Alfred Lo is currently looking for job opportunities around the world in information security. His interests are in techniques in cracking, hacking and how to prevent them. He graduated from The University of Hong Kong in Computer Engineering (1st class) and is finishing his master degree in The University of Birmingham in Sep 2002. If anyone wants to hire him or offers him opportunities (e.g. PhD, short term contracts, etc), please contact him at alfredkmlo@hotmail.com.

Software Protection and its Annihilation

Alfred K.M. Lo

alfredkmlo@hotmail.com

"There is a crack, a crack in everything, that's how the light gets in."

May 2002

Abstract

This project identifies commonly used software protection techniques and their vulnerabilities. By working from the worldviews of crackers, software industry, and researchers, this paper gives analysis on the principles behind the attacks, the investigating psychology, how exploits are constructed, and what can be done to prevent the problems.

Three commercial software programs are studied in depth as case studies. The results ring the alarms of the software industry. It shows that our daily-used commercial software, even being protected by commercial protection solutions, is too easily to be defeated.

Keywords

encryption; packing; unpacking; reverse engineering; cracking; obfuscation; watermarking; software protection; anti piracy

Version History

Publication Date	Changes
May 30, 2002	Add Version History, numerous wording fixes
April 22, 2002	Report first released

Legal Disclaimer

All the materials discussed in this report are served for **educational purposes only**. You should not reverse engineer, debug or crack applications or programs you haven't legitimately bought, or not for your own personal use:

- There is **no intention to encourage cracking**.
- It is merely **a study** of state-of-art software protection systems.
  1. TextPad is a very good program that is deserved to buy.
  2. Dreamweaver is a very good program that is deserved to buy and its trial should be deleted after 30 days anyway.
  3. SmartSaver Pro 3 is a very good program that is deserved to buy and its trial should be deleted after 15 days anyway.
- Any legal issues arising from the misuse of the information presented here ARE NOT the writer's responsibilities.

Table of Contents

Table of Figures

1. Introduction

University researches on security can sometimes be too academic. They will tell you when the program is encrypted, the way to defeat it is to wait until it is decrypted in memory and then extracts the contents, and that's all. But, the fact is that this dump executable won't run correctly unless some necessary conditions are met, and practical protection schemes are designed so that these necessary conditions are difficult to be achieved.

On the other hands, people in the underground community may lack of formal trainings and knowledge. However, they can possess very sophisticated and practical skills that are not commonly known by academic researchers. The combination of these twos can be very powerful and very interesting, and this is the objective of this project.

Software protections appear in many forms, from those be seen by end-users such as textbox asking for serial key, to those invisible watermarks embedded in software. For whatever they are, they serve only one goal – to protect the intellectual property rights of the owner.

On the other sides, there are always some people who want to bypass those protections. These people are called crackers. In their parlance, they called themselves "software hackers", those people who "destroy" the CODE of the application that they are examining. Their acts to breach software protections are called 'cracking'.

Cracking started as long as protection schemes appeared. The first cracking document I have come across dated back 1987. It should be stressed that these twos help evolving each other. Whenever there is a new protection scheme, there must be someone who works out the crack of it and a new scheme will appear which is stronger…

This project contains case studies. Three programs have been selected. They are TextPad, SmartSaver Pro and Dreamweaver. They represent different market segments in the industry: TextPad (US $27:cheap), SmartSaver Pro (US $59.95:medium) and Dreamweaver (US $299:expensive).

In some senses, cracking is good because it "helps" software to be better protected. Needless to say, the race between software protectors and crackers is endless.

2. Simple Threat Model

Possible attacks to software can be best understood with a simple threat model of a computer program [1].

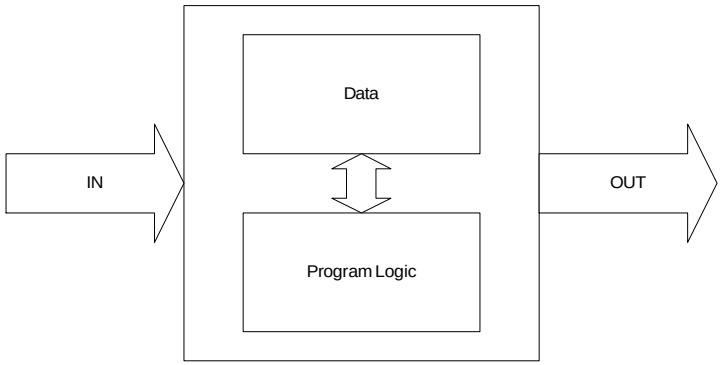

Figure 1 Simple thread model of a computer program

Data

The data area may store confidential information like user passwords, serial numbers, private/decryption keys, etc. Through monitoring the contents in these areas, confidentiality can be breached.

Program Logic

On receiving the input and the state of data, the program acts according to the logic defined in its codes. By reverse engineering, valuable pieces of code flow - "the brain of the program" can be extracted. This allows someone to extract a module from the program and use it in his own. If someone modifies the program logic, a process known as patching/tampering, the execution flow will be modified, e.g. bypassing a security check.

IN/OUT

The input and output of a program can be monitored. By capturing this information, replay attack is made possible.

