Extending .NET Security to Unmanaged Code

Patrick Klinkoff[1], Christopher Kruegel[1], Engin Kirda[1], and Giovanni Vigna[2]

[1] Secure Systems Lab
Technical University Vienna
[pk,chris,ek]@seclab.tuwien.ac.at
[2] Department of Computer Science
University of California, Santa Barbara
vigna@cs.ucsb.edu

Abstract. The number of applications that are downloaded from the Internet and executed on-the-fly is increasing every day. Unfortunately, not all of these applications are benign, and, often, users are unsuspecting and unaware of the intentions of a program. To facilitate and secure this growing class of mobile code, Microsoft introduced the .NET framework, a new development and runtime environment where machine-independent byte-code is executed by a virtual machine. An important feature of this framework is that it allows access to native libraries to support legacy code or to directly invoke the Windows API. Such native code is called *unmanaged* (as opposed to *managed* code). Unfortunately, the execution of unmanaged native code is not restricted by the .NET security model, and, thus, provides the attacker with a mechanism to completely circumvent the framework's security mechanisms.
The approach described in this paper uses a sandboxing mechanism to prevent an attacker from executing malicious, unmanaged code that is not permitted by the security policy. Our sandbox is implemented as two security layers, one on top of the Windows API and one in the kernel. Also, managed and unmanaged parts of an application are automatically separated and executed in two different processes. This ensures that potentially unsafe code can neither issue system calls not permitted by the .NET security policy nor tamper with the memory of the .NET runtime. Our proof-of-concept implementation is transparent to applications and secures unmanaged code with a generally acceptable performance penalty. To the best of our knowledge, the presented architecture and implementation is the first solution to secure unmanaged code in .NET.

1 Introduction

With the growth of the Internet, applications are increasingly downloaded from remote sources, such as Web sites, and executed on-the-fly. Often, little or no knowledge exists about the author or her intentions. Therefore, users are susceptible to executing potentially malicious programs on their computers. Malicious programs contain code that executes in any unauthorized or undesirable way.

To secure users and increase the proliferation of mobile code, Microsoft recently introduced a new development and runtime framework called .NET [5]. This framework leverages the previous experiences gathered with the Java virtual machine concepts and includes a fine-grained security model that allows one to control the level of access associated with software built upon .NET. These applications are referred to as composed of *managed* code. The model significantly limits the damage that can be caused by malicious code. To address the important problem of backward compatibility and legacy code support, .NET also offers a mechanism to tie in native libraries. These libraries, however, execute outside of the .NET security model, and therefore are called *unmanaged code*. As a consequence, the usage of this feature in .NET applications may allow an attacker to completely circumvent the framework's security mechanisms, leading to the unrestricted execution of arbitrary code. This security problem is important because the use of unmanaged code will probably be common in future Windows .NET applications. Millions of lines of legacy native Windows code exist that will need to be integrated and supported over the next decade. Also, software engineering research [10] has shown that it is not realistic to expect existing applications to be entirely rewritten from scratch in order to take advantage of the features of a new language.

This paper describes our approach to extend the current .NET security model to native (unmanaged) code invoked from .NET. To this end, we use a sandboxing mechanism that is based on the analysis of Windows API and system call invocations to enforce the .NET security policy. Our approach ensures that all unmanaged code abides by the security permissions granted by the framework. Our primary contributions are as follows:

– Extension of existing sandboxing methods to .NET unmanaged code invocations.
– Two-step authorization of system calls by placing the security layer in the Windows API and the enforcement mechanisms in a loadable kernel driver.
– Separation of untrusted native library and trusted managed code into two separate processes by way of .NET remoting.

The paper is structured as follows. The next section provides an overview of the .NET framework and its security-relevant components. Section 3 introduces the design of our proposed system. Section 4 discusses the evaluation of the security and performance of the system and shows that our approach is viable. Section 5 presents related work. Finally, Section 6 outlines future work and concludes the paper.

2 Overview of the .NET Framework

Microsoft's .NET framework is an implementation of the Common Language Infrastructure (CLI) [6], which is the open, public specification of a runtime environment and its executable code. A part of the CLI specification describes the Common Type System (CTS), which defines how types are declared and used in the runtime. An important property of the .NET framework is that it is type-safe. Type safety ensures that memory accesses are performed only in well-defined ways, and no operation will be applied to a variable of the wrong type. That is, any declared variable will always reference an object of either that type or a subtype of that type. In particular, type safety prevents a non-pointer from being dereferenced to access memory. Without type safety, a program could construct an integer value that corresponds to a target address, and then use it as a pointer to reference an arbitrary location in memory. In addition to type safety, .NET also provides memory safety, which ensures that a program cannot access memory outside of properly allocated objects. Languages such as C are neither type-safe nor memory-safe. Thus, arbitrary memory access and type casts are possible, potentially leading to security vulnerabilities such as buffer overflows.

