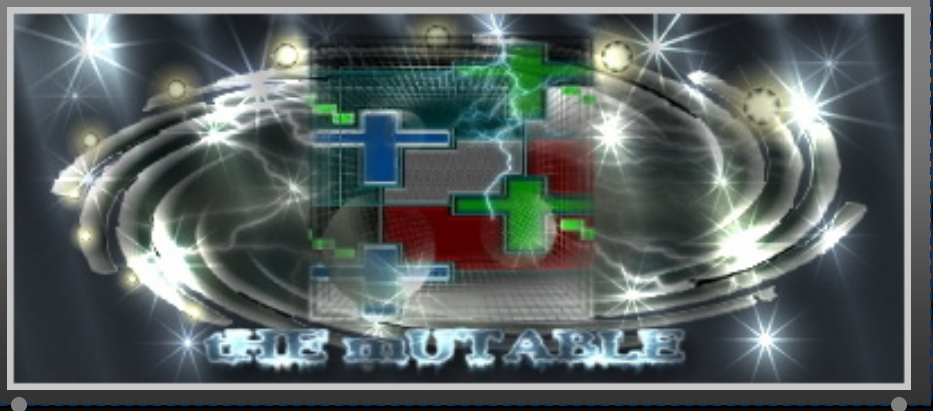
Reverse Code Engineering [RCE]
Emphasizing On
Breaking Software Protection

5265766572736 96E6720746865204D696E64206F6620476F64

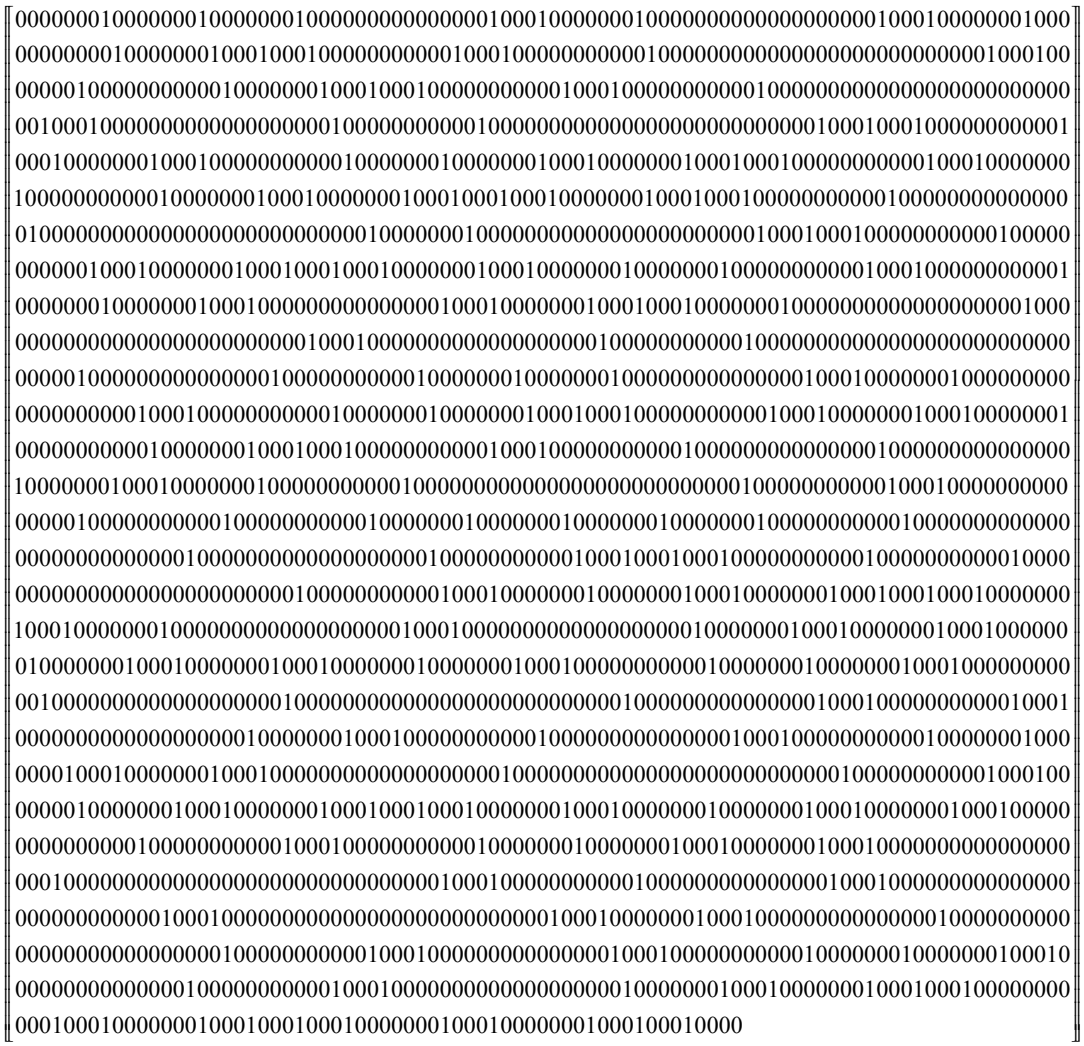
Exclusive Edition 2006

BY



June 29, 2006

BEGIN tHE mUTABLE ENCODING



END tHE mUTABLE ENCODING

History

Version A±0.1A ∃ E.E.

Files Included

Complete Source Code...

I Don't Know

Life Is More Than Human Evolution

#

$\oint .. I .. + A \pm 0.1A = .. \ddot{I} .. \exists E . E .$

S. & C.

You Are Welcome For Any Suggestions, Comments

I hereby declare that I am the sole author of this document.

tHE mUTABLE

I further authorize the ARTeam to reproduce, distribute the document by photocopying or by other means, in total or in part, at the request of other individuals or group's for the purpose of research.

einsteinzero@hotmail.com
cheviva2000@hotmail.com
http://www.themutable.com

tHE mUTABLE

ABSTRACT

Reverse Code Engineering with emphasizing on breaking software protection. For many specialists in this field especially in the field of malware reversing, it's a must to understand what all is about by "analyzing the subject system to identify the system's components and their interrelationships and to create representations of the system in another form or at a higher level of abstraction "(IEEE 1990) in order to take the control over the malware invaders and protect millions of computers around the world from being infected as quick as possible. For breaking protections protocols the rationale is to get the knowledge for the unknown because it's enjoyable and truly truth to reconstruct 0's & 1's for another purpose without knowing the original state (source code) of construction.

The objective is to unhide the castle of secrets behind the beauty of how things works and to present a newly customized approach for better protection against illegal reversing concerning commercial software applications. The methods used to perform this task, that is, analytical, numerical, and experimental.

The study shows the weakness of the Operating System in handling the binaries connections system call, protections in a commercial applications and how it's fully reversed to its newborn phase, which impose a great threat on the customers and companies affecting companies' liability. It reveals the integrity in reversing software executable files and how to break software's protections.

Most of the materials presented are newly designed and implemented for this purpose.

Keywords: Reverse Engineering, Breaking Protections, Algorithm, Packer, UnPacker, Patching, Serial

ACKNOWLEDGEMENTS

I'd like to thank Lena151 for his very well done newbie's to advance RCE Flash Tutorials, and last but certainly not least every member in the ARTeam with special thanks to Goppit & Shub – Nigurrath: Thanks a lot for your proofreading my book, your corrections and suggestions is deeply appreciated and for your generosity to take time out of a busy schedule to answer my questions.

Sincere thanks go out to the true professionals, the RCE's AR The incredibly hard working team. And I would like to thank everyone for his contributions concerning RCE and many aspects of advanced techniques used in the scene.

tHE mUTABLE

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

1.1 Reverse Engineering?

Reverse Engineering is the ultimate apex human mind could ever achieve in the embarking creation of the process and reengineered it back to its original composition through multi sophisticated Neuron-Digital combinational analysis.

Opening up a program's "box," and looking inside, it would be a very interesting project to tackle. Reverse Code Engineering it's one of the most important fields in the development industry concerning security, piracy software, malicious software, reversing cryptographic algorithms. It's all about translating the Low Level Programming Language to High Level Programming Language which requires understanding everything as bits 0's and 1's to reveal the hidden secrets behind the interface; Operating system, Assembly Language, High Level Language, Hardware Development, Human minds,… it's a must to have all of these requirements and much more for better understanding of how things works.

It's a very long chain of connections from top till down where there are multi-branches for each cross-reference from one procedure to another in order for the instructions to flow cumulatively based on the previous block of procedures and functions.

1.2 0's & 1's Ubiquity: Philosophy

Binary! It's all about binary 0's and 1's, wherever you go whatever you think you will fall into the same extreme point as a looser or a winner, and there is nothing in the middle or the system will collapse over itself because there is only one mode at a time. So in order to understand how things work one must accept the fact of the experiment being done.

Everything is a bit with variable time being the ultimate ruler on the behavior of the inter-modular call between the procedures and instructions. Locating a reference and follow it to the source step by step, it's an enjoyable very long journey to tackle, but had better to allocate something dynamically tracing it line by line and grasp the whole pictures of how registers and numbers follow your control till the desired conclusion. Not always things that easy, what if you crashed out of the game and there is no return. No, for every procedure there is a starting point and ending one but be careful sometimes you will never get out of this loop until you setting another breakpoint outside this loop and run freely for another block of 0's and 1's where your imagination start to degenerate after the long trip of fighting. May be there is something monitors your movements and record a log file for all of your actions without noticing anything at all and that's the pleasure. Don't think that you know what you know is what you don't

know; analyzing your mysterious actions in parallel with the victim being hijacked might reveal for you secrets behind the outer shell. Sometimes it's easy, sometimes it's not, it depends on how much experience you have to toggle things on and off and make everything reversible to its raw state. It's truly truth to have something well done and share it with the others.

Binary can represent anything, but you can't make things work the same as each other. Executable code and data are, at the lowest level, the exact same thing a collection of 0'and 1's, you may want to try to run data as code, but most likely this will lead to a crash. Why, because every object is being structured for it's own purpose, and that's what makes things more than just binary. And for every object (Picture, Movie, Executable file …) there is something that understand this structure and will interpreted in the proper way.

1.3 Protection Classifications

Checking *authenticity* is the "heart" of the overwhelming majority of protection mechanisms. In all cases, we have to make sure that the person working with our program is who he or she claims to be, and that this person is authorized to work with the program. The word "person" might mean not only a user, but the user's computer or the medium that stores a licensed copy of the program. Thus, all protection mechanisms can be divided into two main categories:

- Protection based on *knowledge* (of a password, serial number, etc.)
- Protection based on *possession* (of a key disc, documentation, etc.)

Knowledge-based protection is useless if a legitimate owner isn't interested in keeping the secret. An owner can give the password (and/or serial number) to whomever he or she likes, and thus anyone can use a program with such protection on his or her computer.

Naturally, nobody is barging in on users in their homes, and nobody is even considering it (yet) — your house is still your castle. Besides, what can you get from a domestic user? A wide distribution of products is good for manufacturers, and who can distribute better than pirates? Even in that case, serial numbers aren't superfluous— unregistered users cannot use technical support, which may push them to purchase legal versions.

Such protection is ideal for giant corporations, but it isn't suitable for small groups of programmers or individual developers, especially if they earn their bread by writing highly specialized programs for a limited market (say, star spectra analysis, or modeling nuclear reactions). Since they cannot apply sufficient pressure, it's unreal for them to ask users to check their licenses, and it's hardly possible to "beat" the payment out of illegal users. All that can be done is through threat and eloquence.

In this case, protection based on the possession of some unique subject that is extremely difficult to copy, or impossible to copy in general (the ideal case), is more appropriate. The first of this kind were key floppies with information written on them in such a manner that copying the floppy disk was impossible. The simplest way (but not the best) to prepare such a floppy was to gently damage the disk with a nail (an awl, a penknife), and then, having determined the sector in which the defect was located (by writing and reading any test information — up until a certain point, reading will proceed normally, followed by "garbage"), register it in the program. Then, each time the program started, it checked whether the defect was located in the same place or not. When floppy disks became less popular, the same technique was used with compact discs. The more affluent cripple their discs with a laser, while ordinary folk still use an awl or nail.

Other possession-based protection mechanisms frequently modify the subject of possession, limiting the number of program starts or the duration of its use. Such a mechanism is often used in installers. So as to not irritate users, the key is only requested once, when the program is installed, and it's possible to work without the key. If the number of installations is limited, the damage arising from unauthorized installation of one copy on several computers can be slight.

The problem is that all of this deprives a legal user of his or her rights. Who wants to limit the number of installations? (Some people reinstall the operating system and software each month or even several times a day). In addition, key discs are not recognized by all types of drives, and are frequently "invisible" devices on the network. If the protection mechanism accesses the equipment directly, bypassing drivers in order to thwart hackers' attacks more effectively, such a program definitely won't run under Windows NT/2000, and will probably fail under Windows 9x. (This is, of course, if it wasn't designed appropriately beforehand. But such a case is even worse, since protection executing with the highest privileges can cause considerable damage to the system.) Apart from that, the key item can be lost, stolen, or just stop working correctly. (Floppy disks are inclined to demagnetize and develop bad clusters, CDs can get scratched, and electronic keys can "burn out".)

And that’s the summary for protection classifications as reported in figure 1.1

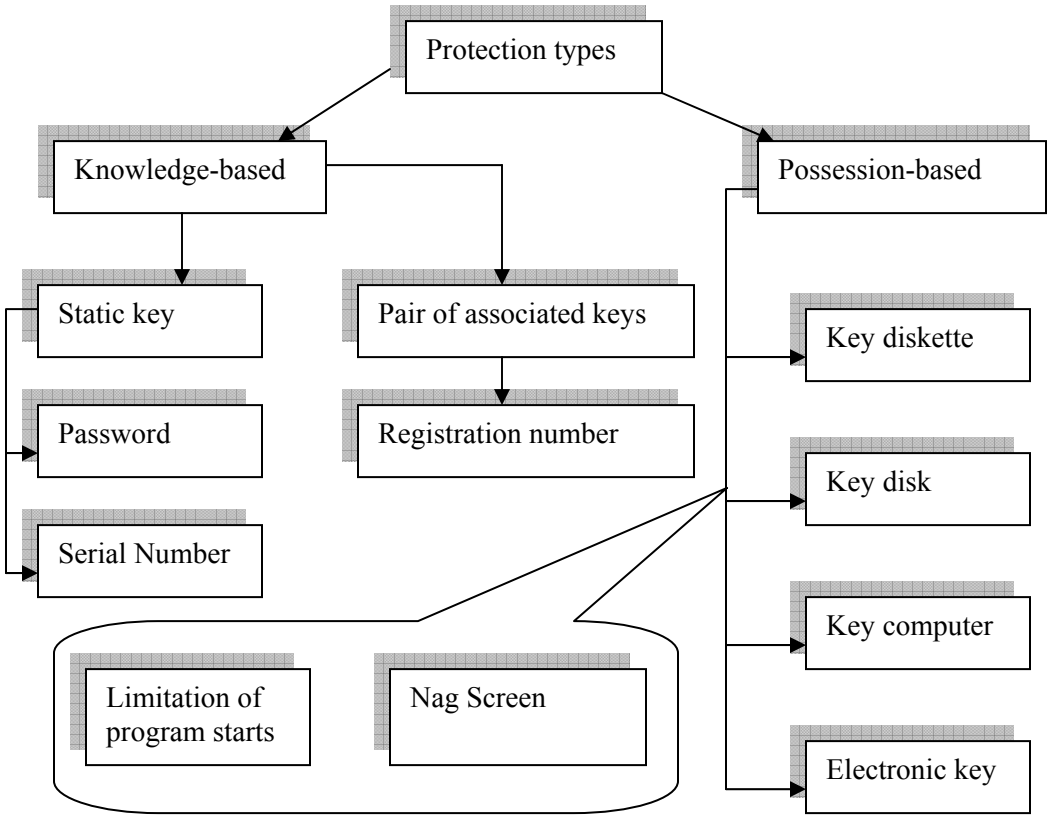


Figure 1.1 The main types of protection

Naturally, these considerations concern the effectiveness of keys in thwarting hackers, and not the concept of keys in general. End users are none the better for this! If protection causes inconveniences, users would rather visit the nearest pirate and buy illegal software. Speeches on morals, ethics, respectability, and so on won't have any effect. Shame on you, developers! Why make users' lives even more complicated? Users are human beings too!

That said, protections based on registration numbers have been gaining popularity: Once run for the first time, the program binds itself to the computer, turns on a "counter", and sometimes blocks certain functionalities. To make the program fully functional, you have to enter a password from the developer in exchange for monetary compensation. To prevent pirate copying, the password is often a derivative of key parameters of the user's computer (or a derivative of their user name, in an elementary case).

1.4 Protection Determination

If protection is based on the assumption that its code won't be investigated and/or changed, it's poor protection. Concealing the source code isn't an insurmountable obstacle to studying and modifying the application. Modern reverse engineering

techniques automatically recognize library functions, local variables, stack arguments, data types, branches, loops, etc. And, in the near future, disassemblers will probably be able to generate code similar in appearance to that of high-level languages.

But, even today, analyzing machine code isn't so complex as to stop hackers for long. The overwhelming number of constant cracks is the best testament to this. Ideally, knowing the protection algorithm shouldn't influence the protection's strength, but this is not always possible to achieve. For example, if a server application has a limitation on the number of simultaneous connections in a demo version (which frequently happens), all a hacker needs to do is find the instruction of the process carrying out this check and delete it. Modification of a program can be detected and prevented by testing the checksum regularly; however, the code that calculates the checksum and compares it to a particular value can be found and deleted.

However many protection levels there are — one or one million — the program *can* be cracked! It's only a matter of time and effort. But, when there are no effective laws protecting intellectual property, developers must rely on protection more than law-enforcement bodies. There's a common opinion that if the expense of neutralizing protection isn't lower than the cost of a legal copy, nobody will crack it. This is wrong! Material gain isn't the only motivation for a hacker. Much stronger motivation appears to lie in the *intellectual struggle*(who's more clever: the protection developer or me?), the *competition* (which hacker can crack more programs?), *curiosity* (what makes it tick?), *advancing one's own skills* (to create protections, you first need to learn how to crack them), and simply as an *interesting way to spend one's time.* Many young hackers spend weeks removing the protection from a program that only costs a few dollars, or even one distributed free of charge.

The usefulness of protection is limited to its competition — other things being equal, clients always select unprotected products, even if the protection doesn't restrain the client's rights. Nowadays, the demand for programmers considerably exceeds supply, but, in the distant future, developers should either come to an agreement or completely refuse to offer protection. Thus, protection experts will be forced to look for other work.

Chapter 2

PORTABLE EXECUTABLE ANATOMY

Portable Executable file is the native Win32 file format. Every win32 executable (except VxDs and 16-bit DLLs) uses PE file format. 32bit DLLs, COM files, OCX controls, Control Panel Applets (.CPL files) and .NET executables are all PE format. Even NT's kernel mode drivers use PE file format.

2.1 File Structure

This analysis will help later in unpacking and code injection technique for most of the shareware software available today in the market.

At a minimum, a PE file will have 2 sections; one for code and the other for data. An application for Windows NT has 9 predefined sections named **.text**, **.bss**, **.rdata**, **.data**, **.rsrc**, **.edata**, **.idata**, **.pdata**, and **.debug**. Some applications do not need all of these sections, while others may define still more sections to suit their specific needs.

The sections that are most commonly present in an executable (depends on the compiler being used) are:

- **Executable Code** Section, named **.text** (Microsoft) or **CODE** (Borland)
- **Data** Sections, named **.data**, **.rdata**, or **.bss** (Microsoft) or **DATA** (Borland)
- **Resources** Section, named **.rsrc**
- **Export Data** Section, named .**edata**
- **Import Data** Section, named **.idata**
- **Debug Information** Section, named .**debug**

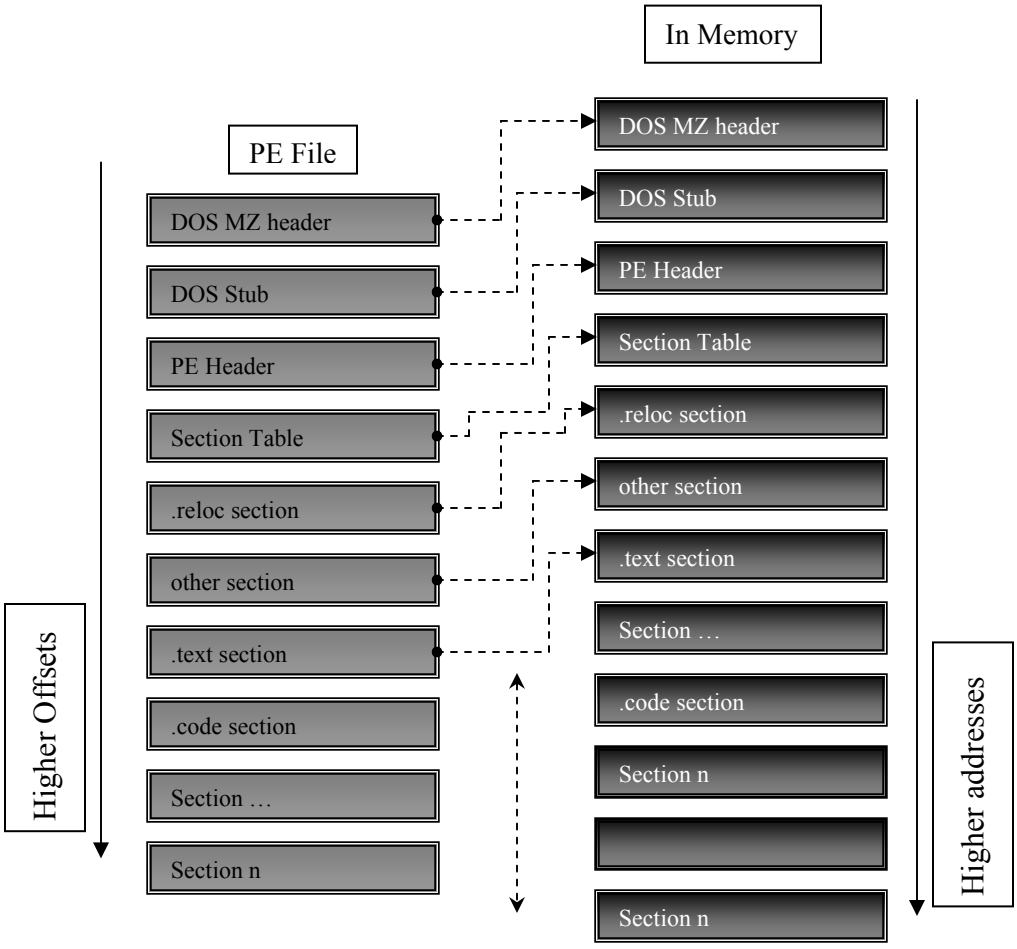


Figure 2.1 Virtual Memory

The names are actually irrelevant as they are ignored by the OS and are present only for the convenience of the programmer. Another important point is that the structure of a PE file on disk is exactly the same as when it is loaded into memory so if you can locate info in the file on disk you will be able to find it when the file is loaded into memory.

However it is not copied exactly into memory. The windows loader decides which parts need mapping in and omits any others. Data that is not mapped in is placed at the end of the file past any parts that will be mapped in e.g. Debug information.

Also the location of an item in the file on disk will often differ from its location once loaded into memory because of the page-based virtual memory management that windows uses. When the sections are loaded into RAM they are aligned to fit to 4Kb memory pages, each section starting on a new page.

The concept of virtual memory is that instead of letting software directly access physical memory, the processor and OS create an invisible layer between the two. Every time an attempt is made to access memory, the processor consults a "page table" that tells the process which physical memory address to actually use. It wouldn't be practical to have a table entry for each byte of memory (the page table would be larger than the total physical memory), so instead processors divide memory into pages. This has several advantages:

- It enables the creation of multiple address spaces. An address space is an isolated page table that only allows access to memory that is pertinent to the current program or process. It ensures that programs are completely isolated from one another and that an error causing one program to crash is not able to poison another program's address space.

- It enables the processor to enforce certain rules on how memory is accessed. Sections are needed in PE files because different areas in the file are treated differently by the memory manager when a module is loaded. At load time, the memory manager sets the access rights on memory pages for the different sections based on their settings in the section header. This determines whether a given section is readable, writable, or executable. This means each section must typically start on a fresh page.

- However, the default page size for Windows is 4096 bytes (1000h) and it would be wasteful to align executables to a 4Kb page boundary on disk as that would make them significantly bigger than necessary. Because of this, the PE header has two different alignment fields; Section alignment and file alignment. Section alignment is how sections are aligned in memory as above. File alignment (usually 512 bytes or 200h) is how sections are aligned in the file on disk and is a multiple of disk sector size in order to optimize the loading process.

- It enables a paging file to be used on the hard drive to temporarily store pages from the physical memory whilst they are not in use. For instance if an app has been loaded but becomes idle, its address space can be paged out to disk to make room for another app which needs to be loaded into RAM. If the situation reverses, the OS can simply load the first app back into RAM and resume execution where it left off. An app can also use more memory than is physically available because the system can use the hard drive for secondary storage whenever there is not enough physical memory.

When PE files are loaded into memory by the windows loader, the in-memory version is known as a **module**. The starting address where file mapping begins is called an ***HMODULE***. A module in memory represents all the code, data and resources from an executable file that is needed for execution whilst the term **process** basically refers to an isolated address space which can be used for running such a module.

2.2 Sections Surgery

I wrote a very basic program in assembly language and compiled it using MASM compiler (Macro Assembler) v9.0 in order to have an overview about sections in the executable file with their different names and addresses for each one.

No	Name	VSize	VOffset	RSize	ROffset	Charact.
01	**.text**	000000B0	00001000	00000200	00000400	60000020
02	**.rdata**	0000012A	00002000	00000200	00000600	40000040
03	**.data**	00000010	00003000	00000200	00000800	C0000040
04	**.rsrc**	00000148	00004000	00000200	00000A00	40000040

Table 2.1 Section analyses [Sample DialogBox.exe]

Manipulating these sections could be done through any hex editor or any automated tool created for this purpose like Stud_PE v2.2.0.5 or LordPE Deluxe by yoda.

Name: [this field is 8 bytes] The name is just a label and can even be left blank.

VirtualSize: [DWORD union] The actual size of the section's data in bytes. This may be less than the size of the section on disk (Size OfRawData) and will be what the loader allocates in memory for this section.