Whole Program

Because the program is a kind of digital information, any copy of it is exactly the same as original. It is possible for someone to make illegal copies of the program and resell them – an act known as software piracy.

Therefore, any programs under this model are subjected these attacks:

1. Monitoring
2. Reverse Engineering
3. Software Piracy
4. Tampering

To cope with these potential threats, measures have been taken to protect the software. Here I classify the techniques into basic and advance levels, according to their complexities and eases of implementation.

3. Cracking Tools

Most protections cannot be bypassed without the use of tools. So let's first take a look at them. These tools fall into two main categories, reverse engineering tools and system-monitoring tools.

3.1 Reverse Engineering Tools

It helps us to know the logics of the underlying program. By using these tools properly, we are able to study the internal of a process, understand its weaknesses and carry out exploitations. They can be further subdivided into 2 categories:

1. Disassembler/Decompiler – allows us to study the static logic of the program. E.g. W32Dasm
2. Debugger – allows us to study the runtime behaviors and status during program execution. E.g. SoftICE and the Debugger in Visual Studio

3.1.1 Disassembler/Decompiler

The Disassembler is used to disassemble the compiled code and generates its assembly equivalents, while the decompiler generates its high-level source codes. Decompliers work very well in Java (almost 1-1 mapping) but don't perform well in C/C++. Since our targets in this project are not Java programs, we will not use decompiler and thus is not discussed further.

A very good disassembler for x86 environment is W32DASM. This allows studying of the internal program structure and useful information to be extracted. By the way, it is a debugger as well.

Figure 2.1 W32Dasm

W32dasm is a Windows Program Disassembler/Debugger featuring:

1. Disassembles both 16 and 32 bit Windows programs
2. Disassembles for MMX instructions
3. Displays for Exports, Imports, Menu, Dialog, and Text References
4. Integrated Debugger for 32 bit Programs (16 bit Debug NOT available)

3.1.2 Debugger

Debuggers work by emulating the processor. Therefore, programs are executed in the debugger container as if it is interacting directly to the processor. By acting as the middleman, the debugger is able to trace the runtime execution, memory/register contents, and setting break points, etc.

There are two kinds of debuggers, application-level debugger and system-level debugger. Application-level debugger, sits itself between the OS and the debugging program, while system-level debugger sits itself between the processor and the OS. Therefore,

system-level debuggers are more "powerful" because it can debug the OS at the driver/kernel level.

Here are the functions that are often provided by the debugger:

1. Execute each source statement, one at a time, with as much time between statements as we would like. This procedure is known as *single step*, or *stepping* for short
2. Step *into*, *out of*, and *over* function calls
3. Have the program execute normally until a specified source statement is reached and then stop execution. This procedure is known as *breakpoints*
4. Display the values in variables, either while the program is running normally, or during single steps and breakpoints. This procedure is known as *watch*
5. Change the values in variables and then have the program continue operation
6. Monitor and modify the run-time memory and register contents
7. Disassembling
8. Monitoring the Stack Context

In this project, these debuggers are used – SoftICE and Visual Studio.

SoftICE – The System-Level Debugger

According to Compuware [5], "SoftICE is a powerful **kernel mode debugger** that supports **device driver debugging** on either a single or dual machine configuration… SoftICE reduces debugging downtime by providing powerful features that extend beyond the limitations of the traditional Windows SDK/DDK tools. SoftICE has unique **system-wide views** and controls that make it easy to understand and diagnose the widest variety of Windows software problems."

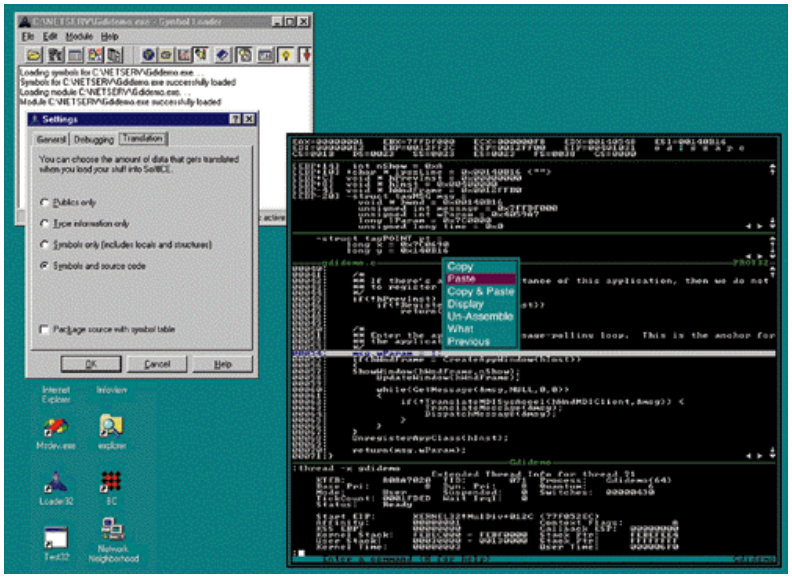

Figure 2 Debugging in SoftICE

Here are the most commonly used commands in SoftICE:

1. Step Into [press F8] – step into the call
2. Step Over [press F10] – step over the call
3. Step Out [press F12] – step out of the current call
4. Register Modifying [R] – e.g. R EAX FFFFFFFF (modify EAX to FFFFFFFF)
5. Memory Dump [D] – e.g. D 400000 (dump the memory content at 0x400000)
6. Memory Search [S] – e.g. S 0 L FFFFFFFF 'str' (search the memory from 0x0 over FFFFFFFF bytes for the string 'str'
7. Breakpoint of execution [bpx] – e.g. bpx 401000 (SoftICE breaks when instructions at 0x401000 is executed)
8. Breakpoint of memory read/write [bpm] – e.g. bpmb 401000 RW (SoftICE breaks when the byte at memory location 0x401000 is access by read/write operations)

Please refer to SoftICE command references for details [14].

