Designing Secure Solutions with .NET

Bill Ferreira

Designing Secure Solutions with .NET
By Bill Ferreira
GSEC
Version 1.4b option 1

Abstract

The proper approach to designing a solution is one that meets business objects and that protects against identified risks with controls that are transparent to the user. The approach sounds simple enough; the challenge is defining what needs to be protected, what are the risks and types of controls needed, and how to implement them in a cost effective way.

There is no such thing of a security mechanism that will protect your secrets for eternity. If the security mechanism is strong, attacks could be mounted against the environment, vulnerabilities in key management or with a person's willingness to be helpful. Aside from secure design and coding practices you need to consider policies, setting expectations, environmental control, and training. Developing a secure solution requires creating a layered security strategy with the support of policies, controls, and training.

Overview

Writing secure code and knowing how the environment impacts security is important to designing secure software. Solutions extend beyond writing secure software; it involves designing a system that will likely interact with users, and legacy systems to meet some business objective. This requires understanding the business objectives and mitigating any risks through policy and controls.

To begin with, security needs to be considered from the onset of a design rather than as an afterthought. You need to know your risks which come with awareness and knowing your adversaries and their capabilities. Mitigating risks will involve building controls and security mechanisms that will impact policies, environment, and coding. If there are any weaknesses in the implementation, security becomes ineffective. Security need not be complex but failing to understand the impact of your choices will mean the difference between what is secure and what is just obscure. We will take a detailed look at how to define security, and how policies, expectations, environment, and coding decisions can impact our security stance. Through awareness and knowledge we begin designing secure solutions.

Before digging into development and implementation, we need to have some understanding of the requirements for security mechanisms. This will ultimately have an impact on everything else that follows. In particular we need to document:

1) What are the business objectives for the solution?
2) What is the useful lifespan of the data?
3) What are the performance constraints?

4) What are the security goals?
5) What are the security risks?

These questions will dictate the security stance and influence our design decisions. Let's take a look at each of these in more detail.

Knowing the business objective is important because without a business objective there is no need to develop a solution. The business objectives will dictate the complexity of the system and the level of interaction with users and other legacy systems.

Identifying sensitive data and its useful lifespan is one of the most important exercises in designing a secure solution. Useful lifespan can be measured by its economic value over time. Its value can be a hard dollar amount; an economic advantage to the company such as in a secret recipe or trade secret; or, a potential liability such as a patient's medical history in the wrong hands. Over time, the economic value diminishes until it becomes worthless. The lifespan measured in time can be as little as a few hours to as long as centuries. The lifespan of data will determine the type of security mechanisms required to protect it.

Performance constraints can have a big impact on the confidentiality of data and the usability of the system. For real-time high volume data stream processing, performance may be a major criteria on the success of a solution. The other performance measurements would measure reliability and availability. Performance constraints will be a major influence on the cost of the system. The performance of the system extends beyond the costs of development to also encompass hardware and infrastructure.

Security goals deal with the balance of data confidentiality, integrity, and availability. If the availability to the data is reduced or if the security mechanisms are not transparent, workarounds end up being created by users at the risk of security. The demand for increased availability comes at a price of equipment redundancy.

Risk analysis determines threat exposure to our assets, operations, and reputation. During the risk analysis you would do a threat and risk assessment to discover the types of threats and the likelihood of occurrence and its impact on assets, operations, and reputation. Part of risk analysis is knowing your adversary and their capabilities and resources. Areas that would need to be reviewed are the environment, software design and construction, operations, controls and policies.

There are three approaches to risk management: we can accept the risk as a normal part of the operation; mitigate the risk by reducing potential impacts; or we could transfer the risk to a third-party by insuring against it.

It is possible to have security goals, risks, performance constraints, and business objectives at odds with each other. In such cases you need to do a cost/ benefit analysis and review your risk management strategy. Once a balance has been struck you could begin designing security mechanisms to mitigate risks.

Mitigating Risk: Policies, Expectations, and Controls

Even in the proper implementation of security mechanisms, data is only as secure as the weakest link. Other factors that will have an affect on security are:

1) Policies and procedures
2) Educating and setting expectations
3) Key and data management & control
4) Environmental control

Organizational policies give guidance on where responsibilities lay, and outline procedures and guidelines on how the organization should deal with information security. They're tools used to control the environment and management of information security. Policies should be individually targeted to all levels of personnel dealing with sensitive information and all personnel should be aware of these policies and why they are in place and how it affects their work. Good policies are relevant and easy to understand and clearly cover the procedures for protecting information and responsibilities expected of employees. Policies should be revised to reflect changes in the environment. Changes should be integrated back in to the employee manual and training curriculum. An example of a policy would be a policy on how to label sensitive information [4].

Education and setting right expectations is very important within the context of security. All of us have certain expectations with the products and services we use. Not all of us have the same expectations, some are warranted and others need to be reevaluated. Employees and contractors need to be educated on policies & procedures and have the right expectations of what encryption offers, including where, when, and how it should be used; and what are the risks and threats.

Key management policies are very important when implementing confidentiality through encryption. [10] Without proper key management, encryption could easily be ineffective at protecting secrets. Key management policies should describe procedures and guidelines on:

1) Labeling of keying material
2) Generation of keying material
3) Storage of keying material
4) Distribution of keying material
5) Destruction of obsolete keying material

Although keys are not physical like the keys for your home and car, they are none the less used for specific roles and applications. Keys should be label indicating when it was created, purpose, target application, owner, access restrictions and privileges. Keying material include the encryption key and initialization vector that are used to support encryption. The generation of keying material should be done using a true

random numbers or at minimum a cryptographic random number generator. Both are explained in the random numbers section of this paper.

The Storage and distribution of keying material must be secure from tampering until it is destroyed. Distributing in electronic form must be done with key wrapping. That is encrypting the keying material with a master key or using a public encryption scheme. Once the keying material is no longer needed it must be destroyed.

Environmental control is an important issue when it comes to security. The environment that the code runs under determines some of the vulnerabilities that it's susceptible to. In most cases we will never have full control of the environment so we risk attacks with reverse engineering, Trojan horses, keypress loggers [9], and not quite destroyed secrets; and packet sniffers can be used to attack if not go around security mechanisms altogether.

When implementing security, we must assume that attackers have detailed knowledge of the algorithms used, as well as copies of source code, components, framework and libraries. With this assumption we must place the strength of confidentiality in the encryption key and the management of those keys and not have it depend on the secrecy of code. This is perhaps on of the most important point I could make regarding security. Because of this lack of foresight the DVD encryption has been cracked and unauthorized software floats on the Internet. [17]

Design and Construction of Secure Software

Understanding the issues and challenges that face programmers is important to reducing the threat of someone coming along and exploiting a common vulnerability. Let us take a look at some coding pitfalls that we should avoid and at some design strategies that could be implemented to improve security. Awareness, good design and coding practices are vital to implementing secure software.

