

# Reuse Of Components In .NET

Dhruba Shankar Ray

Student M. Tech(CSE), University School of Information Technology, GGSIPU

shankardhrub@gmail.com

Udayan Ghosh

Lecturer, University School of Information Technology, GGSIPU

g\_udayan@lycos.com

---

## ABSTRACT

*It is a normal practice by developers to reuse components. But reusing of components can cause a significant security risk, if the components come from untrusted source, as these components may behave badly, either by having malicious code in them or due to negligence on the part of their creators. We have fine-grained mechanisms, in .NET framework, for specifying the trust of software. Permissions are granted based on the source of software, and where it currently resides (on the local disk, or in a particular internet zone). But, these trust guarantees are difficult to manage, and there is no guarantee that an end-user receiving a component created by someone else, will correctly set its trust level. There is a need to understand the trust levels easily, and through simple mechanisms, these trust levels should be applicable both, to already-compiled applications, and libraries within the .NET framework. This will ensure that the end-users and software developers appreciate the work of others, while feeling guaranteed that this software will not, cause damage to their systems or leak confidential information, intentionally or unintentionally. We show how this could be done taking into consideration security aspects provided in .NET framework.*

## I. INTRODUCTION

Ideally, a software developer should not waste time in what have already been developed, rather, be able to grab from a collection of available modules, use the portions that are relevant to them, and focus their efforts on creating new functionalities. The promise of object-oriented systems lies in their promise of code reuse. Although we have a huge collection of libraries and components in .NET, however, the ultimate solution won't be to put everything possibly reusable into the standard programming environment. Using of components written by others introduces a lot of risk [1]. It is a well known fact by now that, component libpng 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, causing embarrassment to Microsoft, Adobe and others.

Vulnerabilities in libpng were supposedly accidental, but in the future there can be cases where backdoors or other vulnerabilities are deliberately planted in components designed

for reuse. Certain individuals and groups deliberately and routinely place malicious code in their freely available components. In future there could be more systematic efforts to infiltrate supposedly safe and reputable repositories. Developers therefore have to put some extra efforts to protect end users by not only ensuring that their own modules are developed securely, but also by ensuring that reused modules do not cause any harm to their computer or information. One possible way that can be achieved is by reverse engineering all reused modules; however, this obviously, greatly reduces the advantages of code reuse, moreover different types of obfuscation technologies and tools available in Visual Studio .NET 2005 or as open source makes reverse engineering very difficult.[2]

We can also take advantage of the existing stack inspection techniques available in Microsoft's Common Language Runtime (CLR) [3]. We can rewrite .NET executables and libraries to add specified security rules at command line or through custom developed tools. These tools if developed should have a set of easy to understand choices, along with an interface for creating custom rules.

## II RELATED WORKS

Previous works in stopping, or tracing applications, trying to access, dangerous or nondesired system resources or files, has been done for Solaris and Linux systems. A system called Janus was proposed for Solaris systems [4]. This system would facilitate process tracing, to monitor, if any, untrusted helper application, tried to access dangerous system calls. But here modification of the kernel of the underlying operating system was necessary [5]. To stop unwanted file access, Alcatraz was developed for Linux OS. It provided a

virtual mirror of the file system, so that any file-writes did not affect the underlying system.

### III. SECURITY ARCHITECTURE IN NET FRAMEWORK

The security architecture of the .NET Framework is composed of a number of core elements, including:

- Evidence-based security
- Code access security
- The verification process
- Role-based security
- Cryptography
- Application Domains

The key elements of the .NET Frameworks evidence-based security subsystem include policy, permissions, and evidence [6]. In essence, .NET Framework policy defines what resources code in executing assemblies may access, preventing software from errantly or maliciously harming the integrity of data. And the permissions lie at the root of policy. Permissions describe one or more resources and associated rights, and implement methods for demanding and asserting access. The developer may extend these permissions definitions to include application-defined resources and methods for verifying access rights. Whenever 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 (hash value, strong name of assembly, publisher certificate)
- 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, defined at machine.config, indicates that an assembly loaded from www.myhome.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.