Microsoft Visual Studio – The Application-Level Debugger

Visual Studio contains this nice debugger that allows you to control the operation of your program, to display and change the values in variables.

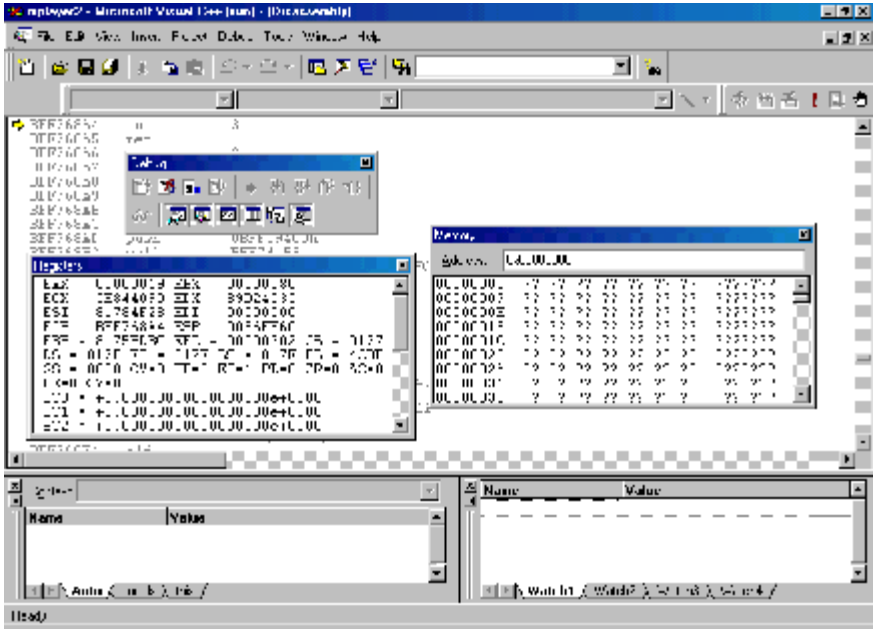

Figure 3 Microsoft Visual Studio Debugger

3.2 System Monitoring Tools

There are two system-monitoring tools that are proved to be very useful throughout the project. They are FileMon and RegMon.

FileMon – The File Monitor

According to Sysinternal [6], "*Filemon* monitors and displays file system activity on a system in **real-time**. Its advanced capabilities make it a powerful tool for exploring the way Windows works, seeing how applications use the files and DLLs, or tracking down problems in system or application file configurations. Filemon's time stamping feature will show you precisely when every open, read, write or delete, happens, and its status

column tells you the outcome."

RegMon – The Registry Monitor

According to Sysinternal [7], "*Regmon* is a Registry monitoring utility that will show you which applications are accessing your Registry, which keys they are accessing, and the Registry data that they are reading and writing - all in **real-time**."

3.3 Others Tools

If we want to tamper the software, binary editing is unavoidable. A hexadecimal text editor is nevertheless necessary. It should allow editing binary files, showing in binary and ASCII view, and is better to be equipped with:

1. HEX Cut, Copy and Paste support
2. HEX Insert and Delete of characters
3. HEX Find, Replace and Replace All
4. HEX Files comparisons

An example of good hex text editor is "Hex Workshop" from Breakpoint Software.

3.4 Discussions

Different tools are available depending on different platforms. The tools listed here are for Microsoft Windows, but the same principles can be applied to other platforms. This list is in any way not comprehensive. More sophisticated and dedicated tools will be introduced later in the report.

It should be noted that debugger, although be put in the category of reverse engineering tools, is also a runtime monitoring and tampering tool as well.

For system-level debugging, the SoftICE provides many robust and powerful functions and can be invoked at any time, even before Windows starts. Thus, it allows debugging of any programs at any time. Besides, application-level debugger requires explicit creating or attaching to the process of the debugging target.

Because of the powerfulness of system-level debuggers, most people may think that it is the only one they need. In fact, this may not true. As it sits between OS and the processor,

it is not possible for it to use the OS API, and can be expected, the user interface provided by these system-level debuggers are very native (DOS like) and hence non-user-friendly. USB mouse support had only been added into SoftICE since last year.

