

Safely Redistributing Untrusted Code using .NET

Martin C. Carlisle, Jeffrey W. Humphries, and John A. Hamilton, Jr., *Senior Member, IEEE*

Abstract—Reusing software components is a textbook software engineering best practice. Developers reuse components written by others, combining them in unique ways to create new software products. Reusing software components can create a significant security risk, as these reused components may behave badly, either by malicious intent or negligence on the part of their authors.

The .NET framework provides fine-grained mechanisms for specifying how software should be trusted. Permissions are granted based on the source of software, and where it currently resides (on the local disk, or in a particular internet zone). Unfortunately, these trust guarantees are difficult to manage, and there is no guarantee that an end-user receiving a redistributed untrusted component will correctly set its trust level.

We propose a framework with a set of easily understood trust levels, and a simple mechanism for applying these trust levels both to already-compiled applications and libraries within the .NET framework. This will allow both end-users and software developers to leverage the work of others, while maintaining guarantees that this software will not, intentionally or otherwise, cause damage to their systems or leak confidential information. This tool should provide significant opportunities for code reuse with security and should be easily extended to handle related applications, such as those using compiled Java class libraries.

Index Terms—D.4.6 Security and Privacy Protection, D.2.13.b Reusable libraries, D.3.4.k Runtime environments.

I. INTRODUCTION

“THE PROMISE of object-oriented systems lies in their promise of code reuse.” [1] Ideally, a software developer should be able to grab from a collection of available modules, use the portions that are relevant to their task, and focus their efforts on creating new functionality. Both .NET and Java provide a huge collection of such libraries and components; however, the ultimate solution won't be to put everything conceivably reusable into the standard

programming environment. Instead, developers should have a smorgasbord of components available, from which they can selectively choose which ones to add to their projects.

Unfortunately, using code, or even a compiler written by others introduces risk [2]. The broadly used graphics processing libpng library was discovered to have numerous vulnerabilities. As a result, a large number of products had exploitable security holes, including such widely used programs as MSN Messenger and Adobe Reader [3], causing embarrassment to Microsoft, Adobe and others.

While the libpng vulnerabilities were presumably accidental, in the future we may see cases where backdoors and vulnerabilities are deliberately planted in components designed for reuse. Individuals and fringe groups routinely place malicious code in their freely available components. We are likely to find more systematic efforts to infiltrate supposedly safe and reputable repositories. Developers therefore need to protect end users by not only ensuring their own modules are developed securely, but also by ensuring that reused modules will not cause mischief. One possibility is to reverse engineer all reused modules; however, this obviously greatly reduces the advantages of code reuse.

Instead, we take advantage of the existing stack inspection techniques available in the Java Virtual Machine (JVM) and Microsoft's Common Language Runtime (CLR) [4]. We propose a tool which rewrites .NET executables and libraries to add specified security rules. We provide a set of easy to understand choices, along with an interface for creating custom rules.

Section 2 describes previous work on sandboxing untrusted applications. In Section 3, we review the available security features in the .NET framework. Section 4 describes a set of threats from untrusted code. Section 5 describes our rewriting tool, which provides a very simple mechanism for managing trust in a way that allows for simple redistribution. Future possibilities for expansion of this work are described in Section 6.

II. PREVIOUS WORK

The idea of dealing with untrusted helper applications is not new. Goldberg et al. [5] proposed a system called Janus that leveraged Solaris' process tracing facility to capture attempts by an untrusted helper application to access dangerous system calls. Ostia [6] added the idea of agents, which simplified the task of system call interposition. Bernaschi, Gabrielli and

Manuscript received February 14, 2006.

Martin C. Carlisle is with the Department of Computer Science and Software Engineering, Auburn University, Auburn, AL 36849 USA, on leave from the United States Air Force Academy, USAFA, CO 80840 USA (phone: 719-333-3590; fax: 719-333-3338; e-mail: carlisle@acm.org).

Jeffrey W. Humphries, was with the United States Air Force Academy, USAFA, CO 80840 USA. He is now with US Air Force, in GA USA (e-mail: humphries@acm.org).