The runtime environment can enforce a variety of security restrictions on the execution of a program by relying on type and memory safety. This makes it possible to run multiple .NET programs with different sets of permissions in the same process (on the same virtual machine). To specify security restrictions, the CLI defines a security model that is denoted as Code Access Security (CAS) [9]. CAS uses *evidence* provided by the program and security policies configured on the machine to generate permissions set associated with the application. Security relevant operations (for example, file access) create corresponding permission objects, which are tested with respect to the granted permission set. If the permission is not found in the granted set, the action is not permitted and a security exception is thrown. Otherwise, the operation continues.

Managed code executes under the control of the runtime, and, therefore, has access to its services (such as memory management, JIT compilation, or type and memory safety). In addition, the runtime can also execute *unmanaged code*, which has been compiled to run on a specific hardware platform and cannot directly utilize the runtime. In general, developers will prefer managed code to benefit from the services offered by the runtime. However, there are cases in which unmanaged code is needed. For example, the invocation of unmanaged code is necessary when there are external functions that are not written in .NET. Arguably, the most important library of unmanaged functions is the Windows API, which contains thousands of routines that provide access to most aspects of the Windows operating system.

To support interoperability with existing code written in languages such as C or C++ (e.g., the Windows API), the CLI uses a mechanism called *platform invoke service* (P/Invoke). This service allows for invocation of code residing in native libraries. Because code in native libraries can modify the security state of the user's environment, the .NET permission to call native code is equal to full trust [18]. Furthermore, native code launched by P/Invoke is run within the same process as the .NET CIL, and, as a consequence, malicious native code could modify the state of the .NET runtime itself. Microsoft suggests to only allow P/Invoke to be used to execute highly-trusted code. Unfortunately, users generally cannot determine the trust level of an application and will likely grant access also to non-trustworthy applications.

3 System Design

Our goal is to bring unmanaged native code invoked with P/Invoke from .NET *under the control* of the CAS rule-set. That is, we aim to combine the flexibility of unmanaged code with the security constraints enforced by managed code. When unmanaged code is executed, we assume that the attacker has complete control over the process' memory space and the instructions that are executed.

As a first approach, one could attempt to use the on-board operating system security model to enforce the desired .NET restrictions at the process level. That is, the downloaded application together with its native components is launched in a dedicated process. Then, operating system access control mechanisms are employed to restrict the privileges of this process such that the .NET Code Access Security settings are mirrored. Unfortunately, this is not easily possible. One problem is that Microsoft's Windows security model, though extensive, is different from the CAS model. That is, Windows security permissions differ from .NET permissions and do not provide a similar level of granularity. For example, in the CAS model, it is possible to allow a program to append to a file while simultaneously deny write access to the file's existing parts. In Microsoft Windows, on the other hand, the file system permissions have to be set to permit write access for a process to be able to append to a file. As another example, one can finely restrict network access to specific hosts using CAS, while this is not possible using OS-level Windows security mechanisms. Furthermore, Windows access control is based on user and role-based credentials. CAS, on the other hand, is based on the identity of the code, via its evidence. A comparable concept of evidence does not exist in the Windows security model. For example, it is not possible to define Windows security based on the URL the program was downloaded from.

Because the Microsoft Windows security mode is significantly different than CAS, we propose a dedicated security layer to extend the .NET code access security to unmanaged code. The goal of this security layer is to monitor the actions performed by the unmanaged code and enforce those restrictions specified by the CAS permission set. In the following sections, we discuss details about the design and implementation of our security layer.

3.1 Security Layer

The first design decision is concerned with the placement of the security layer. Ideally, this layer should be transparent to the application that it encapsulates. Also, it requires full access to all security-relevant functions invoked by the application (with parameters), so that sufficient information is available to make policy decisions. Finally, it must be impossible for malicious code to bypass the security layer.

The fundamental interface used by applications to interact with the environment, and the operating system in particular, is the Windows API. The Windows API is the name given by Microsoft to the core set of application programming interfaces available in the Microsoft Windows operating systems. It is the term used for a large set of user mode libraries, designed for usage by C/C++ programs, that provide the most direct way to interact with a Windows system. We can, therefore, expect all security-relevant operations, such as file access, networking, memory, etc.[3], to pass through the Windows API.