VirtualAddress: The RVA of the section. The PE loader examines and uses the value in this field when it's mapping the section into memory. Thus if the value in this field is 1000h and the PE file is loaded at 400000h, the section will be loaded at 401000h.

SizeOfRawData: The size of the section's data in the file on disk, rounded up to the next multiple of file alignment by the compiler.

PointerToRawData: (Raw Offset) - incredibly useful because it is the offset from the file's beginning to the section's data. If it is 0, the section's data are not contained in the file and will be arbitrary at load time. The PE loader uses the value in this field to find where the data in the section is in the file.

Characteristics: Contains flags such as whether this section contains executable code, initialized data, uninitialized data, can it be written to or read from.

2.2.1 Executable Code Section

In Windows NT all code segments reside in a single section called **.text** or **CODE**. Since Windows NT uses a page-based virtual memory management system, having one large code section is easier to manage for both the operating system and the application developer. This section also contains the entry point mentioned earlier and the jump thunk table (where present) which points to the IAT.

2.2.2 Data Section

The **.bss** section represents uninitialized data for the application, including all variables declared as static within a function or source module.

The **.rdata** section represents read-only data, such as literal strings, constants, and debug directory information.

All other variables (except automatic variables, which appear on the stack) are stored in the **.data** section. These are application or module global variables.

2.2.3 Resource Section

The **.rsrc** section contains resource information for a module. There are many resource editors available today which allows editing, adding, deleting, replacing and copying resources.

For a demo about resource editor we can use Restorator[1] 2006 v3.60 build 1535. This figure shows the Sample DialogBox.exe resources. I didn't include an icon or any other resources only a dialog box with two buttons and one edit box.

Source code (Sample DialogBox.exe) written in MASM assembler:

```
.386
.model flat, stdcall  ;32 bit memory model
option casemap :none  ;case sensitive

include Sample DialogBox.inc

.code
```

[1] Restorator is a resource editor for Windows. Resources are additional data accompanying a Windows application. Resources are usually part of the application interface. E.g. dialogs, menus, images, text, icons etc. They are usually stored with the executable or dll. Restorator can edit those resource files and thereby change the look and feel or language of an application completely independent of the development and compile tools. By bome.com/Florian Bomers

```
start:

      invoke GetModuleHandle,NULL
      mov    hInstance,eax

   invoke InitCommonControls
   invoke DialogBoxParam,hInstance,IDD_DIALOG1,NULL,addr DlgProc,NULL
   invoke ExitProcess,0

DlgProc proc hWin:HWND,uMsg:UINT,wParam:WPARAM,lParam:LPARAM

      .if uMsg == WM_COMMAND
            mov   eax,wParam
            .if eax == IDC_MESSAGE
                  invoke SetDlgItemText,hWin,IDC_EDITBOX,ADDR Message
            .elseif eax == IDC_EXIT
                  invoke SendMessage, hWin, WM_CLOSE, 0, 0
            .endif
      .elseif     uMsg == WM_CLOSE
            invoke EndDialog, hWin, 0
      .endif

   xor eax,eax
   ret
DlgProc endp

end start
```

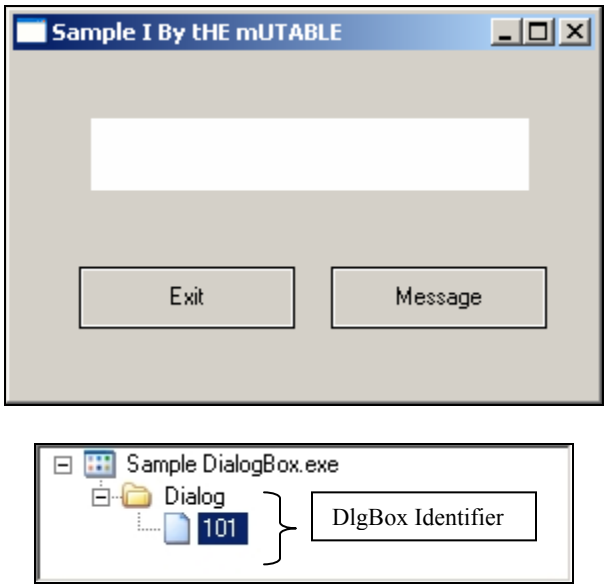


Figure 2.2 Restorator: DlgBox Identification

This tool helps in translating the strings to any language an it could be used as an aesthetical purpose for cracking after locating the *BadMessage*[2] String.

2.2.4 Export Data Section

The **.edata** section contains the Export Directory for an application or DLL. When present, this section contains information about the names and addresses of exported functions.

2.2.5 Import Data Section

The **.idata** section contains various information about imported functions including the Import Directory and Import Address Table.

2.2.6 Debug Information Section

Debug information is initially placed in the **.debug** section. The PE file format also supports separate debug files (normally identified with a .DBG extension) as a means of collecting debug information in a central location. The debug section contains the debug information, but the debug directories live in the .rdata section mentioned earlier. Each of those directories references debug information in the .debug section.

2.2.7 Base Relocation Section

When the linker creates an EXE file, it makes an assumption about where the file will be mapped into memory. Based on this, the linker puts the real addresses of code and data items into the executable file. If for whatever reason the executable ends up being loaded somewhere else in the virtual address space, the addresses the linker plugged into the image are wrong. The information stored in the **.reloc** section allows the PE loader to fix these addresses in the loaded image so that they're correct again. On the other hand, if the loader was able to load the file at the base address assumed by the linker, the .reloc section data isn't needed and is ignored.

2.3 The Import Section

The import section (usually .idata) contains information about all the functions imported by the executable from DLLs. This information is stored in several data structures. The most important of these are the Import Directory and the Import Address Table which we will discuss next. The Windows loader is responsible for loading all of the DLLs that the application uses and mapping them into the process address space. It

[2] *BadMessage* means that the program is not registered yet (annoying message box text), so you have to locate the GoodMessage either using patching technique where the software being cracked automatically place the right one or by any resource editor.

has to find the addresses of all the imported functions in their various DLLs and make them available for the executable being loaded.

The addresses of functions inside a DLL are not static but change when updated versions of the DLL are released, so applications cannot be built using hardcoded function addresses. Because of this a mechanism had to be developed that allowed for these changes without needing to make numerous alterations to an executable's code at runtime. This was accomplished through the use of an Import Address Table (IAT). This is a table of pointers to the function addresses which is filled in by the windows loader as the DLLs are loaded.

By using a pointer table, the loader does not need to change the addresses of imported functions everywhere in the code they are called. All it has to do is add the correct address to a single place in the import table and its work is done.

2.4 The Loader

When an executable is run, the windows loader creates a virtual address space for the process and maps the executable module from disk into the process' address space. It tries to load the image at the preferred base address and maps the sections in memory. The loader goes through the section table and maps each section at the address calculated by adding the RVA[3] of the section to the base address. The page attributes are set according to the section's characteristic requirements. After mapping the sections in memory, the loader performs base relocations if the load address is not equal to the preferred base address in ImageBase.

The import table is then checked and any required DLLs are mapped into the process' address space. After all of the DLL modules have been located and mapped in, the loader examines each DLL's export section and the IAT is fixed to point to the actual imported function address. If the symbol does not exist (which is very rare), the loader displays an error. Once all required modules have been loaded execution passes to the app's entry point.

The area of particular interest in RCE is that of loading the DLLs and resolving imports. This process is complicated and is accomplished by various internal (forwarded) functions and routines residing in ntdll.dll which are not documented by Microsoft. As we said previously function forwarding is a way for Microsoft to expose a common Win32 API set and hide low level functions which may differ in different versions of the OS. Many familiar kernel32 functions such as GetProcAddress are simply thin wrappers around ntdll.dll exports such as LdrGetProcAddress which do the real work.

---

[3] Relative Virtual Address: In an executable file or DLL, an RVA is always the address of an item once loaded into memory, with the base address (ImageBase) of the image file subtracted from it: *RVA = VA – ImageBase* ∴ *VA = RVA + ImageBase*

Chapter 3

ASM AND C++ COMPILER LAB TEST

For Reverse Code Engineering RCE mechanism, Assembly language it's a must to understand it fully at least as a prerequisite for better understanding of how things flow between lines.

Machine Language is a numeric language that is specifically understood by a computer's processor (the CPU). Intel processors, for example, have a machine language that is automatically understood by other Intel processors. Machine languages consists purely of numbers.
Assembly language consists of statement that uses short mnemonics such as ADD, MOV, SUB, and CALL. Assembly language has one to one relationship with machine language, meaning that one assembly language instruction corresponds to one machine-language instruction.

3.1 ASM & C++ Compiler Study

After readings about computer languages and learning most of it from DOS, Qbasic, Delphi, C++, VB, to ASM, I came to a conclusion based on my experience concerning compiler architecture especially in the protection field that the programmer imagination and the compiler capabilities must unite for better performance and better protection.

In the following section I will write a small code which only shows a message box at startup on the screen in assembly language and another one in C++ language (the same functionality).Using API[4] (Application Programming Interface) technique.

ASM Code (Size After compilation: 3.0 KB):

```
.386
.model      flat, stdcall
Option      casemap:none

include     windows.inc
include     kernel32.inc
include     user32.inc
includelib  kernel32.lib
includelib  user32.lib

.data
MsgBoxCaption  db "Thesis DemoBox II",0
MsgBoxText     db "Welcome To LIU University",0
```

[4] Functions that user applications can use to request specific operations to be performed by the kernel.

```
.code
start:
invoke MessageBox, NULL, ADDR MsgBoxText, ADDR MsgBoxCaption, MB_OK
invoke ExitProcess,0
end start
```

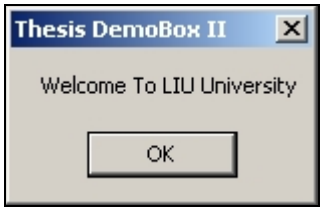


Figure 3.1 ASM Executable MessageBox

C++ Code (Size After compilation: 152 KB):

```
#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MessageBox(NULL, "Welcome To LIU University", "Thesis DemoBox II
               C++", MB_OK);
    return 0;
}
```

Figure 3.2 C++ Executable MessageBox

I've used an API technique in these two examples (API): Which is a set of functions that the operating system makes available to application programs for communicating with the operating system. For reversing under Windows, it is *imperative* that you develop a solid understanding of the Windows APIs and of the common methods of doing things using these APIs.

After the compilation for these two small programs comes the disassembler in order to investigate the behavior of the program as an assembly language with hex code mnemonic.

The disassembler is one of the most important reversing tools. Basically, a disassembler decodes binary machine code (which is just a stream of numbers) into a readable assembly language text. This process is somewhat similar to what takes place

within a CPU while a program is running. The difference is that instead of actually performing the tasks specified by the code (as is done by a processor), the disassembler merely decodes each instruction and creates a textual representation for it.
Needless to say, the specific instruction encoding format and the resulting textual representation are entirely platform-specific. Each platform supports a different instruction set and has a different set of registers. Therefore a disassembler is also platform-specific (though there are disassemblers that contain specific support for more than one platform).

I'm going to use IDA (Interactive Disassembler) by DataRescue (www.datarescue.com) which is an extremely powerful disassembler that supports a variety of processor architectures, including IA-32, IA-64 (Itanium), AMD64, and many others. IDA also supports a variety of executable file formats, such as PE (Portable Executable, used in Windows), ELF (Executable and Linking Format, used in Linux), and even XBE, which is used on Microsoft's Xbox.

Using IDA Pro Advanced v 4.9.0.863 (32bit)

3.1.1 ASM Disassembled Code

Disassembling the compiled ASM Code:

```
.text:00401000                  public start
.text:00401000 start            proc near
.text:00401000                  push    0               ; uType
.text:00401002                  push    offset Caption  ; "Thesis
                                                         DemoBox II"
.text:00401007                  push    offset Text     ; "Welcome To
                                                        LIU University"
.text:0040100C                  push    0               ; hWnd
.text:0040100E                  call    MessageBoxA
.text:00401013                  push    0
.text:00401015                  call    $+5
.text:0040101A                  jmp     ds:ExitProcess
.text:0040101A start            endp
```

As you notice the code is very clear and understandable from the first looking without any additional garbage from the MASM compiler. MessageBox calling convention is in the reverse order to invoke (C convention where for Delphi it's almost the same as the source code). Everything is clear. If we compare this disassembled code to ASM source code: no huge differences.

According to the MSDN library the MessageBox:

The MessageBox function creates, displays, and operates a message box. The message box contains an application-defined message and title, plus any combination of predefined icons and push buttons.

Syntax

```
    int MessageBox(

    HWND hWnd, ; in this case "0" no owner window referred to."hWnd"

    LPCTSTR lpText, ; "Welcome To The LIU University"

    LPCTSTR lpCaption, ; "Thesis DemoBox II"

    UINT uType ; for every type (Icon, Buttons Combination assigned
nb)

);
```

Parameters

hWnd
[in] Handle to the owner window of the message box to be created. If this parameter is NULL, the message box has no owner window.

lpText
[in] Pointer to a null-terminated string that contains the message to be displayed.

lpCaption
[in] Pointer to a null-terminated string that contains the dialog box title. If this parameter is NULL, the default title **Error** is used.

uType
[in] Specifies the contents and behavior of the dialog box. This parameter can be a combination of flags.

The Exit Process function ends a process and all its threads.

```
Void ExitProcess(

  UINT uExitCode. ; "0"

);
```

3.1.2 C++ Disassembled Code

Disassembling the compiled C++ Code:

```
.text:00401010 ; int __stdcall WinMain(HINSTANCE hInstance,HINSTANCE
                 hPrevInstance,LPSTR lpCmdLine,int nShowCmd)

.text:00401010 WinMain         proc near              ; CODE XREF:
```

```
                                                    WinMain(x,x,x,x)↑j
.text:00401010
.text:00401010 var_40          = dword ptr -40h
.text:00401010 hInstance       = dword ptr  8
.text:00401010 hPrevInstance   = dword ptr  0Ch
.text:00401010 lpCmdLine       = dword ptr  10h
.text:00401010 nShowCmd        = dword ptr  14h
.text:00401010
.text:00401010                 push    ebp
.text:00401011                 mov     ebp, esp
.text:00401013                 sub     esp, 40h
.text:00401016                 push    ebx
.text:00401017                 push    esi
.text:00401018                 push    edi
.text:00401019                 lea     edi, [ebp+var_40]
.text:0040101C                 mov     ecx, 10h
.text:00401021                 mov     eax, 0CCCCCCCCh
.text:00401026                 rep stosd
.text:00401028                 mov     esi, esp
.text:0040102A                 push    0               ; uType
.text:0040102C                 push    offset Caption ; "Thesis
DemoBox
                                                        II C++"
.text:00401031                 push    offset Text    ; "Welcome To
                                                       LIU
University"
.text:00401036                 push    0               ; hWnd
.text:00401038                 call    ds:__imp__MessageBoxA@16 ;
                                          MessageBoxA(x,x,x,x)
.text:0040103E                 cmp     esi, esp
.text:00401040                 call    __chkesp
.text:00401045                 xor     eax, eax
.text:00401047                 pop     edi
.text:00401048                 pop     esi
.text:00401049                 pop     ebx
.text:0040104A                 add     esp, 40h
.text:0040104D                 cmp     ebp, esp
.text:0040104F                 call    __chkesp
.text:00401054                 mov     esp, ebp
.text:00401056                 pop     ebp
.text:00401057                 retn    10h
.text:00401057 WinMain         endp
```

```
.text:00401BB1 loc_401BB1:                     ; CODE XREF: doexit+BD↑
.text:00401BB1                 mov     _C_Exit_Done, 1
.text:00401BBB                 mov     ecx, [ebp+uExitCode]
.text:00401BBE                 push    ecx             ; uExitCode
.text:00401BBF                 call    ds:__imp__ExitProcess@4 ;
                                             ExitProcess(x)
```

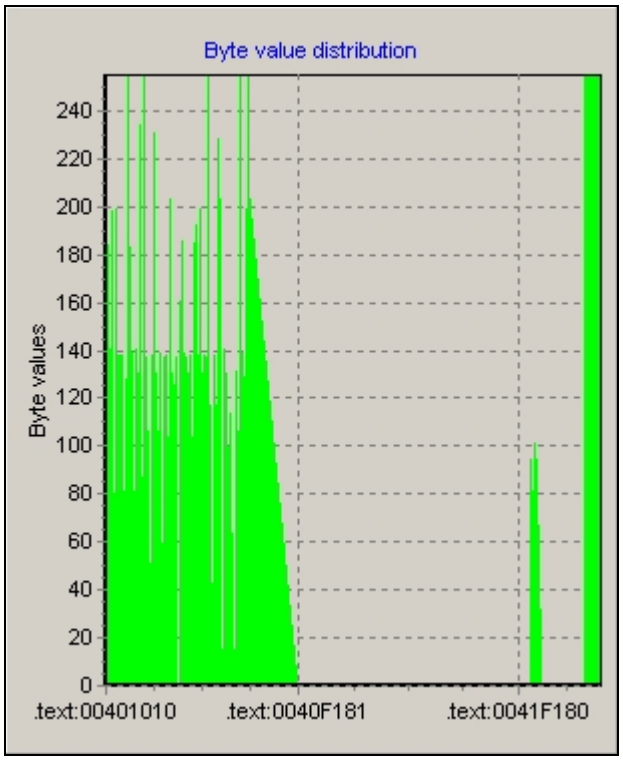
In the compiled C++ program the disassembled code is very lengthy with many cross references, Import functions from other DLL (Dynamic Link Library) file
It seems clear enough which code is more readable and compact for better performance.

	ASM Code	C++ Code
Size	3 KB	152 KB
Imports	2	50
D.ASM.Code readability	More	Less
Compactness	High	Low
Names	2	531
Functions	2	239

Table 3.1 Differences between ASM & C++ compiler implementation

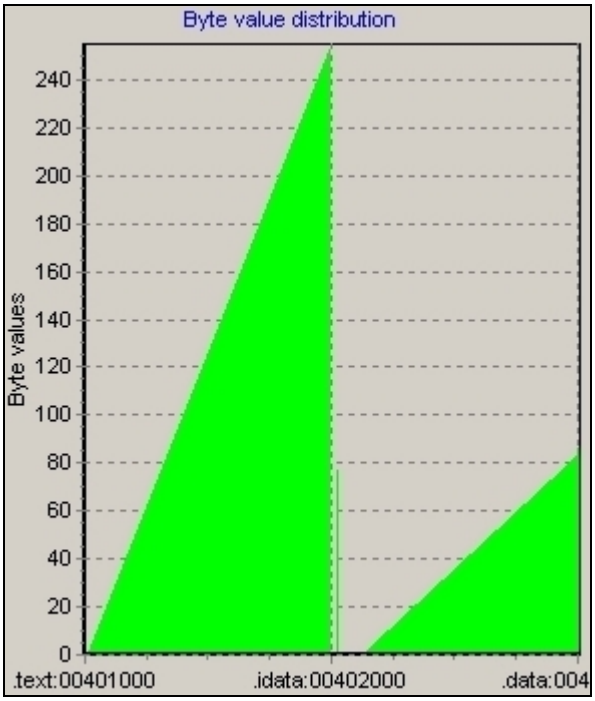
From compiler point of view MASM compiler is better when writing Assembly language instructions without any help from other resources only the required ones, whereas concerning the Microsoft Visual Studio v6.0 C++ compiler, writing in high level language (C++) requires many additional resources from other DLL's to interact with the operating system. But for reversing with such huge informative analysis depicted by IDA the task become less complex especially in malware analysis.

We can also check the distribution of the byte value in each one with respect to the sections size and how messy it's the graph in C++ compiler.



C++ compiler Byte Value Distribution: it's clear how much variation there are in the same section and that's declare the extremity in the near and far jumping in the program in order to interconnect with its own procedures from one place to another.

Figure 3.3 C++ Byte Value Distribution



ASM compiler Byte Value Distribution: here the graph is very well distributed with no extremities between sections; also we notice the three sections being varied in size and distribution in a normal way.

Figure 3.4 ASM Byte Value Distribution

Chapter 4

BREAKING PROTECTIONS

Before start breaking the protection of any software, let me first introduce some definitions which is important for later discussions.

4.1 Debugger

The term debugger is something of a misnomer[5]. Strictly speaking, a debugger is a tool to help track down, isolate, and remove bugs from software programs. Bugs are software defects that have been affectionately known as bugs. In truth, debuggers are tools to illuminate the dynamic nature of a program-they are used to understand a program as well as find and fix defects. Debuggers are the magnifying glass, the microscope, the logic analyzer, the profiler, and the browser with which a program can be examined.

Debuggers are software tools that help determine why the program does not behave correctly. They aid a programmer in understanding a program and then in finding the cause of the discrepancy. The programmer can then repair the defect and so allow the program to work according to its original intent. A debugger is a tool that controls the application being debugged so as to allow the programmer to follow the flow of program execution and, at any desired point, stop the program and inspect the state of the program to verify its correctness.

4.2 Packer Theory

The executable file is packed? What does that mean. Just as we pack files using Winzip or Winrar we can pack executables to protect them and conserve space. You can't open a zip file without a program to unpack it. The same is true for packed executables; except the program that unpacks it is part of the executable. The unpacker program is called a STUB. When you run a packed EXE the STUB first decrypts/unpacks the original EXE into memory. Then it executes the original program. The beginning of the original program is called the Original Entry Point "OEP". What to do next is waiting until the program is decrypted into memory, find the OEP and then dump[6] the decrypted file to the hard drive. However, the OEP does not mean the beginning of a "working" EXE. Even if the programs code is known but an executable

[5] A wrong or unsuitable name.

[6] Dump: every program executed is mapped in memory as it’s, even if it’s packed. After a while everything will be clear. So, in order to overcome the packed executable file we must load it into debugger and then save it to a disk file, of course after we found the Original Entry Point, hence the dumped program will be freed from the unpacking stub and will again run as before. But sometimes the OEP redirected in a very complicated way in order to increase the protection.

can have many different sections outside of the code. One very important section is called the Import Address Table "IAT". The Import Address Table allows a program to use functions stored outside of the program. A Messagebox from the Windows API is an example of an outside function. When a program wants to use an outside function Windows loads the DLL with that function into memory address space and then gives the IAT the code location for the desired functions. A table is created with called functions, and addresses of those functions within the DLL's; hence the Import Address Table.

The compressed executable file requires:

- Less storage space in the file system
- Less time to transfer data from the file system into memory
- More time to decompress the data before execution begins than the uncompressed original .

These advantages come at a price. For viruses, virii, worms and other malicious software (Malware) which spread rapidly on the internet the antiviruses face a big problem because of the packers being used in these viruses and make things more complicated for AV programmer to identify the signature of the virus. Because the cracker themselves build their own advanced packers (Private License only for crackers group not for public use) especially in polymorphic and Import Address Table redirection means.

4.3 Newton Third Law: Protectors

For every action there is an equal and opposite reaction.

The aim of the protector is emphasizing more or even completely on protection against Reverse Engineering than the simple packer. So, protectors try adding another sections and layers of obfuscations in order to defeat the reverser. The size of the program being protected is large. But in fact sometimes things are easier to defeat with protectors than packer.(my experience with highly sophisticated packers and protectors).

Nothing is impossible, for every packing, protecting mechanism there are tens of unpacking and unprotecting mechanisms, for some packers, protectors there is a fully automated software which delete the whole protection of the software in one click.

What the most important thing a protector mess with is the Import Address Table (IAT). The following paragraphs are very important, so read it carefully and try to visualize the mechanism of the possibilities of what the protectors' abilities inside this area.

There are different Microsoft windows operating systems, and they all have different addresses for their API functions, because of different structured DLL's. when an application starts, it has a list of all functions that aren't originally a part of the application. These functions, called imports, are located in the operating system DLL's,

but the application doesn't know where. Every win32 executable application has an Import address table (IAT) residing inside the program. The IAT is used as a lookup table when the application is calling a windows API function. So before starting, the windows loader has to find each address of each API that the program wants to call and constructs an IAT with them. When the program is running and it wants to call API, it simply looks in the IAT and thus finds immediately the address it needs to go in the DLL. When an executable has been packed or protected the reverse engineer must recover the original executable file because a lot of packers/protectors destroy the IAT (while taking care of finding the API's for the program). The import address table needs to be either rebuilt or fixed to allow for the executable to run properly. Import rebuilding is the reconstruction of the import address table (IAT).

Now, let's start this over but look in it in more detail. First of all: when an executable is first loaded, the windows loader is responsible for reading in the PE structure and loading the executable image into memory. One of the other steps it takes is to load all of the DLL's that the application uses and map them into the process address space. The executable also lists all of the functions it will require from each DLL. Because the function addresses are not static, a mechanism was developed that allows for these variables to be change without needing to alter all of the compiled code at runtime. This was accomplished through the use of an import address table (IAT). This is a table of function pointers filled in by the windows loader as the DLL's are loaded. When the application was first compiled, it was designed so that none of the API calls use direct hardcoded addresses but rather work through a function pointer. In follow, this pointer table can be accessed in several ways. Either directly by a call [pointer address] or via a jump thunk table. By using the pointer table, the loader does not need to fix up all of the places in the code that want to use the API call, all it has to do is add the pointer to a single place in a table and its work is done.

When it comes to packed executables, they almost invariably mess with the processes import table in order to make the executable smaller and make it harder for people to unpack and get running again. Packed programs were still generated with standard compilers, and of course they were still designed to work this fixed mechanism (which is a very efficient way to handle the problem anyway). If a packer has destroyed the default import table mechanism, which simply means that the packer/protector will have to figure out which DLL's and functions to load and where to place the pointers so that the original program still operates as normal after it has done its decompression and restorations routines.

4.4 A Little Bit About Assembly

But before embarking on the Reverse Engineering, let me first describe a little bit about registers because they are the primary information holders when using a debugger. So, registers are high-speed storage locations directly inside the CPU, designed to be accessed at much higher speed than conventional memory. When a processing loop is optimized for speed, for example, registers are used inside the loop rather than variables.