4. Basic protection techniques

4.1 Software Tokens

It is the most commonly used techniques for software protection. It can have the following forms:

1. Registration Key – one single serial key hard-coded in the program code. Our input is compared with it.
2. Multiple Serials – the serial number is broken into parts (e.g. [xxx]-[xxxxx]-[xxx]). A serial validating algorithm exists to check against these sub-parts. Using the algorithm, the program can accept many different serials without hard-coding them.
3. Serial/Name – the software token here is a serial/name pair. Checking is based on algorithm like multiple serials (e.g. check if f(name)=serial)
4. Key File – the software token exists as a license file stored inside the hard disk or system registry. In many cases, this key file, apart from storing user profile, may also contain usage information (e.g. how many days it has been used).

4.2 Hardware Tokens

Because software is a kind of digital information that is so easy to be duplicated, people invented hardware tokens, and make the operation of their programs dependant on the presence of these physical keys. The root assumption to this protection method is that hardware tokens are difficult to be copied. The art of making them difficult to be copied is called “Copy Protection”.

Physical keys can also be in many forms:

1. Key disk – specially produced diskette. E.g. By boring a hole in the magnetic media at a specific location. The program then checks for bad sectors at that location for validation.
2. Dongle - small hardware attached at the I/O (serial/parallel/USB) port of the computer. The checking routine queries those ports for values. If the hardware token is there, it will detect the electric pulses and then generate appropriate responses.
3. Smart Cards - a plastic card about the size of a credit card, with an **embedded microchip** that can be loaded with data. Some smart cards contain both code and data and therefore it can execute routines using the built-in microchip. Smart Card is tamper-resistant, whenever it detects intrusion, it will destroy the data inside it.

4. CD – Most CDs in the past doesn't have any copy-protection at all. The CD in itself is already a very good token because in the old days, most people don't have CD copying equipment (e.g. CDR/CDRW) and the capacity of CD was even larger than that of the hard drive. It was impossible to copy the entire CD into the hard disk. However, with the advance of CD copying technology, measures have been taken to protect the CD from copying. Some tricks used by manufacturers are discussed in [43].

4.3 Manual Look-ups

This was the protection method used in early days. It is a scheme in between hardware and software tokens. The protection is like this: when you enter a game, the game asks you: "What is the color of the pattern at the left hand corner of page 32?" The protection assumption is based on – it was more difficult to copy the manual at that time (especially for color one) than diskettes. It is "hard" because it is a manual but is also "soft" because one can ask others to lookup the manual for the answers.

4.4 Nag Screens

They are those annoying screens that prompt up usually at the start of the program, claiming the rights of the owner, prompt the user for registration or so. It is a very simple technique used to prove ownership.

4.5 Limits

There are many forms of limits. The most common ones are time limits imposed by shareware. The program will disable itself after the limit exists.

4.6 Crippleware

Some functions are deliberately disabled, e.g. save. Those functions may be unlocked if the user registers the software – commonly used in shareware.

4.7 Discussions

In view of the protected program, hardware and software token protections are essentially the same. The formula includes invoking some protection checking routines inside the program to see if required tokens are present and correct. (Note: this is not true until hardware tokens possessing code execution abilities appeared in the market, e.g. Smart Card. The implication of this will be discussed in the section "Code Partitioning" later in the report.) Therefore, in terms of cracking, bypassing these checking in the program are also the same.

Hardware token schemes and manual-lookups are controversial measures to discourage piracy, the act of unauthorized copying of software. These strategies are "effective but failed". It is effective because they are really difficult to be duplicated, but it is also inconvenient for legitimate users as they are not able to make backup (in case of copy-protected hardware tokens) and annoying (asking for manual lookup every time the game starts). More importantly, they fail because many cracks that patch the program to bypass protections can be found on the shared media. Therefore, piracy can be achieved without duplication difficulties.

On the other hands, shareware uses an entirely different approach to combat piracy. Shareware, instead of being copy-protected, actually encourages copying and spreading of itself. Nag screens, limits, crippleware are measures often used by shareware to claim ownership, reminding registering, and enforcing its freedom of use is not being abused.

5. Basic protection countermeasures

5.1 The Simple Scenario

Simple protection schemes discussed above can be easily defeated if they are not further protected by encryption/obfuscation. This is because many of them can fall into this simple model:

```
result=security_check(condition1, condition2)
if (result = = TRUE)
    then <authorize and goto proper program execution>
else <prompt up error and penalty>
```

Condition 1 may be the user input serial number, and condition 2 may be the required number. They may also be detected hardware response and the required response, etc. The security check can range from simple string comparisons to system I/O queries (like file checking, port checking, etc). The penalty may be disabled function, program termination, etc. Using your imagination, many simple protection schemes can be fitted into this simple model.

5.2 The Simple Challenge

My previous work [13] on Windows Media Player hacking describes in very details what happens when Win32 functions are translated into assembly, that I won't repeat here. The above simple model will probably be translated into assembly like this:

```
push condition2
push condition1
call security_check
test eax, eax
jnz address1 (authorized)
<prompt up error and penalty>
```

Just a brief to the assembly code – the last parameter to the function is always pushed first, then the second last one… the first one. The result of the called function is stored in register EAX. The test operation performs a logical AND operation without modifying

input parameters. Therefore, if the result is FALSE (0x0 in most cases), the AND operation of two zero parameters will flag up the "Z flag" in the flag register. Therefore the conditional jump (jnz = jump if not zero) will not be carried out and penalty is executed.