In a first step, we decided to place the security layer between the Windows API and the native library. More precisely, we intercept calls to security-relevant Windows API functions and evaluate their function parameters. Fortunately, .NET security permissions map well to Windows API calls. Thus, we can evaluate the parameters of Windows API calls by creating and checking corresponding .NET permission objects. For example, we can evaluate the parameters of the `CreateFile`[4] API call and create a corresponding .NET permission object representing the filename and the requested action (*create* or *open*). Then, this permission object can be checked against the granted CAS permissions, appropriately permitting or denying the request.

To intercept Windows API functions, we make use of Detours [14]. Detours is a general purpose library provided by Microsoft for instrumenting `x86` functions. This is achieved by overwriting the first few instructions of a target function with an unconditional jump to a self-provided function. Using this technique, we create hooks in security-relevant functions of the Windows API. 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 security exception is thrown. A valid request is passed on to the original Windows API call to perform the requested operation. By placing the security layer on top of the Windows API, it is possible to make the mechanism transparent to applications, and, in addition, it allows for comprehensive access to security-critical functions and their arguments.

Unfortunately, an attacker who has access to native code has great flexibility and can use a range of possible techniques to evade our naive security layer. The main reason is that the Windows API is user-level code that can be easily bypassed by interacting with the operating system directly. This could be achieved, for example, by invoking functions from `ntdll.dll`, which is the user space wrapper for kernel-level system calls, or by calling the system calls directly with assembly code. Another attack vector that needs to be mitigated is that parts of the .NET framework can be modified. Unmanaged code has complete and unrestricted access to the virtual address space that it is executed in. Unrestricted memory access can be leveraged by an attacker to overwrite management objects of the .NET runtime. For example, the variables holding the granted permission set could be modified. The attacker could also modify executable parts necessary for security enforcement, or simply tamper with objects on the managed heap, thereby crashing other .NET threads running on the same virtual machine. To protect from these kinds of attacks, the security

[3] For details on the Windows API, refer to [25].

[4] The name of this call is slightly misleading, as it is also used to open files.

layer has to shield the .NET runtime and concurrently executing processes from tampering with their allocated memory.

In the following Section 3.2, we introduce our approach to prevent unmanaged code from bypassing the Windows API when calling security-relevant functions. Then, in Section 3.3, we discuss our techniques to protect memory objects of the runtime from modifications.

3.2 Securing the Security Layer

In this section, we discuss our mechanism to prevent an attacker from bypassing the Windows API. To this end, we require a mechanism that allows us to enforce that certain user-mode library functions are called *before* corresponding operating system calls are performed. This mechanism is a second security layer that resides in kernel space. In a fashion similar to the previously mentioned layer at the API level, this second layer intercepts and analyzes operating system invocations. In particular, it enforces that each system call invocation must first pass through our security layer in the Windows API. To this end, the functions in the Windows API are modified such that subsequent system calls must be *authorized*. That is, whenever a security relevant Windows API function is invoked, this function authorizes the corresponding operating system calls that it is supposed to make. To make sure that only the security layer can authorize system calls (and not the native code controlled by the attacker), we have to ensure (i) that the authorization call originates from the security layer and (ii) that the security layer was not modified by the attacker. The mechanisms to enforce these two conditions are explained in more detailed later.

When unsafe code attempts to bypass the checks in the first security layer and performs a system call directly, the kernel space layer identifies this invocation as unauthorized and can abort the operation. The kernel driver is the only trusted component in the system, as it cannot be modified directly by a user process. Thus, if the attacker circumvents the Windows API, the invoked system call is not authorized 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 grant the system call. We prevent this with a two-step authorization process. The check routine in the security layer immediately grants authorization for the system call. The parameters are then evaluated and, if any check fails, the security layer revokes its authorization. Thus, to authorize a system call, the attacker must always jump *before* the actual argument check routines and run through the entire process. Figure 1 shows the two-step authorization process.