John A. Hamilton, Jr. is with the Department of Computer Science and Software Engineering, Auburn University, Auburn, AL 36849 USA (e-mail: hamilton@eng.auburn.edu).

Mancini [7] propose a similar mechanism to Janus, but for setuid programs and root daemons. Each of these systems requires modifying the kernel of the underlying operating system.

Alcatraz [8] also uses a system call interception to stop unwanted file access. It provides a virtual mirror of the file system, so that writes don't affect the underlying system. It also runs without modification of the Linux kernel. Alcatraz does not, however, solve our problem of mixing trusted and untrusted code within a single application.

III. SECURITY FEATURES IN THE .NET FRAMEWORK

The Microsoft .NET framework provides evidence-based security to address issues of running untrusted or partially trusted code [9]. When an assembly (library or executable) is loaded, it presents "evidence" to the framework. This evidence includes items such as:

- Source of the code (name of web site, URL, directory)
- Code signature (signed name of assembly, publisher)
- Custom security object

The CLR then uses policy from the domain, machine, user and application to determine what permissions the assembly will have. At each level, the set of permissions can be restricted, but not expanded. For example, if the machine policy indicates that an assembly loaded from www.badguys.com will not have access to files on the local hard drive, the user policy cannot grant this permission.

Evidence based security provides an additional level of security to that provided by the operating system. Most OS security is based on the identity of the current user. When logged in as an administrator, all programs executed have permission to do anything that an administrator can do. Evidence based security allows for discrimination between various programs without having to log out and log back in to change security levels.

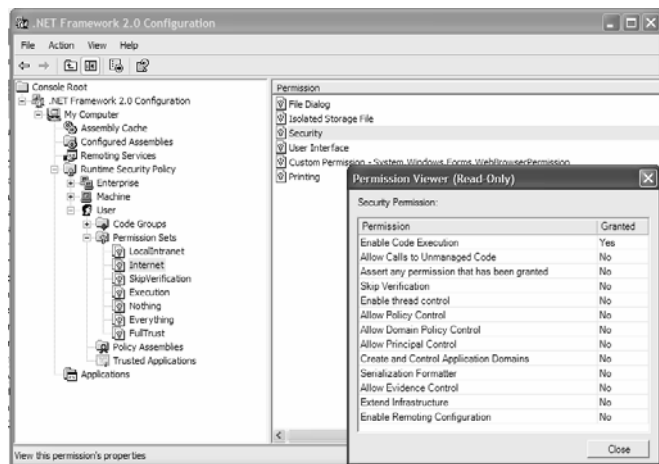


Figure 1: Microsoft .NET Security Configuration Tool

Administrators and users can create permission sets and assign these using the .NET Framework Configuration tool (see Figure 1). Several permission levels are provided by default, including LocalIntranet, Internet and FullTrust. By default, assemblies that are located on the local hard drive are given full trust. This creates a serious problem for redistributing untrusted code as part of an application, as the result will, by default, be the end user having given this code full trust. Furthermore, creating a policy for a particular module using this tool can be tedious, as it involves managing code groups, zones, and permission sets. If not fully specified, the permissions of a particular file could change simply by copying it to a different folder.

Another major portion of the security model involves code access security. Here, the code itself requests or refuses particular permissions [10]. These requests can either be done declaratively or imperatively. A declarative request is static, and appears before the declaration of an assembly, class or method. An imperative request is dynamic, and is included inside a method. Code access security requests do not permit a module to obtain more permissions than allowed by the evidence-based .NET framework configuration. That is, the permissions a module actually obtains results from the intersection of the requested permissions and the policy permissions. Modules that do not utilize code access security request full trust by default, and are therefore given the permissions specified by policy.

These sets of permissions are important, as the .NET framework libraries use them to determine whether or not to allow certain operations. For example, in the Connect method of System.Net.Sockets.Socket, the permission NetworkAccess from the class System.Net.SocketPermission is demanded. Evaluating this permission demand results in a stack walk [10].