To keep my report brief and precise, I will not explain assembly instructions any more. Please reference to some 80x86 Intel instruction references, such as [11,12].

5.2.1 By Debugger

Therefore, crackers, by setting appropriate breakpoints (e.g. break if system executes StrCmp), and upon the debugger breaks, they can do:

1. "Serial fishing" – by looking at the contents at condition 1 or 2, the required parameter to pass the security check is leaked.
2. Tampering – by modifying the instruction from conditional jump (jnz) to unconditional one (jmp), the penalty will never be executed. If the call to security check is disabled (replaced by nop), the security checking will never be invoked. Cracker may note down this instruction address and patch it permanently into the executable file.
3. Result modifying – if tampering instructions is not possible, e.g. because of CRC checking, etc, crackers can invert the flag after the call (e.g. changing EAX from 0 to 1 or modify the Z bit of the flag register so as to affect the jump).
4. Key Generator – if the required key is not hard-coded, crackers can reverse engineer the key generating algorithm inside the program, and release a key generator to the public. Some commercial key generating schemes are discussed in [8].

5.2.2 By Disassembler

Sometimes crackers don't need to use debugger at all. By noting down the error message after the security check, say "Wrong serial key! Program exits", crackers can just disassemble the file and look through the "String Data References" in the file. Most disassembler (like W32Dasm) supports the extraction of static string data in the initialized data section of the executable. Therefore, by locating where in the program references to these strings, they are able to locate the security checking routine and bypass it, e.g. through patching.

5.3 Useful Breakpoints

As demonstrated, we can break software protections if we can locate the security checking routines. The most convenient way to do this is by setting breakpoints. Below is a list of commonly used breakpoints for operations related to:

1. **Windows** - bpx CreateWindow, bpx CreateWindowEx(A/W), bpx ShowWindow, bpx UpdateWindow, bpx GetWindowText(A/W)
2. **Message box** - bpx MessageBox(A/W)
3. **Alarm beep** - bpx MessageBeep
4. **Dialogbox** - bpx DialogBox, bpx DialogBoxParam(A/W), bpx GetDlgItemText(A/W)
5. **Registry operations** - bpx RegOpenKey(A/W), bpx RegOpenKeyEx, bpx RegQueryKeyValue(A/W), bpy RegQueryKeyValueEx, bpx RegSetValue(A/W), bpx RegSetValueEx(A/W)
6. **Crippled functions** - bpx EnableMenuItem, bpx EnableWindow
7. **Timing** - bpx GetLocalTime, bpx GetSystemTime, bpx GetFileTime, bpx GetTickCount, bpx GetCurrentTime, bpx SetTimer
8. **File I/O** – bpx CreateFile(A/W), bpx OpenFile, bpx ReadFile, bpx WriteFile, bpx _lcreat, bpx _lopen, bpx _lread, bpx _lwrite, bpx _hread, bpx _hwrite
9. **Drive operations** – GetDriveType(A/W), bpx GetLogicalDrives, bpx GetLogicalDriveString(A/W)
10. **Port I/O**, useful for "dongles" – bpio 378 (378, 278, 3BC are the usual port address for parallel port), bpio 3F8 (3F8, 2F8, 3E8, 2E8 are the usual port address for serial port)
11. **String manipulations** – bpx CompareString(A/W), bpx lstrcmp, bpx lstrcmpi
12. **Visual Basic String manipulations** – bpx __vbaStrCmp, bpx vbaStrComp, bpx __vbaStrCopy, bpx __vbaStrMove

For functions come with (A/W), its name is appended with either 'A' or 'W'. They are the result of ANSI or Unicode support:

- 8 bit ANSI – String 'ABCD' is stored as 41 42 43 44
- 16 bit Unicode – stored as 00 41 00 42 00 43 00 44

Many of Microsoft Win32 functions and structures have wrappers to provide Unicode support. The functions or structures that have both ANSI and Unicode support have a

note in the information section of their reference pages. When the application is compiled, the function (or structure) will be substituted with the appropriate version ("A" version for ANSI or "W" version for Unicode). Therefore, if in our program, our call is CreateFile, the compiled code will call CreateFileA (if ANSI) in Windows. Obviously, if we set breakpoints, we need to append 'A' for ANSI functions but 'W' for Unicode.

There are many more functions in Win32 API that can be useful to be breakpoints. For details of these operations, please refer to the Microsoft Win32 API reference [15].

5.4 Useful Op Codes

Typical op codes that can interest crackers are:

- JE (jump if equal) / 74
- JNE (jump if not equal) / 75
- JMP (unconditional jump) / EB
- NOP (no operation) / 90

Tampering can be done by changing these op codes, e.g. from 74 to EB.

5.5 Case Study 1 - TextPad v4.5

The first case study in this project is TextPad, a popular editor. The interesting thing of Textpad is that it is **not free**, but allows for **unlimited trial**. Therefore, it relies totally on the users' honesty on buying the software. The user, can "technically evaluate" the product "forever" without paying.