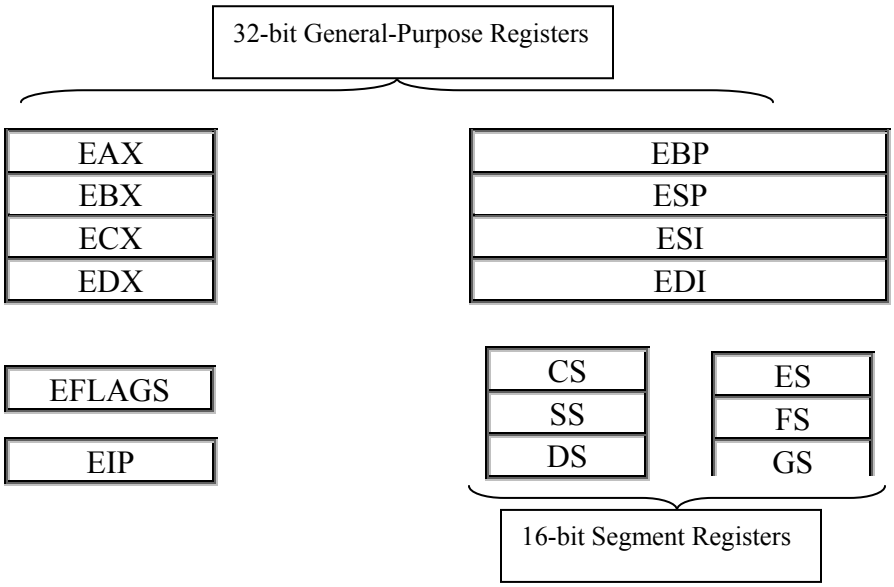
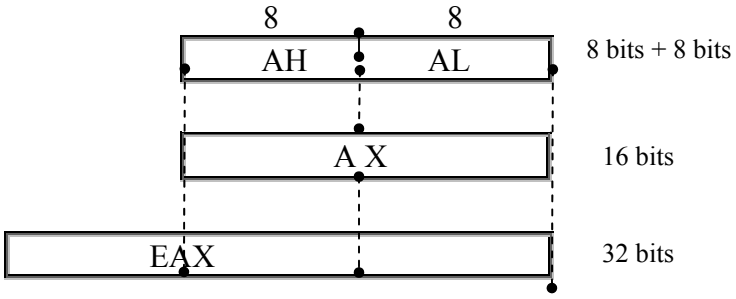


Figure 4.1 IA-32 Basic Program Execution Registers

Figure 4.1 shows the basic program execution registers. There are eight general purpose registers, six segment registers, a register that holds processor status flags (EFLAGS), and an instruction pointer.

General-Purpose Registers: The general-purpose registers are primarily used for arithmetic and data movement. As shown in the following figure, each register can be addressed as either a single 32-bit value or a 16-bit value:



Some 16-bit registers can be addressed as two separate 8-bit values. For example, the EAX register is 32 bits. Its lower 16 bits are also named AX. The upper 8 bits of AX are named AH, and the lower 8 bits are named AL.

This overlapping relationship exists for the EAX, EBX, ECX, and EDX registers:

32-bit	16-bit	8-bit(High)	8-bit(Low)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

Table 4.1 Overlapping relationship For EAX, EBX, ECX, EDX registers

The remaining general purpose registers have separate names for their lower 16-bits, but cannot be divided further.

32-bit	16-bit
ESI	SI
EDI	DI
EBP	BP
ESP	SP

Table 4.2 16-bit registers used in real –address mode

Specialized Used Some general-purpose registers have specialized uses:

- EAX is automatically used by multiplication and division instructions. It is often called the *extended accumulator* register. "*EAX is also the return register for almost all API functions."*
- The CPU automatically uses ECX as loop counter.
- ESP addresses data on the stack (a system memory structure). It should never be used for ordinary arithmetic or data transfer. It is often called the *extended stack pointer* register
- ESI and EDI are used by high-speed memory transfer instructions. They are sometimes called the *extended source index* and *extended destination index* registers.
- EBP is used by high-level languages to reference function parameters and local variables on the stack. It should not be used for ordinary arithmetic or data transfer except at an advanced level of programming. It is often called the *extended frame pointer* register.

Segment Registers the segment registers are used as base locations for preassigned memory areas called segments. Some segments hold program instructions (code), others hold variables (data), and another segment called the stack segment holds local function variables and function parameters.

Instruction pointer The EIP, or instruction pointer register contains the address of the next instruction to be executed. Certain machine instruction manipulate this address, causing the program to branch to a new location.

EFLAGS Register The EFLAGS (or just Flags) register consists of individual binary bits that either control the operation of the CPU or reflect the outcome of some CPU operation. There are machine instructions that can test and manipulate the processor flags.

Note: A flag is *set* when it equals 1; it is *clear* (or reset) when it equals 0.

Instructions (*ASM mnemonics*)

An instruction is a statement that is executed by the processor at runtime after the program has been loaded into memory and started. For example:

```
MOV destination,source ; Copies a byte or word from a source operand
                         to a destination operand. Ex. MOV eax,10000h
                         MOV (copies) the integer 10000h to EAX regi-
                         ster, Now EAX holds the value 10000h, EAX =
                         10000h and so on for another instructions the
                         same procedure( relatively speaking).
```

Here is some of the fundamental instructions is ASM which often considered to be the key point in breaking protections depending on conditional and unconditional jump's and how it affects the flags status.

Mnemonic	Comment
AND dest,src	(Logical AND). Each bit in the destination operand is ANDed with the corresponding bit in the source operand. This will equal to 1 only when two 1 are stand below each other otherwise equal to 0. This instruction will clear the O-Flag and the C-Flag and can set the Z-Flag. (Ex. AND EAX, EDX), where EAX holds 10101101 & EDX holds 11101100 10101101 11101100 10101100
CALL address	call a procedure, pushes the location of the next instruction on the stack and transfers to the destination location, executes a function at the specified address, (Ex. CALL 00404056, CALL EAX, CALL DWORD PTR [EAX] , CALL DWORD PTR [EAX+3]): call address 00404056 , call register EAX, calls the address that is stored at [EAX], calls the address that is stored at [EAX+3] and then when the function has finished the code will continue the line after the call.
CMP dest,src	Compares the destination to the source by performing an implied subtraction of the source from the destination and updates the flags, this is a very important instruction, because it verify something whether it's to check if the program is registered or to compare the entered serial (fake) to the real one. Usually comes

	after a call instruction... Ex. CMP EAX,ECX JZ 00401648 ;("address") Compare EAX to ECX if equal the zero flag is set to 1, hence if the zero flag is set then jump to 00401648
INC	Adds 1 to register (Ex. INC EAX)or a memory operand (Ex. INC [00406234], INC [EAX], INC [EAX+00406234]): increase the dword that is stored at [00406234], [EAX], [EAX+00406234].
DEC	Subtracts 1 from an operand. The opposite of the INC
INT	Interrupt. Generates a software interrupt, which in turn calls an operating system subroutine (Ex. INT 3)
RET	To return from a function, located usually at the end of a function, and it simply instructs the processor to RETurn to the address of the CALL to the function
NOP	This instruction does nothing (No Operation). (it's used for patches so often)
OR dest,src	Performs a boolean (bitwise) OR operation between each matching bit in the destination operand and each bit in the source operand. Only when there are two 0 on top of each other, the resulting bit is 0. Else the resulting bit is 1. This instruction will clear the O-Flag and the C-Flag and can set the Z-Flag. (Ex. OR EAX, EBX), let's say EAX holds 01011001 & EBX holds 10001010 therefore ORing them will yield: 01011001 10001010 11011011
XOR dest,src	Each bit in the source operand is exclusive ORed with its corresponding bit in the destination. The destination bit is a 1 only when the original source and destination bits are different. This instruction will clear the O-Flag and the C-Flag and can set the Z-Flag. (Ex. OR EAX,EBX), let's say EAX holds 01011001 & EBX holds 10001010 therefore XORing them will yield: 01011001 10001010 11110011 One of the most often seen use of XOR is XOR EAX,EAX This will set EAX to 0, because when you XOR a value with itself, the result is always 0
TEST dest,src	Tests individual bits in the destination operand against those in the source operand. Performs a logical AND operation that affects the flags but not the destination operand. (Ex. TEST EAX,EAX). It does not save the values. It only sets the Z-Flag, when EAX is 0 or clears it, when EAX is not 0. The Overflow/Carry flags are always cleared.

POP dest	(Pop from Stack[7]). Copies a word or a doubleword at the current stack pointer location into the destination operand, and adds 2(or 4) to (Extended) Stack Pointer (E)SP. POP loads the value of byte/word/dworp ptr [esp] and puts it into dest. Additionaly it increases the stack by the size of the value that was popped of the stack, so that the next POP would get the next value. *(LaZaRuS)*
PUSH operand	(Push on Stack). Subtracts 2 from (E)SP and copies the source operand into the stack location pointed to by (E)SP. PUSH is the opposite of POP. It stores a value on the stack and decreases it by the size of the operand that was pushed, so that ESP points to the value that was PUSHed. *(LaZaRuS)*
JMP	Jump unconditionally to... ("address") a code. Ex. JMP 00412075

Jxx	Meaning	Jump Condition
JA	Jump if (unsigned) above	CF=0 and ZF=0
JAE	Jump if (unsigned) above or equal	CF=0
JB	Jump if (unsigned) below	CF=1
JBE	Jump if (unsigned) below or equal	CF=1 or ZF=1
JC	Jump if carry flag set	CF=1
JCXZ	Jump if CX is 0	CX=0
JE	Jump if equal	ZF=1
JECXZ	Jump if ECX is 0	ECX=0
JG	Jump if (signed) greater	ZF=0 and SF=OF (SF = Sign Flag)
JGE	Jump if (signed) greater or equal	SF=OF
JL	Jump if (signed) less	SF != OF (!= is not)
JLE	Jump if (signed) less or equal	ZF=1 and SF != OF
JMP	Jump	Jumps always
JNA	Jump if (unsigned) not above	CF=1 or ZF=1
JNAE	Jump if (unsigned) not above or equal	CF=1
JNB	Jump if (unsigned) not below	CF=0
JNBE	Jump if (unsigned) not below or equal	CF=0 and ZF=0
JNC	Jump if carry flag not set	CF=0
JNE	Jump if not equal	ZF=0

[7] The **Stack** is a group of memory locations in the Read/Write memory that is used for temporary storage of binary information during the execution of a program. The starting memory location of the stack is defined in the main program, and space is reserved, usually at the high end of the memory map. The method of information storage resembles a stack of books. The contents of each memory location are, in a sense, "Stacked" –one memory location above another –and information is retrieved starting from the top. Hence, this particular group of memory locations is called the stack. (Gaonkar, Ramesh. *Microprocessor Architecture, Programming, and Applications with the 8085*, Third Edition. 1996

JNG	Jump if (signed) not greater	ZF=1 or SF!=OF
JNGE	Jump if (signed) not greater or equal	SF!=OF
JNL	Jump if (signed) not less	SF=OF
JNLE	Jump if (signed) not less or equal	ZF=0 and SF=OF
JNO	Jump if overflow flag not set	OF=0
JNP	Jump if parity flag not set	PF=0
JNS	Jump if sign flag not set	SF=0
JNZ	Jump if not zero	ZF=0
JO	Jump if overflow flag is set	OF=1
JP	Jump if parity flag set	PF=1
JPE	Jump if parity is equal	PF=1
JPO	Jump if parity is odd	PF=0
JS	Jump if sign flag is set	SF=1
JZ	Jump if zero	ZF=1

4.5 Packer Theory Demonstration

Real demonstration about packers and IAT theory:

- ✓ First stage: writing a small C++ program and loaded in a debugger in order to investigate IAT theory.
- ✓ Second stage: pack the same program using a free packer and investigate how the OEP being changed upon using the packer and making a step by step manual unpacking demo.
- ✓ Third stage: make the software accept any password or changed to another one.

In this part, the debugger role is crucial for code analysis to visualize the implementation of the theory in real application. One of the most important debugger with many advanced features is OllyDbg v1.10 for Ole Yuschuk and is free you can download it from *http://www.ollydbg.de*. **OllyDbg** is a 32-bit assembler-level analyzing debugger with intuitive interface. It is especially useful if source code is not available or when you experience problems with your compiler.

I wrote a small C++ console based program (Compiled using MS Visual Studio 6.0)in order to demonstrate the three stages mentioned above.
First of all. I will explain the C++ source code functionality .

```
#include <iostream>
using namespace std;

int main()

{
	int counter = 0;

	while(counter !=3)
	{
		const int k = 13;
```

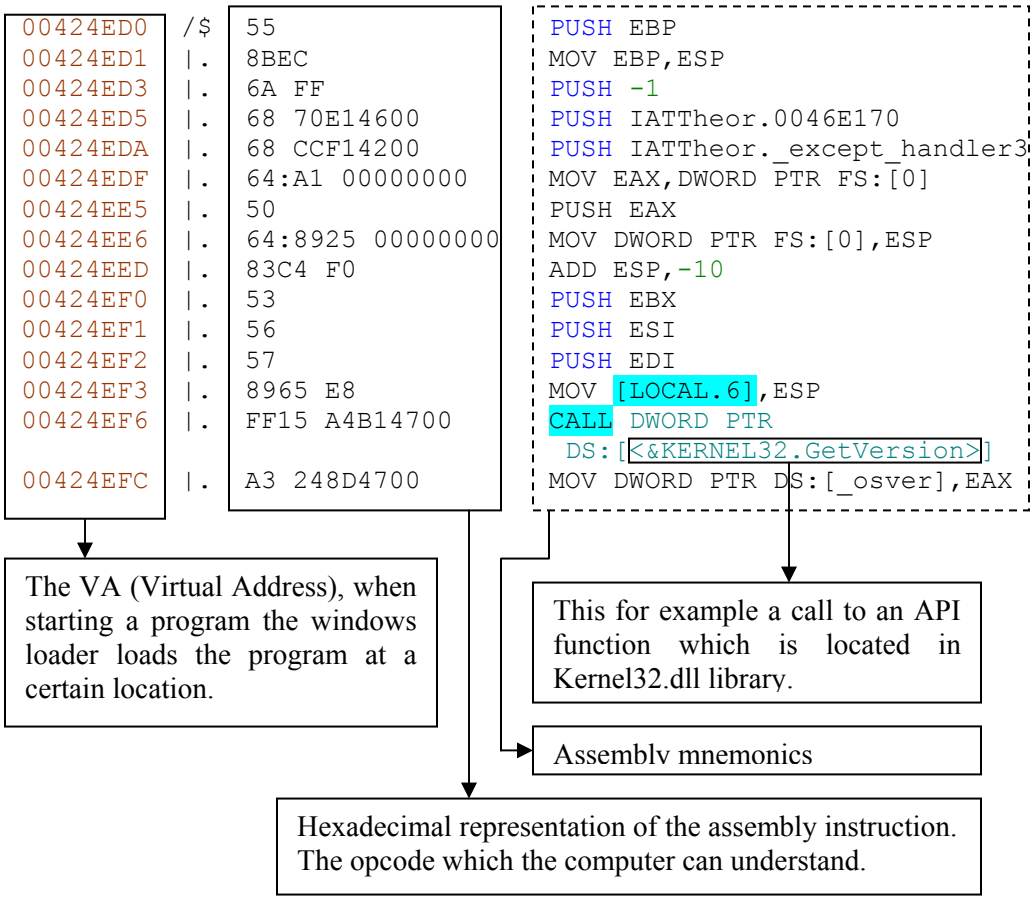
```
            const int l = 13;

            char password[k] = "drhasanbazzi";
            char user[l] = {0};

            cout <<"Thesis IAT, Packer, Serial Protection Demo"<<endl;

            cout <<"Please Enter Your Password: ";

            cin>>user;

      for (int i=0; i<=11; i++)
      {
            if (user[i] == password[i])
            {
                  cout<<"Character:"<<"["<<i<<"]"<<"="<<user[i]<<"
                        :Access Approved"<<endl;
            }

            else
            {
                  cout<<"Character:"<<"["<<i<<"]"<<"="<<user[i]<<"
                        :Access Denied"<<endl;
            }
      }

            counter++;
      }

      return 0;
}
```

Explanation: what's the functionality of this software? This software will serve as a guidance to apply the three stages after the compilation.

The purpose is: when it's executed the software will receive a password from the user and check it with a predefined one, if they match an Access Approved Message will appear otherwise an Access Denied Message will appear. The password created using an array of size 13 one for the predefined password and another one for user checking. The while loop will decide how many times you try in checking the password, in this case it's allowed for three times only. The usefulness of this software is that it will check each entered character with predefined one and after that the output will show with respect to the array index which matches and which doesn't. Why this? Because in the reversing we can try many thing for better understanding. When it comes to the disassembly we no longer have the source code only the executable, we deal with it in a disassembler and debugger in parallel.

4.5.1 First Stage: Deciphering The Bits

❖ First Stage Anatomy: the executable being loaded in OllyDbg



Apply the rule: Relative Virtual Address: For EXE file the image base is 00400000 and differs from the DLL files.

Let's take the second row as an example with VA = 00424ED1 (Hex), Therefore:

RVA = 00424ED1 – 00400000 = 24ED1

The Virtual Address (VA) of the first row is the Entry Point (EP)

`CALL DWORD PTR DS:[<&KERNEL32.GetVersion>]`, now if we assemble this line in OllyDbg by double-click on it, this will looks like any other call: `CALL DWORD PTR DS:[47B1A4]`, jumps to a value of dw pointer. where 47B1A4 is the destination address to where this functions goes. If we press enter in Olly this will lead to the Kernel32.dll routine. So, from whatever place from the program we call this function the same address "47B1A4" will be used an this makes things easier for the windows loader, and only the junk table needs to point to the right address instead of changing

every call anywhere to GetVersion API function. If we search for this address in the dump window in Olly by right click -> Go To - > Expression and then we cath the IAT with all the addresses for the API's in their respective DLL's.

```
0047B184 >FC A7 E7 77 49 A9 E7 77 44 0C F6 77 37 38 E7 77  ü§çwI©çwD.öw78çw
0047B194 >04 5A E7 77 0D 5B E7 77 48 C7 E6 77 58 E3 E7 77  ♪Zçw.[çwHÇæwXãçw
0047B1A4 >42 D1 E7 77 FD 98 E7 77 B8 16 E6 77 B9 E6 E7 77  BÑçwý˜çw¸┬æw¹æçw
0047B1B4 >31 A0 E7 77 7E DE E7 77 1C EE EA 77 94 E4 E7 77  1 çw~Þçw îêw"äçw
0047B1C4 >3A F1 E7 77 71 A6 E7 77 B7 49 E9 77 32 B3 E7 77  :ñçwq¦çw·Iéw2³çw
0047B1D4 >61 D9 E7 77 60 A6 E7 77 A9 AD E7 77 5E E3 E7 77  aÙçw`¦çw©-çw^ãçw
0047B1E4 >F8 5C E7 77 95 D9 E7 77 6B 15 F5 77 A1 E5 E7 77  ø\çw•Ùçwk└õw¡åçw
0047B1F4 >A1 16 F5 77 5F 8C F5 77 02 15 F5 77 16 89 E7 77  ¡┬õw_Œõw┐└õw┬‰çw
0047B204 >D2 E2 E7 77 CB 15 E8 77 72 AC E7 77 86 AD E7 77  ÒâçwË└èwr¬çw†-çw
0047B214 >C0 30 E9 77 42 75 E9 77 95 E5 E7 77 F2 94 E6 77  À0éwBuéw•åçwò"æw
0047B224 >85 E5 E7 77 51 E3 E7 77 FB E3 E7 77 7E 17 E6 77  …åçwQãçwûãçw~┤æw
0047B234 >2E F0 E7 77 59 53 E7 77 F0 A6 E7 77 E6 75 E7 77  .ðçwYSçwð¦çwæuçw
0047B244 >CE 77 E7 77 62 73 E7 77 0F 31 E6 77 82 67 E7 77  Îwçwbsçw☼1æw,gçw
0047B254 >AF DD E7 77 54 C9 E6 77 C8 E0 E7 77 95 81 E7 77  ¯ÝçwTÉæwÈàçw•ü çw
0047B264 >ED 95 E7 77 99 06 E7 77 8B B0 E7 77 97 25 E7 77  í•çw™─çw‹°çw—%çw
0047B274 >4E AB E7 77 1D 20 E8 77 55 E8 E7 77 6B 90 E6 77  N«çw  èwUèçwkü æw
0047B284 >00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Dump Window from Olly

Therefore each call in the code that wants to call an API will call the respective address in the jump thunk table which will point to the API's respective address in the IAT.

The windows loader first reads the header of the program. Specifically for the construction of the IAT, the bytes at RVA 3C are read. In this case, that will be at VA 40003C (RVA + ImageBase). This is done because the import table's RVA is stored in the PE header at its value plus 80h.

4.5.2 Second Stage: Packing and Unpacking

- ❖ Second Stage Anatomy: Packing and Unpacking

Now we need to envelop the executable file by packing it with a free packer UPX v1.24w for Markus F.X.J. Oberhumer & Laszlo Molnar. Which a command line tool without GUI (Graphical User Interface).
And here a screenshot for this nice packer/compressor because it has nothing to do with any anti debugging or protection. The main objective of this packer is to reduce the size of the executable file.

◆----------◆

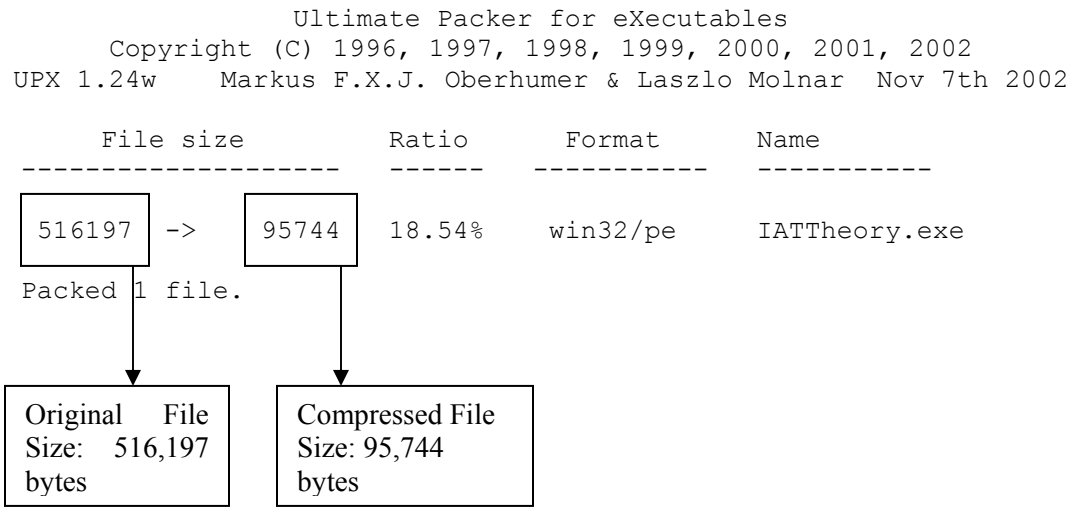
```
                    Ultimate Packer for executables
          Copyright (C) 1996, 1997, 1998, 1999, 2000, 2001, 2002
UPX 1.24w        Markus F.X.J. Oberhumer & Laszlo Molnar  Nov 7th 2002

Usage: upx [-123456789dlthVL] [-qvfk] [-o file] file..