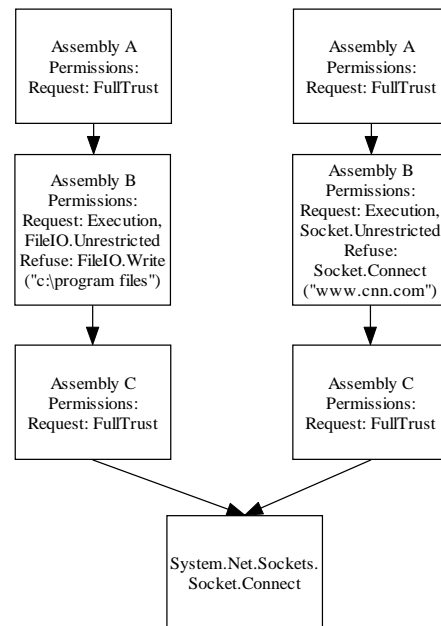


Figure 2: Two different call stacks to Socket.Connect

Consider two different code paths to `Socket.Connect` as shown in Figure 2. Although in both cases, the method which calls `Socket.Connect` is in a fully trusted module, only the call on the right will succeed (and then only if the host is not `www.cnn.com`). This is because a demanded permission is only granted if all of the stack frames above it have the required permission.

This stack walk can be stopped by using a security assertion. If Assembly C asserts the permission to connect to the socket, then the stack walk will stop at the assertion and the call will succeed. The point of an assertion is to perform a trusted operation on behalf of an untrusted caller, and should be used only sparingly [11]. Assertions are only effective if the asserting method actually has the asserted permission.

IV. UNTRUSTED CODE THREATS AND ATTACKS

We now turn our attention to what adverse outcomes could result from running untrusted code and how this code might attempt to subvert the security framework provided by .NET. We ignore the possibility that the untrusted code might engage in a denial of service attack by simply consuming CPU resources. We first consider how untrusted code might compromise the integrity or confidentiality of the system, and then consider three different techniques whereby untrusted code might abuse trusted modules.

A. *Modifying or Destroying Local Data*

Particular attention must be given to code that is allowed to access the local hard drive. Even if the code is prevented from modifying the operating system or applications on the machine, it still can end up compromising the entire machine. Here's one possibility. Suppose the malicious module `BadModule` is able to discover that `BuggyDocReader` is installed on the machine. It could then modify a `BuggyDocReader` document to cause `BuggyDocReader` to have a buffer overflow the next time that document is loaded. The buffer overflow would then have whatever permissions `BuggyDocReader` had, and could wreak further havoc from there.

Microsoft provided `IsolatedStorage` [12] in the .NET framework. This provides an isolated file where an assembly can cache data, with a disk quota. Unfortunately, this is not particularly useful for storing user data, as it is hard for the user to back up this data, or distribute it to other users, but it is a secure way to allow an assembly to store users settings between sessions.

B. *Leaking Information*

If the untrusted code has the ability to use the network, it can create a channel for sending data to an outside party. This is true even if the code is only able to download web pages, as the URL requests can themselves be used for transmitting data, simply by including the data as extra parameters, e.g. `http://www.badguy.com/index.html#outgoing_data` [13]. The ability for untrusted code to read system environment

variables or the file system should never be combined with the ability to access the network. Even though the data from system environment variables may seem harmless, it can be used by an attacker to determine what software is installed on a machine and check for additional vulnerabilities [13].

C. *Luring Attacks*

A luring attack involves an untrusted module getting a trusted module to perform a secure operation on its behalf. Using the left call stack in Figure 2 as an example, a luring attack would occur if Assembly B could get Assembly C to connect to the internet on its behalf. The stack walk is designed to prevent luring attacks; however, there are ways that this can be subverted. For example, if Assembly C makes an incorrect security assertion, then it might perform operations for B that B could not do itself. Also, if C contains public attributes, B may be able to maliciously change these. For example, if C stores the name of a web site to connect to in a public attribute, B might simply change this to a malicious proxy, creating an information leak. Similarly, if C maintains a directory or filename in a public attribute, B may be able to change this method so that the more trusted module C will write over critical system files.