Version: 4.5, by Helios Software Solutions
Price: US $27.00 per single user license
Website: http://www.textpad.com/index.html
Free Evaluation:

- Unlimited time
- Filename: txpeng450.exe
- File size: 2.52 MB

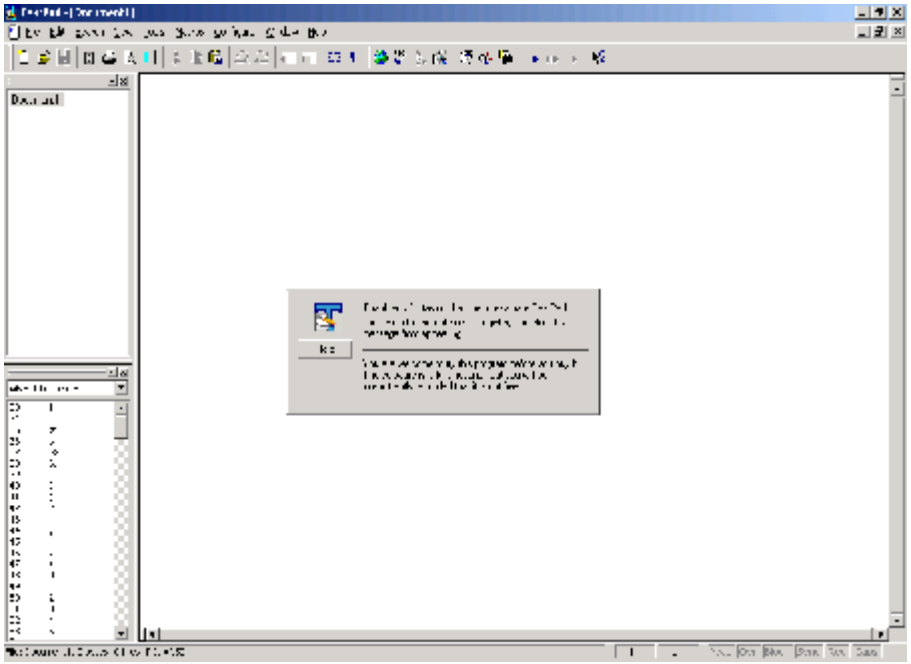

Figure 4 NAG screen of TextPad

The protection used by Textpad is a NAG screen – It holds for several seconds, asking for registration.

The approaches to get around the NAG should be:

1. Use SoftICE to set breakpoints before/during NAG
2. Step through instructions
3. Disables suspicious CALLs, modifying CALL results, etc

So which breakpoints to set?

Because we want to break before the NAG appears, therefore any possible points NOT AFTER NAG is okay. To minimize the number of times stepping through irrelevant instructions, we need to choose one breakpoint closest to NAG.

Notice the NAG displays for several seconds. Therefore, it is intuitive to try breakpoints on functions about time. First we disassemble TextPad.exe, read through the import table

for functions related to time. We found it imports three time-related functions in KERNEL32.

1. GetTickCount - retrieves the number of milliseconds that have elapsed since Windows was started.
2. GetSystemTime - retrieves the current system date and time. The system time is expressed in Coordinated Universal Time (UTC).
3. GetLocalTime - retrieves the current local date and time.

Then we set breakpoint on these functions in SoftICE. Here is the result:

- GetSystemTime doesn't break before NAG
- GetLocalTime doesn't break before NAG
- GetTickCount first breaks before NAG. Then we disabled the breakpoint by "bd *", press "F12" 20 times to step out of the program stack and back into the TextPad.

```
:0045F500 85C0          test eax, eax
:0045F502 7535          jne 0045F539
:0045F504 8D4508        lea eax, dword ptr [ebp+08]
:0045F507 50            push eax
:0045F508 FF7508        push [ebp+08]
:0045F50B 56            push esi
:0045F50C FF156C965800  call dword ptr [0058966C]
:0045F512 85C0          test eax, eax  ß we are here
:0045F514 750E          jne 0045F524
:0045F516 FF750C        push [ebp+0C]
:0045F519 FF7508        push [ebp+08]
:0045F51C FF1594965800  call dword ptr [00589694]
:0045F522 8BF0          mov esi, eax
```

Then press "F10" to step over instructions, notice the screen changes on displaying the NAG. Sometimes SoftICE may block the screen, in this case, press "F4" to get a clear view. After stepping over around 440 times, we reach:

```
:00404EC5 FF75EC        push [ebp-14]
:00404EC8 FF90D0000000  call dword ptr [eax+000000D0]
:00404ECE 85C0          test eax, eax
:00404ED0 7424          je 00404EF6     ß we land here
```



```
:00404ED2 8B8ED0000000  mov ecx, dword ptr [esi+000000D0]
:00404ED8 6A05          push 00000005
:00404EDA E809C00F00    call 00500EE8    ß this creates NAG
:00404EDF 8B86D0000000  mov eax, dword ptr [esi+000000D0]
:00404EE5 FF701C        push [eax+1C]
:00404EE8 FFD7          call edi
```

If we step over the call at 00404EDA, the NAG appears. Notice the jump highlighted at 00404ED0 and the call, this pattern falls in our simple scenario - “if result is good then proceed else penalty”.