Commands:
  -1     compress faster                   -9  compress better
  -d     decompress                        -l  list compressed file
  -t     test compressed file              -V  display version number
  -h      give more help                       -L   display software
license
Options:
  -q     be quiet                          -v  be verbose
  -oFILE write output to `FILE'
  -f     force compression of suspicious files
  -k     keep backup files
  file.. executables to (de)compress

This version supports: dos/exe, dos/com, dos/sys, djgpp2/coff,
watcom/le,win32/pe, rtm32/pe, tmt/adam, atari/tos, linux/386

UPX comes with ABSOLUTELY NO WARRANTY; for details type `upx -L'.
```

This compressor able to decompress the file without any manual unpacking but for checking the mechanism of how packer/compressor works we will do everything manually.

Now place the executable file in the same directory where the UPX located.
If the name of our executable is “IATTheory.exe”. now we need to compress it using the command line options and write :

```
upx -9 -k IATTheory.exe
```

“upx” is the name of the compressor, “-9” stands for better compression which is the maximum level of compression , “-k” for backing the executable file we work on for safe if something goes wrong, and finally “IATTheory.exe” the name of our executable to be compressed.
Now after executing this statement in the command line the result will be like this:



This is an amazing job for such compressor for reducing the size of the executable to such level.

After that, we need to load the compressed executable inside Olly, but Olly will complain about that, because of the PE Header realignment the compressor had done on it.

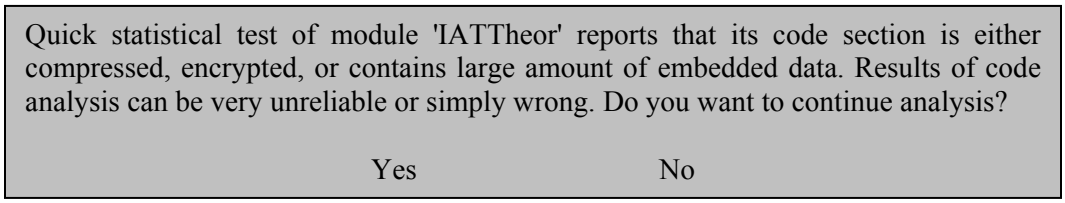


Figure 4.2 Compressed code error from OllyDbg

It doesn't matter just press yes and Olly will analyze the compressed executable. And here how the code looks like at first in Olly:

```
00481E50 > $  60              PUSHAD
00481E51   .  BE 00B04600     MOV ESI,IATTheor.0046B000
00481E56   .  8DBE 0060F9FF   LEA EDI,DWORD PTR DS:[ESI+FFF96000]
00481E5C   .  57              PUSH EDI
00481E5D   .  83CD FF         OR EBP,FFFFFFFF
00481E60   .  EB 10           JMP SHORT IATTheor.00481E72
00481E62      90              NOP
00481E63      90              NOP
00481E64      90              NOP
00481E65      90              NOP
00481E66      90              NOP
00481E67      90              NOP
00481E68   >  8A06            MOV AL,BYTE PTR DS:[ESI]
00481E6A   .  46              INC ESI
```

```
00481E6B    .  8807              MOV BYTE PTR DS:[EDI],AL
```

Another look at the Register Window in Olly:

```
EAX 00000000
ECX 0012FFB0
EDX 7FFE0304
EBX 7FFDF000
ESP 0012FFC4
EBP 0012FFF0
ESI 76397C78
EDI 77E94809 kernel32.77E94809
EIP 00481E50 IATTheor.<ModuleEntryPoint>
```

This is the *extended stack pointer* register, and it's clear how the value of this register is being pushed onto the stack. (Stack Window)

First of all we need to find the OEP (Original Entry Point) of the program which will allow for dumping the program to it's original state (original file size) without any error.

UPX uses the stack mechanism (PUSH ....POP) in order to unpack the compressed file in memory and run freely of the STUB.

A look at the Stack Window in Olly:

```
0012FFC4   77E814C7  RETURN to kernel32.77E814C7
0012FFC8   77E94809  kernel32.77E94809
0012FFCC   76397C78
0012FFD0   7FFDF000
0012FFD4   F2381CF0
0012FFD8   0012FFC8
0012FFDC   8053C88F
0012FFE0   FFFFFFFF  End of SEH chain
0012FFE4   77E94809  SE handler
0012FFE8   77E91210  kernel32.77E91210
0012FFEC   00000000
0012FFF0   00000000
0012FFF4   00000000
0012FFF8   00481E50  IATTheor.<ModuleEntryPoint>
0012FFFC   00000000
```

We notice that the value of the register (`ESP 0012FFC4`) has been pushed onto the stack, This is the result of the **PUSHAD** function at the beginning of the program. The function that POP's the register value from the stack is appropriately called **POPAD**. This will pop the value off of the stack and back into the register.

Now the method is by tracing the code and setting a breakpoint[8] in Dump Window in order to reach the OEP. The next step is to Step over[9] the code by pressing F8 in

[8] Setting a breakpoint at a specific location in the executing code such that when the processor reaches this location, execution will stop; this can be provided for by simply writing some illegal instruction into the code stream for the debuggee. (ex, INT 3)

OllyDbg, or shortcut (Ctrl-F12). Now that we have executed the PUSHAD, watch how the ESP register is changed to (ESP 0012FFA4). our register is on the stack. We need to break when the register is POPed off the stack.
In the Register Window: Right-click on the ESP register → Follow in Dump. (ESP register contains the address to the top of the stack).
In the Dump Window: Highlight the first four byte and Right-click → Breakpoint → Hardware[10], on access → Dword. Now Olly will stop when the first four bytes are accessed.
Now Press F9 (Run) wait for the program to be unpacked and we will be break at a JMP. The code looks like the following:

```
00481F80   .- E9 4B2FFAFF      JMP IATTheor.00424ED0
00481F85      00               DB 00
00481F86      00               DB 00
00481F87      00               DB 00
00481F88      00               DB 00
00481F89      00               DB 00
00481F8A      00               DB 00
```

If we Step Into this JMP, it will lead immediately to the OEP (depends on the experience)

Now press Step into F7 (Enter the JMP address). The code looks like the following:

```
00424ED0    55                  PUSH EBP

00424ED1    8BEC                MOV EBP,ESP
00424ED3    6A FF               PUSH -1
00424ED5    68 70E14600         PUSH IATTheor.0046E170
00424EDA    68 CCF14200         PUSH IATTheor.0042F1CC
00424EDF    64:A1 00000000      MOV EAX,DWORD PTR FS:[0]
00424EE5    50                  PUSH EAX
00424EE6    64:8925 00000000    MOV DWORD PTR FS:[0],ESP
00424EED    83C4 F0             ADD ESP,-10
00424EF0    53                  PUSH EBX
00424EF1    56                  PUSH ESI
00424EF2    57                  PUSH EDI
00424EF3    8965 E8             MOV DWORD PTR SS:[EBP-18],ESP
00424EF6    FF15 A4B14700       CALL DWORD PTR DS:[47B1A4]
                                ; kernel32.GetVersion
00424EFC    A3 248D4700         MOV DWORD PTR DS:[478D24],EAX
00424F01    A1 248D4700         MOV EAX,DWORD PTR DS:[478D24]
00424F06    C1E8 08             SHR EAX,8
00424F09    25 FF000000         AND EAX,0FF
```

[9] Trace over (Single Step) means that the processor is instructed to execute a single machine instruction when it is next processing instructions for the debuggee. But this doesn't allow entering the call; it's like a watcher for variables variations with respect to its line of code being executed.

[10] Hardware Breakpoints are directly supported by the CPU, using some special registers, called debug registers.

The virtual address: `00424ED0`, is the same as when the executable file was in its original state without being UPX'ed . and that does demonstrate that we are on the right track. (the code is the same as in First Stage analysis)
Dumping the executable file from memory to the hard drive. Using OllyDump Plug-in: Now Right-click in the Code Workplace → Dump debugged process. And it should looks like the following screenshot:

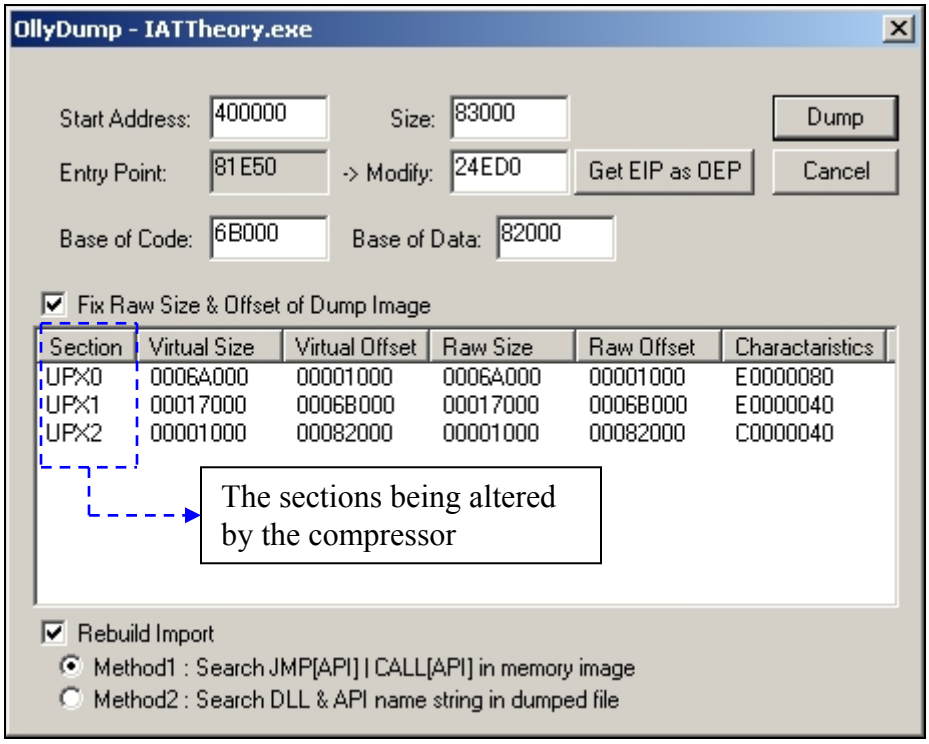


Figure 4.3 OllyDump: Dumping and Rebuilding the packed program

Then press on Dump button, a save Dump to file dialog will appear asking a name for the dumped file (in this case choose: dumped.exe)

The dumped file get back to its original state (size, configuration, IAT, Strings,...): everything work fine now.

4.5.3 Third Stage: Cracking The Micro-Universe of Bits

Third Stage: this is a very interesting stage for understanding how bits works together in a very organized structure to translate the exe back to assembly. And how to decrypt the algorithm step by step to its HL (High Level). The objective is to locate the routine which responsible about password checking mechanism and try to defeat the whole algorithm in a many different ways. It's a little bit complicated to grasp it at first, so had better to make a chart for every step of procedures.

A little bit explanation about this stage:

- Entering a False and True password to understand the output
- Defeat the counter limitation
- Defeat password checking array
- Patching the executable (static changes)
- Code Injection for the packed executable

4.5.3.1 Third Stage: Understanding False and True Password

✓ Starting with Entering a False and True password to understand the output.

❖ Entering a False password: 1r326autrzhm

Output Screen

```
Thesis IAT, Packer, Serial Protection Demo
Please Enter Your Password: 1r326autrzhm
Character:[0]=1 :Access Denied
Character:[1]=r :Access Approved
Character:[2]=3 :Access Denied
Character:[3]=2 :Access Denied
Character:[4]=6 :Access Denied
Character:[5]=a :Access Approved
Character:[6]=u :Access Denied
Character:[7]=t :Access Denied
Character:[8]=r :Access Denied
Character:[9]=z :Access Approved
Character:[10]=h :Access Denied
Character:[11]=m :Access Denied
```

❖ Entering a True password: drhasanbazzi

Output Screen

```
Thesis IAT, Packer, Serial Protection Demo
Please Enter Your Password: drhasanbazzi
Character:[0]=d :Access Approved
Character:[1]=r :Access Approved
Character:[2]=h :Access Approved
Character:[3]=a :Access Approved
Character:[4]=s :Access Approved
Character:[5]=a :Access Approved
Character:[6]=n :Access Approved
Character:[7]=b :Access Approved
Character:[8]=a :Access Approved
Character:[9]=z :Access Approved
Character:[10]=z :Access Approved
Character:[11]=i :Access Approved
```

As we notice in False password, some characters are equivalent to the right password corresponding to the array index and that's why we have an Access Approved, where in entering the true password we have an Access Approved for all characters.

Short identification:
T.P: True Password
F.E.P: False Entered Password
R.E.P: Right Entered Password

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
T.P	d	r	h	a	s	a	n	b	a	z	z	i
F.E.P	1	3	3	2	6	a	u	t	r	z	h	m
R.E.P	d	r	h	a	s	a	n	b	a	z	z	i

Table 4.3 Output Summary: F.E.T, R.E.P.

4.5.3.2 Third Stage: Defeat The Counter Limitation

✓ Defeat the Counter Limitation: the program allows only three times of checking, after that exit. Locating the routine responsible for that and change it for unlimited or increase the value. In order to get into the procedure responsible for that, we have to figure out a way to enter in. ***STRINGS (Characters)*** are found to be in the same .code section in relevant with the procedure relating to. Olly has a feature for searching for all referenced text strings. Therefore, strings will lead us to the heart of the routine. Load the executable file (IATTheory.exe) in Olly and Right-click in the Workplace code → Search for → All referenced text strings. And the output looks like this: the list is more than that, but that's what we want.

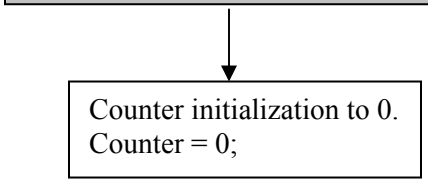
```
Text strings referenced in IATTheor:.text
Address    Disassembly                    Text string
0040170D   PUSH IATTheor.0046D088 ASCII "Thesis IAT, Packer, Serial
                                            Protection Demo"
00401726   PUSH IATTheor.0046D064 ASCII "Please Enter Your Password: "
0040177E   PUSH IATTheor.0046D04C ASCII " :Access Approved"
0040179E   PUSH IATTheor.0046D030 ASCII "Character:"
004017F2   PUSH IATTheor.0046D01C ASCII " :Access Denied"
00401812   PUSH IATTheor.0046D030 ASCII "Character:"
```

Now, double click at `ASCII "Thesis IAT, Packer, Serial Protection Demo"`
And we land here: (routine (Start...End) recognized as (Push...Ret))

```
004016A0   > \55                PUSH EBP
004016A1   .  8BEC              MOV EBP,ESP
004016A3   .  83EC 70           SUB ESP,70
004016A6   .  53                PUSH EBX
```

```
004016A7   .  56                PUSH ESI
004016A8   .  57                PUSH EDI
004016A9   .  8D7D 90           LEA EDI,DWORD PTR SS:[EBP-70]
004016AC   .  B9 1C000000       MOV ECX,1C
004016B1   .  B8 CCCCCCCC       MOV EAX,CCCCCCCC
004016B6   .  F3:AB             REP STOS DWORD PTR ES:[EDI]
004016B8   .  C745 FC 00000000  MOV DWORD PTR SS:[EBP-4],0
004016BF   >  837D FC 03        CMP DWORD PTR SS:[EBP-4],3
004016C3   .  0F84 A9010000     JE IATTheor.00401872
004016C9   .  C745 F8 0D000000  MOV DWORD PTR SS:[EBP-8],0D
004016D0   .  C745 F4 0D000000  MOV DWORD PTR SS:[EBP-C],0D
004016D7   .  A1 BCD04600       MOV EAX,DWORD PTR DS:[46D0BC]
004016DC   .  8945 E4           MOV DWORD PTR SS:[EBP-1C],EAX
004016DF   .  8B0D C0D04600     MOV ECX,DWORD PTR DS:[46D0C0]
004016E5   .  894D E8           MOV DWORD PTR SS:[EBP-18],ECX
004016E8   .  8B15 C4D04600     MOV EDX,DWORD PTR DS:[46D0C4]
004016EE   .  8955 EC           MOV DWORD PTR SS:[EBP-14],EDX
004016F1   .  A0 C8D04600       MOV AL,BYTE PTR DS:[46D0C8]
004016F6   .  8845 F0           MOV BYTE PTR SS:[EBP-10],AL
004016F9   .  C645 D4 00        MOV BYTE PTR SS:[EBP-2C],0
004016FD   .  33C9              XOR ECX,ECX
004016FF   .  894D D5           MOV DWORD PTR SS:[EBP-2B],ECX
00401702   .  894D D9           MOV DWORD PTR SS:[EBP-27],ECX
00401705   .  894D DD           MOV DWORD PTR SS:[EBP-23],ECX
00401708   .  68 F5104000       PUSH IATTheor.004010F5
0040170D   .  68 88D04600       PUSH IATTheor.0046D088
      ;  ASCII "Thesis IAT, Packer, Serial Protection Demo"
00401712   .  68 F8874700       PUSH IATTheor.004787F8
00401717   .  E8 EBFBFFFF       CALL IATTheor.00401307
0040171C   .  83C4 08           ADD ESP,8
0040171F   .  8BC8              MOV ECX,EAX
00401721   .  E8 32FBFFFF       CALL IATTheor.00401258
00401726   .  68 64D04600       PUSH IATTheor.0046D064
      ;  ASCII "Please Enter Your Password: "
0040172B   .  68 F8874700       PUSH IATTheor.004787F8
00401730   .  E8 D2FBFFFF       CALL IATTheor.00401307
.     .

00401867   .  83C2 01           ADD EDX,1
0040186A   .  8955 FC           MOV DWORD PTR SS:[EBP-4],EDX
.     .

00401872   > \33C0              XOR EAX,EAX
```

```
004016B8   .  C745 FC 00000000  MOV DWORD PTR SS:[EBP-4],0
```



```
004016BF    >  837D FC 03          CMP DWORD PTR SS:[EBP-4],3
```

Checking the counter value in the While loop if the counter does not exceed 3 times of trials yet. While (counter!= 3)

```
004016C3    .  0F84 A9010000       JE IATTheor.00401872
```

If register EAX(counter) = 3 then jump to VA 00401872, which will zeroed the counter again and exit from the routine (XOR EAX,EAX)

```
00401872    > \33C0                XOR EAX,EAX[1]
```

This could be written like this:

```
004016A0:  Starting Routine Address
.      .
004016B8    .  C745 FC 00000000    MOV [LOCAL.counter],0
004016BF    >  837D FC 03          CMP [LOCAL.counter],3
.      .
00401884:  Ending Routine Address
```

```
00401867    .  83C2 01             ADD EDX,1
0040186A    .  8955 FC             MOV DWORD PTR SS:[EBP-4],EDX
```

Otherwise increase the counter by 1 each loop ("*the counter is increased at the end of the while loop*"). Counter++;

This could be defeated in either: changing the initialization counter value to a higher value than the one in the while loop (limit counter), or changing the 3 value in the comparison statement to "-1", and many other ways. To do this just select the line in Olly and press space button and change the value to whatever you want then press ok.

List of methods to defeat the checking counter limitation			
004016B8	.	C745 FC 04000000	MOV DWORD PTR SS:[EBP-4],4
004016BF	>	837D FC FF	CMP DWORD PTR SS:[EBP-4],-1
004016C3	.	0F85 A9010000	JNE IATTheor.00401872
004016C3	.	9090 90909090	NOP : (NO OPERTAION)

Table 4.4 Methods to defeat checking counter limitation

4.5.3.3 Third Stage: Defeat Password Checking Array

✓ Defeat password checking array: for every character of the false entered password there is an equivalent in the right password (hardcoded). So, the program will make a loop for each character and checking if its match or not. Another check for the array size and ... now the full assembly code (Routine) snippet from Olly:

```
004016A0   > \55                PUSH EBP
004016A1   .  8BEC              MOV EBP,ESP
004016A3   .  83EC 70           SUB ESP,70
004016A6   .  53                PUSH EBX
004016A7   .  56                PUSH ESI
004016A8   .  57                PUSH EDI
004016A9   .  8D7D 90           LEA EDI,DWORD PTR SS:[EBP-70]
004016AC   .  B9 1C000000       MOV ECX,1C
004016B1   .  B8 CCCCCCCC       MOV EAX,CCCCCCCC
004016B6   .  F3:AB             REP STOS DWORD PTR ES:[EDI]
004016B8   .  C745 FC 00000000  MOV DWORD PTR SS:[EBP-4],0
004016BF   >  837D FC 03        CMP DWORD PTR SS:[EBP-4],3
004016C3   .  0F84 A9010000     JE IATTheor.00401872
004016C9   .  C745 F8 0D000000  MOV DWORD PTR SS:[EBP-8],0D
004016D0   .  C745 F4 0D000000  MOV DWORD PTR SS:[EBP-C],0D
004016D7   .  A1 BCD04600       MOV EAX,DWORD PTR DS:[46D0BC]
004016DC   .  8945 E4           MOV DWORD PTR SS:[EBP-1C],EAX
004016DF   .  8B0D C0D04600     MOV ECX,DWORD PTR DS:[46D0C0]
004016E5   .  894D E8           MOV DWORD PTR SS:[EBP-18],ECX
004016E8   .  8B15 C4D04600     MOV EDX,DWORD PTR DS:[46D0C4]
004016EE   .  8955 EC           MOV DWORD PTR SS:[EBP-14],EDX
004016F1   .  A0 C8D04600       MOV AL,BYTE PTR DS:[46D0C8]
004016F6   .  8845 F0           MOV BYTE PTR SS:[EBP-10],AL
004016F9   .  C645 D4 00        MOV BYTE PTR SS:[EBP-2C],0
004016FD   .  33C9              XOR ECX,ECX
004016FF   .  894D D5           MOV DWORD PTR SS:[EBP-2B],ECX
00401702   .  894D D9           MOV DWORD PTR SS:[EBP-27],ECX
00401705   .  894D DD           MOV DWORD PTR SS:[EBP-23],ECX
00401708   .  68 F5104000       PUSH IATTheor.004010F5
0040170D   .  68 88D04600       PUSH IATTheor.0046D088
                  ;   ASCII  "Thesis  IAT,  Packer,  Serial  Protection
          Demo"
00401712   .  68 F8874700       PUSH IATTheor.004787F8
00401717   .  E8 EBFBFFFF       CALL IATTheor.00401307
0040171C   .  83C4 08           ADD ESP,8
0040171F   .  8BC8              MOV ECX,EAX
00401721   .  E8 32FBFFFF       CALL IATTheor.00401258
00401726   .  6864D04600        PUSH IATTheor.0046D064
                  ;  ASCII "Please Enter Your Password: "
0040172B   .  68 F8874700       PUSH IATTheor.004787F8
00401730   .  E8 D2FBFFFF       CALL IATTheor.00401307
00401735   .  83C4 08           ADD ESP,8
00401738   .  8D55 D4           LEA EDX,DWORD PTR SS:[EBP-2C]
0040173B   .  52                PUSH EDX
0040173C   .  68 88884700       PUSH IATTheor.00478888
00401741   .  E8 2DF9FFFF       CALL IATTheor.00401073
00401746   .  83C4 08           ADD ESP,8
00401749   .  C745 D0 00000000  MOV DWORD PTR SS:[EBP-30],0
```

```
00401750   .  EB 09              JMP SHORT IATTheor.0040175B
00401752   >  8B45 D0            MOV EAX,DWORD PTR SS:[EBP-30]
00401755   .  83C0 01            ADD EAX,1
00401758   .  8945 D0            MOV DWORD PTR SS:[EBP-30],EAX
0040175B   >  837D D0 0B         CMP DWORD PTR SS:[EBP-30],0B
0040175F   .  0F8F FF000000      JG IATTheor.00401864
00401765   .  8B4D D0            MOV ECX,DWORD PTR SS:[EBP-30]
00401768   .  0FBE540D D4        MOVSX EDX,BYTE PTR SS:[EBP+ECX-2C]
0040176D   .  8B45 D0            MOV EAX,DWORD PTR SS:[EBP-30]
00401770   .  0FBE4C05 E4        MOVSX ECX,BYTE PTR SS:[EBP+EAX-1C]
00401775   .  3BD1               CMP EDX,ECX
00401777   .  75 74              JNZ SHORT IATTheor.004017ED
00401779   .  68 F5104000        PUSH IATTheor.004010F5
0040177E   .  68 4CD04600        PUSH IATTheor.0046D04C
                 ;  ASCII " :Access Approved"
00401783   .  8B55 D0            MOV EDX,DWORD PTR SS:[EBP-30]
00401786   .  8A4415 D4          MOV AL,BYTE PTR SS:[EBP+EDX-2C]
0040178A   .  50                 PUSH EAX
0040178B   .  68 48D04600        PUSH IATTheor.0046D048
00401790   .  68 44D04600        PUSH IATTheor.0046D044
00401795   .  8B4D D0            MOV ECX,DWORD PTR SS:[EBP-30]
00401798   .  51                 PUSH ECX
00401799   .  68 40D04600        PUSH IATTheor.0046D040
0040179E   .  68 30D04600        PUSH IATTheor.0046D030
                 ;  ASCII "Character:"
004017A3   .  68 F8874700        PUSH IATTheor.004787F8
004017A8   .  E8 5AFBFFFF        CALL IATTheor.00401307
004017AD   .  83C4 08            ADD ESP,8
004017B0   .  50                 PUSH EAX
004017B1   .  E8 51FBFFFF        CALL IATTheor.00401307
004017B6   .  83C4 08            ADD ESP,8
004017B9   .  8BC8               MOV ECX,EAX
004017BB   .  E8 76F9FFFF        CALL IATTheor.00401136
004017C0   .  50                 PUSH EAX
004017C1   .  E8 41FBFFFF        CALL IATTheor.00401307
004017C6   .  83C4 08            ADD ESP,8
004017C9   .  50                 PUSH EAX
004017CA   .  E8 38FBFFFF        CALL IATTheor.00401307
004017CF   .  83C4 08            ADD ESP,8
004017D2   .  50                 PUSH EAX
004017D3   .  E8 49FAFFFF        CALL IATTheor.00401221
004017D8   .  83C4 08            ADD ESP,8
004017DB   .  50                 PUSH EAX
004017DC   .  E8 26FBFFFF        CALL IATTheor.00401307
004017E1   .  83C4 08            ADD ESP,8
004017E4   .  8BC8               MOV ECX,EAX
004017E6   .  E8 6DFAFFFF        CALL IATTheor.00401258
004017EB   .  EB 72              JMP SHORT IATTheor.0040185F
004017ED   >  68 F5104000        PUSH IATTheor.004010F5
004017F2   .  68 1CD04600        PUSH IATTheor.0046D01C
                 ;  ASCII " :Access Denied"
004017F7   .  8B55 D0            MOV EDX,DWORD PTR SS:[EBP-30]
004017FA   .  8A4415 D4          MOV AL,BYTE PTR SS:[EBP+EDX-2C]
004017FE   .  50                 PUSH EAX
004017FF   .  68 48D04600        PUSH IATTheor.0046D048
00401804   .  68 44D04600        PUSH IATTheor.0046D044
```

```
00401809   .  8B4D D0            MOV ECX,DWORD PTR SS:[EBP-30]
0040180C   .  51                 PUSH ECX
0040180D   .  68 40D04600        PUSH IATTheor.0046D040
00401812   .  68 30D04600        PUSH IATTheor.0046D030
                  ;  ASCII "Character:"
00401817   .  68 F8874700        PUSH IATTheor.004787F8
0040181C   .  E8 E6FAFFFF        CALL IATTheor.00401307
00401821   .  83C4 08            ADD ESP,8
00401824   .  50                 PUSH EAX
00401825   .  E8 DDFAFFFF        CALL IATTheor.00401307
0040182A   .  83C4 08            ADD ESP,8
0040182D   .  8BC8               MOV ECX,EAX
0040182F   .  E8 02F9FFFF        CALL IATTheor.00401136
00401834   .  50                 PUSH EAX
00401835   .  E8 CDFAFFFF        CALL IATTheor.00401307
0040183A   .  83C4 08            ADD ESP,8
0040183D   .  50                 PUSH EAX
0040183E   .  E8 C4FAFFFF        CALL IATTheor.00401307
00401843   .  83C4 08            ADD ESP,8
00401846   .  50                 PUSH EAX
00401847   .  E8 D5F9FFFF        CALL IATTheor.00401221
0040184C   .  83C4 08            ADD ESP,8
0040184F   .  50                 PUSH EAX
00401850   .  E8 B2FAFFFF        CALL IATTheor.00401307
00401855   .  83C4 08            ADD ESP,8
00401858   .  8BC8               MOV ECX,EAX
0040185A   .  E8 F9F9FFFF        CALL IATTheor.00401258
0040185F   >^ E9 EEFEFFFF        JMP IATTheor.00401752
00401864   >  8B55 FC            MOV EDX,DWORD PTR SS:[EBP-4]
00401867   .  83C2 01            ADD EDX,1
0040186A   .  8955 FC            MOV DWORD PTR SS:[EBP-4],EDX
0040186D   .^ E9 4DFEFFFF        JMP IATTheor.004016BF
00401872   >  33C0               XOR EAX,EAX
00401874   .  5F                 POP EDI
00401875   .  5E                 POP ESI
00401876   .  5B                 POP EBX
00401877   .  83C4 70            ADD ESP,70
0040187A   .  3BEC               CMP EBP,ESP
0040187C   .  E8 CF030200        CALL IATTheor.00421C50
00401881   .  8BE5               MOV ESP,EBP
00401883   .  5D                 POP EBP
00401884   .  C3                 RETN
```

Now, stepping over the code line by line will reveal many important information in how the high level language being translated to low level language. By setting a breakpoint (Toggle) at the beginning of the routine (select `004016A0:` Starting Routine Address line and press F2 in Olly) → Run F9 when the program start after a while, it will stop at the breakpoint point we set→ and then press step over the code with F8 button in order to investigate the registers changing mode till VA 004016C9

```
004016C9   .  C745 F8 0D000000   MOV DWORD PTR SS:[EBP-8], 0D
004016D0   .  C745 F4 0D000000   MOV DWORD PTR SS:[EBP-C], 0D
```

Load the pointer register address (EBP-8) and (EBP-c) with These two values"0D" which are in hexadecimal. They are responsible for array constant size init., $0D \xrightarrow{Decimal} 13$

```
const int k = 13;
const int l = 13;
```

Continue F8

```
004016D7   .  A1 BCD04600        MOV EAX,DWORD PTR DS:[46D0BC]
```

Load the value at address 46D0BC into register EAX which is as showed in the pane window in Olly [0046D0BC]=61687264. The value 61687264 is in hexadecimal in the reverse order (Little Endian Order[11]), now if we convert the hexadecimal value to string: "ahrd" in the reverse order, this is the first four letter of the right password.

Continue F8

```
004016DC   .  8945 E4            MOV DWORD PTR SS:[EBP-1C],EAX
004016DF   .  8B0D C0D04600      MOV ECX,DWORD PTR DS:[46D0C0]
```

The same. Load the value at address 46D0C0 into register ECX which is as showed in the pane window in Olly DS:[0046D0C0]=626E6173. The value 626E6173 is in hexadecimal in the reverse order (Little Endian Order), now if we convert the hexadecimal value to string: "bnas" in the reverse order, this is the second four letter of the right password.