D. *Serialization Attacks*

If untrusted code is allowed to do serialization and deserialization, this can create security problems. First, the untrusted code may gain access to private data by reading it from the serialized representation (which it would be unable to do directly because of the access restrictions on private attributes). Further, this may also allow the untrusted code to create trusted objects with invalid attributes. For example, if a trusted object has an integer attribute that is always between 0 and 255, the untrusted code might create data to be deserialized that has a value of 1000 for this field. If the deserialization routine in the trusted code does not verify the data, this could lead to undesired effects. To protect against this attack, the .NET framework requires assemblies to be given serialization permission to perform serialization or deserialization. This permission must not be given to untrusted modules.

E. *Exception Attacks*

Mismanaged exception handlers can also introduce potential security holes [14]. For example, consider the following code from a trusted module:

```
try {
    PerformSecurityAction();
    // perform some file IO
}
catch {...}
finally {
    RevertSecurityAction();
}
```

If the untrusted code can get the trusted module to throw an

exception (perhaps by passing it bad data), then an exception filter from the untrusted module could run before the security action had been reverted. It is necessary to wrap the try block with the security action in a second to prevent an untrusted exception filter from running with additional privilege, as:

```
try {
    try {
        PerformSecurityAction();
        // perform some file IO
    }
    finally {
        RevertSecurityAction();
    }
}
catch {...}
```

This attack is somewhat counterintuitive, as you would expect that the frames of the trusted module would be removed before any exception handling code from the caller occurs. Unfortunately, however, the .NET Framework does exception handling in a two-pass process. First, all of the exception filters are evaluated. Then, frames are removed and the appropriate exception handlers are called. Since the exception filters are called before the stack frames are removed, the exception filter can run with elevated privilege.

V. CODE REWRITER FOR SECURITY

Our scenario is that an application developer wishes to reuse a .NET module downloaded from the internet in their application, and then distribute that application to end users. Although the .NET Configuration Manager could be used to restrict the permissions of this downloaded module, these permissions are both difficult to manage, and extremely difficult to redistribute. Also, as the end user is likely to install an application containing a redistributed dynamically linked library (DLL) to the local hard drive, this will mean that the assemblies will, by default, be given full trust. Butler Lampson, a Microsoft Distinguished Engineer, commented in his NSF CyberTrust 2005 invited presentation [15] that using role-based access control (such as the .NET evidence-based security) to achieve least privilege puts the responsibility in the hands of the programmer, where it belongs. However, he believed it to be “hopeless” because it is unmanageably complicated. We agree that, in its current form, managing these permissions is untenable; but, rather than abandoning the mechanism provided by .NET, we instead propose a simple way of using the mechanism to achieve the desired security result.

Our code rewriter will use a combination of adding declarative security and rewriting byte code to ensure that an untrusted module can be safely redistributed. The code rewriter will use Microsoft’s ILDASM disassembler to get a disassembled text file. It then will parse this file and create a second assembly text file containing the modifications. Finally the modified assembly is reassembled using Microsoft’s ILASM assembler.

First, as shown in Figure 3, the rewriter will allow the user to pick from a simple list of functionality that the DLL should be allowed to perform.

Following are the basic permissions that may be granted to an assembly:

- * Execute: this means that the assembly will be allowed to load and execute.
- * Display user interface: this allows the assembly to create SafeTopLevel windows and also to read and write data from its own clipboard.
- * Access isolated storage: allows the assembly to create an isolated storage file (as described in Section 3.1, with the specified quota).
- * Files via Windows dialogs: allows the assembly to read and write files; however, the assembly can not specify filenames. It can display Open and Save dialogs, and then read and write the files that are selected by the user only.
- * Network sockets: allows the assembly to communicate with network sockets.
- * Serialization of binary data: allows the assembly to read and write binary data via serialization.