So, we changed the statement from

```
:00404ED0 7424      je 00404EF6     to
:00404ED0 EB24      jmp 00404EF6
```

This modification would force the program to bypass the penalty anyway.

Done. A search of the code statement in W32DASM revealed that the statement corresponds to offset 4ED0h in .exe. Finally, we modified the .exe file to patch it permanently.

TextPad was cracked by **changing 1 byte only.**

5.6 Discussions

The implication of the results in our first case study is that, simple software protections, under our threat model, can be cracked easily by changing **1 byte** only! Textpad, although is an unlimited trial software, which is expected to be easy to crack, many other commercial programs, can be defeated similarly. Even if the program is protected by hardware tokens that we cannot duplicate, if the protection can be bypassed in this way, the use of hardware is meaningless. Security is as weak as the weakest link.

So what is wrong? The problem lies in **the routine providing security to the software (called the guard module in [04]) is itself not secure**. Therefore, under our threat model, it is possible to see and mimic what the guard module does, and fool it to let us pass

without the valid key.

As a result, a new protection model is needed, and this cannot be done without advanced protection mechanisms such as encryptions, obfuscation, etc.

6. Advanced protection techniques

Securing "data" has been for long. Securing important data such as keys, database, and password files are very well known and aware by the people. However, the executable code is also a valuable intellectual asset that should be protected. As discussed before, basic software protections can be easily bypassed if the code itself is not secure. The art of securing the code is called "code security".

Here we will look at four different ways to achieve code security – encryption, packing, obfuscation and anti-debugging.

6.1 Code Encryption

The most common way to protect data from eavesdropping is to encrypt it. It is already a prerequisite in electronic commerce today. From the point of view of the encryption algorithm, code and data are essentially the same, therefore code encryption and data encryption can be done in the same way. After encryption, the code will then be immune to normal disassembling and decompiling.

There are many kinds of encryption. In early days, when the computer was very slow, encryption is simply XOR tricks – encrypting and decrypting using the same XOR value. With the increase in processing power, we have more advanced encryption algorithms like DES or RSA. In any cases, the key length remains the most important measurements for how easy the encryption can be defeated.

6.2 Executable Packing

Executable packing is originally designed for compressing executable and yet still let it be runnable, with reduced disk spaces without runtime or memory penalty. Because the original data is scrambled during the "zip" process, it also protects the packed code from normal disassembly/decompiling process.

Packing is commonly used in the software industry because it protects the code with reduced image size. More importantly, making a packed executable can be as easy as making an executable zipped file. Everything is automatic. Some packer programs also have the ability to add anti-cracking measures such as anti-debugging routines in the

packed executable.

Here is a list of commonly used packers: UPX, ASPACK, PECOMPACT, PETTITE, PEPACK, NEOLITE, WWPACK32, PKLITE32, SHRINKER.

6.3 Obfuscation

Obfuscation is the process of transforming the software to unintelligible but still functional code. The aim of obfuscation is to make examine of disassembled or decompiled code yields no useful information; thereby dramatically increases the time required to reverse engineer the code.

There are several ways to add obfuscation to the code [2]:

1. Lexical transformations – e.g. scramble identifiers to replace name of classes, methods and variables by meaningless strings.
2. Control transformations – by inserting **opaque predicates**, e.g. changing the sequential instruction executions "a followed by b" <a;b> to:

   ```
   a;
   if (p = = true)
       b;
   else
       b';
   ```

   This gives an illusion to the reverse engineer that b may not always follow the execution of a, and a may be followed by b'. The predicate p here should always be evaluated to true but very difficult to be deduced by crackers.
3. Data transformations – e.g. through splitting variables to turn the representation of a boolean into two integers. The program is modified to use these two integers to be interpreted as boolean values, such as 0, 0 as TRUE and 0, 1 as FALSE.

Since reverse engineering Java byte code almost yields 1-1 mapping to the source, obfuscation is commonly used in securing java byte code, e.g. SourceGuard. For x86 programs, some packers claim they provide obfuscation to the binaries as well, e.g. PECompact.

6.4 Anti-Debugging

It is a roundabout way to code security. It works by confusing the debugger so that the debugger cannot investigate the internal of the program.

The tricks to confuse debugger are divided into two main categories:

1. Preventive actions – actions that are taken by the program to make the user unable to trace it during program running (e.g. playing with the interrupt)
2. Self-modifying code

These tricks are described in details in [42]. However, to combat with these anti-debugging tricks, crackers also have tricks to do anti-anti-debugging.

6.5 Discussions

Encryption and packing of the code are in principles the same: they transform the code and restore them back to original during execution.

For the encrypted/packed program to be executed, the executable must be equipped with a small decryption/unpacking routine, which must be itself unencrypted/unpacked. When it is executed, the encrypted/packed program will then either be:

1. Fully decrypted/unpacked in memory at runtime before its first instruction starts, or
2. Dynamically decrypted/unpacked thereby remaining most parts of the program encrypted/packed in runtime, partial decryption/unpacking is on-demand.

The first approach is commonly used because the program's performance is unaffected during runtime **after** it is full unpacked/decrypted. The second one will incur heavy performance penalty and thus is not typical in the market.