All software solutions follow a life cycle known as the *Software Development Life Cycle* (SDLC). The stages of the SDLC follow:

1) System requirements
2) Software requirements
3) Analysis of requirements
4) Program Design
5) Coding
6) Testing
7) Maintenance

Gathering requirements and business rules is not a vulnerability in of itself, it seams though when there is a problem with requirements usually the developer shoulders the brunt of retrofitting changes nearing the final stages of development. The cost of changes in the development process becomes exponentially greater as the product

nears the delivery date. The risks also increase if changes are not handled in change management processes. During this stage assumptions should be documented. Depending on whom you ask, assumptions on any give topic can vary. A documentation of assumptions brings everyone on the same page.

Testing is an important phase in the SDLC that is used for quality assurance of accuracy, reliability and performance. Defects will always exist in large systems this is partly do to the fact that as the quality of the system improves the cost of testing increases. To measure the quality we need to collect metrics on the number and types of defects found. Developers should not be the ones testing their software because they tend to test for scenarios that we have already considered and also because of a potential conflict of interest when recording their software defects for quality control metrics.

Before any testing can begin, there needs to be a document that defines the testing scope, expected production environment and its state; testing environment and its state if it differs from the production environment; expected system behavior and specification; and bug fix turnaround time. It is important that all assumptions are recorded since they will also form the basis for test cases. [13] There are various types of testing:

1) Unit testing
2) Integration testing
3) User acceptance testing (UAT)

Unit testing tests the functionality of a procedure and how well it conforms to design specification. Integration testing takes the various procedures together to test how well they integrate with each other. User acceptance testing looks at the final product to see how well it performs in the user's environment.

The inventory of test cases should include stress testing where computing resources are gradually reduced until the procedure breaks. Stress testing should also be done over an extensive period of time to measure resource usage and any memory leaks. Input parameter bounds testing checks for buffer overflows and input that falls outside specifications.

Without going too deep into the SDLC it is important to formalize and follow some design, development and testing methodology. The benefits of formalizing a development methodology is that there is less misinterpretation, scope of work is documented, and better design, which results in fewer revisions.

From a coding perspective there are a number of vulnerabilities that seam to reoccurs that weakens the security of a solution. The vulnerabilities include:

1) Security through obscurity
2) Buffer overflow & format string vulnerabilities

3) Race conditions
4) Locking problems
5) Trusting user input and not validating parameters
6) Poor exception handling.
7) Complexity
8) Control granularity
9) Excessive privilege
10) Compiler intricacies

Security through obscurity and secrecy is a bad approach to information security. The idea is that if security mechanisms are secretive no one will know of any vulnerability. The idea is interesting but a little naïve to think that attackers do not have any skills in reverse engineering code to analyze security mechanisms.

A Buffer overflow exploits vulnerabilities in unchecked buffer boundaries with buffers that use static allocations. The threat is overwriting the buffer's boundaries into sections of code or data. Usually the overflow contains malicious code that gets injected into a processes address space and eventually executes. Format string is similar to buffer overflows but exploits vulnerabilities with formatted I/O constructs. An example of a formatted I/O construct is the *printf* types of commands. Buffer overflow and format string threats have been around for many years, validating user input is the only way for dealing with these types of threats.

In database environments locking is used to protect the atomicity, consistency, isolation, and durability (ACID) of changes to a database. The ACID properties protect the integrity of the data and resources during updates and querying of resources. Locking mechanisms are used to implement the ACID properties of a transaction.

A transaction contains one or more data modification language (DML) set commands that manipulate records in tables. A transactional lock protects all the set commands in the transaction, or rolls back all the changes. Various lock types are escalated on resources until the transaction is complete. Some types of locks are not compatible with each other. This ensures that transactions complete in its entirety by blocking other transactions that might conflict with the transaction integrity.

The risk is when transactions occur outside a transaction lock and risks breaking the database integrity. Take an example of making a purchase at an online store. When an order is placed payment is accepted, inventory is changed, and shipping is arranged. Payment, inventory change, and shipping is considered a transaction, each task considered a DML set command. If the inventory of an item was shown as having one unit in stock, and multiple customers attempted initiate a transaction to purchase the unit at the same time, a failure will result with one of the transactions. The set command that fails will cause the entire transaction to roll back any changes made during that transaction. Without transaction locks the payment table will contain more payments received than orders recorded, the integrity of the database is now shot.

A deadlock is a bad locking situation that could occur when not properly placing locks in a transaction. Deadlocks occur when a transaction has placed locks on records but is block from completing because of locks from another transaction that are also blocked from completing because of locks from the first transaction. This situation could have been avoided if we consistently placed locks on resources in the same order. In a deadlock situation, usually the database system decides on a victim allowing the other transaction to complete. The victim transaction rolls back any changes it started making during the transaction.

Locks need to be considered as part of an overall security strategy because it is a mechanism for protecting the integrity of data. Avoiding deadlock situations should be in the back of your mind when developing. How a database server handles deadlocks should also be considered when deciding on a database system. Deadlocks affect availability and database systems that do not handle deadlocks could be vulnerable to a denial of service by having vital records locked up blocking legitimate transactions from completing.

Race conditions take on many forms but could all be characterized by scheduling dependencies between multiple threads that are not properly synchronized that cause an undesirable timing of events. An example of how a race condition could have a devastating outcome on security would be say you have a system that is multi threaded, you have one thread handling the security and another thread handling the storage. If the thread handling the storage works faster than the thread handling the security, you could be faced with a scenario that stores sensitive information unsecured. Race conditions usually do not manifest themselves until it is too late. There are a number of programming constructs that could be used to control the synchronization of threads and they are semaphore, mutex, and critical sections. In the .NET world there are thread management classes that could be used to handle thread synchronization and scheduling.

Trusting user and parameter input can lead to disasters. Even if code is protected against buffer overflows, vulnerabilities such as semantic or SQL injection can still pose risks to confidentiality and integrity. Semantic injection is a well formed user input that when processed produces unexpected results. SQL injection is similar to semantic injection in that a well formed SQL syntax that is passed on to a database server for processing.

Here is an example of an SQL injection. A user is prompted for a branch and account number to view an account summary. The specification of the procedure expects an account number and a branch number. The procedure takes the two arguments and constructs a SQL command to execute against an SQL Server. The line that is used to construct the SQL command is as follows:

```
SQLCmd = “select * from tblaccounts where branch = “ + branch + “ and account = “ + account
```

If branch = *277* and account = *3455* then the value of SQLCmd will be:

*Select * from tblaccounts where branch = 277 and account = 3455*

The SQL command is a well formed select command. However the procedure has a flaw in that there is a potential that the user may enter more then an account number. What if branch = *277* and account = *3455; select * from tblaccounts* then the value of SQLCmd will be:

*Select * from tblaccounts where branch = 277 and account = 3455; select * from tblaccounts*

The above SQL select commands are also well formed. The injection of *;select * from tblaccounts* would cause the procedure to return all the accounts in the database table.