```
004016E5   .  894D E8            MOV DWORD PTR SS:[EBP-18],ECX
004016E8   .  8B15 C4D04600      MOV EDX,DWORD PTR DS:[46D0C4]
```

Same as previously analyzed: DS:[0046D0C4]=697A7A61 $\xrightarrow{String}$ "izza"

Reverse each block of the hexadecimal value:
61687264 → 64726861
626E6173 → 73616E62
697A7A61 → 617A7A69

Concatenate the whole hexadecimal values to big one, we will have the hardcoded right password. (6472686173616E62617A7A69)

Hexa	64	72	68	61	73	61	6E	62	61	7A	7A	69
string	d	r	h	a	s	a	n	b	a	z	z	i

[11] Intel processors store and retrieve data from memory using what is referred to as *little endian order*. This means that the least significant byte of a variable is stored at the lowest address. The remaining bytes are stored in the next consecutive memory positions.

We can change the hexadecimal value to another one which will make the software accept only the newly inserted password as simple as that.

Continue F8 till this line `00401746 . 83C4 08 ADD ESP,8`
Now the software is being loaded into memory and we can enter a false password to check the comparison with the right one. Let's enter a false password "mfm1hdiatuzi" and press enter, Olly will break

Continue F8 till VA `00401765`

```
00401765   .  8B4D D0           MOV ECX,DWORD PTR SS:[EBP-30]
00401768   .  0FBE540D D4       MOVSX EDX,BYTE PTR SS:[EBP+ECX-2C]

Load the first character of the false entered password "m" from the
pointer address 0012FF54 into register EDX

                Stack SS:[0012FF54]=6D ('m')
                EDX=0046D134 (IATTheor.0046D134)

0040176D   .  8B45 D0           MOV EAX,DWORD PTR SS:[EBP-30]
00401770   .  0FBE4C05 E4       MOVSX ECX,BYTE PTR SS:[EBP+EAX-1C]

Load the first character of the right hardcoded password "d" from the
pointer address 0012FF64 into register ECX
                Stack SS:[0012FF64]=64 ('d')
                ECX=00000000
00401775   .  3BD1              CMP EDX,ECX

Compare the registers value EDX & ECX: ECX=00000064 ('d')
                                       EDX=0000006D ('m')

00401777   .  75 74             JNE SHORT IATTheor.004017ED

Jump if not zero (controlled through flag) to VA 004017ED and print on
the screen Access Denied message, otherwise Access Approved message.
In this case the jump will be taken to the bad message because no
match between the two registers.

00401779   .  68 F5104000       PUSH IATTheor.004010F5
0040177E   .  68 4CD04600       PUSH IATTheor.0046D04C
                ;  ASCII " :Access Approved"
.   .   .
.   .   .
.   .   .
004017ED   >  68 F5104000       PUSH IATTheor.004010F5
004017F2   .  68 1CD04600       PUSH IATTheor.0046D01C
                ;  ASCII " :Access Denied"
```

Continue F8 till this line `0040185F >^ E9 EEFEFFFF JMP IATTheor.00401752`

This line will jump back to VA 00401752 to read the next character and applying the same procedure as above till the end of the array.

For defeating the checking password procedure we need to alter the decision maker statement which is in this case at line:

```
00401777    .  75 74            JNZ SHORT IATTheor.004017ED
```

If we change the JNZ equivalent in hexadecimal (75h) to JZ (74h) this will not function perfectly, it will accept as an access approved only the characters which differ than the right password characters.
So, if we enter the right password an access denied message will appear. The best solution is by nopping (no operation (nop: 90h)) this statement and the code will follow directly with no checking. Or make the cmp mnemonic checking itself. We know that the register ECX is holding the right character to compare with the wrong entered one in the register EDX, so if we replace the register EDX with ECX with comparison will always be true (CMP ECX,ECX).

Methods for defeating the password checking procedure			
Original	00401777	. 75 74	JNZ SHORT IATTheor.004017ED
Modified	00401777	. 90 90	nop
Original	00401775	. 3BD1	CMP EDX,ECX
Modified	00401775	. 3BC9	CMP ECX,ECX

Table 4.5 Methods to defeat the password checking procedure

4.5.4 Patching: Static Changes

✓ Patching the executable (static changes)

Apply the changes for counter limitation and password protection as summarized in the steps before.

how to make a permanent change for the modified code, so that the executable will have a new phase of instructions. This is easy to be done using Olly. Right-click→ Copy to executable → All modifications → Copy all → Save file. A dialog box (Save file as) appear give it a name and press save, and that's it.

4.5.5 Code Injection: Tracking the Unbounded

✓ Code Injection for the packed executable

What is Code injection? You don't have the source code only the executable, and you want to insert or add some functionality. This task could be done by searching inside the executable for empty area (left from the compiler alignment) which is usually

denoted by zeros, and then redirect your Entry Point to this location where you want to add your new code. After that jump again to the EP++(...) to make the software work in its normal flow.
This process is tedious, especially if you need to add a lot of code, it requires a very good understanding of the PE structure and mastering in the assembly language. Another way to add new code is by linking the executable to DLL (Dynamic Link Library) to do some function.

In our case we need to do a lot of things, not for the original executable but for the packed one. We can't change the instructions (Counter Limitation, Password Checking) in this mode, because as stated in the packer's theory memory mapping will be different than what we see in the disassembler, we are not yet in the original place and the packer isn't yet unpacked the executable. That's why we need to locate the magic jump to the Original Entry Point and then redirect the execution to our cave.

Steps required for doing this process:

a. Search for padded area of zeros (usually at the end of the file)
b. Locating the magic point: the line which responsible for jumping to the OEP.
c. Instead of entering the OEP, force the flow of the program to jump to our cave.
d. Write the newly inserted code in the empty stomach area.
e. If some lines of code deleted at the redirection step(because of no match between the size of the previous instruction with the JMP instruction) write it at the end of the cave.
f. Redirect the cave to the Magic point++(...)

In our example we need to do some aesthetic modification: insert a message box at the start of the packed program. Configure the changes we've done for the executable in the last stage(Counter Limitation, Password Checking). Note: the packed program (UPXe'd) is the target and not the original one, and that's impose a different scenario as mentioned in the steps required.

✓ a. Search for the zeros zone. Load the packed executable in Olly, scroll down till VA 00481FC7, looks like the following

```
00481F90      00                DB 00
00481F91      00                DB 00
00481F92      00                DB 00
00481F93      00                DB 00
00481F94      00                DB 00
```

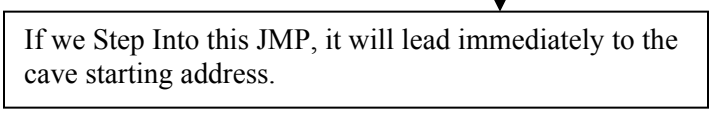
✓ b. Locating the magic point : as you remember when we unpacked this software in the previous part :

```
00481F80    .- E9 4B2FFAFF      JMP IATTheor.00424ED0
```

If we Step Into this JMP, it will lead immediately to the OEP

✓ c. Instead of entering the OEP, redirect it to the cave started at VA 00481FC7

```
00481F80    .- E9 4B2FFAFF          JMP IATTheor.00481FC7
```



If we Step Into this JMP, it will lead immediately to the cave starting address.

✓ d. Write the modified code: this step is very important.

Insert a message box at the beginning of the packed executable. There is a problem concerning API function MessageBoxA which is located inside user32.dll windows library system, because this function isn't loaded in the Imported Functions Table to use it, so we have to insert it manually using any hex editor or automatically using a tool: Code Snippet Creator version 1.05 build 2 by Iczelion. Follow along: start Code Snippet tool → Action → New Target → a dialog box appear → Press Browse → Locate the packed executable (IATTheory.exe) and press open → Press Save → Action → Add New Imports → Browse for user32.dll in the system32 directory → Press Open → In the Import Functions Group select MessageBoxA → Click Add → Press Exit. And that's it.

Imported Functions [Before]			
KERNEL32.DLL			
	LoadLibraryA	ord:0	rva: 00082028
	GetProcAddress	ord:0	rva: 0008202C
	ExitProcess	ord:0	rva: 00082030

Imported Functions [After]			
KERNEL32.DLL			
	LoadLibraryA	ord:0	rva: 00082028
	GetProcAddress	ord:0	rva: 0008202C
	ExitProcess	ord:0	rva: 00082030
user32.dll			
	AppendMenuW	ord:0	rva: 0008401B
	MapDialogRect	ord:0	rva: 0008401F
	MessageBoxA	ord:0	rva: 00084023

Now we can call our newly inserted function.

As I explained the MessageBoxA functionality before, a handle of owner window, address of text in message box, address of text in message box, style of message box.
In assembler this written in the reverse order starting with: Style, text, caption, handle.

Making the ASCII texts for message box text and caption. Select in Olly loads of those zeros line starting at VA 00481F90 and then press CTRL+E and write : "Modified Packed IATTheory" as a caption for the message box, and select another load of zeros and write: "The Software Is Defeated" → press CTRL+A to analyze the code. Now, we start patching the MessageBoxA function in its reverse order. After that we need to insert the defeated protection code. Here is the complete modification:

```
00481F80     EB 45           JMP SHORT IATTheor.00481FC7
00481F82     90              NOP
00481F83     90              NOP
00481F84     90              NOP
00481F85     00              DB 00
00481F86     00              DB 00
00481F87     00              DB 00
00481F88     00              DB 00
00481F89     00              DB 00
00481F8A     00              DB 00
00481F8B     00              DB 00
00481F8C     00              DB 00
00481F8D     00              DB 00
00481F8E     00              DB 00
00481F8F     00              DB 00
00481F90   . 4D 6F 64 69 66>ASCII "Modified Packed "
00481FA0   . 49 41 54 54 68>ASCII "IATTheory",0
00481FAA     00              DB 00
00481FAB   . 54 68 65 20 53>ASCII "The Software Is "
00481FBB   . 44 65 66 65 61>ASCII "Defeated",0
00481FC4     00              DB 00
00481FC5     00              DB 00
00481FC6     00              DB 00
00481FC7   > 6A 00           PUSH 0      ; /Style = MB_OK|MB_APPLMODAL
00481FC9   . 68 AB1F4800     PUSH IATTheor.00481FAB   ; |Title = "The
                                                      Software Is Defeated"
00481FCE   . 68 901F4800     PUSH IATTheor.00481F90   ; |Text =
                                                     "Modified Packed IATTheory"
00481FD3   . 6A 00           PUSH 0                   ; |hOwner = NULL
00481FD5   . E8 9C448E77     CALL user32.MessageBoxA  ; \MessageBoxA
00481FDA     C605 BB164000 >MOV BYTE PTR DS:[4016BB],4
00481FE1     66:C705 771740>MOV WORD PTR DS:[401777],9090
00481FEA    -E9 E12EFAFF     JMP IATTheor.00424ED0
```

∴ The packed executable file (IATTheory.exe) is fully defeated as following:

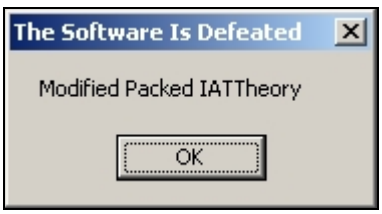


Figure 4.4 MessageBoxA Injected

And the protection on the counter limitation + password checking does no longer exist.

Chapter 5

CASE STUDIES: REVERSING THE INVISIBLE

The pleasure of breaking protections always found to be in the commercial protected applications. Cracking is the "dark art" of defeating, bypassing, or eliminating any kind of copy protection scheme. In its original form, cracking is aimed at software copy protection schemes such as serial-number-based registrations, hardware keys (dongles), and so on. More recently, cracking has also been applied to digital rights management (DRM) technologies, which attempt to protect the flow of copyrighted materials such as movies, music recordings, and books. Unsurprisingly, cracking is closely related to reversing, because in order to defeat any kind of software-based protection mechanism crackers must first determine exactly how that protection mechanism works.

Note for the beginners: *"Chapter 5 case studies are probably not for the learning beginner. They are probably more for the more experienced. So, do not try and follow along, but refer back to the case studies later as you gain experience."*

5.1 Serial Fishing: L0phtCrack v5.02 Victim

Target: LC5 (L0phtCrack) v5.02, the award-winning password audit and recovery tool for Windows and UNIX passwords. This software cost around 500$ for the Administrator version Unlock Code, and it accept three mode of registration (basic, Professional, Administration). Why am I apply this on a commercial software, I know it's illegal but after a search in the specialized media i found this software to be keygenned[12] already.

LC5 provides two critical capabilities to system administrators:

- LC5 helps administrators secure Windows and Unix-authenticated networks through comprehensive auditing of Windows NT, Windows 2000, Windows XP, and Unix user account passwords.
- LC5 recovers Windows and Unix user account passwords to streamline migration of users to another authentication system or to access accounts whose passwords are lost.

Generate keys based on a combination of user input and computer-specific information.

After starting this program, a dialog box appears as in figure 5.1. choose either to continue as a trial version or register to enter the valid unlock code for this software.

[12] *Keygenning* is the process of creating programs that mimic the key-generation algorithm within a protection technology and essentially provide an unlimited number of valid keys, for everyone to use.

Press on Register button and another dialog box as in figure 5.2

Figure 5.1 LC5 Trial Version Startup Window

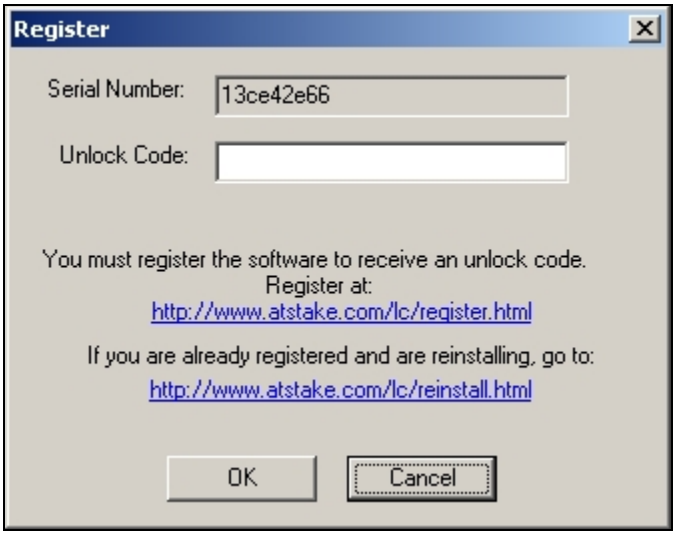


Figure 5.2 LC5 Registration Window

Note: Serial Number: 13ce42e66, this key is generated based on computer-specific information. This number will always change from computer to another. So, the mechanism for the Unlock Code means that a product key will only be valid on the computer on which it was installed—users can't share product keys.
To overcome this problem software pirates use keygen programs that typically contain exact replicas of the serial number generation algorithms in the protected programs.

The next step is to try to type a random values into the text box (Unlock Code: r94e2d) and then clicking the "OK" button produces a message box as in figure 5.3

Figure 5.3 Invalid Unlock Code message.

This is important because if this is indeed a conventional Windows message box, we could use a debugger (Olly) to set a breakpoint on the message box(MessageBoxA) APIs. From there, we could try to reach the code in the program that's telling us that we have a bad serial number. This is a fundamental cracking technique—find the part in the program that's telling you you're unauthorized to run it. Once you're there it becomes much easier to find the actual logic that determines whether you're authorized or not.

We open the program in OllyDbg and we search for MessageBoxA API which is responsible for displaying the Invalid Unlock Code message on the screen. This could be done by typing "bp MessageBoxA" in the command bar in Olly and then press enter as in figure 5.4

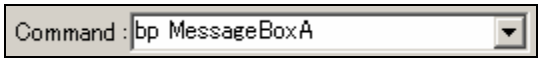


Figure 5.4 Setting a Breakpoint on MessageBoxA API

Press F9 in Olly and follow as previously we did, enter an invalid Unlock Code and then press on "OK" button. Olly at this point will break (stop) executing because of the breakpoint we set. We can check the Stack Window in OllyDbg which reveal a lots of important information from where this message box is being called (the source which will lead us to the registration mechanism). As in figure 5.5

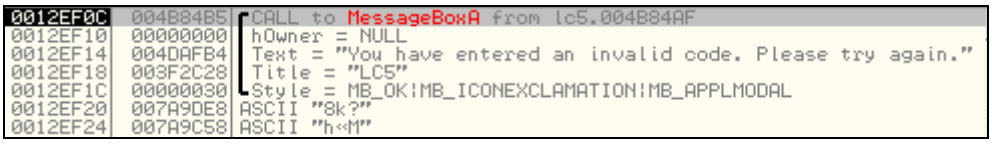


Figure 5.5 Stack Window in OllyDbg

CALL to MessageBoxA from lc5.004B84AF: select this line in stack window and press Enter (Follow in Disassembler), then scroll up till the starting routine VA 004B83E3. and set a Breakpoint F2 (Toggle) at this address (004B83E3). We no longer need the Breakpoint at MessageBoxA function, delete it by typing in the command bar

bc MessageBoxA as in figure 5.6

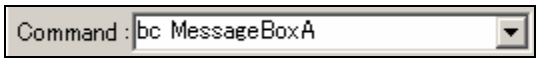


Figure 5.6 How to clear the Breakpoint

Now, if we restart the program inside Olly the bp we set at VA 004B83E3 still activated and that's what we want. Restart LC5.EXE : press CTRL+F2 or press on the icon → press yes when the warning message box appear→ after a while the program restarted and we are ready for the next journey→ press Run (F9)→ and follow the same procedure as above. When we enter the invalid Unlock Code Olly will break at the bp we set at VA 004B83E3 this time→ and step over (F8)→ the invalid message box appear again press on button "OK"→ and continue stepping over the code with F8 shortcut→ this time the dialog box in the figure 5.3 will popup→ press on button "OK' again→ continue with F8 till VA 0044159A. now we are in the determination block of code for the valid Unlock Code. We notice at this VA the register ECX is being loaded with the Serial Number: 13ce42e66 (Registers Window in OllyDbg)

```
0044159A   .  8D4C24 10          LEA ECX,DWORD PTR SS:[ESP+10]
```

Continue F8 till VA 004415A7 where also the register EDX is loaded with the invalid Unlock Code: r94e2d we entered

```
004415A7   .  8D5424 18          LEA EDX,DWORD PTR SS:[ESP+18]
```

At this point in the Pane window and in the Stack window in OllyDbg the real Unlock Code is calculated and appear as an ASCII plain text. (Basic version)

```
Stack address=0012F068, (ASCII "d80bdb20")
EDX=0012F06F
```

How do I know that this is the real Unlock code because of the logic in the analysis steps. For better visualization here is the code for basic, professional, administration algorithm calculation.

```
004415A7   .  8D5424 18          LEA EDX,DWORD PTR SS:[ESP+18]
004415AB   .  52                 PUSH EDX
004415AC   .  50                 PUSH EAX
004415AD   .  E8 D1F70400        CALL lc5.00490D83
004415B2   .  83C4 10            ADD ESP,10
004415B5   .  85C0               TEST EAX,EAX
004415B7   .  75 43              JNZ SHORT lc5.004415FC
004415B9   .  8B07               MOV EAX,DWORD PTR DS:[EDI]
004415BB   .  50                 PUSH EAX           ; /Arg3
004415BC   .  68 E4B14D00        PUSH lc5.004DB1E4 ; |Arg2 = 004DB1E4
                                                    ASCII "Unlock Code"
004415C1   .  68 D4B14D00        PUSH lc5.004DB1D4 ; |Arg1 = 004DB1D4
                                                    ASCII "Registration"
```

```
004415C6   .  8BCE              MOV ECX,ESI       ; |
004415C8   .  899E 94010000     MOV DWORD PTR DS:[ESI+194],EBX ; |
004415CE   .  E8 28700700       CALL lc5.004B85FB ; \lc5.004B85FB
004415D3   .  53                PUSH EBX          ; /Arg3
004415D4   .  53                PUSH EBX          ; |Arg2
004415D5   .  68 74B04D00       PUSH lc5.004DB074 ; |Arg1 = 004DB074
ASCII "You have successfully registered the Basic version of LC5."
004415DA   .  C786 A4010000 0100>MOV DWORD PTR DS:[ESI+1A4],1 ; |
004415E4   .  E8 066F0700       CALL lc5.004B84EF ; \lc5.004B84EF
004415E9   .  33C0              XOR EAX,EAX
004415EB   .  894424 10         MOV DWORD PTR SS:[ESP+10],EAX
004415EF   .  894424 14         MOV DWORD PTR SS:[ESP+14],EAX
004415F3   .  884424 18         MOV BYTE PTR SS:[ESP+18],AL
004415F7   .  E9 9B010000       JMP lc5.00441797
004415FC   >  8B45 00           MOV EAX,DWORD PTR SS:[EBP]
004415FF   .  8D4C24 10         LEA ECX,DWORD PTR SS:[ESP+10]
00441603   .  51                PUSH ECX
00441604   .  50                PUSH EAX
00441605   .  E8 E610FCFF       CALL lc5.004026F0
0044160A   .  8B07              MOV EAX,DWORD PTR DS:[EDI]
0044160C   .  8D5424 18         LEA EDX,DWORD PTR SS:[ESP+18]
00441610   .  52                PUSH EDX
00441611   .  50                PUSH EAX
00441612   .  E8 6CF70400       CALL lc5.00490D83
00441617   .  83C4 10           ADD ESP,10
0044161A   .  85C0              TEST EAX,EAX
0044161C   .  0F85 AA000000     JNZ lc5.004416CC
00441622   .  53                PUSH EBX
00441623   .  8D8C24 28010000   LEA ECX,DWORD PTR SS:[ESP+128]
0044162A   .  E8 319B0000       CALL lc5.0044B160
0044162F   .  8D8C24 24010000   LEA ECX,DWORD PTR SS:[ESP+124]
00441636   .  C68424 80040000 05 MOV BYTE PTR SS:[ESP+480],5
0044163E   .  E8 4D930600       CALL lc5.004AA990
00441643   .  83F8 01           CMP EAX,1
00441646   .  0F85 59010000     JNZ lc5.004417A5
0044164C   .  8B07              MOV EAX,DWORD PTR DS:[EDI]
0044164E   .  50                PUSH EAX    ; /Arg3
0044164F   .  68 E4B14D00       PUSH lc5.004DB1E4; |Arg2 = 004DB1E4
                                                  ASCII "Unlock Code"
00441654   .  68 D4B14D00       PUSH lc5.004DB1D4 ; |Arg1 = 004DB1D4
                                                  ASCII "Registration"
00441659   .  8BCE              MOV ECX,ESI       ; |
0044165B   .  899E 94010000     MOV DWORD PTR DS:[ESI+194],EBX ; |
00441661   .  E8 956F0700       CALL lc5.004B85FB ; \lc5.004B85FB
00441666   .  53                PUSH EBX          ; /Arg3
00441667   .  53                PUSH EBX          ; |Arg2
00441668   .  68 30B04D00       PUSH lc5.004DB030; |Arg1 = 004DB030
ASCII "You have successfully registered the Professional version of
LC5."
0044166D   .  C786 A4010000 0200>MOV DWORD PTR DS:[ESI+1A4],2 ; |
00441677   .  E8 736E0700       CALL lc5.004B84EF  ; \lc5.004B84EF
0044167C   .  33C0              XOR EAX,EAX
0044167E   .  894424 10         MOV DWORD PTR SS:[ESP+10],EAX
00441682   .  894424 14         MOV DWORD PTR SS:[ESP+14],EAX
00441686   .  884424 18         MOV BYTE PTR SS:[ESP+18],AL
0044168A   .  8B8424 94010000   MOV EAX,DWORD PTR SS:[ESP+194]
00441691   .  83C0 F0           ADD EAX,-10
```