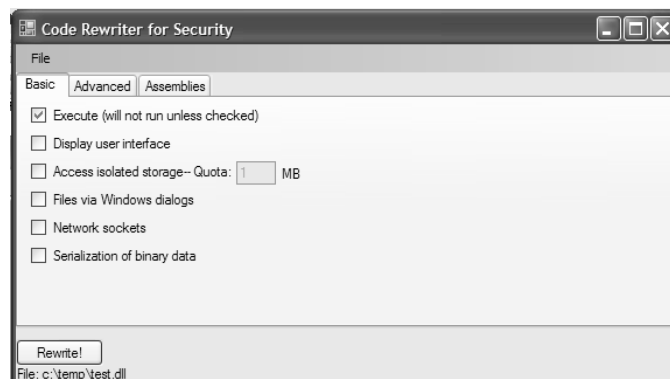


Figure 3: Code Rewriter for Security basic tab

It may be the case that the developer needs finer grained control of the permissions given to an assembly. The advanced tab, shown in Figure 4, may be used to allow or deny any permission attribute defined by the .NET framework. As can be seen, it has been selected to allow the assembly to read and write “c:\temp\test.txt”, but not to write to the folder “c:\program files” (or any of its files or subfolders). Generally, it is expected that the basic tab will be sufficient; however, the advanced tab provides flexibility when needed.

The luring and exception attacks described in Section 3 exist because the untrusted DLL may call methods, access properties, or extend classes from the main application. While it is possible to restrict untrusted DLL calls, accesses and class extensions using application domains [16], this requires complicated coding. Instead, we simply rewrite the assembly, replacing every reference to an assembly that is not explicitly listed with a throw of a SecurityException.

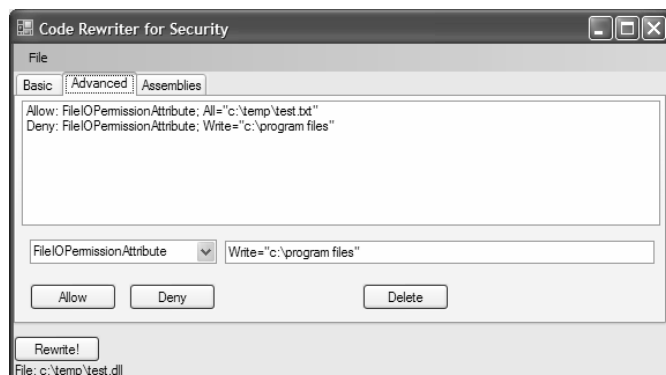


Figure 4: Code Rewriter for Security advanced tab



Figure 5: Code Rewriter for Security assemblies tab

Figure 5 shows the portion of the interface that will allow the developer to select which external assembly references will be allowed. By default, references to the .NET framework assemblies are allowed and references to other assemblies are not. The header of the .NET assembly lists all of these external references as follows:

```
.assembly extern System.Windows.Forms
{
  publickeytoken=(B7 7A 5C 56 19 34 E0 89 )
  .ver 1:0:5000:0
}
```

Having the tool parse the header allows the user to quickly see what assemblies the module is trying to reference, and also which references may require careful consideration.

There are two possibilities for how the external references may appear in the code. The reference will be either in an intermediate language instruction, such as calling an external method, or in the specification of a class, specifying inheritance from an external class or interface.

A disassembled call to an external method is similar to the following:

```
callvirt instance void
[System.Windows.Forms] System.Windows.
Forms.Control::set_Text(string)
```

This is replaced by throwing a security exception using the

following code:

```
newobj instance void
[mscorlib]System.SecurityException::.ctor(
)
throw
```

Handling an attempt to extend a class from a disallowed assembly is slightly more complicated.

```
.class public auto ansi beforefieldinit
ExtendDisallowed extends
[disallowed]SecureObject
```

Here, we cannot simply remove the class, as the result may not be a valid assembly. Instead, we simply add the throw of the security exception (as above) to each of the constructors for this class. This guarantees that no object of the class will ever be created, thus preventing any luring attacks via a child class.

VI. CONCLUSIONS AND FUTURE WORK

The use of type-safe managed code or virtual machines allows for useful security guarantees (in particular, the absence of buffer overflow errors). Although .NET provides a significant security framework, its focus is on individual administrators protecting their machines or networks from untrusted code, not on allowing developers to include untrusted modules in new software projects. Furthermore, the framework is overly complex, which means it is unlikely to be used correctly.