Exceptions are events which disrupts the normal flow of code. Exception handling is used for trapping exceptions that would otherwise crash or worse continue executing in an unstable state and produce unexpected results. Although unhandled exceptions are bad, exception handling is essential for keeping control of code flowing through the program. Depending on the severity of the exception you may want a mechanism to record the exception and perform a graceful shutdown.

Exceptions could be avoided by testing for conditions that could lead to an exception. For example before opening a file for reading, the presence of the file should be tested first. In Visual C++.NET, exception handling is done through the *try*, *catch*, and *finally* code blocks. When an exception occurs inside a *try* block such as a read command to a missing file, an exception is thrown and caught by the nearest *catch* block designed to catch that specific exception. The flow of code then continues with the *finally* regardless of an exception to do some cleanup before continuing with the rest of the code. If there are no catch blocks designed to catch the exception, the program risks crashing.

Complexity is a big threat to the security because it can easily mask logic errors, and could be used to hide logic bombs. Complex procedures make it more difficult to spot vulnerabilities. The definition and measurement of complexity can be different from programmer to programmer. I would measure complexity by the number of logic conditions and nesting depth in a procedure. Sometimes it is impossible to avoid writing complex procedures but by implementing coding and documenting conventions, and hiding or segregating complexity from other code, complex procedures would be less of a threat to security.

Software that processes user input may need various levels of control to match the separation of duties for users on the system. The privileges that management requires will be greater than those for users entering data. Management my request a higher granularity of control based on the levels of management or functional roles users play on the system. System with a low level of granularity may lack the controls to implement proper segregation of duties and therefore prevent abuse or misuse of the system. An

exclusive reliance on software controls is bad because you are placing all your security measures within the software. The risk is that if a software defect crashes or causes the software to become unstable, the confidentiality and integrity of the system or data could be lost. Software controls should be balanced with system privileges and native control mechanisms.

Privileges are the authorities that a system provides to users to complete a task. Generally code privileges should inherit the privileges of the user on the system. Excessive privileges are privileges that code offers to a user that they would not otherwise have on the system. The risk of excess privileges is that if the users is able to escape the controls of the application, data would be vulnerable to abuse, misuse, or corruption. If excessive privileges are required, the code should be kept to a minimum with controls in place to track usage and misuse.

Compiler intricacies are important to know because the code you develop may be optimized to an extent that it may not be implemented the way it was intended in the executable. Take the following three lines of pseudo code, compiling with the Visual C++.NET compiler with optimization turned on, the third line that scrubs the secret from memory would be omitted in the executable. Since *Memorybuffer* is no longer read from memory after the second line, the compiler ignores the memory scrubbing on the third line. The executable will resemble the code on the first and second lines only [11].

```
1: MemoryBuffer = Validate (Secret_User_Input)
2: SecretMemoryBuffer = Encrypt (MemoryBuffer)
3: MemoryBuffer = RandomData
```

A work around would be to turn off optimization or read the *Memorybuffer* variable after scrubbing it. The following couple of lines demonstrate how to read the *Memorybuffer.*

```
if MemoryBuffer = SecretMemoryBuffer then
        Remark do nothing
```

The above two lines are meaningless and may even be optimized out with future compiler patches or revisions. Compiler intricacies can vary from compiler build types, compiler patches, and from compiler to compiler. This underlines the importance of keeping compiler patches up to date. Build types such as the debug build contain debug information that is used to help the developer trace through code and contains source code comments.

Usually debug builds have optimization turned off and release builds remove debug info and source code comments from the executable with optimization turned on. Because the compiler treats code differently from build types it is important to test the release you plan on using in production. This underlines the importance of testing, peer review, as well as possibly considering an open design.

In addition to avoid the vulnerabilities we have just talked about we should follow the following design strategies into our software.

1) Fail safe defaults
2) Input and parameter validation
3) Encrypting secrets in memory and in storage
4) Scrubbing secrets in memory when finished
5) Placing security measures closest to data & minimizing scope
6) Least privilege
7) Loose coupling and high cohesion
8) Peer code review and open designs
9) Change management & version control
10) Accommodating change

Designing for fail safe is defensive programming. It involves validating user input and parameters, setting defaults, and handling exceptions to protect the confidentiality, integrity and availability of data and resources when a failure occurs or when garbage is processed. Generally this means that a safe state should always be maintained through the course of code execution. Variables should be initialized with safe defaults, input/ parameters must be validated, error handling should be in place to trap potential exceptions; and return parameters must conform to design specification. In the event of a failure resource locks must be removed and return an error code to the calling routine.

Input and parameter validation verifies that user input and parameters conform to the design specification for the procedure. Procedures with parameters that have little to no checking are tightly coupled with the calling routine. Tightly coupled procedures are dependent on the calling routine to provide proper and well formed parameters. Loosely coupled procedures do not assume the quality of parameters and will verify that they conform to specification. Any variances to specification should be corrected or the procedure should return an error code or throw an exception. The risk of not validating input data or verifying parameters is having the procedure run outside of specification.

Secrets in memory or on storage need to be protected. Encrypting secrets or wiping memory after you done with the secret protects it from ending up in a crash dump file or on a page file. The process of encrypting secretes should be done using a proven cryptographic primitive. Ideally, you should implement a double buffer, one for the plaintext secret and the other for the ciphertext. This is to protect against any possible race conditions that could use the secret before it is fully secured. Any secrets in memory should be scrubbed along with wiping secrets persisting on the disk before deleting.

Scrubbing memory requires changing its value to a safe default before you are done with the variable. The code required to scrub the memory will depend on the data type. The code snippet below demonstrates scrubbing a secret from an integer data type.

```
m_SecretRounds = 19630213;
```

```
ProcessSecret (m_SecretRounds);
m_SecretRounds = 0; // Scrub secret from memory using a safe default
```

The code sample below demonstrates an example of wiping a file before deleting it from the filesystem. The code is written for the .NET environment using Visual C++.NET.

```
bool WipeFile (String * inFilePath, int nWipes ){

      bool status = false;

      FileStream * sw;
      Byte Patterns[];
      __int64 fileLength=0;
      int patIndex =0;

      if (nWipes<1) nWipes =1;
      else if (nWipes>200) nWipes=200;

      try {
            // Open the file with exclusive access
            sw = new FileStream (inFilePath, FileMode::Open  ,
                              FileAccess::Write, FileShare::None);

            Patterns = new Byte[nWipes] ;

            // Create a pattern index
            for (patIndex = 0; patIndex < Patterns->Count ; patIndex++)
                  Patterns[patIndex] = patIndex;


            fileLength = sw->Length ;
            for (patIndex=0; patIndex < Patterns->Count ; patIndex++) {
                  sw->Seek (0,SeekOrigin::Begin );
                  for (int i =0; i <= fileLength ;i++)
                        sw->Write (Patterns,patIndex,1);
            }
            sw->Flush ();
            sw->Close ();

            // Safe to remove the file now
            File::Delete (inFilePath);
            status = true;
      }
      catch (Exception * e) {
            Console::Writeline (e->ToString());
      }
      return status;
}
```

Security and control measures should be placed closest to sensitive data either by implementing native access control mechanisms provided by the operating system or database server. The further the control measures are away from data, the more opportunity there is to circumvent the control. An exclusive reliance on software control

can potentially be bypassed altogether by accessing the sensitive data using another program without controls.