The second security layer is implemented as a device driver loaded directly into kernel space. Russinovich [22] describes a method for hooking operating system calls in Windows NT by replacing the function pointer in the system call table. The driver employs this method to hook operating system calls and monitors the invocation of these calls from the unmanaged code. The security layer that resides at the Windows API level communicates with the kernel driver via `IOCTL` messages. These messages allow user space applications to communicate

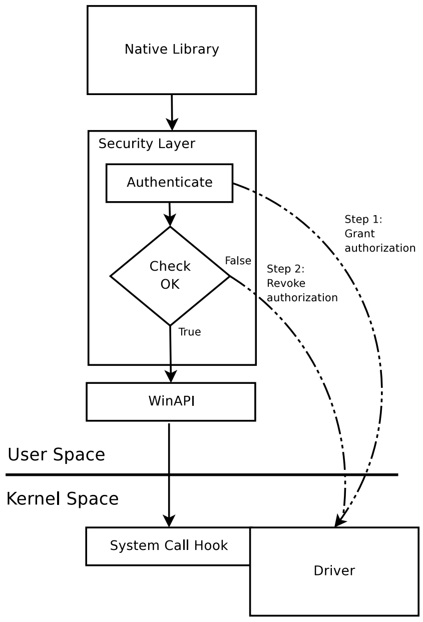

Fig. 1. Two-Step Authorization

with kernel-level drivers by passing buffers between them. In particular, IOCTL messages are used to perform authorization and revocation of system calls.

As discussed previously, the system must not allow the native code to communicate with the kernel driver directly (via IOCTL messages). Otherwise, the attacker could authorize (and later invoke) a certain system call without going through the security layer. Thus, only the security layer can be allowed to grant and revoke system calls. The problem is that both the security layer (at the Windows API level) and the native code are executed in the same address space, and it is not immediately obvious how a call from the security layer can be distinguished from one of the native code. To solve this problem, we permit IOCTL calls only from Windows API library code segments (where the security layer is implemented), and not from the native code itself (or from other segments such as the heap or stack). To this end, the system call handler for the IOCTL call first determines the address of the instruction that invoked the system call. If this address is not in the code segment of a library, it is not forwarded to the kernel driver. When the attacker attempts to jump directly to the instruction in the library that authorizes a call, the two-step authorization process ensures that arguments are checked properly. Otherwise, the authorization would be revoked immediately.

In addition, the correct operation of the two-step authorization process relies on the fact that the native code cannot alter the code executed by Windows API functions. Otherwise, it would be easy for an attacker to rewrite the code parts that check arguments or simply remove the statements that are responsible for revoking authorization when a CAS policy violation is detected. Fortunately, ensuring that code sections are not modified is relatively straightforward. The reason is that executable code sections are stored in memory pages that are marked execute-only. Thus, to modify these sections, the attacker must first change the protection of the corresponding pages. To prevent this, the driver hooks the system call that handles page protection modifications. Pages containing executable code are typically 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. This approach prevents an attacker from modifying executable code, but still allows for dynamic library loading. When a library is loaded dynamically, for example through the `LoadLibrary` call, memory is first allocated with `PAGE_READWRITE` protection [21]. After the library is loaded, the protection is changed to `PAGE_EXECUTE`. Because of this, the unmanaged code is effectively prevented from writing to executable pages in memory.

The security of the whole system relies on the fact that a user process cannot modify objects that reside in kernel space, and thus, cannot tamper with our second security layer. The astute reader might wonder why the security layer was placed in the Windows API in the first place, given the security advantages from placing it in kernel space. One important reason is the absence of a published documentation of the native API[5], which is subject to changes without notice even between different service packs of Windows. In contrast, the Windows API is well-documented and explicitly designed to shield application code from subtle changes of the native API. In addition, Windows API calls exist that map to multiple system calls. In such cases, the Windows API function parameters indicate the actual purpose of the invocation and checks are easier to perform at the Windows API level than based on arguments of individual system calls.

As mentioned previously, unmanaged code cannot tamper with the driver because it is located in kernel space. We can, therefore, use the driver as a trusted storage for important data. In particular, to mitigate the danger of an attacker modifying the CAS permission set, we safely store it in the trusted storage. To this end, we serialize the permission set and store it in the driver before we launch any native code. This is, again, achieved with `IOCTL` messages. Note that permission sets are stored on a per-process basis. That is, multiple processes with different permission sets can be sandboxed at the same time. 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 then checks against this set. Of course, the permission set stored in the driver cannot be modified

[5] Even though [19] does an excellent job at documenting the native API, the documentation can never be complete without support from Microsoft.

directly by the unmanaged code through another IOCTL, because that invocation would be trapped and checked with respect to the established permission set.