We propose a tool which allows developers to rewrite .NET assemblies so that they can be redistributed with security guarantees that are enforced by the .NET framework. This tool will have a very simple interface and is sufficiently flexible to create any possible security policy. The code rewriter also provides the ability to choose simple security policies that should cover most cases. The rewriter does not need access to the original source code of the assembly, but works with the intermediate language (bytecode).

This tool should provide significant opportunities for code reuse with security, as it removes the need to write special security code when including an untrusted assembly. We anticipate that the tool will have negligible effect on running time, as the stack walk is already being performed by the framework, and we are simply creating additional cases where this will fail. We also expect that, for well-behaved libraries, the rewriter will have no semantic effect. This follows as the only instructions that are rewritten are those which involve calling disallowed assemblies. We anticipate being able to run the tool against a collection of open source modules and determining the effect on running time and functionality.

In the future, it would be worthwhile to create a similar program to handle compiled Java class libraries.

REFERENCES

- [1] McManis, Chuck. "Code Reuse and Object-Oriented Systems." Online. Available at: <http://www.javaworld.com/javaworld/jw-12-1996/jw-12-indepth.html>
- [2] Thompson, Ken. "Reflections on Trust." *Communications of the ACM*, Vol. 27, No. 8, August 1984, pp. 761-763.
- [3] NIST. Vulnerability Summary CAN-2004-05-97. Online. Available at: <http://nvd.nist.gov/nvd.cfm?cvename=CAN-2004-0597>.
- [4] Gordon, A. and C. Fournet. "Stack Inspection: Theory and Variants." Symposium on Principles of Programming Languages, Portland OR, January 2002, pp. 307-318.
- [5] Goldberg, Ian; Wagner, David; Thomas, Randi and Eric Brewer. "A Secure Environment for Untrusted Helper Applications (Confining the Wily Hacker)." USENIX Security Symposium, San Jose CA, July 1996.
- [6] Garfinkel, Tal; Pfaff, Ben and Mendel Rosenblum. "Ostia: A Delegating Architecture for Secure System Call Interposition." Internet Society Symposium on Network and Distributed Systems Security, February 2004.
- [7] Bernaschi, M., Gabrielli, E., and Mancini, L. V. "Operating system enhancements to prevent the misuse of system calls." ACM Conference on Computer and Communications Security, Athens Greece, November 2000.
- [8] Liang, Zhenkai; Venkatakrishnan, V.N. and R. Sekar. "Isolated Program Execution: An Application Transparent Approach for Executing Untrusted Programs." Annual Computer Security Applications Conference, Las Vegas, December 2003.
- [9] Watkins, D. and S. Lange. "An Overview of Security in the .NET Framework." Online. Available at <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/netframesecover.asp>.
- [10] Microsoft Corp. "Code Access Security Basics." Online. Available at <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconcodeaccesssecuritybasics.asp>.
- [11] Farkas, Shawn. "All About Assert." Online. Available at: <http://blogs.msdn.com/shawnfa/archive/2004/08/23/219155.aspx>.
- [12] Tavares, Chris. "Understanding Isolated Storage." Online. Available at: <http://www.dotnetdevs.com/articles/IsolatedStorage.aspx>.
- [13] Dean, Drew; Felten, Edward and Dan Wallach. "Java Security: From HotJava to Netscape and Beyond." IEEE Symposium on Security and Privacy, Oakland CA, May 1996.
- [14] Microsoft Corp. "Securing Exception Handling." Online. Available at: <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconsecuringexceptionhandling.asp>.
- [15] Lampson, Butler. "Computer Security in the Real World." NSF CyberTrust 2005 Principle Investigators Meeting, September 2005. Online. Available at: <http://www.ics.uci.edu/~cybrtrst/schedule.htm>.
- [16] Robbins, Thom. "Application Domain Basics." Online. Available at: <http://blogs.msdn.com/trobbins/articles/343958.aspx>.