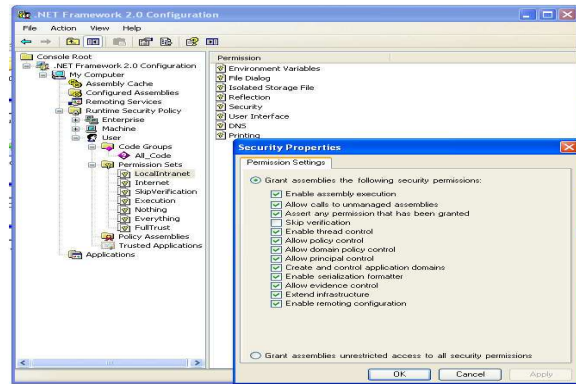


Figure 1: Microsoft .NET Security Configuration Tool

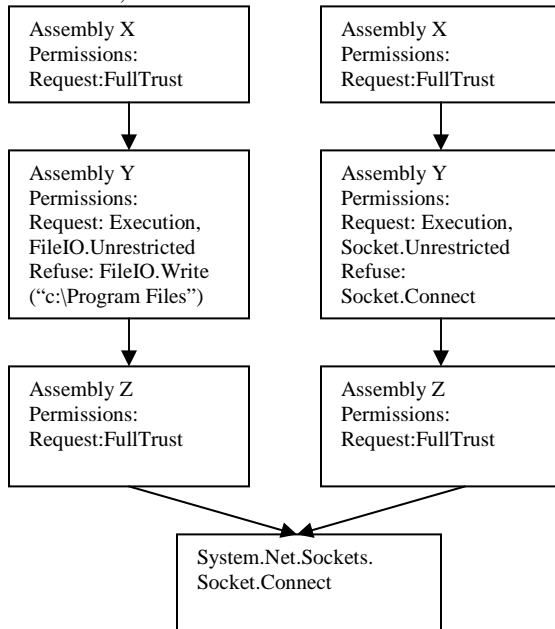
Administrators and users can create permission sets and assign these using the .NET Framework Configuration tool (figure 1). Several permission levels are provided by default, including LocalIntranet, Internet and FullTrust [7]. By default, assemblies that are located on the local hard drive are given 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 when it is copied to a different folder.

Code access security (CAS) is the enforcement engine that ensures assembly code does not exceed its granted permissions while executing on a computer system. As managed code assemblies are loaded for execution, they are associated with a corresponding set of permissions. If a method in an assembly needs permission to access a resource, the code providing access to that resource will demand the appropriate permission object. When this occurs, a stack walk is initiated. This checks that each assembly in the call-chain has the demanded permission granted to it, not just the immediate caller. If any of the callers fail this test, a security exception is generated and the requested operation is not performed. The developers here programmatically express security constraints within the code. There are two styles for doing this declarative and imperative. The declarative style uses custom attribute syntax. 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.

These sets of permissions obtained by a module 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.

Consider two different code paths to Socket.Connect as shown in Figure 2. Although in both cases, the method which calls



**Figure 2: Two different call stacks to Socket.Connect**

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.ipu.ac.in). 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 Z 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. Assertions are only effective if the asserting method actually has the asserted permission.

**IV. POSSIBLE THREATS AND ATTACKS**

Now let us 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. Here let us 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 different techniques whereby untrusted code might abuse trusted modules.

*A. Modification-Destruction and Leaking of 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 GameModule is able to discover that XDocReader is installed on the machine. It could then modify a XDocReader document to cause XDocReader to have a buffer overflow the next time that document is loaded. The buffer overflow would then have whatever permissions XDocReader had, and could wreck further havoc from there. Microsoft provided IsolatedStorage [8] 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.

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.myhome.com/index.html#outgoing\\_data\[9\]](http://www.myhome.com/index.html#outgoing_data[9]). 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 [9].

*B. 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 Y could get Assembly Z 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 Z makes an incorrect security assertion, then it might perform operations for Y which Y could not do by itself. Also, if Z contains public attributes, Y may be able to maliciously change these. For example, if Z stores the name of a web site to connect to in a public attribute, Y might simply change this to a malicious proxy, creating an information leak. Similarly, if Z maintains a

directory or filename in a public attribute, Y may be able to change this method so that the more trusted module Z will write over critical system files.

#### C. Attacks Originating From Serialization

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.

#### D. Exception Handling Attacks

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

```
try
{
DoSecurityAction(); // perform some file
IO
}
catch {...}
finally
{
RevertSecurityActivity();
}
```

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 action
}
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. USING COMMANDS OR TOOLS FOR SECURITY

We here have a scenario wherein application developers wish to reuse a .NET module downloaded from somewhere in the internet, in their application, and then distribute that application to end users. Although the .NET Configuration Manager [7] 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. Through the custom tools or from command line options we need to rewrite the code adding declarative security and rewriting byte code to ensure that an untrusted module can be safely redistributed. To rewrite the code we have to use Microsoft's ILDASM disassembler to get a disassembled text file. Then we need to parse this file and create a second assembly text file containing the modifications. Finally the modified assembly is reassembled using Microsoft's ILASM assembler.

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, 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.

We allow references to the .NET framework assemblies but references to other assemblies are not

disallowed. 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
}
```

By parsing 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

.NET provides a significant security framework, but, 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 they are working on. Presently we are using command line option to create and enforce security policies for the untrusted components. In future an advanced tool could be developed which would allow developers to rewrite .NET assemblies so that they can be redistributed with security guarantees that are enforced by the .NET framework. This tool should have a very simple interface and should be sufficiently flexible to create any possible security policy. The tool should also have the ability to choose simple security policies that should cover most cases. It should not require any access to the original source code of the assembly, and work only with the intermediate language. As extended functionalities it should create back the possible source code in desired language. The tool should provide significant opportunities for code reuse with guaranteed security.

## REFERENCES

- [1] Thompson, Ken. "Reflections on Trusting Trust." *Communications of the ACM*, Vol. 27, No. 8, August 1984, pp. 761-763
- [2] Microsoft Corp. "Thwart Reverse Engineering of Your Visual Basic .NET or C# Code"
- [3] Gordon, A. and C. Fournet. "Stack Inspection: Theory and Variants." *Symposium on Principles of Programming Languages*, Portland OR, January 2002, pp. 307-318.
- [4] 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.
- [5] 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.
- [6] Security in the Microsoft .NET framework-An analysis by Foundstone, Inc. and CORE Security Technologies
- [7] Microsoft Corp. "An Overview of Security in the .NET Framework."
- [8] Microsoft Corp. Introduction to Isolated Storage
- [9] 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.