In the previous discussion, the two steps of granting and revoking authorization were explained in the context of a process. However, when considering multi-threaded applications, this two-step authorization process would contain a race condition. This race condition can be exploited when one thread attempts a particular forbidden call, while another thread attempts to sneak in the same call between the time it is originally authorized and the time it is revoked. This problem is solved by granting and revoking authorization for system calls on a per-thread basis. That is, whenever the kernel driver is consulted to grant or revoke permissions for a system call, it checks the thread identified of the currently running thread instead of its process ID.

3.3 Remoting

Using security layers and the two-step authorization process, the CAS protection is successfully extended to unmanaged code. That is, the CAS model is enforced by monitoring all relevant interaction with the operating system and the permission set is safely stored in the trusted kernel driver. Unfortunately, the objects in the managed heap and data structures of the runtime can still be altered by an attacker, possibly causing the virtual machine or other .NET threads to crash or behave unexpectedly. Another problem is that there are system calls invoked by the runtime (or certain managed classes) that do not necessarily pass through the Windows API. Although these system calls are not authorized by our security layer, they are still valid. Of course, these calls must be permitted as blocking them would prevent managed classes from functioning correctly.

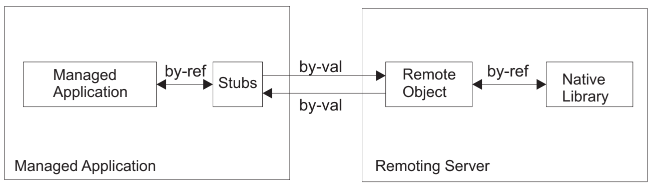

Fig. 2. Remote Parameter Passing

To protect the managed heap of .NET threads (and the runtime) and to make tracking of system calls easier, we isolate the unsafe code from the managed code that invokes it. More specifically, we create a process boundary between the managed code and the unmanaged code. Existing sandboxing techniques consider the entire process untrusted. For our purposes, however, we must distinguish between managed and unsafe code, even though these run in the same process. We therefore isolate the untrusted native library from the trusted managed code by running them in two different processes. In this way, we leverage the basic memory protection mechanisms offered by the operating system and prevent unmanaged code from accessing memory allocated by managed code.

When the native, unmanaged parts of an application are executed in a process different from the one where the managed part of the application resides, the question naturally arises how communication between these processes is realized. In particular, we need to explain how parameters and return values can be exchanged between the process that runs managed code and the process with the native code piece. While simple data types such as integers can be easily passed (copied) between address spaces, the situation is more difficult when complex data structures such as linked lists are involved. In these cases, the data structures have to be serialized by the sender and appropriately rebuilt by the receiver.

To accomplish the data exchange between the managed and the native processes, we make use of .NET remoting, the Remote Procedure Call (RPC) mechanism of .NET. To use .NET remoting, two proxy libraries have to be generated. The first proxy library contains the stubs for the native calls and is linked with the managed part of the application. More precisely, this library acts as an interceptor that replaces the original native library. It contains one method stub for each function of the unmanaged code that the managed code can invoke. Each method stub uses .NET remoting to invoke its corresponding method in the second proxy library. The second proxy library, called remote object, exposes a remote method for each function that the managed code uses in native libraries. These remote methods then perform the actual invocation of the native library in the remote process. Conceptually, the .NET remoting process can be viewed as an additional level of indirection between the managed code and the native libraries. Instead of passing values directly to the native code via the P/Invoke function, these values are first copied to the remote process using .NET remoting and only there passed to the native library. Note that both proxy libraries are automatically generated from the managed assembly. To this end, we use a combination of .NET reflection and an analysis of the disassembly of the intermediate language code. The goal is to obtain the required information to generate the proxy libraries, namely, the number of parameters of each native function and their respective types.

One problem that has not been discussed so far are parameters that are passed *by-reference* from managed code to the native library. The problem is that variables cannot be transfered across the process boundary with remoting when they are by-reference. This is because pointer values have no meaning outside the process address space. As a result, remoting parameters are always passed by-value. However, P/Invoke allows for by-reference parameters and we must take this into account. To solve this problem, we have to simulate by-reference parameter passing by copying the variables back and forth by-value. More precisely, the proxy library on the managed side transforms a by-reference argument into the corresponding value that is then copied to the remote process. Once the call into the native library is completed by the remote object, the stub method requests the parameter variables back. Then, the reference parameters are copied back into the original locations, as changes in the remote process must be reflected in the original object. Figure 2 shows the process of simulating by-reference parameter passing.