Restricting the accessibility scope of an object interface offers better control over sensitive methods, properties and attributes for the object. Control mechanisms within objects unless well defined could be potentially bypassed or escaped by using object inheritance and thus avoiding any authorization or authentication to access. [12] Control mechanisms provided by the operating system or database server have little to no changes of being escaped from.

Privileges to the data should be the minimal set required to perform a task. The risk with privileged code is that it may be vulnerable to luring attacks which is when a user or other procedure will less privileges makes use of a privileged procedure to gain access to methods or attributes they would not otherwise have. Code that must be privileged to perform some task must be kept as short as possible. Reducing the complexity and size of privileged code makes it easier to audit and track. An example would be a user who normally has read only access against a database could use a system with full access to make modifications. Or, a developer could implement a component that could be used to gain greater access to data.

Loosely coupled and high cohesion describe how procedures related to each other. Loosely coupled procedures are very independent with other procedures. High cohesion describes procedures that perform a single function. An example of a function with high cohesion would *CalculatePi()*. Poor cohesion would define a function *CalculatePiAndSin ().* A procedure that is loosely coupled and has high cohesion is more reliable and is usually easier to maintain and read.

Configuration management is a well defined process to control the integrity of a system changes during its lifetime. It allows all the stakeholders of the system and solution to know of upcoming proposed changes and evaluate the impact to their environment, priorities, and processes. From a business perspective change management controls development and integration costs, and since changes are known by all the stakeholders, changes are better integrated into the production environment.

There is a risk when changes to code are required, there is a potential that vulnerabilities might be introduced during development or rollout. Usually programmers are under pressure to complete changes as quickly as possible and have little time to analyze how a change may affect security. Change requirements are unavoidable but their effects could be limited by anticipating them. Changes requirements can be brought about by changes to the business climate, environment, policies, or new opportunities. During the initial design of a system, requirements and business rules that are susceptible to change should be identified so that they could be segregated from the rest of the code by placing the unstable procedures into a module or assembly. Layers of abstraction should be considered to abstract the stable code from the unstable code. For example, if the database environment is susceptible to change over the next year. Database access procedures should be placed in a separate assembly

and a layer of abstraction created to separate code specific for the database environmental with the business rules.

The list of programming pitfalls grows larger as new vulnerabilities are discovered. Staying ahead requires a staying on top of vulnerabilities and changes.

You could never be fully assured that your code is secure from any implementation flaws. Peer review helps uncover any glaring logic flaws and weaknesses. Open design also offers the same benefits but through public scrutiny. There are debates on if open design promotes security through public review or make it easier for attackers to find vulnerabilities to exploit.

.NET Environment Overview

.NET is a new initiative from Microsoft that changes the software development paradigm [6]. The new paradigm similar to Java focuses on security by creating an abstraction layer that sits between system resources and the program code. The difference between .NET and Java is that .NET also creates a language abstraction in that you could mix various languages that conform to the .NET language specification within an assembly.

The abstraction layer known as the common language runtime (CLR) provides a controlled environment and is responsible for security enforcement through:

1) Code verification
2) Role based security
3) Code access security
4) Security policy management
5) Isolated Storage

The CLR treats code as having varying degrees of trust in the system that is dictated by security policies and based on evidence that the CLR collects to determine if the code is trustworthy before executing it. Some of the technology behind the CLR is based on Microsoft's Authenticode Technology. It is designed to identify the publisher of the code to verify that no one has tampered with it.

Code strictly developed for the CLR using a .NET language is known as managed code. Both managed and traditional code forms an assembly which can be in the form of an EXE or DLL. Regardless of the extension of the file, code developed for the .NET environment compiles to Microsoft Intermediary Language (MSIL) by the compiler. The MSIL file needs to be further compiled to native code which is done by the CLR on the fly at runtime along with performing security checks.

.NET applications run in what is like a silo but called an *application domain.* Application domains are where the .NET applications gets initiated and controlled from. It is at this point that code gets loaded from an assembly and begins execution. The types of hosts

that can create an application domain are the IE browser, server host such as ASP.NET; and command shell host. The CLR does a just-in-time (JIT) compile on the managed code sections in the MSIL creating native code that is executed in the domain. Unmanaged code runs outside of the CLR and therefore can still pose a security risk even when it is part of an assembly.

During the JIT compile process of managed code, code verification occurs to ensure type safety, bounds checking which avoids buffer overflows; proper initialization of objects; and, stack frame corruption to avoid attacks that transfer execution to an arbitrary memory location. As you can see the verifications process adds a lot of security without having to program the constraints in. These constraints prevent most types of attacks that occur to code. Unfortunately, when it comes to unmanaged code, the CLR does not do any security checks or policy enforcements. A recent attack to ASP.NET [8] using a buffer overflow in an unmanaged section of code can attest to that.

Role based security allows a developer to build constraints into the application that modifies the flow of code based on the user's role on the system or in the enterprise. The roles can be taken from group membership in the domain or active directory.

Code access security prevents luring attacks when code with a lower set of privileges to call on code with a higher set of privileges and evoke methods that the original code couldn't otherwise. When invoking methods or constructing .NET objects, the CLR will do a stack walk to see if the caller along the call chain has the security privilege to instantiate the object or call on methods. Code access security can also restrict access to unmanaged code within the assembly.

Security policy management defines the set rules that are applied to code groups. The CLR analyses the evidence in the assembly to determine which code group the assembly falls under [7], then applies those rules to the assembly before executing the code. Examples of evidence that the CLR looks for in an assembly include publisher detail, digital signature, location of installation directory, and the type of host that created the application domain.

Isolated storage is a mechanism that allows .NET applications to use a Framework controlled file storage system isolated from the operating system's file system. [16] The isolated storage protects the file structure of the operating system without preventing applications requesting File IO that otherwise would not have the permissions or privileges with the file system.

Although the increased security built into the .NET environment does come with a slight performance penalty, the performance of managed code is very good. However, because a lot of work has been done to secure an environment, a poor security design could still leak sensitive data.

Protecting confidentiality and integrity with .NET

We will take a look at implementing data confidentiality and integrity using the .NET Framework's Cryptographic Service namespace using some code samples written in Managed Extensions for C++. The language semantics would be different from other .NET languages, but the interaction of the classes and method calls within the .NET Framework would be the same.

The components of the symmetric cipher are the algorithm, initialization vector, key, and stream. A discussion on the encryption key can not be made without taking about the effective key space, entropy, modes of operation; hashing, and the random number generator.

The .NET Framework base class library or BCL includes four symmetric encryption algorithms:

1) RC2
2) DES
3) 3DES
4) Rijndael

Each of the algorithms has had public peer review and have all had research papers published on the Internet. The security of each of the algorithms is located in the size and quality of the key. The key being is nothing more than a very large binary number to the encryption algorithm. The strength of an encryption is based on the algorithm and the effective key size.

