedDLL.dll");

// no more page protect from now on
hookNtProtectMem();

Console::ReadLine();

closeDriver();
return 0;
}

```

Listing C.6: Hooking `CreateFile` with Detours. This routine uses Detours to hook the WinAPI call.

```

// function pointer for trampoline
#pragma managed
HANDLE (* trampolineCreateFile)(LPCTSTR fname,
    DWORD dwDesiredAccess,
    DWORD dwShareMode,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    DWORD dwCreationDisposition,
    DWORD dwFlagsAndAttributes,
    HANDLE hTemplateFile) = NULL;

#pragma unmanaged
void setDetourCreateFile()
{
    void * func = DetourFindFunction("kernel32.dll", "CreateFileA");
    if (func == NULL)
        cout << "Could not find function in binary" << endl;

    trampolineCreateFile = (HANDLE (*)(LPCTSTR fname,
        DWORD dwDesiredAccess,
        DWORD dwShareMode,
        LPSECURITY_ATTRIBUTES lpSecurityAttributes,
        DWORD dwCreationDisposition,
        DWORD dwFlagsAndAttributes,

```

```

        HANDLE hTemplateFile)) DetourFunction((PBYTE) func, (PBYTE)
            detouredCreateFile);
    }

```

Listing C.7: Sample Security Layer Check Routine for `CreateFile`. This function is the hook function for the above detours usage.

```

#pragma managed
HANDLE __stdcall detouredCreateFile(
    LPCTSTR fname,
    DWORD dwDesiredAccess,
    DWORD dwShareMode,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    DWORD dwCreationDisposition,
    DWORD dwFlagsAndAttributes,
    HANDLE hTemplateFile)
{
    // first grant permission
    nextValidSysCall(SYSCALLNTCREATEFILE);

    // setup trusted permission set
    XmlDocument * xml = new XmlDocument();
    PermissionSet * trustedSet = new PermissionSet(PermissionState::None);
    trustedSet->FromXml(createSecElement(xml->LoadXml(
        getTrustedPermissionSet()->DocumentElement));

    // create permission object and test against trusted set
    FileIOPermission * ioPermission;
    if ((dwDesiredAccess & GENERIC_READ) != 0)
        ioPermission = new FileIOPermission(FileIOPermissionAccess::Read,
            fname);
    if ((dwDesiredAccess & GENERIC_WRITE) != 0)
        ioPermission = new FileIOPermission(FileIOPermissionAccess::Write,
            fname);

    // create demand permission set
    PermissionSet * demandSet = new PermissionSet(PermissionState::None);
    demandSet->AddPermission(ioPermission);

    // test
    if (demandSet->Intersect(trustedSet) == NULL)
    {
        nextValidSysCall(0);
        throw new SecurityException("File_permission_not_granted");
    }

    // call the dynamic trampoline
    return (HANDLE) trampolineCreateFile(fname,
        dwDesiredAccess,
        dwShareMode,
        lpSecurityAttributes,
        dwCreationDisposition,
        dwFlagsAndAttributes,
        hTemplateFile);
}

```

C.3 Remoting Proxies Sample Source

Listing C.8: Sample of a generated interceptor proxy for CreateFile

```
#pragma managed
extern "C" _declspec(dllexport) System::Int32 CreateFile(
    System::String* lpFileName,
    System::UInt32 dwDesiredAccess,
    System::UInt32 dwShareMode,
    System::IntPtr lpSecurityAttributes,
    System::UInt32 dwCreationDisposition,
    System::IO::FileAttributes dwFlagsAndAttributes,
    System::IntPtr hTemplateFile)
{
    System::Int32 returnValue = Remote::self()->CreateFile(
        lpFileName,
        dwDesiredAccess,
        dwShareMode,
        lpSecurityAttributes,
        dwCreationDisposition,
        dwFlagsAndAttributes,
        hTemplateFile);
    return returnValue;
}
```

Listing C.9: Sample of a generated remote object proxy for CreateFile

```
[DllImport("kernel32.dll", EntryPoint="CreateFile")]
extern "C" System::Int32 CreateFile_unmanaged(
    System::String* lpFileName,
    System::UInt32 dwDesiredAccess,
    System::UInt32 dwShareMode,
    System::IntPtr lpSecurityAttributes,
    System::UInt32 dwCreationDisposition,
    System::IO::FileAttributes dwFlagsAndAttributes,
    System::IntPtr hTemplateFile);

#pragma managed
public __gc class RemoteObject : public MarshalByRefObject
{
    public:
        System::Int32 CreateFile(
            System::String* lpFileName,
            System::UInt32 dwDesiredAccess,
            System::UInt32 dwShareMode,
            System::IntPtr lpSecurityAttributes,
            System::UInt32 dwCreationDisposition,
            System::IO::FileAttributes dwFlagsAndAttributes,
            System::IntPtr hTemplateFile)
        {
            Console::WriteLine("Entering _CreateFile");
            System::Int32 returnValue = CreateFile_unmanaged(
                lpFileName,
                dwDesiredAccess,
```

```

        dwShareMode,
        lpSecurityAttributes,
        dwCreationDisposition,
        dwFlagsAndAttributes,
        hTemplateFile);
    }
    return returnValue;
}

```

Listing C.10: Sample of a generated interceptor proxy using simulated by-reference passing

```

#pragma managed
extern "C" _declspec(dllexport) System::Int32 testFunction(
    System::Text::StringBuilder** a)
{
    System::Int32 returnValue = Remote::self()->testFunction(*a);
    *a = Remote::self()->get_System_Text_StringBuilder_a();
    return returnValue;
}

```