The remoting server (see Figure 2) hosts both the remote object (which contains managed code) and the native library that should be confined. Before the unmanaged code is executed, the remoting server has to perform a number of initialization tasks. First, the Windows API hooks are installed to perform API function monitoring. Then, the .NET security manager is used to generate the granted permission set based on the evidence provided by the managed application. This permission set is then serialized to an XML format and sent to the trusted storage. Finally, the kernel driver has to be initialized. To this end, the remoting server registers its own process ID for subsequent monitoring. From that point onward, the remoting server process is subject to the CAS policy enforcement and can no longer perform any unauthorized system calls. Of course, the native library can freely tamper with the process memory and possibly crash the virtual machine or return arbitrary results to the managed code. However, such actions only affect this single process, while the managed code and the runtime (together with other threads) is successfully shielded by the process barrier. In particular, note that values returned by the unmanaged code are automatically integrated into the .NET type system when received by the proxy on the managed side. If values are returned that do not correspond to valid types, the situation is detected and an appropriate unmarshaling exception thrown.

4 Evaluation

To evaluate the proposed approach, we developed a proof-of-concept implementation of our system. Our prototype implements both the security layer at the kernel level and the layer at the Windows API level. Also, we support running the native process in a dedicated process with the automatic generation of the .NET proxy libraries. The system extends CAS to the following areas: file access, registry handling, and interaction with environment variables.

We investigated whether the current prototype achieves our stated goal of extending .NET's CAS mechanism to native libraries. We report on results of our simulations of the attack methods discussed previously. We continue by shedding light onto the performance penalty incurred by the design and conclude with experiments that demonstrate that our system can successfully isolate the native libraries of real-world applications.

4.1 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 (CAS) policy. For this purpose, we first constructed a CAS rule set that denies access to a certain file. The check of the file name to enforce this policy is performed in the `CreateFile` Windows API function, which in turn has to authorize the invocation of the corresponding operating system call. Then, we attempted to bypass our checks and illegitimately obtain access to this file.

In a first test, we attempted to bypass the Windows API function and called `NtCreateFile` from `ntdll.dll` directly. As expected, our kernel-level security layer denied the call as it was not authorized by the Windows API security layer. In the second approach, we decided to avoid using libraries altogether and used in-line assembly code to invoke system calls directly. Again, our kernel driver prevented the system call invocation. Next, we simulated an attacker's attempt to subvert the runtime or the security layer. We simulated this attack by attempting to modify an executable function. As expected, the driver hook for page protection denied this modification.

The results obtained from our attacks indicate that our system works as expected, and we successfully showed that all system call invocations must first pass through the security layer, and the checks therein.

4.2 Performance

After testing the system's functionality, we ran performance analysis to determine the overhead incurred by the security layer and, in particular, the remoting infrastructure. To this end, we conducted a series of micro benchmarks to measure the performance overhead of individual calls to native library functions. All experiments were run on a machine with an Intel Pentium 4 1.8GHz and 1GB of RAM, running Windows XP with Service Pack 2.

We anticipated 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 first 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 delays from skewing the results. The first entry (i.e., Test 1) in Table 1 compares the average running time over ten calls of a direct P/Invoke call to a call redirected over .NET remoting. As we expected, the .NET remoting mechanism creates a considerable performance penalty, which arises from the need to perform interprocess communication. In our next test, we used the remoting server as outlined in Section 3.3. That is, the security layer was in place and interacted with the driver. Our test function was the same as above, i.e., it took no parameters and returned no value. The second entry (i.e., Test 2) in Table 1 shows the average running time over ten calls. The results indicate that our security layer introduces no measurable performance penalty (less than one millisecond). Finally, we investigated how parameter passing affects performance. To this end, our next test compared the overhead produced by parameters in the .NET remoting call. This overhead stems from the need to marshal arguments at the sender and restore them at the receiver. The `CreateFile` call has seven parameters and one return parameter, which need to be serialized and exchanged between processes. The last entry (i.e., Test 3) in Table 1 shows that including parameters exacerbates the performance penalty.

Table 1. P/Invoke vs. Remoting

Test	**Test Description**	**Direct Call (P/Invoke) (ms)**	**Remoting Call (ms)**
1	No Security Layer	15	234
2	Active Security Layer	15	234
3	Active Security Layer + Function Parameters	15	286

While the overhead of a remote procedure call is an order of magnitude larger than invoking unmanaged code within a process, this is not surprising. Also, note that the in-process P/Invoke call incurs significantly more overhead than a regular function call. Thus, we do not expect this mechanism to be used frequently by performance-critical applications and believe that the increase in security clearly outweighs the performance loss.