```
00441694   .  C68424 80040000 06 MOV BYTE PTR SS:[ESP+480],6
0044169C   .  8D48 0C            LEA ECX,DWORD PTR DS:[EAX+C]
0044169F   .  83CA FF            OR EDX,FFFFFFFF
004416A2   .  F0:0FC111          LOCK XADD DWORD PTR DS:[ECX],EDX;
                                                      LOCK prefix
004416A6   .  4A                 DEC EDX
004416A7   .  85D2               TEST EDX,EDX
004416A9   .  7F 08              JG SHORT lc5.004416B3
004416AB   .  8B08               MOV ECX,DWORD PTR DS:[EAX]
004416AD   .  8B11               MOV EDX,DWORD PTR DS:[ECX]
004416AF   .  50                 PUSH EAX
004416B0   .  FF52 04            CALL DWORD PTR DS:[EDX+4]
004416B3   >  8D8C24 24010000    LEA ECX,DWORD PTR SS:[ESP+124]
004416BA   .  C68424 80040000 04 MOV BYTE PTR SS:[ESP+480],4
004416C2   .  E8 988B0600        CALL lc5.004AA25F
004416C7   .  E9 CB000000        JMP lc5.00441797
004416CC   >  8B45 00            MOV EAX,DWORD PTR SS:[EBP]
004416CF   .  8D4C24 10          LEA ECX,DWORD PTR SS:[ESP+10]
004416D3   .  51                 PUSH ECX
004416D4   .  50                 PUSH EAX
004416D5   .  E8 B60FFCFF        CALL lc5.00402690
004416DA   .  8B07               MOV EAX,DWORD PTR DS:[EDI]
004416DC   .  8D5424 18          LEA EDX,DWORD PTR SS:[ESP+18]
004416E0   .  52                 PUSH EDX
004416E1   .  50                 PUSH EAX
004416E2   .  E8 9CF60400        CALL lc5.00490D83
004416E7   .  83C4 10            ADD ESP,10
004416EA   .  85C0               TEST EAX,EAX
004416EC   .  53                 PUSH EBX
004416ED   .  0F85 7F000000      JNZ lc5.00441772
004416F3   .  8D4C24 38          LEA ECX,DWORD PTR SS:[ESP+38]
004416F7   .  E8 649A0000        CALL lc5.0044B160
004416FC   .  8D4C24 34          LEA ECX,DWORD PTR SS:[ESP+34]
00441700   .  C68424 80040000 07 MOV BYTE PTR SS:[ESP+480],7
00441708   .  C78424 A8000000 01>MOV DWORD PTR SS:[ESP+A8],1
00441713   .  E8 78920600        CALL lc5.004AA990
00441718   .  83F8 01            CMP EAX,1
0044171B   .  0F85 9B000000      JNZ lc5.004417BC
00441721   .  8B07               MOV EAX,DWORD PTR DS:[EDI]
00441723   .  50                 PUSH EAX           ; /Arg3
00441724   .  68 E4B14D00        PUSH lc5.004DB1E4 ; |Arg2 = 004DB1E4
                                                    ASCII "Unlock Code"
00441729   .  68 D4B14D00        PUSH lc5.004DB1D4 ; |Arg1 = 004DB1D4
                                                    ASCII "Registration"
0044172E   .  8BCE               MOV ECX,ESI        ; |
00441730   .  899E 94010000      MOV DWORD PTR DS:[ESI+194],EBX ; |
00441736   .  E8 C06E0700        CALL lc5.004B85FB ; \lc5.004B85FB
0044173B   .  53                 PUSH EBX           ; /Arg3
0044173C   .  53                 PUSH EBX           ; |Arg2
0044173D   .  68 E8AF4D00        PUSH lc5.004DAFE8 ; |Arg1 = 004DAFE8
ASCII "You have successfully registered the Administrator version of
      LC5."
00441742   .  C786 A4010000 0300>MOV DWORD PTR DS:[ESI+1A4],3 ; |
0044174C   .  E8 9E6D0700        CALL lc5.004B84EF ; \lc5.004B84EF
00441751   .  33C0               XOR EAX,EAX
00441753   .  894424 10          MOV DWORD PTR SS:[ESP+10],EAX
00441757   .  894424 14          MOV DWORD PTR SS:[ESP+14],EAX
```

```
0044175B   .  8D4C24 34          LEA ECX,DWORD PTR SS:[ESP+34]
0044175F   .  884424 18          MOV BYTE PTR SS:[ESP+18],AL
00441763   .  C68424 80040000 04 MOV BYTE PTR SS:[ESP+480],4
0044176B   .  E8 C01CFDFF        CALL lc5.00413430
00441770   .  EB 25              JMP SHORT lc5.00441797
00441772   >  33C9              XOR ECX,ECX                    ; |
00441774   .  894C24 14          MOV DWORD PTR SS:[ESP+14],ECX ; |
00441778   .  53                 PUSH EBX                       ; |Arg2
00441779   .  894C24 1C          MOV DWORD PTR SS:[ESP+1C],ECX ; |
0044177D   .  68 B4AF4D00        PUSH lc5.004DAFB4 ; |Arg1 = 004DAFB4
ASCII "You have entered an invalid code. Please try again."
00441782   .  884C24 24          MOV BYTE PTR SS:[ESP+24],CL  ; |
00441786   .  E8 646D0700        CALL lc5.004B84EF   ; \lc5.004B84EF
```

As you notice the block of code for Basic, Professional, Administration calculations is the same. So, if you continue stepping over the code with F8 you'll get the real Unlock code for Professional and Administration version. As in the following:

Professional version: `8deeb68b`

```
Stack address=0012F068, (ASCII "8deeb68b")
EDX=0012F06F
```

Administration version: `27d63d04`

```
Stack address=0012F068, (ASCII "27d63d04")
EDX=0012F06F
```

Pane Window in OllyDbg

The method I follow to fish the serial is little bit complicated. There are many other methods which is more simple and fast by searching for the String reference and set a breakpoint at starting routine address and follow as above. But what if the strings are encrypted you'll end up with nothing in this way.

5.2 Patching The EXE: **8085 Simulator IDE v2.35 Victim**

Target: 8085 Simulator IDE v2.35 at http://www.oshonsoft.com/. 8085 Simulator IDE is powerful application that supplies 8085 educators and developers with user-friendly graphical development environment for Windows with integrated BASIC compiler, assembler, simulator, debugger and disassembler for Intel 8085 8-bit microprocessor.

While I was studying the Microprocessor Architecture and Organization Course, I needed a software for simulating the Intel 8085 8-bit microprocessor in order to get more acquainted with the assembly language. I like to work with 32-bit not 8-bit ASM.

I studied the behavior of this software and I found that there is no place to enter a user name & serial number to activate the software as fully function as usual in the

trial applications. It could be registered as Personal, Commercial, Educational, Site, Institution License.

Limitations:

- ❖ You can start the program only 30 times.
- ❖ Session duration 120 minutes after that the soft. exit automatically.
- ❖ Nag Screen at startup inform us that we need to register the Software.

This is the nag screen figure 5.7 when we start the program

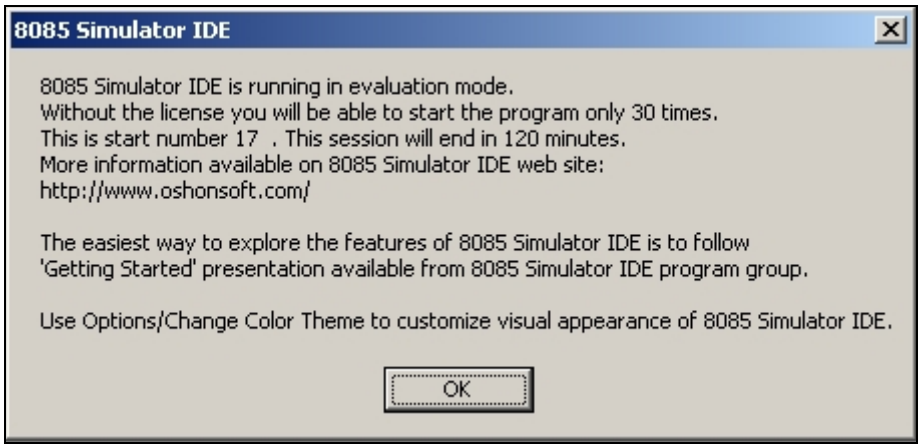


Figure 5.7 8085 Simulator IDE nag screen

Now, if we exceed the 30 times trials, the program will not start anymore stating that as in figure 5.8:

Figure 5.8 8085 Simulator IDE expired version Nag Screen

This software is programmed using Visual Basic Programming Language and that's clear from the library it uses. All VB programs relies on an external Dll (MSVBVM60.DLL for VB version 6.0 and similar DLL's for the other versions) which implements all the language APIs and also the event dispatchers: the result is that a VB app takes almost all its time inside this Dll or generally goes into it and out of it very

frequently. This is a by-design behavior: being VB a very high language all the VB APIs are implemented into MSVBVM60.DLL. The only thing, almost, that the application owns are the events handlers, used as callbacks from the VB Dll to answer to specific events/messages. The rest of a compiled VB application are the resources, the variables and the functions used to associate events-handlers. There are decompilers for Visual Basic applications which help in understanding the structure of the compiled program. But I will not get into this method, rather using OllyDbg features.(check Load the program into Olly and it's obvious from the first two lines : VB application)

```
00405158 > $  68 F0554000        PUSH QWE.004055F0 ;  ASCII "VB5!6&*"
0040515D   .  E8 EEFFFFFF        CALL <JMP.&MSVBVM60.#100>
```

Referenced Text Strings Method

Load the program in Olly and do a search for referenced text strings→ Right-click in the code workplace→ Search for→ All referenced text strings. Now, we have to search for the strings "`The special evaluation period has expired.`", which will lead us to the main routine responsible for this behavior. Right-click in the Text strings referenced in 8085simu:.txt window→ Search for text→ Uncheck Case sensitive check box and check Entire scope and write in the edit box: `The special evaluation period has expired.` mode → Press on "OK" button→ Press CTRL+L twice and bingo as in the following:

```
Address=004A2B3C
Disassembly=MOV EDX,8085simu.0041AB04
Text string=UNICODE "The special evaluation period has expired."
```

double click on the string and we are here:

```
004A2861   .  6A 03              PUSH 3
004A2863   .  FF15 2C104000      CALL DS:[<&MSVBVM60.__vbaFreeVarLis>;
                                          MSVBVM60.__vbaFreeVarList
004A2869   .  83C4 10            ADD ESP,10
004A286C   .  FF15 30104000      CALL DWORD PTR
                  DS:[<&MSVBVM60.__vbaEnd>]      ;  MSVBVM60.__vbaEnd
004A2872   >  A1 44D04C00        MOV EAX,DWORD PTR DS:[4CD044]
004A2877   .  85C0               TEST EAX,EAX
004A2879   .  0F8E 67030000      JLE 8085simu.004A2BE6
004A287F   .  50                 PUSH EAX
004A2880   .  E8 0B170000        CALL 8085simu.004A3F90
004A2885   .  8945 E4            MOV DWORD PTR SS:[EBP-1C],EAX
004A2888   .  E8 F31C0000        CALL 8085simu.004A4580
004A288D   .  3B45 E4            CMP EAX,DWORD PTR SS:[EBP-1C]
004A2890   .  0F8F A6020000      JG 8085simu.004A2B3C
004A2896   .  8B45 E0            MOV EAX,DWORD PTR SS:[EBP-20]
004A2899   .  85C0               TEST EAX,EAX
004A289B   .  0F8C 9B020000      JL 8085simu.004A2B3C
004A28A1   .  BA C0AB4100        MOV EDX,8085simu.0041ABC0;
UNICODE "8085 Simulator IDE is running in special evaluation mode."
004A28A6   .  8D4D D8            LEA ECX,DWORD PTR SS:[EBP-28]
004A28A9   .  FF15 B0114000      CALL DWORD PTR
```

```
DS:[<&MSVBVM60.__vbaStrCopy>];  MSVBVM60.__vbaStrCopy
. . .
. . .
. . .
004A2BE6    > \66:837D EC FF      CMP WORD PTR SS:[EBP-14],0FFFF
004A2BEB    .  0F85 9B020000      JNZ 8085simu.004A2E8C
. . .
. . .
. . .
004A2E8C    > \8B45 E0            MOV EAX,DWORD PTR SS:[EBP-20]
004A2E8F    .  8B0D 38D04C00      MOV ECX,DWORD PTR DS:[4CD038]
004A2E95    .  3BC1               CMP EAX,ECX
004A2E97       0F8E 72010000      JLE 8085simu.004A300F
004A2E9D    .  BA 48AF4100        MOV EDX,8085simu.0041AF48;  UNICODE
                                "The evaluation period has expired."
004A2EA2    .  8D4D D8            LEA ECX,DWORD PTR SS:[EBP-28]
004A2EA5    .  FF15 B0114000      CALL DWORD PTR
             DS:[<&MSVBVM60.__vbaStrCopy>]  ;  MSVBVM60.__vbaStrCopy
004A2EAB    .  8B4D D8            MOV ECX,DWORD PTR SS:[EBP-28]
004A2EAE    .  51                 PUSH ECX
004A2EAF    .  68 B8574100        PUSH 8085simu.004157B8 ;  UNICODE ""
```

Explanation: the code at VA 004A2879 (Jump if less than or equal) is the key for deciding whether to continue in the trial mode below 30 times or not (upon testing the register EAX), but also this will be tested again at VA 004A2BE6 if we change the JLE to JMP this will not solve the problem. The best solution is at VA 004A2E97 by forcing the program to always jump to VA 004A300F (JMP 004A300F). Now we have unlimited times of trials.

To defeat the nag screen. Visual Basic API's is differs than the C++ concerning MessageBox calling. VB programs use rtcMsgBox API for displaying a message on the screen. To bypass the nag screen (figure 5.7): set a bp at this API (go to command bar in Olly and type bp rtcMsgBox) → Run F9 Olly→ after a while Olly will break→ go to the Stack window in Olly and press Enter (Follow in Disassembler) at this line

```
0012FBD4    004A2FEC RETURN to 8085simu.004A2FEC from
                                MSVBVM60.rtcMsgBox
```

Then go to VA 004A3416 where this message is being called as in this line:

```
004A3416    .  FF15 98104000      CALL DWORD PTR DS:[<&MSVBVM60.#595>]
                                  ;  MSVBVM60.rtcMsgBox
```

Nopping the call at this line will bypass the nag screen

Session duration 120 minutes: the same procedure as above search for the text string: This evaluation session is finished and double click.

```
00442259     /75 68              JNZ SHORT 8085simu.004422C3
0044225B   . |8D55 AC            LEA EDX,DWORD PTR SS:[EBP-54]
0044225E   . |8D4D DC            LEA ECX,DWORD PTR SS:[EBP-24]
00442261   . |897D C4            MOV DWORD PTR SS:[EBP-3C],EDI
00442264   . |8975 BC            MOV DWORD PTR SS:[EBP-44],ESI
00442267   . |897D D4            MOV DWORD PTR SS:[EBP-2C],EDI
0044226A   . |8975 CC            MOV DWORD PTR SS:[EBP-34],ESI
0044226D   . |C745 A4 28D04C00   MOV DWORD PTR SS:[EBP-5C]
                                          ,8085simu.004CD028
00442274   . |C745 9C 08400000   MOV DWORD PTR SS:[EBP-64],4008
0044227B   . |C745 B4 F8514100   MOV DWORD PTR SS:[EBP-4C],
8085simu.004151F8   ;  UNICODE "This evaluation session is finished."

00442282   . |C745 AC 08000000   MOV DWORD PTR SS:[EBP-54],8
00442289   . |FF15 F8114000      CALL DWORD PTR
      DS:[<&MSVBVM60.__vbaVarDup>]   ;  MSVBVM60.__vbaVarDup
0044228F   . |8D4D BC            LEA ECX,DWORD PTR SS:[EBP-44]
00442292   . |8D55 CC            LEA EDX,DWORD PTR SS:[EBP-34]
00442295   . |51                 PUSH ECX
00442296   . |8D45 9C            LEA EAX,DWORD PTR SS:[EBP-64]
00442299   . |52                 PUSH EDX
0044229A   . |50                 PUSH EAX
0044229B   . |8D4D DC            LEA ECX,DWORD PTR SS:[EBP-24]
0044229E   . |53                 PUSH EBX
0044229F   . |51                 PUSH ECX
004422A0   . |FF15 98104000      CALL DWORD PTR DS:[<&MSVBVM60.#595>]
                                    ;  MSVBVM60.rtcMsgBox
004422A6   . |8D55 BC            LEA EDX,DWORD PTR SS:[EBP-44]
004422A9   . |8D45 CC            LEA EAX,DWORD PTR SS:[EBP-34]
004422AC   . |52                 PUSH EDX
004422AD   . |8D4D DC            LEA ECX,DWORD PTR SS:[EBP-24]
004422B0   . |50                 PUSH EAX
004422B1   . |51                 PUSH ECX
004422B2   . |6A 03              PUSH 3
004422B4   . |FF15 2C104000      CALL DWORD PTR
      DS:[<&MSVBVM60.__vbaFreeVarLis>;  MSVBVM60.__vbaFreeVarList
004422BA   . |83C4 10            ADD ESP,10
004422BD   . |FF15 30104000      CALL DWORD PTR
            DS:[<&MSVBVM60.__vbaEnd>]      ;  MSVBVM60.__vbaEnd
004422C3   > \895D FC            MOV DWORD PTR SS:[EBP-4],EBX
```

This also very easy to defeat just change the JNZ (Jump if not zero) to JMP (Jump) which will bypass the API `MSVBVM60.__vbaEnd` at VA 004422BD because this API terminates the software.

Target Patches Needed To Defeat The Following		
Original (Trials 30)		
004A2E97	0F8E 72010000	JLE 8085simu.004A300F
Modified (Trials 30)		
004A2E97	/E9 73010000	JMP 8085simu.004A300F
Original (Session duration 120 minutes)		
00442259	/75 68	JNZ SHORT 8085simu.004422C3

Modified (Session duration 120 minutes)			
00442259	/EB 68	JMP SHORT 8085simu.004422C3	
Original (Nag Screen)			
004A3416	. FF15 98104000	CALL DWORD PTR DS:[<&MSVBVM60.#595>]	; MSVBVM60.rtcMsgBox
Modified (Nag Screen)			
004A3416	. 909090909090	NOP	

5.3 Keygenning the KeygenMe: Reengineering the Ripped Algorithm

This is one of the most important topics related to software cracking. The purpose of the keygen is to provide an intellectual challenge to crackers. This is not a cracking demonstration, actually in this case study; I'll explain everything step by step: how to find the serial generation algorithm and how to make a keygen out of it, translating the low level instructions to more readable in high level language(C++). Why am I apply this technique on a special program called KeygenMe (Google it with keyword "KeygenMe", and you'll found many of this KeygenMe available on the net on several RE websites) because it would be illegal and immoral at the same time if I applied on commercial software.

Now let' go beyond the beautifulness of the GUI, and seeking the invisible secretes in how the algorithm work. I will take as an example the keygenMe : KeyGen-me#1 by devilz. When we start it a nice screen appear with two edit boxes: one for the user name and another for the serial number as in figure 5.9

Figure 5.9 Keygen Screen

The Serial is calculated in conjunction with the Name. So, you cannot enter a valid serial with different name. (Parallel combination). If we enter an invalid serial a message box appears telling us that: (Fatal Error: Bad boy, Read more and more Tuts man !!). How do we locate the code which is responsible for this behavior? As we did in the previous analysis, follow the same procedures using the MessageBox API technique (setting a Breakpoint and you are in the game) or search for the referenced strings method.

We will enter as an example, Name: dradeldabou and Serial: r029f4 in order to calculate the valid serial for this name (dradeldabou).

And the following is the snippet assembly code from OllyDbg:

```
004010E2    .  6A 0C              PUSH 0C      ; /Count = C (12.)
```

// Maximum entered characters in the Name text box is 12 including the null character at the end of the array ('\0').

. . .

```
004010F5    . /75 19              JNZ SHORT KeyGen-m.00401110
004010F7    .  6A 00              PUSH 0 ; /Style = MB_OK|MB_APPLMODAL
004010F9    .  68 96304000        PUSH KeyGen-m.00403096
                                     ; |Title = "Fill in the blank"
004010FE    .  68 A8304000        PUSH KeyGen-m.004030A8
                                     ; |Text = "The name please !!!"
00401103    .  6A 00              PUSH 0   ; |hOwner = NULL
00401105    .  E8 18010000        CALL <JMP.&USER32.MessageBoxA>
                                     ; \MessageBoxA
```

// This message box at VA 00401105 popup when the edit box for the Name is empty

```
0040110A    .  C9                 LEAVE
0040110B    .  C2 1000            RETN 10
0040110E    .  EB 2F              JMP SHORT KeyGen-m.0040113F
00401110    > \6A 0C              PUSH 0C    ; /Count = C (12.)
00401112    .  68 80334000        PUSH KeyGen-m.00403380
                                     ; |Buffer = KeyGen-m.00403380
00401117    .  68 C8000000        PUSH 0C8  ; |ControlID = C8 (200.)
0040111C    .  FF75 08            PUSH DWORD PTR SS:[EBP+8]  ; |hWnd
0040111F    .  E8 F2000000        CALL <JMP.&USER32.GetDlgItemTextA>
                                     ; \GetDlgItemTextA
00401124    .  0BC0               OR EAX,EAX
00401126    .  75 17              JNZ SHORT KeyGen-m.0040113F
00401128    .  6A 00              PUSH 0  ; /Style = MB_OK|MB_APPLMODAL
0040112A    .  68 BC304000        PUSH KeyGen-m.004030BC
                                     ; |Title = "Fill in the blank"
0040112F    .  68 CE304000        PUSH KeyGen-m.004030CE
                                     ;  |Text  =  "The  serial  please
!!!"
00401134    .  6A 00              PUSH 0   ; |hOwner = NULL
00401136    .  E8 E7000000        CALL <JMP.&USER32.MessageBoxA>
                                     ; \MessageBoxA
```

// This message box at VA 00401136 popup when the edit box for the Serial is empty

```
0040113B    .  C9                 LEAVE
0040113C    .  C2 1000            RETN 10
0040113F    >  6A 0C              PUSH 0C    ; /Count = C (12.)
00401141    .  68 80334000        PUSH KeyGen-m.00403380
                                     ; |Buffer = KeyGen-m.00403380
00401146    .  6A 64              PUSH 64    ; |ControlID = 64 (100.)
00401148    .  FF75 08            PUSH DWORD PTR SS:[EBP+8] ; |hWnd
0040114B    .  E8 C6000000        CALL <JMP.&USER32.GetDlgItemTextA>
                                     ; \GetDlgItemTextA
```

This is the algorithm for generating the equivalent Serial for the entered Name. To get comfortable with each line of code set a Breakpoint at VA 00401150 and step over the Code with F8 in OllyDbg (don't forget to enter the Name: dradeldabou and Serial:r029f4), and you'll notice how registers values varies from line to line in accordance with the calculations.

```
00401150    .  33D2             XOR EDX,EDX
00401152    .  33DB             XOR EBX,EBX
00401154    .  33C9             XOR ECX,ECX
00401156    .  33C0             XOR EAX,EAX
00401158    .  BE 80334000      MOV ESI,KeyGen-m.00403380
0040115D    >  8A1C31           MOV BL,BYTE PTR DS:[ECX+ESI]
00401160    .  03C3             ADD EAX,EBX
00401162    .  41               INC ECX
00401163    .  80FB 00          CMP BL,0
00401166    .^ 75 F5            JNZ SHORT KeyGen-m.0040115D
00401168    .  BA 28000000      MOV EDX,28
0040116D    .  F7E2             MUL EDX
0040116F    .  83C0 19          ADD EAX,19
```

5.3.1 Serial Generator Algorithm: Analyses

A Fully Descriptive Analyses for the Serial Generator Algorithm

```
00401150    .  33D2             XOR EDX,EDX
00401152    .  33DB             XOR EBX,EBX
00401154    .  33C9             XOR ECX,ECX
00401156    .  33C0             XOR EAX,EAX
```

These four lines of code will XOR'ed the registers EDX, EBX, ECX, EAX to 0: Initialize registers EDX == 0, EBX == 0, ECX == 0, EAX == 0

```
00401158    .  BE 80334000      MOV ESI,KeyGen-m.00403380
```

The Name entered will be loaded into register ESI: ESI == dradeldabou

```
0040115D    >  8A1C31           MOV BL,BYTE PTR DS:[ECX+ESI]
```

Load the first character from the Name entered into the register BL (8-bit). In this case BL == 'd' == 64h, everything manipulated as Hexa.
h: denotes hexadecimal

```
00401160    .  03C3             ADD EAX,EBX
```

Add the value in the source register EBX into destination register EAX and save the result into EAX (32-bit). In this case

EAX == EAX + EBX == 00000000 + 00000064 == 00000064

```
00401162    .  41              INC ECX
```

Increments register ECX by 1. An indicator to know how many characters are in the Name entered (counter). ECX == 1

```
00401163    .  80FB 00         CMP BL,0
```

Compare register BL (8-bit) with value '0': the end of the array

```
00401166    .^ 75 F5           JNZ SHORT KeyGen-m.0040115D
```

Jump if not Zero; Jump if BL is not Zero yet at VA 0040115D (which is not zero in this case). So, this will make a loop for reading the next character till BL == 0 the end of the array ('\0') (entered Name). It will loops 12 times (reading the full Name),the last loop is for the ending of the array with BL == 0

Note: in the next steps, I will not explain everything as above only what the registers are holding in each loop.