Each of the symmetric algorithms has different performance and encryption strength properties that are best suited for specific scenarios.

To begin using the cryptographic services found in the .NET Framework we need to reference the namespace.

```
#using <system.dll>
using namespace System::Security::Cryptography ;
```

You may ask which of these algorithms is best. The answer as is that it all depends on what you define as best, your business objectives, performance requirements, and the value of your data over time and sensitivity of the information. Although each of these algorithms has had expert peer review and has been extensively tested over time, there is no guarantee of everlasting security. The selection of security mechanism will depend on your adversaries and what are their perceived technical abilities and financial resources.

Lets look at each of the algorithms and when and how to use them.

RC2 was developed in 1987 by Ron Rivest of RSA Securities. RC2 is a block cipher with a variable key and block length. This encryption is used in many commercial products.

To Create an RC2 encryption object from the class:

```
RC2CryptoServiceProvider * myRC2 = new RC2CryptoServiceProvider();
```

DES developed by IBM in 1974 and finally adopted by the National Institute of Standards and Technology in 1976. [1] DES uses a 64 bit key, but since the last byte is used for parity, the effect key strength is 56 bits. The DES key is known to be brute forced in a few days. The ideal application for this algorithm is one that needs to be backwards compatible with DES or that the information lifespan becomes worthless in a day. An example of data that would have a short lifespan would be Internet/intranet session tokens.

Here is an example of creating a DES object from the class.

```
DESCryptoServiceProvider * myDES = new DESCryptoServiceProvider();
```

The 3DES algorithm is also developed by IBM. [2] 3DES is basically three iterations of DES each iteration with a different key. The sequence would be:

$$E (D (E (m)))$$

That is: encrypt – decrypt – encrypt. With 3DES you could use either two keys or three keys. The .NET Framework uses a three key version. In a two key version you would have:

$$E_{k1} (D_{k2} (E_{k1} (m))).$$

In a three key version you would have all keys different:

$$E_{k1} (D_{k2} (E_{k3} (m))) .$$

The key space on 3DES is 168 bits (56 bits x 3) but because of a “Meet in the middle” attack the effect key size is 112 bits (56 bits x 2). 3DES is slower than DES since it has to go through the extra iterations.

To create a 3DES object from the class.

```
TripleDESCryptoServiceProvider * myTDes = new
TripleDESCryptoServiceProvider();
```

The Rijndael algorithm was developed by Joan Daemen and Vincent Rijmen as a candidate for the U.S. Advanced Encryption Standard (AES) which was selected on

October 2, 2000 [3]. It uses key sizes of 128, 192, or 256 bits and block lengths of 128, 192, or 256 bits. You can use any combination of key sizes and block lengths. The design goals of the algorithm is that it must resist all known attacks, must have design simplicity, code compactness and speed on a wide spectrum of platforms.

To create a RijndaelManaged object from the class:

```
RijndaelManaged * Rm = RijndaelManaged();
```

Of the block ciphers listed the Rijndael cipher is the fastest by far followed by RC2, DES, 3DES. Rijndael also has the largest key space of the algorithms. To put the size of the key space into perspective, if there was a machine fast enough that could brute force a DES key in one second, it would take 149 trillion years to brute force a 128 bit key for the Rijndael algorithm.

Streams

The symmetric encryption primitives in .NET are all block oriented but use data streams for the transformation. The transformations pass through the *cryptostream* that links a regular data stream object to the transformation. A block is taken from the data stream and is transformation based on the properties of algorithm. This approach is different from the traditional stream cipher that makes the transformation on bits as it travels through the stream. From the .NET Framework perspective, a stream represents an abstract sequence of bytes that flows through a device. Some examples of the derived stream objects that could be used are: File IO, Memory, Network IO, and Buffered IO.

The cryptostream class constructor is as follows:

```
Public: CryptoStream(
  Stream*,
  ICryptoTransform*,
  CryptoStreamMode );
```

The *ICryptoTransform* interface inherits from either a derived symmetric *decryptor*, or *encryptor* object. Use the encryption algorithm's *CreateDecrytor()* method to create a *decryptor* object used to do the decryption transformation, and the *CreateEncryptor()* method for the encryption transformation.

The *CryptoStreamMode* mode is used to set the direction of the stream and can be read or write depending on whether you are encrypting or decrypting. If you are encrypting you would set the *CryptoStreamMode* to write mode, otherwise set it for read mode for decrypting.

An example of creating a *cryptostream* for the RC2 encryption algorithm to encrypt a memory stream is as follows:

```
1: MemoryStream * myMemoryStream = new MemoryStream;
2: RC2CryptoServiceProvider * myRC2 = new RC2CryptoServiceProvider();
3: ICryptoTransform * myEncryptor = myRC2->CreateEncryptor ();
4: // Create an encryption stream that will use the memory stream for
5: // a storage area.
6: CryptoStream * myEncryptStream = new CryptoStream(myMemoryStream,
myEncryptor, CryptoStreamMode::Write  );
```

If we need to decrypt a memory stream we would change the lines 3 and 6 to read:

```
3: ICryptoTransform * myDecryptor = myRC2->CreateDecryptor ();
6: CryptoStream * myDecryptStream = new
CryptoStream(myEncryptedMemoryStream, myDecryptor,
CryptoStreamMode::Read );
```

.NET Symmetric Encryption Algorithm Properties

Property	RC2	DES	3DES	Rijndael
Default mode	CBC	CBC	CBC	CBC
Allowed modes	CBC, CFB, ECB, OFB	CBC, CFB, ECB, OFB	CBC, CFB, ECB, OFB	CBC, CFB, ECB, OFB
Unsupported Modes	CTS	CTS	CTS	CTS
.NET Class	RC2CryptoServiceProvider	DESCryptoServiceProvider	TripleDESCryptoServiceProvider	RijndaelManaged
Legal key sizes	40 ~ 128 (8 bit increments)	64 bits	128, 192	128, 192, 256
Default key size (bits)	128	64	192	256
Legal block size (bits)	64	64	64	128, 192 256
Default block (bits)	64	64	64	128
Feedback (bits)	8	8	8	128
Effective bit size (bits)		56	112 (Theoretical)	

Modes of Operation:

Since block ciphers encrypt a block at a time, two plaintext blocks that are identical would results in the cipher text blocks also being identical. This pattern could be used to recover the keys. To avoid this from occurring the previous cipher text block is chained back into the encryption process modifying the next cipher block. This continues until the entire plaintext is encrypted. There are different chaining modes that could be use. The acronyms are CBC, ECB, CFB, CTS, and OFB.

```
myRC2->Mode = CipherMode::CFB; //Sets mode to CFB
```

The cipher block chaining or (CBC) is the default mode for the encryption algorithms included with the .NET Framework. It is also one of the most secure. It takes the previous ciphertext block and does an XOR operation with the current plaintext block before it is encrypted to produce the next ciphertext block. Initially the initialization vector is XOR with the first plaintext block before it is encrypted. If the plaintext always begins the same way (Dear Sir:) and the initialization vector never changes, the

beginning of the ciphertext block will also always be the same. This is why the IV should change from session to session.