4.3 Remoting

Another feature that we evaluated is the remoting infrastructure and the generated stub libraries. In particular, we want to ensure that we have not introduced limitations on parameter passing and that we maintain transparency for managed real-world applications that use native library components.

As mentioned in Section 2, an important reason for the introduction of P/Invoke and the ability to include unmanaged code into .NET applications is the need to call Windows API functions. Thus, we have to ensure that our protection infrastructure supports the invocation of (almost) all API functions. To test the ability of our system to call Windows API functions, we selected a representative subset of ten routines from important areas such as process management, file handling, and helper functions (all implemented in the `kernel32.dll`, the core Windows kernel library). We then tested whether these functions can be invoked from managed code running in a different process. We observed that our design successfully passed the relevant parameters across the process boundary via .NET remoting and invoked the native functions in the remote server process. After invoking the respective function, possible return parameters were successfully passed back to the original process.

Besides tests with Windows API functions, we also investigated our system when running real-world managed applications that make use of native library routines. To this end, we tested our infrastructure on two popular libraries: Sleepycat Software's Berkeley Database [23] and the OpenGL graphics library [20].

Berkeley Database (BDB) is an embedded database. This is, the database engine component is compiled as a library and linked with the application. BDB is officially available as libraries for multiple languages such as C, C++, and Java. In addition, an unofficial C# wrapper [1] exists to port BDB to .NET. This wrapper uses P/Invoke to call the functions of the original BDB library. To test our system, we used the C# wrapper to invoke functions of the BDB library. More precisely, our test application uses the C# BDB wrapper to open a database, store and retrieve records, and close the database. Function parameters include strings, integers and enums for supporting flags.

For testing OpenGL, we used a C# wrapper called CsGL [4] that encapsulates a native OpenGL library. To test our prototype, we exercised basic OpenGL functionality, such as filling the background of a window and drawing a rectangle. However, because most OpenGL functions use a similar syntax, we are confident that this covers the majority of OpenGL.

In both cases, our system automatically generated the necessary proxy libraries to split the managed part and the native library into two processes. That is, instead of invoking unmanaged library functions directly with P/Invoke, the parameters were first transfered to a remote process via .NET remoting. Only there were the native functions executed (via P/Invoke). Also, in case where a function returned a value, these values were properly returned to the managed application. This demonstrates that our system can automatically and transparently isolate native components from managed code.

5 Related Work

The system presented in this paper uses a *sandbox* to confine the execution of potentially untrusted applications. Sandboxing is a popular technique for creating confined execution environments that has been of interest to systems researchers for a long time.

An important class of sandboxing systems uses system call interposition to monitor operating system requests. That is, system calls are intercepted at the kernel interface. Then, these calls and their arguments are evaluated against security policies and denied when appropriate. Numerous approaches have been proposed [2, 11, 17, 3] that implement a variation of a sandboxing mechanism based on system calls. These approaches typically differ in the flexibility and ease-of-use of the policy language and the fraction of system calls that are covered.

One problem with kernel-level sandboxing mechanisms is the need to install the necessary policy enforcement infrastructure (e.g., kernel drivers or operating system modifications). To circumvent this problem, techniques [15, 12] have been proposed that rely on existing monitoring infrastructure in the kernel (e.g., APIs used for tracing and debugging such as *ptrace*) to intercept system calls, which are then processed by a monitor that resides in user space.

The main differences between our proposed approach and sandboxing techniques that operate on system calls are twofold. First, we do not only analyze the invoked system calls but can also force native code to go through user-mode libraries first (in our case, Windows API functions) before invoking a system call.

That is, our two-step authorization process extends system call interposition to user libraries. The second difference is that we distinguish between a trusted, managed part and an untrusted, native part of an application, which originally run together in the same address space. To protect the managed code from malicious, unmanaged code, both parts have to be run in separated processes.

Forcing native code to go through user libraries can also be achieved with program shepherding [16], a method for monitoring control flow transfers during program execution to enforce security policies. The difference to our system is that program shepherding cannot prevent data values from being overwritten, a property that we obtain by executing managed and unmanaged code in two separate address spaces.

Being at the boundary between potentially untrusted user programs and the trusted kernel, system calls have received interest also from other areas of security research. In particular, system calls have been extensively used for performing host-based intrusion detection. To this end, specifications of permitted system calls were either learned by observing legitimate application runs [8] or extracted statically from the application [24, 7].