```
Loop [2]   :   BL  == 'r' == 72h
               EAX == EAX + EBX == 00000064 + 00000072 == 000000D6
               ECX == 2
               BL  != 0

Loop [3]   :   BL  == 'a' == 61h
               EAX == EAX + EBX == 000000D6 + 00000061 == 00000137
               ECX == 3
               BL  != 0
```

```
Loop [4]   :   BL  == 'd' == 64h
               EAX == EAX + EBX == 00000137 + 00000064 == 0000019B
               ECX == 4
               BL  != 0
```

```
Loop [5]   :   BL  == 'e' == 65h
               EAX == EAX + EBX == 0000019B + 00000065 == 00000200
               ECX == 5
               BL  != 0
```

```
Loop [6]   :   BL  == 'l' == 6Ch
               EAX == EAX + EBX == 00000200 + 0000006C == 0000026C
               ECX == 6
               BL  != 0
```

```
Loop [7]   :   BL  == 'd' == 64h
               EAX == EAX + EBX == 0000026C + 00000064 == 000002D0
               ECX == 7
               BL  != 0
```

```
Loop [8]   :   BL  == 'a' == 61h
               EAX == EAX + EBX == 000002D0 + 00000061 == 00000331
               ECX == 8
               BL  != 0
```

```
Loop [9]   :   BL  == 'b' == 62h
               EAX == EAX + EBX == 00000331 + 00000062 == 00000393
               ECX == 9
               BL  != 0
```

```
Loop [10]  :   BL  == 'o' == 6Fh
               EAX == EAX + EBX == 00000393 + 0000006F == 00000402
               ECX == 10
               BL  != 0
```

```
Loop [11]  :   BL  == 'u' == 75h
               EAX == EAX + EBX == 00000402 + 00000075 == 00000477
               ECX == 11
               BL  != 0
```

```
Loop [12]  :   BL  == '0' == 00h
               EAX == EAX + EBX == 00000477 + 00000000 == 00000477
               ECX == 12
               BL  == 0
```

```
00401168   .  BA 28000000        MOV EDX,28
```

Load the register EDX with 28h: EDX == 00000028h

```
0040116D   .  F7E2               MUL EDX
```

Multiply the value in the register EDX with The value in the accumulator EAX and save the result into EAX.

EAX == EAX * EDX == 00000477 * 00000028 == 0000B298

```
0040116F   .  83C0 19            ADD EAX,19
```

Add 00000019 to the value in the register EAX and save the result into EAX.

EAX == EAX + 19 == 0000B298 + 00000019 == 0000B2B1

Now, what do you think the next step is? We have in the final calculation EAX == B2B1. of course we must converted to decimal

$$B2B1 \xrightarrow{Decimal} 45745$$

That's it our valid Serial number is 45745 for the entered name dradeldabou. As in Figure 5.10

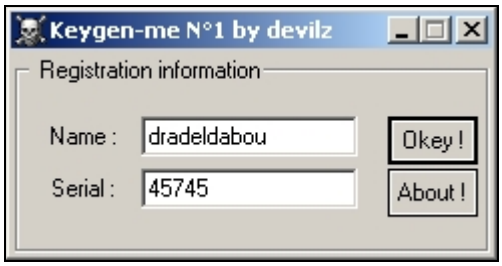


Figure 5.10 Valid Entered Serial

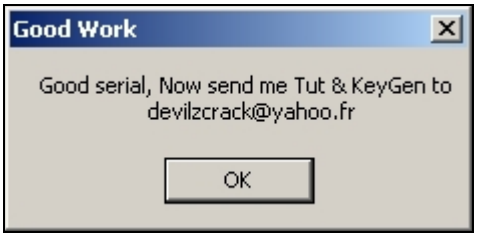


Figure 5.11 Congratulation Message

5.3.2 Serial Generator Algorithm: C++ Translation

This is the equivalent in C++ for the ripped serial generator from the KeygenMe, which will simulate the original one for any entered name.

```
#include <iostream>
using namespace std;
int main()

{
            float EAX = 0; // XOR EAX,EAX
            float EBX = 0; // XOR EBX,EBX
            int   ECX = 0; // XOR ECX,ECX
            float EDX = 0; // XOR EDX,EDX


            const int Size = 12; // PUSH 0C ; Count = C (12.)

            char Name[Size];

            cout<< "Please Enter Your Name (Maximum 11): ";

            cin.getline(Name,Size); // MOV ESI,KeyGen-m.00403380

            for ( ECX=0; Name[ECX] != '\0'; ECX++)
            {

                  EBX = Name[ECX]; // MOV BL,BYTE PTR DS:[ECX+ESI]
                  EAX = EAX + EBX; // ADD EAX,EBX

            }

            EDX = 0x28; // MOV EDX,28
```

```
            EAX = EAX * EDX; // MUL EDX
            EAX = EAX + 0x19; // ADD EAX,19

            cout <<dec<<"Your Serial Number Is: "<<EAX;

            return 0;
}
```

5.4 Deciphering The Algorithm

Target: CoffeeCup Applet Password Wizard v3.0 Registered Version from http://www.coffeecup.com. Its software which will allows you to add a simple password protection to your site, with customization of usernames and passwords, links to different forwarding site one for "access approved" and another for "access denied" permission for each user. It's very easy to implement such a password schema to your site but it's weakness in the encryption algorithm lies behind the simplicity. You can implement the protection either by using Java or Flash mode and the two are very dangerous to protect your site with such software, you will see later why.

Goals: Decrypting the algorithm step by step for the Password Wizard with Java & Flash (Algorithm Decryption with ASCII Plain Text Search Approach). I will decrypt the Java implementation with DJ Java Decompiler Software in the first part of the reversing and the Flash implementation with 010 Editor "Hex Editor" in the second part of this tutorial.

5.4.1 Target & Tools Description

- Applet Password Wizard v3.0

Upon starting up the software you'll be asked to choose between the Java or Flash mode as in figure 5.12

Figure 5.12 CoffeeCup Startup Screen

Studying the program behavior is very important to know the limitations and the purposes of its functionalities.

- *DJ Java Decompiler v3.7.7.81*: With DJ Java Decompiler you can decompile java CLASS files and save it in text or other format. It's simple and easy. DJ Java

Decompiler is decompiler and disassembler for Java that reconstructs the original source code from the compiled binary CLASS files (for example Java applets). DJ Java Decompiler is able to decompile complex Java applets and binaries, producing accurate source code. It's a must to have it in your lab tools.

In Java analysis, I'm going to use this great software to decompile the Java Applet Class to its original high level language construction and then decoding the flow of statements step by step in order to reassemble the encoding of the password protection routine.

- *010 Editor V2.0.2*: It's a professional Hex Editor designed to quickly and easily edit the contents of binary files. You can use any hex editor but I advice you to give it a try and discover its potential especially it's script language for automation in the binary analysis. (used in the second approach)

5.4.2 Java Reversing Approach

The flow of the analysis will be applied as live demonstration by taking an example of how the software and the Applet Java Class will interact with each other. Before start explaining anything: Run the software -> Choose "Click Here to Use Java" as in figure 5.12. So, now we are inside the game but even though not yet there is much more to discover. We will follow a step by step procedure to create an example:

1. File-> New Applet... ("Applet Preview" screen appears as a preview of how the system will look like). Figure 5.13

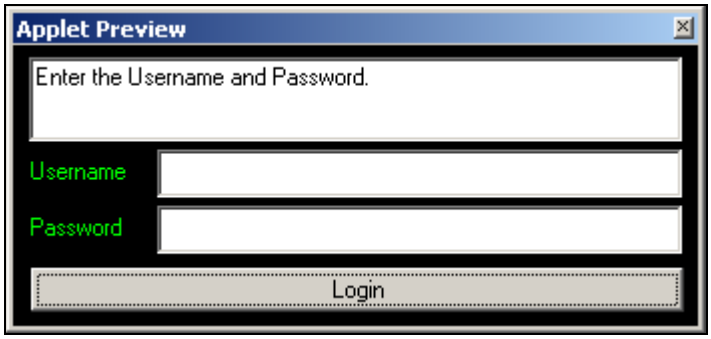


Figure 5.13 Applet Preview

1. General Tab: Startup configuration (default values), in the "Link" section (Edit Box) write http://www.themutable.com as a default value. Figure 5.14

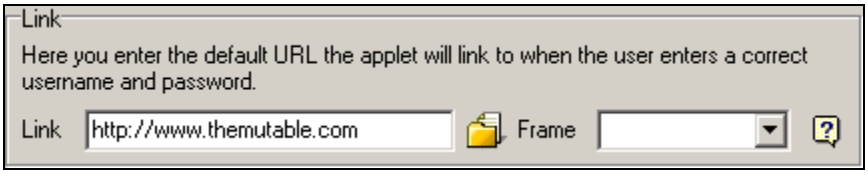


Figure 5.14 General Tab

3. Login Tab: everything is clear. Figure 5.15

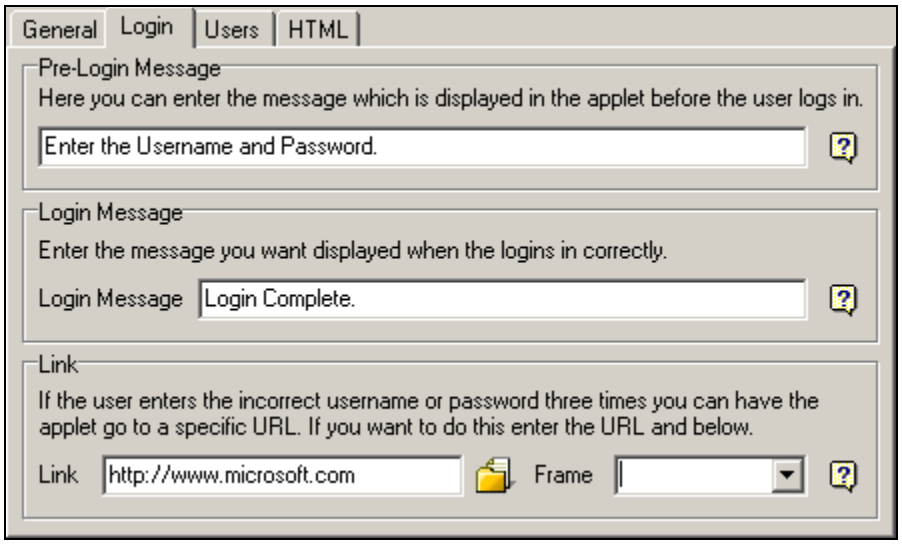


Figure 5.15 Login Tab settings

4. Users tab: This is the important area where you add a new user with name, password, and access approved forward link. Click on -> New User and enter the following value as in figure 5.16.

Figure 5.16 User Tab Properties

5. HTML tab: Here comes the HTML required for this Applet Java Class to work. Figure 5.17

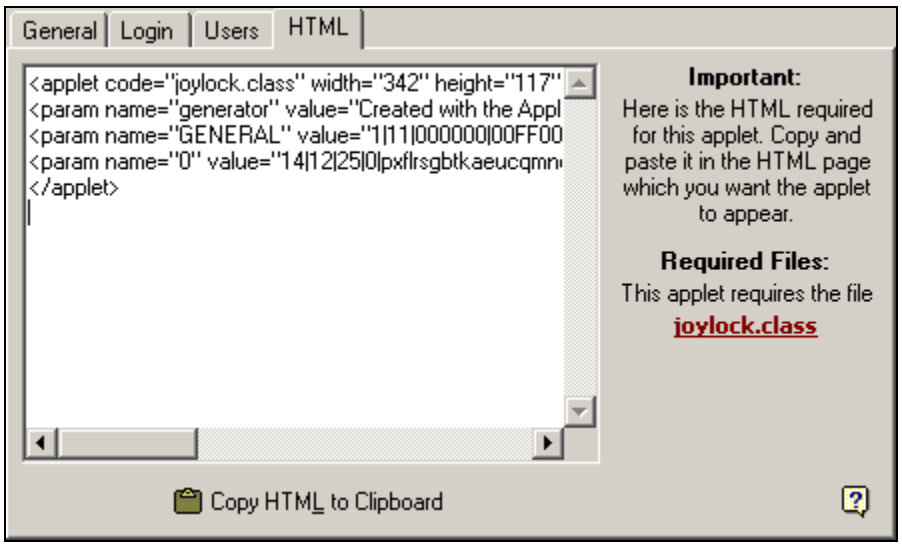


Figure 5.17 HTML Required Parameters Tab

Here is the HTML required for this applet to work in the edit box. by copying and pasting these HTML lines, the connection between the username and password implementation will be made with the joylock.class file which handle the decryption routine inside it in order to decode the param name="0" ... the first username and password. The fie joylock.class must be in the same folder where the page being implemented is. And this file is our target in this session with combination with Figure 5.18

```
<applet code="joylock.class" width="342" height="117">
<param name="generator" value="Created with the Applet Password Wizard
3.0 www.coffeecup.com">
<param name="GENERAL"
value="1|11|000000|00FF00|pbihxzuqjkmcnalwsvfregytdodxxa://ppp.xdukgxnbou
.lzk| |Login Complete.|Enter the Username and
Password.|http://www.microsoft.com| |">
<param name="0"
value="14|12|25|0|ugiceqsrvzothxbmajlfwpkdnyayxehdkylhks66pqswkicgm798mll
v://uuu.lmepalqose.dkp">
</applet>
```

Figure 5.18 HTML required

```
<applet code="joylock.class" width="342" height="117">
```

Startup: Loading the joylock.class file this will be our next victim

```
<param name="generator" value="Created with the Applet Password Wizard 3.0
                                                    www.coffeecup.com">
```

Software creator name & version number

Check Figure 5.16 & 5.16: Initialization and default value configuration

```
<param name="GENERAL"
value="1|11|000000|00FF00|pbihxzuqjkmcnalwsvfregytdodxxa://ppp.xdukgxnbou.lzk|
|Login Complete.|Enter the Username and Password.|http://www.microsoft.com|
|">
```

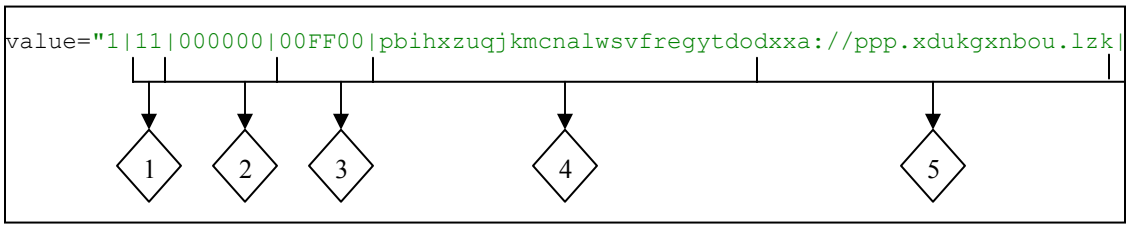


Figure 5.19 default value encrypted

1. This segment will check if the software is registered or not. If you use the trial version this would be different. This confirms that the registration routine is a part of the Applet Java Class File and not the executable file. This software is fully registered so it's yes you'll see later why. do you know what does that mean, try to obtain a trial version of this software and the full one and create a New Applet file you'll notice what makes this software registered or not. Isn't so simple!

```
if(Integer.parseInt(stringtokenizer.nextToken()) == 11)
registered = true;
```

2. This segment will check for the Background Color (Hex Value) as specified in the General tab.

3. This segment will check for Text Color (Hex Value) as specified in the General tab.

4. This segment "LinkURL" contains the encrypted format of the default Access Approved Forwarding Website. Check figure 5.14. this encryption will act as an inter-modular between the (5."LinkFrame") and the key for the decryption mechanism.

5. This segment "LinkFrame" is the real default Access Approved Forwarding Website after the encryption don't get confused with segment 4 do you remember what we set here as a default value figure 5.14.

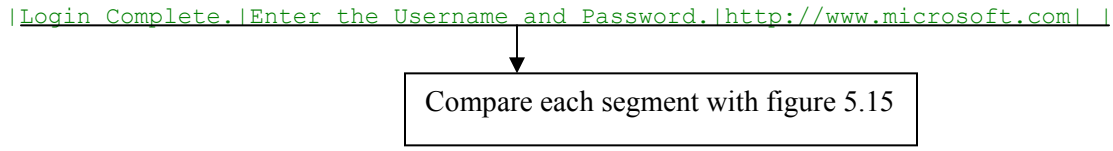


Figure 5.20 Text representation

name="0": this segment will indicate the user number (in this case "0") to configure for each one its own settings. You can create more than one user.

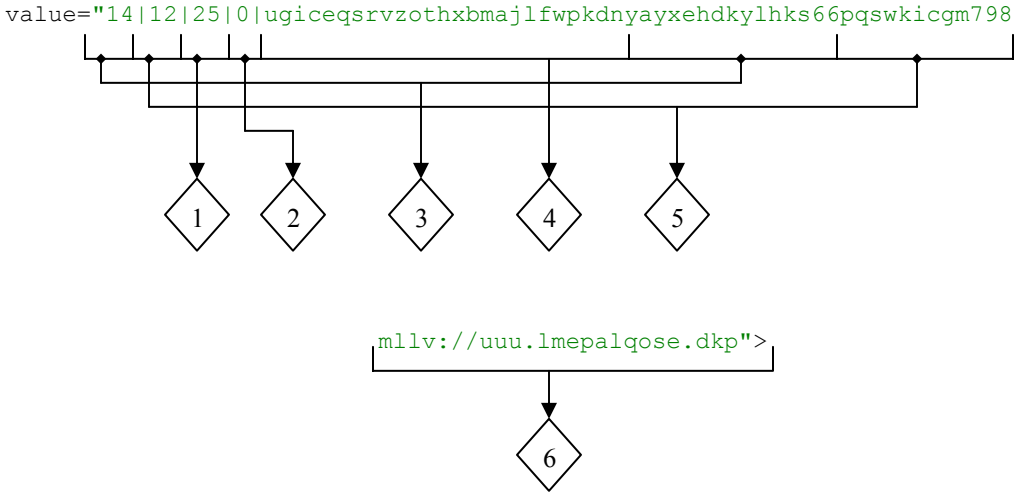


Figure 5.21 Encrypted format for the first username

1. This segment tells about total number of the alphabet(26).

2. This segment tells about alphabet number starting from 0.

3. This segment tells about Username Length with its encrypted format, do you notice that the decimal numbers aren't encoded you can check this by the last two numbers (Username: undecontrol66). Figure 5.16

4. This segment tells about Access Approved Forwarding Website in its encrypted format, (Link: http://www.themutable.com). Figure 5.16

5. This segment tells about Password Length with its encrypted format, (Passowrd: malkovish798). Figure 5.16

5.4.3 Applet Java Class Source Code Anatomy

Now, the time to have the power in front of your eyes with the decompiled version of the Applet Java Class file ("joylock.class"). It's nothing more than to open this file with DJ Java Decompiler software and you have the source code. that's it.

```
// Decompiled by DJ v3.7.7.81 Copyright 2004 Atanas Neshkov  Date:
2/22/2006 8:23:29 AM
// Home Page : http://members.fortunecity.com/neshkov/dj.html  - Check
often for new version!
// Decompiler options: packimports(3)
// Source File Name:   joylock.java
```

```
import java.applet.Applet;
import java.applet.AppletContext;
import java.awt.*;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.StringTokenizer;

public class joylock extends Applet
{

    public void init()
    {
        super.init();
        int i = size().width - 80;
      StringTokenizer stringtokenizer =
            new
            StringTokenizer(getParameter("GENERAL"), "|", false);
        numUsers = Integer.parseInt(stringtokenizer.nextToken());
                  if(Integer.parseInt(stringtokenizer.nextToken())
            == 11); Interesting!, Figure 5.19.1
            registered = true; Good Good.
        else
        if(getDocumentBase().toString().startsWith("file"))
            registered = true;
        if(registered)
        {
  bkColor = new Color(Integer.parseInt(stringtokenizer.nextToken(),
16)); Figure 5.19.2
  textColor = new Color(Integer.parseInt(stringtokenizer.nextToken(),
16)); Figure 5.19.3
  linkURL = stringtokenizer.nextToken();Figure 5.19.4
            if(!linkURL.equalsIgnoreCase(" "))
  decript(linkURL, 0, 0, 0, 0, 0, false); The azimuth of our analysis
            linkFrame = stringtokenizer.nextToken();Figure 5.19.5
      loginText = stringtokenizer.nextToken();Figure 5.15 & Figure 5.20
 preLoginMessage = stringtokenizer.nextToken();Figure 5.15 & Figure 5.20
            reLinkURL = stringtokenizer.nextToken();Figure 5.21.4
            reLinkFrame = stringtokenizer.nextToken();Figure 5.21.6
            username = new String[numUsers]; Figure 5.21.3
            password = new String[numUsers]; Figure 5.21.5
            urls = new String[numUsers];
            frames = new String[numUsers];
 for(int j1 = 0; j1 < numUsers; j1++); Read first user number "0" parameters
            {
StringTokenizer stringtokenizer1 = new
StringTokenizer(getParameter(Integer.toString(j1)), "|", false);
            int j = Integer.parseInt(stringtokenizer1.nextToken());
            Username Length, Figure 5.21.3
            int k = Integer.parseInt(stringtokenizer1.nextToken());
            Password Length, Figure 5.21.5
            int l = Integer.parseInt(stringtokenizer1.nextToken());
            Figure 5.21.1
```

Java Library Loading

Read each segment one by one with respect to its parameter name. for example "General" & User Number "0"

```
            int i1 = Integer.parseInt(stringtokenizer1.nextToken());
```

Figure 5.21.2

```
          String s = stringtokenizer1.nextToken();
decript(s, j1, j, k, l, i1, true); Call function decript to decode
the encryption of these values (s, j1, j, k, l, i1), String S:
reLinkURL
  }
setBackground(bkColor);
            setLayout(null);
   loginButton = new Button();
   loginButton.setLabel("Login");
   loginButton.reshape(8, size().height - 30, size().width - 16, 24);
   loginButton.setBackground(new Color(0xc0c0c0));
   add(loginButton);
   lUsername = new Label("Username:");
   lUsername.reshape(8, size().height - 88, 64, 24);
   lUsername.setForeground(textColor);
   add(lUsername);
   lPassword = new Label("Password:");
   lPassword.reshape(8, size().height - 60, 64, 24);
   lPassword.setForeground(textColor);
   add(lPassword);
   eUsername = new TextField();
   eUsername.reshape(72, size().height - 88, i, 24);
   eUsername.setBackground(new Color(0xffffff));
   add(eUsername);
   ePassword = new TextField();
   ePassword.setEchoCharacter('*');
   ePassword.reshape(72, size().height - 60, i, 24);
   ePassword.setBackground(new Color(0xffffff));
   add(ePassword);
   textArea = new TextArea(preLoginMessage);
   textArea.reshape(8, 6, size().width - 16, size().height - 98);
   textArea.setBackground(new Color(0xffffff));
   textArea.setEditable(false);
   add(textArea);
        }
    }
  public void paint(Graphics g)
      {
          if(!registered)
          {
```

Visualization Theme Settings, Figure 5.13, General tab

This message will not appear when your are offline even if you are using the trial version.

```
   g.drawString("You have the unregistered", 0, 10);
   g.drawString("version of this program", 0, 20);
   g.drawString("You need the registered version", 0, 30);
   g.drawString("for this Applet to work on the internet", 0, 40);
   g.drawString("Click HERE for registration instructions", 0, 50);
          }
      }

      void clickLoginButton(Event event)
      {
```

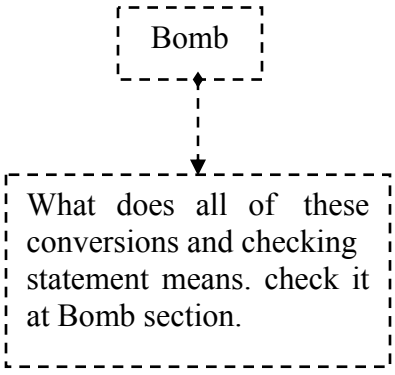
```
       for(int i = 0; i < numUsers; i++)
       {
       if(username[i].equalsIgnoreCase(eUsername.getText()) &&
        password[i].equalsIgnoreCase(ePassword.getText()))
              {
                  try
                  {
                      URL url;
                      if(!urls[i].equalsIgnoreCase(""))
                          url = new URL(getDocumentBase(), urls[i]);
                      else
                          url = new URL(getDocumentBase(), linkURL);
                      String s;
                      if(!frames[i].equalsIgnoreCase(""))
                          s = frames[i];
                      else
                          s = linkFrame;
                      if(s != null && !s.equalsIgnoreCase(" "))
                          getAppletContext().showDocument(url, s);
                      else
                          getAppletContext().showDocument(url);
                  }
                  catch(MalformedURLException _ex) { }
                  textArea.setText(loginText);
                  return;
              }
             if(i == numUsers - 1)
              {
               eUsername.setText("");
               ePassword.setText("");
               textArea.setText("Incorrect Username or Password.");
    if(reLinkURL != null && !reLinkURL.equalsIgnoreCase(" ") &&
     numWrongPass == 2)
                  {
                      numWrongPass = 0;
                      try
                     {
URL url1 = new URL(getDocumentBase(), reLinkURL);
  if(reLinkFrame != null && !reLinkFrame.equalsIgnoreCase(" "))
       getAppletContext().showDocument(url1, reLinkFrame);
     else
           getAppletContext().showDocument(url1);
                }
     catch(MalformedURLException _ex) { }
        } else
            {
                  numWrongPass++;
                }
             }
          }

     }

     public boolean handleEvent(Event event)
     {
        if(event.key == 10 || event.target == loginButton &&
           event.id == 1001)
```

```
            {
                clickLoginButton(event);
                return true;
            } else
            {
                return super.handleEvent(event);
            }
        }

        public boolean mouseUp(Event event, int i, int j)
        {
            if(!registered)
                try
                {
                  URL url = new URL(getDocumentBase(),
                   "http://www.coffeecup.com/");
                    getAppletContext().showDocument(url);
                }
                catch(MalformedURLException _ex) { }
            return true;
        }
```

The Insider Function.