The electronic code book or (ECB) encrypts each block independent of the previous block. This creates a one to one relationship the plaintext and the ciphertext. If the is duplicate blocks in the plaintext there will be duplicate ciphertext blocks. This independence of the previous block makes this the highest performance mode and also the weakest mode of operation in terms of data security. The plaintext has to be larger then the block size.

The cipher feedback (CFB) is similar to CBC except that it begins encryption with a single byte rather than the entire block. This mode is ideal for data streaming. If there is an error in the encryption of the byte the remainder of the plaintext block will be corrupted.

The ciphertext stealing (CTS) produces ciphertext that is the same size as the plaintext where the plaintext is larger than the block size. If the plaintext is smaller then the block size padding is added to the message before it is encrypted. CTS works similarly to the CBC mode until the second last block than the last block and second last block are XORed with each other to produce the final encrypted block. The CTS mode is not supported by any of the symmetric encryption algorithms currently shipped with the .NET Framework BCL. It is included to support new symmetric algorithms that might derive from the *SymmetricAlgorithm* class at a later time.

The output feedback (OFB) this is similar to the CFB except that if an error occurred in the encryption the remainder of the cipher text will be corrupted.

Legal Key Sizes

The legal key sizes are the key spaces that the algorithm supports. Keys that do not match any of the key sizes will throw an exception.

Legal block size

The block size is the number of bits of data that is encrypted at a time. The legal block size lists your block size options. If the amount being encrypted is less than the block size the plaintext is padded. The default block is set to the largest legal block size.

Initialization Vector

The initialization vector (IV) is a random sequence of bytes pre-appended to the plaintext before the initial block is encrypted. The IV plays a big role by reducing the chances of successfully factoring the key using a chosen plaintext attack. The IV does not need to be secret but should vary from session to session. It is important to note that although I believe that the security should be dependent only on the quality of key

alone. The U.S. Federal Government states that with regards to the government's usage, the IV should also be encrypted if the data encryption uses the CBC encryption mode and you need to transmit the IV over an unsecured channel. [5] I can't argue that encrypting the IV makes the makes the data encryption more secure. However, the added complexity of encrypting an IV before transmitting it may not be worth the effort when the data is already encrypted and the key is secure. The same IV is required to retrieve the plaintext from the ciphertext.

To initially produce the random sequence for an *IV* you would make a call to the *GenerateIV*() method of the encryption object. The *IV* property will contain the generated value that we need to store to be used later to decrypt the message.

```
RijndaelManaged * Rm = new RijndaelManaged();
Rm->GenerateIV();
```

If the *IV* is retrieved from the encryption stream or some other communication channel you would set the *IV* property directly with the IV value.

```
Rm->IV = myIV; // myIV is a Byte array
```

Effective Key Space and Entropy

The effective key space is one of the determining factors of the encryption strength. The difference between effective key space and key space is that the effective key space represents the maximum work effort to brute force recover the keys. The key space on DES is 64 bits; however, since eight bits is used for parity, the maximum work effort to recover the key is based on an effective key space of 56 bits. Regardless of effective key space, if the method to generate keys is predictable, which means it has little entropy, recovering the keys could be relatively easy using statistical analysis.

Entropy is used to describe the amount of disorder in a sequence or system. Higher entropy has greater disorder. A 128 bit key may be equivalent to 30 bits of entropy if we base the key on a 20 character password or phrase entered by the user. Entropy is the amount of randomness in the bits. In this case the effective key size is 30 bits even though the key space is 128 bits.

The problem with relying on a user to generate a good pass phase is that the more difficult the password/phrase is, the harder it is to remember it. The passwords that we could remember would be too small to be considered safe.

If using a Standard English passphrase with each letter taking a full byte, there would be 1.3 bits of entropy per byte. This is taken from the fact that in a Standard English phrase there would be a statistically higher occurrence of certain (e,r,s,t) letters than others. [18] A passphrase would have to be 85 characters long to have 110 bits of entropy, and you simply couldn't use 110 bits without first distilling it to match the

encryption key requirements. To produce enough entropy for a hash function such as MD5 to consider random you would need a passphrase at least 99 characters long.

(8 bits to a byte / 1.3 entropy bits to a byte) * (128 bits in MD5 / 8 bits to a byte) = 98.5 bytes

If each letter had a statistically equal occurrence in a password (any letter occurs as often as any other letter) the amount of entropy per letter would be 4.7 bits per byte. The increased entropy would require a little as 27 random letters to hash by MD5 to produce a good random key.

log (26) / log (2) = 4.7 bit of entropy with random letter occurrences

(8 bits to a byte / 4.7 entropy bits to a byte) * (128 bits in MD5 / 8 bits to a byte) = 27 bytes

One of the problems with hashing a password or passphrase as a key is that we have to assume that whoever attacking your software also has a copy of the source code and knows the hashing algorithm used to generate the key. With this knowledge an attacker can easily download a very large dictionary with common English phrases and hash each entry to try to recover your key.

The selection of a hashing function to use for distilling entropy for encryption will depend on the encryption algorithm's key space requirement. For RC2 which has a default key space of 128 bits you would use MD5 to create the message digest of the entropy pool for the key. Your option of encryption algorithms becomes limited to ones that have a key space of 128, 160, 256, 384, and 512 bits. These key spaces are the available key spaces of the hashing functions that are available with the .NET Framework. The preferable way for generating keys would be to use a true random number generator.

Greater the effective key space makes a brute force attack less feasible. The effective key space also depends largely on the encryption algorithm being used and the amount of entropy found in your keys. For private key encryption, an effective key size of 56 bits is considered to be weak; similarly 512 bit keys are weak for public key encryption systems. For each bit that the key increases by there is a doubling of work effort required to brute force the key.

User Authentication Routines

It is preferable to rely on the .NET Framework role based authentication model to supply user authentication and authorization. The .NET Framework BCL contains a number of classes that support role based security.

.NET Cryptographic Hashing Algorithms

Hashing algorithms are generally used to assure data integrity by producing a unique numerical message digest or fingerprint that represents the data being hashed. Hashing takes an arbitrary amount of data and produces a message digest that is fixed length. Hashing works one way because you can't reproduce the data given the message digest and it is computational impossible to produce two documents that produce the same digest. This type of hashing is known as *Message Detection Code* (MDC).

There are different hashing algorithms that produce varying lengths of the message digest. Greater the length, the less likely there would be any collisions with two documents producing a similar message digest. Although MDC primitives can be used to detect changes to data, if the message digest is sent along with the data, both pieces of information could be intercepted and altered before being sent along. A solution to this is to use a keyed hash primitive.

The .NET Framework Cryptographic Namespace contains a number of MDC primitives with varying hash sizes.