Finally, Herzog and Shahmehri [13] present an approach that extends the Java policy syntax for resource control. While we do not extend the .NET policy syntax *per se*, we extend its reach by applying it to native code.

6 Conclusions

The number of applications that are being downloaded from Web sites and automatically executed on-the-fly is increasing every day. Unfortunately, some of these applications are malicious. The .NET framework provides a security mechanism called Code Access Security (CAS) to help protect computer systems from malicious code, to allow code from unknown origins to run with protection, and to help prevent trusted code from intentionally or accidentally compromising security. CAS succeeds in restricting undesired actions of managed code. However, the permission to invoke unmanaged (i.e., native) code gives a potential attacker complete freedom to circumvent all restrictions.

This paper introduced a system to extend the CAS rule-set to unmanaged code. The evaluation of the proof-of-concept prototype of our proposed system shows that our design is viable. In particular, we successfully extended the CAS rule set to important Windows API functions. By confining a possible attacker to using the Windows API, we subjected unmanaged code to our security layer. Further, we successfully protected our system against possible attack vectors, such as circumvention of the security layer and memory corruption. To the best of our knowledge, the presented architecture and implementation is the *first solution* to secure unmanaged code in .NET.

References

[1] Berkeley DB for .NET. `http://sourceforge.net/projects/libdb-dotnet`.

[2] A. Berman, V. Bourassa, and E. Selberg. TRON: Process-specific file protection for the UNIX operating system. In *Winter USENIX Technical Conference*, 1995.

[3] S. Chari and P. Cheng. BlueBox : A Policy-Driven, Host-Based Intrusion Detection System. In *Network and Distributed Systems Security Symposium (NDSS)*, 2002.

[4] CsGL. `http://csgl.sourceforge.net/`.

[5] .NET Framework Development Center. `http://msdn.microsoft.com/netframework/`.

[6] ECMA. ECMA 335 - Common Language Infrastructure Partitions I to VI; 3rd Edition, 2005.

[7] H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee, and B. Miller. Formalizing Sensitivity in Static Analysis for Intrusion Detection. In *IEEE Symposium on Security and Privacy*, 2004.

[8] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A Sense of Self for Unix Processes. In *IEEE Symposium on Security and Privacy*, 1996.

[9] A. Freeman and A. Jones. *Programming .NET Security.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2003.

[10] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering.* Prentice Hall Inc., New York, 1991.

[11] D. Ghormley, D. Petrou, S. Rodrigues, and T. Anderson. SLIC: An Extensibility System for Commodity Operating Systems. In *USENIX Technical Conference*, 1998.

[12] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications: Confining the wily hacker. In *6th USENIX Security Symposium*, 1996.

[13] A. Herzog and N. Shahmehri. Using the Java Sandbox for Resource Control. In *7th Nordic Workshop on Secure IT Systems (NordSec)*, 2002.

[14] G. Hunt and D. Brubacher. Detours: Binary Interception of Win32 Functions. In *3rd USENIX Windows NT Symposium*, 1999.

[15] K. Jain and R. Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *Network and Distributed Systems Security Symposium (NDSS)*, 2000.

[16] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution Via Program Shepherding. In *11th USENIX Security Symposium*, 2002.

[17] C. Ko, T. Fraser, L. Badger, and D. Kilpatrick. Detecting and Countering System Intrusions Using Software Wrappers. In *9th USENIX Security Symposium*, 2000.

[18] .NET Framework Class Library Documentation - Security.Permissions. `http://msdn.microsoft.com/library/en-us/cpref/html/frlrfSystemSecurityP%ermissions.asp`, 2006.

[19] G. Nebbett. *Windows NT/2000 Native API Reference.* New Riders Publishing, Thousand Oaks, CA, USA, 2000.

[20] OpenGL. `http://www.opengl.org`.

[21] R. Osterlund. Windows 2000 Loader, What Goes On Inside Windows 2000: Solving the Mysteries of the Loader. *MSDN Magazine*, March 2002.

[22] M. Russinovich and B. Cogswell. Windows NT System-Call Hooking. *Dr. Dobb's Journal*, January 1997.

[23] Sleepycat Software. Berkeley DB Database. `http://www.sleepycat.com/`.

[24] D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *IEEE Symposium on Security and Privacy*, 2001.

[25] Platform SDK: Windows API. `http://www.microsoft.com/msdownload/platformsdk/`.