```
final void decript(String s, int i, int j, int k, int l,
                   int i1, boolean flag)
    {
        String s1 = "";
        String s2 = s.substring(0, 26);
        String s3 = s.substring(26, s.length());
        String as[] = new String[52];
        for(int j1 = 0; j1 < 52; j1++)
            as[j1] = "";

        as[0] = as[0] + s2.charAt(alphabet.indexOf("a"));
        as[1] = as[1] + s2.charAt(alphabet.indexOf("b"));
        as[2] = as[2] + s2.charAt(alphabet.indexOf("c"));
        as[3] = as[3] + s2.charAt(alphabet.indexOf("d"));
        as[4] = as[4] + s2.charAt(alphabet.indexOf("e"));
        as[5] = as[5] + s2.charAt(alphabet.indexOf("f"));
        as[6] = as[6] + s2.charAt(alphabet.indexOf("g"));
        as[7] = as[7] + s2.charAt(alphabet.indexOf("h"));
        as[8] = as[8] + s2.charAt(alphabet.indexOf("i"));
        as[9] = as[9] + s2.charAt(alphabet.indexOf("j"));
        as[10] = as[10] + s2.charAt(alphabet.indexOf("k"));
        as[11] = as[11] + s2.charAt(alphabet.indexOf("l"));
        as[12] = as[12] + s2.charAt(alphabet.indexOf("m"));
        as[13] = as[13] + s2.charAt(alphabet.indexOf("n"));
        as[14] = as[14] + s2.charAt(alphabet.indexOf("o"));
        as[15] = as[15] + s2.charAt(alphabet.indexOf("p"));
        as[16] = as[16] + s2.charAt(alphabet.indexOf("q"));
        as[17] = as[17] + s2.charAt(alphabet.indexOf("r"));
        as[18] = as[18] + s2.charAt(alphabet.indexOf("s"));
        as[19] = as[19] + s2.charAt(alphabet.indexOf("t"));
        as[20] = as[20] + s2.charAt(alphabet.indexOf("u"));
```

```
as[21] = as[21] + s2.charAt(alphabet.indexOf("v"));
as[22] = as[22] + s2.charAt(alphabet.indexOf("w"));
as[23] = as[23] + s2.charAt(alphabet.indexOf("x"));
as[24] = as[24] + s2.charAt(alphabet.indexOf("y"));
as[25] = as[25] + s2.charAt(alphabet.indexOf("z"));
as[26] = as[26] + s2.charAt(alphabet.indexOf("a"));
as[27] = as[27] + s2.charAt(alphabet.indexOf("b"));
as[28] = as[28] + s2.charAt(alphabet.indexOf("c"));
as[29] = as[29] + s2.charAt(alphabet.indexOf("d"));
as[30] = as[30] + s2.charAt(alphabet.indexOf("e"));
as[31] = as[31] + s2.charAt(alphabet.indexOf("f"));
as[32] = as[32] + s2.charAt(alphabet.indexOf("g"));
as[33] = as[33] + s2.charAt(alphabet.indexOf("h"));
as[34] = as[34] + s2.charAt(alphabet.indexOf("i"));
as[35] = as[35] + s2.charAt(alphabet.indexOf("j"));
as[36] = as[36] + s2.charAt(alphabet.indexOf("k"));
as[37] = as[37] + s2.charAt(alphabet.indexOf("l"));
as[38] = as[38] + s2.charAt(alphabet.indexOf("m"));
as[39] = as[39] + s2.charAt(alphabet.indexOf("n"));
as[40] = as[40] + s2.charAt(alphabet.indexOf("o"));
as[41] = as[41] + s2.charAt(alphabet.indexOf("p"));
as[42] = as[42] + s2.charAt(alphabet.indexOf("q"));
as[43] = as[43] + s2.charAt(alphabet.indexOf("r"));
as[44] = as[44] + s2.charAt(alphabet.indexOf("s"));
as[45] = as[45] + s2.charAt(alphabet.indexOf("t"));
as[46] = as[46] + s2.charAt(alphabet.indexOf("u"));
as[47] = as[47] + s2.charAt(alphabet.indexOf("v"));
as[48] = as[48] + s2.charAt(alphabet.indexOf("w"));
as[49] = as[49] + s2.charAt(alphabet.indexOf("x"));
as[50] = as[50] + s2.charAt(alphabet.indexOf("y"));
as[51] = as[51] + s2.charAt(alphabet.indexOf("z"));
for(int k1 = 26; k1 < 51; k1++)
    as[k1] = as[k1].toUpperCase();

for(int l1 = 0; l1 < s3.length(); l1++)
    switch(s3.charAt(l1))
    {
    case 65: // 'A'
        s1 = s1 + as[26];
        break;

    case 66: // 'B'
        s1 = s1 + as[27];
        break;

    case 67: // 'C'
        s1 = s1 + as[28];
        break;

    case 68: // 'D'
        s1 = s1 + as[29];
        break;

    case 69: // 'E'
        s1 = s1 + as[30];
        break;
```



```
case 70: // 'F'
    s1 = s1 + as[31];
    break;

case 71: // 'G'
    s1 = s1 + as[32];
    break;

case 72: // 'H'
    s1 = s1 + as[33];
    break;

case 73: // 'I'
    s1 = s1 + as[34];
    break;

case 74: // 'J'
    s1 = s1 + as[35];
    break;

case 75: // 'K'
    s1 = s1 + as[36];
    break;

case 76: // 'L'
    s1 = s1 + as[37];
    break;

case 77: // 'M'
    s1 = s1 + as[38];
    break;

case 78: // 'N'
    s1 = s1 + as[39];
    break;

case 79: // 'O'
    s1 = s1 + as[40];
    break;

case 80: // 'P'
    s1 = s1 + as[41];
    break;

case 81: // 'Q'
    s1 = s1 + as[42];
    break;

case 82: // 'R'
    s1 = s1 + as[43];
    break;

case 83: // 'S'
    s1 = s1 + as[44];
    break;

case 84: // 'T'
```

```
            s1 = s1 + as[45];
            break;

        case 85: // 'U'
            s1 = s1 + as[46];
            break;

        case 86: // 'V'
            s1 = s1 + as[47];
            break;

        case 87: // 'W'
            s1 = s1 + as[48];
            break;

        case 88: // 'X'
            s1 = s1 + as[49];
            break;

        case 89: // 'Y'
            s1 = s1 + as[50];
            break;

        case 90: // 'Z'
            s1 = s1 + as[51];
            break;

        case 97: // 'a'
            s1 = s1 + as[0];
            break;

        case 98: // 'b'
            s1 = s1 + as[1];
            break;

        case 99: // 'c'
            s1 = s1 + as[2];
            break;

        case 100: // 'd'
            s1 = s1 + as[3];
            break;

        case 101: // 'e'
            s1 = s1 + as[4];
            break;

        case 102: // 'f'
            s1 = s1 + as[5];
            break;

        case 103: // 'g'
            s1 = s1 + as[6];
            break;

        case 104: // 'h'
            s1 = s1 + as[7];
```

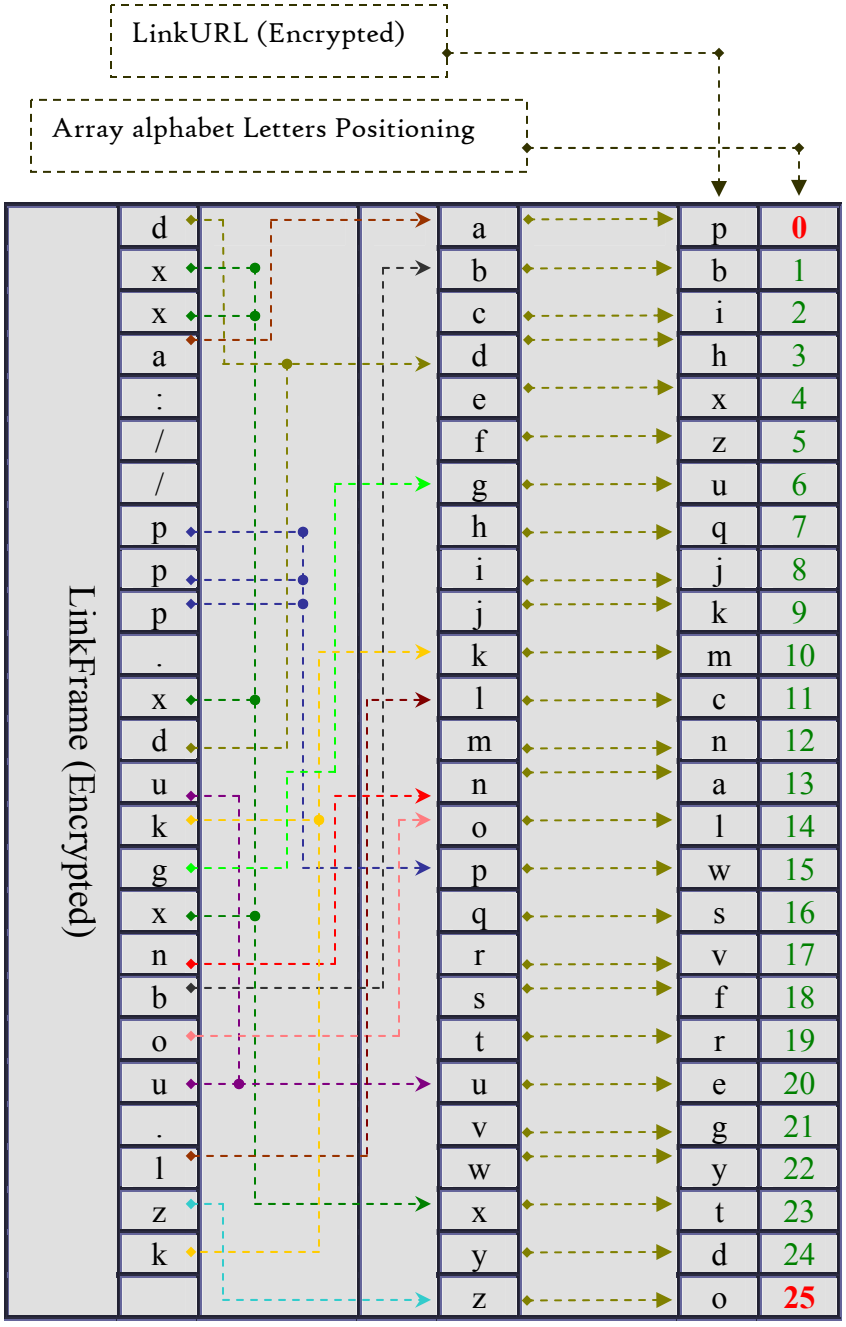
```
        break;

    case 105: // 'i'
        s1 = s1 + as[8];
        break;

    case 106: // 'j'
        s1 = s1 + as[9];
        break;

    case 107: // 'k'
        s1 = s1 + as[10];
        break;

    case 108: // 'l'
        s1 = s1 + as[11];
        break;

    case 109: // 'm'
        s1 = s1 + as[12];
        break;

    case 110: // 'n'
        s1 = s1 + as[13];
        break;

    case 111: // 'o'
        s1 = s1 + as[14];
        break;

    case 112: // 'p'
        s1 = s1 + as[15];
        break;

    case 113: // 'q'
        s1 = s1 + as[16];
        break;

    case 114: // 'r'
        s1 = s1 + as[17];
        break;

    case 115: // 's'
        s1 = s1 + as[18];
        break;

    case 116: // 't'
        s1 = s1 + as[19];
        break;

    case 117: // 'u'
        s1 = s1 + as[20];
        break;

    case 118: // 'v'
        s1 = s1 + as[21];
        break;
```

```
            case 119: // 'w'
                s1 = s1 + as[22];
                break;

            case 120: // 'x'
                s1 = s1 + as[23];
                break;

            case 121: // 'y'
                s1 = s1 + as[24];
                break;

            case 122: // 'z'
                s1 = s1 + as[25];
                break;

            case 91: // '['
            case 92: // '\\'
            case 93: // ']'
            case 94: // '^'
            case 95: // '_'
            case 96: // '`'
            default:
                s1 = s1 + s3.charAt(l1);
                break;
            }

        if(flag)
        {
            username[i] = s1.substring(0, j);
            password[i] = s1.substring(j, j + k);
            urls[i] = s1.substring(j + k, j + k + l);
            frames[i] = s1.substring(j + k + l, s1.length());
            return;
        } else
        {
            linkURL = s1;
            return;
        }
    }

    public joylock()
    {

      The Key for deciphering the algorithm
      ---------------------------------------------

        alphabet = "abcdefghijklmnopqrstuvwxyz";
        registered = false;
    }

    Label lUsername;
    Label lPassword;
    TextArea textArea;
    Button loginButton;
```

```
    TextField ePassword;
    TextField eUsername;
    int numUsers;
    int numWrongPass;
    String loginText;
    String linkURL;
    String linkFrame;
    String preLoginMessage;
    String reLinkURL;
    String reLinkFrame;
    String username[];
    String password[];
    String urls[];
    String frames[];
    String alphabet;
    Color textColor;
    Color bkColor;
    boolean registered;
}
```

5.4.4 Bomb Section Analysis

What I've done so far is tracing the algorithm Source Code step by step and check the relations with Figure 5.18. The conclusion for all of these lines of code is: how the Java Class reads the segments from the HTML required parameters Figure 5.18 and assign each segment it's variable. The whole decryption procedures occurs in the decript function which analyze the input from Figure 5.18 and try to check it's decoding equivalence in the decript function. Here, as you notice the alphabet letters plays a major role in the decryption (no case sensitive). So, as I told you before the LinkUrl Figure 5.19.4 is the inter-modular between the alphabet characters and the LinkFrame Figure 5.19.5 for decoding process.



For the default value parameters Figure 5.19 consider this schema: Read it from left to right. Take for example the first letter of the encrypted LinkFrame "d" (Left to Right) and follow the path horizontally with respect to the equivalence of the alphabet arrangement it will gives you an "h" so the LinkURL is also the key for the decryption.

Repeat the rest of the LinkFrame letters and you'll end up with the Access Approved Forwarding website in its original state: http://www.themutable.com

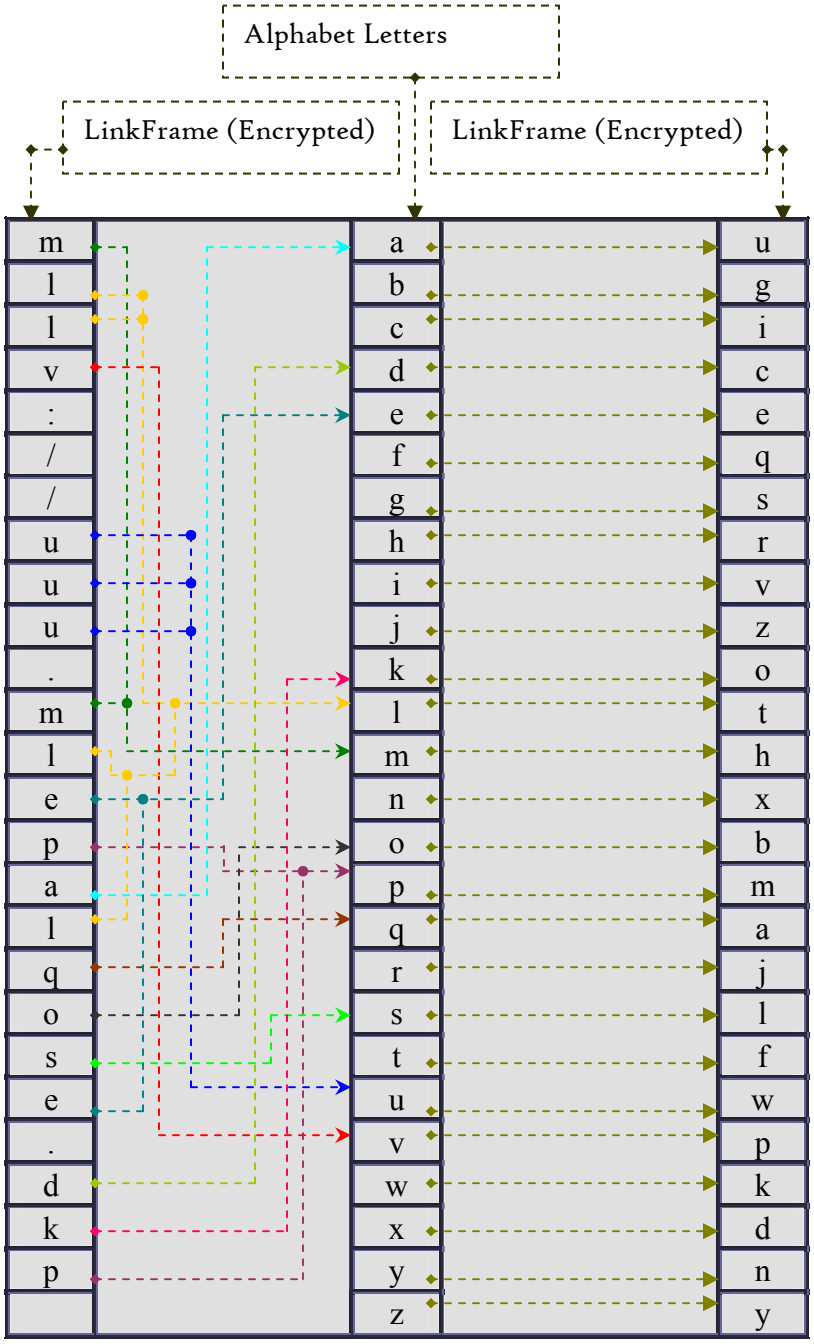


Figure 5.21 [Section “4” & Section “6”] Anatomy
Configuration for username="0"
Follow the same procedure as previously done and you will and up with the same Access Approved Forwarding website in its original state: http://www.themutable.com

Note: the result is same as before because when I create a new user as in Figure 5.16 I checked the "Use Default Link" Box, so it's right no problem. Read it from left to right.

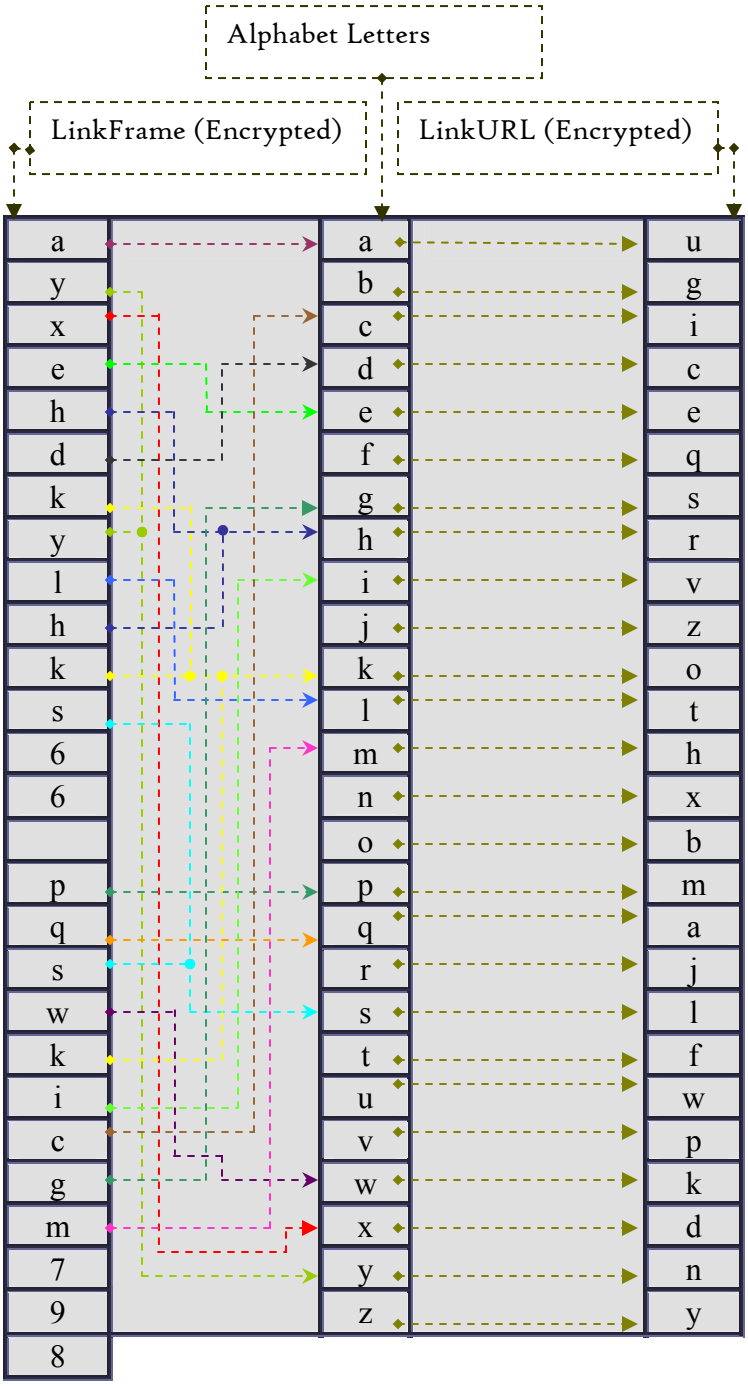


Figure 5.21 [Section “3” & Section “5”] Anatomy
Configuration for username="0"
Follow the same procedure as before and you will and up with the original Username and Password (deciphered)
Username: undercontrol66
Password: malkovish798

5.4.5 Flash Plain Text Searching Approach

Using this software allows you to implement the site password protection in Flash mode, but it's protection is very poor compared to the Java mode where all what you need to do is to open the flash file (*.swf) with any Hex Editor and do a search for an ASCII characters and you'll end up with the username and password as it's with no encryption being involved at all on the ASCII characters: It's in plain text format (what's an easy fishing), creating a new user is same as in Java mode (a little bit settings differences but it doesn't matter), here the HTML required parameters offers no help for the decoding process, because all the information is embedded inside the Flash file so our target in this session will be the Macromedia Flash file with an extension .swf .

```
<OBJECT classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
codebase="http://active.macromedia.com/flash2/cabs/swflash.cab#version=5,0,0,0
"
ID=LogSWF WIDTH=342 HEIGHT=117>
<PARAM NAME=movie VALUE="LogSWF.swf">
<PARAM NAME=quality VALUE=high>
<PARAM NAME=loop VALUE=false>
<EMBED src="LogSWF.swf" loop=false quality=high
WIDTH=342 HEIGHT=117 TYPE="application/x-shockwave-flash"
PLUGINSPAGE="http://www.macromedia.com/shockwave/download/index.cgi?P1_Prod_Ve
rsion=ShockwaveFlash">
</EMBED>
</OBJECT>
```

Let's take a scenario where you access a site using this protection but in this case you don't have the flash file and as i said before the HTML required are useless in this case (used only to make the file appear on the web site...) so what, do you know that when you access a site for a first time, the time required to be fully loaded is longer than the second entrance and that's refer to the Temporary Internet Files which makes surfing the net faster if no update occurs on the site (because of the second entrance).

Follow the steps in order to locate where the victim resides:

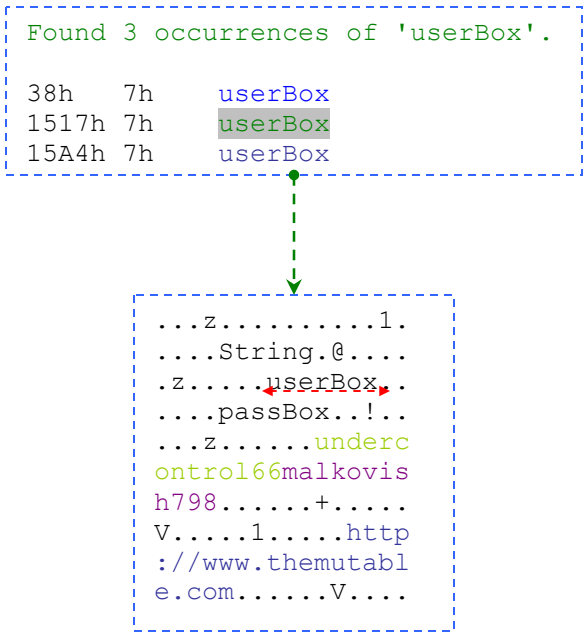
First I recommend that you delete the files from the Temporary Internet Files so that locating the file will be easier. This can be done by open the Internet Explorer (in my case) ---> Tools --> Internet Options... --> Delete Files.

Ok, now the file "Temporary Internet Files" is clear, your next step is to enter the site which using this protection in Flash mode now you are in (page fully loaded).Go to My Computer--> C:\ Documents and Settings \ **UserName** \ Local Settings \ Temporary Internet Files

and now do a search for the .swf extension (ex, *.swf) if there are more than one file try to open each one till you find the required one and have a copy in another place.

Now open the file with 010 Editor (Hex Editor) or any other hex editor and do a search for: userBox or passBox and you'll end up with occurrences of three for each one (depends on the

one you choose for search), stay at the second occurrence what do you notice, are we in the Bomb Section or Not! I think so. As in the following box (taken from 010 Editor):



UserBox OR passBox: Indicator to find the embedded username & password

undercontrol66: Username

malkovish798: Password

http://www.themutable.com: Access Approved Forwarding Website

Chapter 6

CONCLUSIONS

The importance of this project lies in the newly developed methods with step by step analysis to make things easier to follow. Reverse Code Engineering with emphasizing on breaking software protection has the influence on the overall developer programmers who must know the weakness in their protections implementations to better shield their programs from crackers attacks.

It shows how easy it is to change only one instruction to defeat the protection from its root. There is no protection that can not be easily removed (my experience). It's a matter of time.

The information's presented in my book and how it's being processed starting with the theory chapters till the experimental fact demo, it will serve as a basis for another development and contributions concerning RCE and the interrelationship with breaking protections and deciphering the algorithm.

What we've learnt from this newly customized approach. Using the right tools. How to read the inner and outer shell for many algorithms in the case studies chapter. Breaking the unbreakable. Deciphering the algorithm. And the most important thing is to have the pleasure for the unknown.

References

BOOKS

[1] Elam, Eldad. *Reversing: Secrets of Reverse Engineering.* Indianapolis: Wiley Publishing, Inc., 2005.

[2] Irvine, Kip. *Assembly Language for Intel®*-Based Computers. New Jersey: Pearson Education, Inc., 2003.

[3] Rosenberg, Jonathan. *How Debuggers Work: algorithms, data structure, and architecture.* New York: John Wiley & Sons, Inc., 1996.

[4] Kaspersky, Kris. *Hacker disassembling Uncovered.* East Swedesford: A-LIST Publishing, 2003.

[5] Lena151, *Imports rebuilding #21* March 14, 2006

[6] Goppit, *Portable Executable File Format – A Reverse Engineering View*. CBJ, Security Analysis; VOL. 2, NO. 3 2005. August 15, 2005.

TOOLS

[7] IDA Pro Advanced: The Interactive Disassmbler, Version 4.9.0.863 (32-bit). http://www.datarescue.com. TOO Geliosoft., 2005.

[8] OllyDbg, v1.10. http://www.ollydbg.de/. Oleh Yuschuk, 2000-2004

[9] Stud_PE by ChristicG, Version 2.2.0.5-Build date: 20/03/2006 12:21:00 AM. http://www.itimer.home.ro/. CGSoftLabs., 2006.

[10] UPX: Ultimate Packer for eXecutables by Markus F.X.J. Oberhumer & Laszlo Molnar, V1.24w, Nov 7th 2002. http://upx.sourceforge.net. 2002.

[11] Code Snippet Creator by Iczelion, version 1.05 build 2. 1999-xxxx

[12] Restorator 2006 v3.60 build 1535 by bome.com/Florian Bomers, http://www.bome.com/Restorator/. 1996-2006

[13] RadASM version 2.2.0.6 by Ketil Olsen, http://www.radasm.com/. 2001-2005

[14] Microsoft Visual C++ Enterprise Edition V 6.0, http://www.microsoft.com/., Microsoft Corporation. 1994-98

[15] MASM32 version 9.0 by Steve Hutchesson, http://www.movsd.com. 1998-2006

Vita Auctoris

The Outer Worlds
Kuiper Belt, Pluto
(000) ↵00 ↵000000

Name: tHE mUTABLE

Place of Birth: I don't know where, but I think first moment of creation "Quantum"

Year of Birth: -00↙↗-00↙↗-0000

Education: $\frac{\not{X}}{\lambda}, \frac{\beta}{\pi}, \frac{\phi}{\xi}, \frac{\infty}{\infty}$

$$\Gamma(-\infty)+\Gamma(\infty)$$

einsteinzero@hotmail.com
cheviva2000@hotmail.com
http://www.themutable.com