1) MD5
2) SHA1
3) SHA256
4) SHA384
5) SHA512

The MD5 algorithm was developed by Rivest to succeed his previous version MD4. MD5 is a little slower than the previous version but more secure. The MD5 algorithm produces a hash of 128 bits and the SHA1 produces hashes of 160 bits. So it is less likely that there would be a collision with SHA1 than with MD5. There are also variations of SHA1 which create message digests of the following bit sizes 256, 384, and 512.

Here is a code sample written in Visual C++.NET that shows an example of creating a message digest of a unicode message using SHA1 outputting the digest to console.

```
// This is the main project file for VC++ application project
// generated using an Application Wizard.

#include "stdafx.h"
#using <system.dll>
#using <mscorlib.dll>
#include <tchar.h>

using namespace System;
using namespace System::Text ;
using namespace System::Diagnostics ;
using namespace System::Security::Cryptography ;

// This is the entry point for this application
int _tmain(void)
{
	String * myMessage ;
	myMessage = S"Bill Ferreira";

	SHA1Managed * mySHA1 = new SHA1Managed();
```

```
        UnicodeEncoding * myUE = new UnicodeEncoding();

        try {
                Byte myMessageBytes[] = myUE->GetBytes(myMessage);

                // Generate a 160 bit hash
                Byte myHash[] = mySHA1->ComputeHash (myMessageBytes);
                String * myHashResult = BitConverter::ToString (myHash);

                // Remove the "-" from the hash
                Trace::WriteLine (myHashResult->Replace ("-",""));
        }
        catch (Exception * e) {
                Trace::WriteLine (e->ToString());
        }

    return 0;
}
```

To hash a file, we open a filestream on the file and pass the handle of the stream to the *ComputeHash()* method. Here is an example of creating a message digest of a file.

```
// This is the main project file for VC++ application project
// generated using an Application Wizard.

#include "stdafx.h"
#using <system.dll>
#using <mscorlib.dll>
#include <tchar.h>

using namespace System;
using namespace System::Text ;
using namespace System::Diagnostics ;
using namespace System::Security::Cryptography ;

// This is the entry point for this application
int _tmain(void)
{

        FileStream * sw;
        Byte calculatedhash[];

        SHA1Managed * mySHA1 = new SHA1Managed();

        calculatedhash = new Byte[mySHA1->HashSize / 8];

        try {
                sw = new FileStream ("c:\\myfile.exe", FileMode::Open,
                                FileAccess::Read, FileShare::None);
                calculatedhash = mySHA1->ComputeHash (sw);
        }
        catch (Exception * e) {
                Trace::WriteLine (e->ToString());
        }
        __finally {
                sw->Close ();
        }

        return 0;
}
```

Keyed hash primitives are hashes that produce a message digest based on the data and a secret key. Keyed hash algorithms are known as *Message Authentication Codes* (MAC). MAC serves two purposes: assure data integrity and authentication. There are two types of MAC, ones based on hash algorithms such as SHA1 and ones based on encryption algorithms such as TripleDES. The .NET Framework BCL includes both types of MAC algorithms and both are derive from the abstract *KeyedHashAlgorithm* class, the *HMACSHA1* is based on the SHA1 hash algorithm; and the *MACTripleDES* class is based on the TripleDES algorithm.

The main difference between both key hash algorithms is the restriction on the key and the size of the message digest. The HMACSH1 takes any size key and produces a 20 byte message digest. The *MACTripleDES* is restricted to key sizes of 8, 16, or 24 bytes and produces an 8 byte message digest.

The code below is an example of creating a message digest using the key hashed algorithm *HMACSHA1*. The key that I used for the code comes from the cryptographic random number generator. To reproduce the same message digest value on the data you would need to use the same key.

```
// This is the main project file for VC++ application project
// generated using an Application Wizard.

#include "stdafx.h"
#using <system.dll>
#using <mscorlib.dll>
#include <tchar.h>

using namespace System;
using namespace System::Text ;
using namespace System::Diagnostics ;
using namespace System::Security::Cryptography ;

// This is the entry point for this application
int _tmain(void)
{
	String * myMessage ;
	myMessage = S"Bill Ferreira";

	HMACSHA1 * myhmac = new HMACSHA1 ();
	RNGCryptoServiceProvider * rng = new RNGCryptoServiceProvider;

	Byte  rndhashkey[] = new Byte[10]; // allocate a 10 byte array.

	rng->GetNonZeroBytes (rndhashkey); // Array gets filled with random numbers.

	myhmac->Key = rndhashkey;  // sets the hash key

	UnicodeEncoding * myUE = new UnicodeEncoding();

	try {
		Byte myMessageBytes[] = myUE->GetBytes(myMessage);

		// Generate a 160 bit hash
		Byte myHash[] = myhmac->ComputeHash (myMessageBytes);
```

```
            String * myHashResult = BitConverter::ToString (myHash);

            // Remove the "-" from the hash
            Trace::WriteLine (myHashResult->Replace ("-",""));
        }
        catch (Exception * e) {
            Trace::WriteLine (e->ToString());
        }

    return 0;
}
```

At this point we covered enough to begin encrypting and decrypting secrets. Let's take a look at the following source code that encrypts and decrypts a secret in memory. The example uses the 128 bit RC2 encryption algorithm, and the 128 bit MD5 hash to convert the pass phrase into a usable encryption key for RC2. Since we are encrypting and decrypting within the same module there is no need to store the IV. If we were to transmit the encrypted message we would also need to transmit the IV with the message.

```
// This is the main project file for VC++ application project
// generated using an Application Wizard.

#include "stdafx.h"
#using <system.dll>
#using <mscorlib.dll>
#include <tchar.h>

using namespace System;
using namespace System::Diagnostics ;
using namespace System::IO ;
using namespace System::Security::Cryptography ;
using namespace System::Text ;

// This is the entry point for this application
int  tmain(void)
{

        MD5CryptoServiceProvider * myMD5 = new MD5CryptoServiceProvider();
        RC2CryptoServiceProvider * myRC2 = new RC2CryptoServiceProvider();

        MemoryStream * myMemoryStream = new MemoryStream;

        UnicodeEncoding * myUnicodeEncoding = new UnicodeEncoding();

        Encoding * myEncodingMethod = new UnicodeEncoding();

        String * myPassphrase = S"Bill Ferreira can not think of a good passphrase";
        String * myMessage = S"This message will be encrypted!";

        try {

                Byte myPassphraseBytes[] = myUnicodeEncoding->GetBytes( myPassphrase );
                Byte myMessageBytes[] = myUnicodeEncoding->GetBytes( myMessage );

                // Create a 128 bit Hash of the passphrase to be used as a key
                Byte myHash[] = myMD5->ComputeHash ( myPassphraseBytes );

                // Use the initialization vector and passphrase for both
                // encrypting and decrypting functions
                myRC2->GenerateIV ();
                myRC2->Key = myHash ;

                ICryptoTransform * myDecryptor = myRC2->CreateDecryptor ();
                ICryptoTransform * myEncryptor = myRC2->CreateEncryptor ();


                // Create an encryption stream that will use the memory stream for
                // a storage area.
                CryptoStream * myEncryptStream = new CryptoStream(myMemoryStream, myEncryptor,
```

```
                                                CryptoStreamMode::Write  );

                myEncryptStream->Write (myMessageBytes,0,myMessageBytes->Length );
                myEncryptStream->FlushFinalBlock ();

                // The memory stream now holds an encrypted message.
                Byte myEncryptedMessage[] = myMemoryStream->ToArray ();

                Trace::WriteLine (Convert::ToBase64String ( myEncryptedMessage ));

                myEncryptStream->Close ();
                myMemoryStream->Close ();

                ////////////////////////////////////////
                // Decrypt an encrypted memory stream
                ////////////////////////////////////////
                MemoryStream * myEncryptedMemoryStream = new MemoryStream;
                myEncryptedMemoryStream->Write (myEncryptedMessage,0,myEncryptedMessage->Length );

                // Reset stream pointer to the beginning of the memory stream
                myEncryptedMemoryStream->Seek (0,SeekOrigin::Begin );

                // Create a decryption stream that will read from the
                // memory stream to decrypt.
                CryptoStream * myDecryptStream = new CryptoStream(myEncryptedMemoryStream,
                                myDecryptor, CryptoStreamMode::Read );

                // The stream reader will pull data through the myDecryptStream CryptoStream
                // using the Unicoding character set.
                StreamReader * sr = new StreamReader ( myDecryptStream, myEncodingMethod, false );

                Trace::WriteLine ( sr->ReadToEnd () );

                myDecryptStream->Close ();
                myEncryptedMemoryStream->Close ();
        }
        catch (Exception * e) {
                Trace::WriteLine ( e );
        }
    return 0;
}
```

.NET Cryptographic Pseudo Random Number Generator

Generating random numbers is not as easy as it sounds. Random numbers are an effect a series of numbers that arbitrary, unknowable, and unpredictable. [14] Every number has an equal probability of coming up. Because of the characteristics of random numbers using a predictable or deterministic machine such as a computer as a source of random data is not a preferable way for generating cryptographic key material. Random numbers generated by a computer is done through a pseudorandom number generators (PRNG) use a mathematical formula and an initial seed value.

The preferable way is to use a non deterministic source which produces randomness outside the control of humans. Examples of non deterministic sources would be sampling atmospheric noise from radio or measuring radioactive decay. This type of source would produce genuine random numbers.

Cryptographic PRNG algorithms use cryptographic hash or encryption functions to produce the random data. A seed used to initialize the initialization vector and key. The problem with using a mathematical algorithm is that if you use the same seed value you will be able to reproduce that same series of numbers. Sampling the computer environment such as typing rates and mouse movements could potentially be tampered with by programs that take over control of the keyboard or mouse buffer.

The .NET Cryptographic Services includes the PRNG class *RNGCryptoServiceProvider* that you could use for your keying material. Generating random bytes is as easy as calling the method *GetBytes()* passing in an array. The *GetNonZeroBytes()* method could be called instead to return a random sequence without a zero value. *GetNonZeroBytes()* is preferable since a byte value of zero represents eight bits. This example shows generating 16 bytes of random data.

```
RNGCryptoServiceProvider * prng = new RNGCryptoServiceProvider;
Byte  rndbytes[] = new Byte[16]; // 16 bytes or 128 bits
prng->GetNonZeroBytes (rndbytes);// Byte array filled with random data.
```

Conclusion

As you have seen, there are a few things that you need to consider when developing secure solutions. The .NET environment helps with a lot of the security plumbing but you still have to design and code the solution. Regardless of how secure you develop your software or solution you will always be susceptible to a clever attack.

If there was just one thing that you take away from this article that should be: What ever you design, develop, and implement, regardless of security measures, your code will be reverse engineered and your security mechanisms will be analyzed. Because of this, secrecy should be fully dependant on a proven encryption algorithm on the size and quality of the encryption key.

References:

1. **DES Encryption** http://www.tropsoft.com/strongenc/des.htm
2. *RSA Security,* **"What is Triple DES?"** URL: http://www.rsasecurity.com/rsalabs/faq/3-2-6.html
3. *Cryptomathic,* "**Setting the standard – The selection of AES"** URL: http://www.cryptomathic.com/news/aesstandard.html
4. *Sans Institute,* "**Information Security Policy**" http://www.sans.org/newlook/resources/policies/Information_Sensitivity_Policy.pdf
5. *National Institute of Standards and Technology,* **"Key Management Using ANSI X9.17"** URL: http://csrc.nist.gov/publications/fips/fips171/fips171.txt
6. *Fawcette .NET Magazine*, **"Reduce Security Vulnerabilities"** http://www.fawcette.com/dotnetmag/2002_04/magazine/features/jschramm/
7. *Microsoft Corporation,* **"Security in the Microsoft .NET Framework"** URL: http://www.microsoft.com/technet/treeview/default.asp?url=/technet/itsolutions/net/evaluate/fsnetsec.asp
8. *Microsoft Corporation***, "Unchecked Buffer in ASP.NET Worker Process"** URL: http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/bulletin/MS02-026.asp
9. *Keyghost Ltd.*, **Keypress snooping hardware** http://www.keyghost.com/
10. *National Institute of Standards and Technology***, "Key Management Guidelines"** URL: http://csrc.nist.gov/encryption/kms/guideline-1.pdf
11. *Michael Howard, Microsoft Corporation* "**Scrubbing secrets in Memory**" URL http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure10102002.asp
12. *Microsoft Corporation,* **"Secure Coding Guidelines for the .NET Framework**" URL: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/seccodeguide.asp
13. Internet Development Associates **"Software Testing and Methods"**, URL: http://www.ideva.com/Mits/MITs_02.htm
14. Jon Callas, **"Using and Creating Cryptographic-Quality Random Numbers**" URL: http://www.merrymeet.com/jon/usingrandom.html
15. *Microsoft Corporation,* "**Microsoft.NET Framwork Security Overview**" URL: http://msdn.microsoft.com/vstudio/techinfo/articles/developerproductivity/frameworksec.asp
16. *Foundstone Inc. & CORE Security Technologies,* **"Security in the .NET Framework"** URL: http://www.microsoft.com/technet/treeview/default.asp?url=/technet/itsolutions/net/evaluate/fsnetsec.asp
17. *Mathew Schwartz, Computerworld Inc.,* "**DVD Encryption hacked"** URL http://www.cnn.com/TECH/computing/9911/05/dvd.hack.idg/
18. *Bruce Schneier*, "**Secrets & Lies Digital Security In a Networked World"** Published by John Wiley & Sons, Inc. ISBN: 0471253111 http://www.amazon.com/exec/obidos/ASIN/0471253111/qid=1028575527/sr=8-3/ref=sr_8_3/103-1466955-0315020