Because of similar principles in packing/encryption, from now on, unless otherwise specified, 'packing', 'unpacking', 'packed', 'unpacked', also include the meanings of their encryption counterparts.

So what is the challenge of encryption and packing to crackers? Apart from have immunity to disassembling/decompiling, it also adds anti-tampering functionality. Just an analogy with zipping, changing a sentence in a plain text is easy, but changing the plain

text directly in the zipped text is difficult! Unlike zip, some packers doesn't include the unzip function so even if the cracker knows which packer the program is using, he cannot unpack with ease.

Since changing only 1 byte can crack many programs, crackers can easily disseminate small crack files that is programmed to locate a particular file offset and modify that byte. But if the file is "zipped", the entire "zipped" executable will be different - the 1-byte change becomes many byte changes. This makes the crack much larger to be effectively disseminated.

Encryption and packing make the protected code impossible to be read, as the encrypted content is no more valid instructions, in contrast, obfuscation protects the code by making it more difficult to be read, but the obfuscated codes are still valid instructions.

Practically speaking, obfuscated codes do not show structures, usually overwhelm with a large amount of conditional jumps and calls, and include loops that are heavily nested, inter-referencing each other.

With these advanced techniques, a more robust protection model can be made possible, which is described next.

6.6 A more robust protection model

First, we need to modify our program to work with the dependency of the guard module. This may be as simple as containing calls to the guard module, however, to prevent others to disable these calls easily, the program should be encrypted/packed. The guard module initializes the program by decrypting/unpacking it. In this way, the program won't work without the guard module. Before encryption, the codes can be further obfuscated to better protect from reverse engineering.

The guard module checks the presence of the key (either hardware or software) and if it is satisfied, it initializes the program. As described in [04], "the guard module must do its job in complete secrecy. It must be impossible to see what it does, impossible to imitate what it does and impossible to trick it into dosing its job when the key is not really present". Therefore, the guard module should also be obfuscated, encrypted and also protected by anti-debugging measures.

The copy protection measures used by the key should be effective, and hence, we can assume that the key (either software or hardware tokens) here is secured and cannot be duplicated.

This model will make the program significantly more difficult to be cracked and is adopted by professional commercial protection schemes.

7. Advanced protection countermeasures

The techniques discussed above also have their weaknesses, by noting the following:

1. For the code to be executed, it should be decrypted/unpacked in memory – thereby reverse engineering is possible.
2. Obfuscation can only increase the difficulty in code reverse engineering, but not impossible.

Reversing obfuscated codes is just a matter of time, and moreover, whenever crackers encounter these codes, they will probably find another way to get around the protection, instead of spending time into this prepared trap. Our focus here is how to get around the encryption.

7.1 Manual Unpacking

As many programs are not protected by dynamic encryption/decryption of code, therefore, in most cases, when the first instruction of the protected program is to be executed, it must be fully unpacked. By dumping the unpacked content, we will get the "naked" executable. In cracker's parlance, the act to extract these fully unpacked codes is called "manual unpacking". It is an advanced stuff. We will deal with this later in case studies 4.

7.2 Process Patching

If unpacking is impossible and it is difficult to get rid of the encryption, and given the crackers can find the run-time locations of code to be tamper-with, the challenge to them is: how to effectively create a crack file to patch the executable permanently and effectively disseminate it.

The 'solution' is 'process patching'.

After the packer's routine unpacks the program and before transferring control to it, we somehow, seize the control and run our code to patch the unpacked process in **run-time memory**, after that, we transfer the control back to the program as if nothing happens. In this way, we are patching at the time when the program is decrypted/unpacked in memory. Packed program is NEVER changed on disk.

7.3 Case Study 2 - Process Patching TextPad

This case study will give us hand-on experiences with packed programs, therefore, better reinforcing our knowledge and prepare us to do advanced unpacking.

We first use a packer to pack up Textpad. To make it simple, the packer should not provide other protective measures other than code scrambling, here we choose UPX (http://upx.sourceforge.net/). We choose to pack Textpad because it is simple and we are familiar with it as well as its crack.

Use UPX (The Ultimate Packer for eXecutables) to pack Textpad to simulate protection:

```
C:\Program Files\TextPad 4>\upx120w\upx TextPad.exe
                       Ultimate Packer for eXecutables
              Copyright (C) 1996, 1997, 1998, 1999, 2000, 2001
UPX 1.20w           Markus F.X.J. Oberhumer & Laszlo Molnar           May 23rd 2001
        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
   1900544 ->    756224   39.79%    win32/pe     TextPad.exe
Packed 1 file.
```

Rename Textpad.exe to topatch.exe. We treat this as the protected target.

For crackers, it is not difficult to identify a file that has been packed. Some good signs are: disassembler cannot disassemble the program, generating exceptions, or if it can, it shows only the packer's routine. We will discuss some more methods to detect packing in case study 4.

The principle of how packers work is depicted in Figure 5. First, the instruction of the packer's routine is executed (labeled as packed program entry point). At this moment, the program is not yet unpacked into memory (depicted as a block of zero). After the packer finishes unpacking the program into the memory, it will transfer the control to the unpacked program through a jump or call.

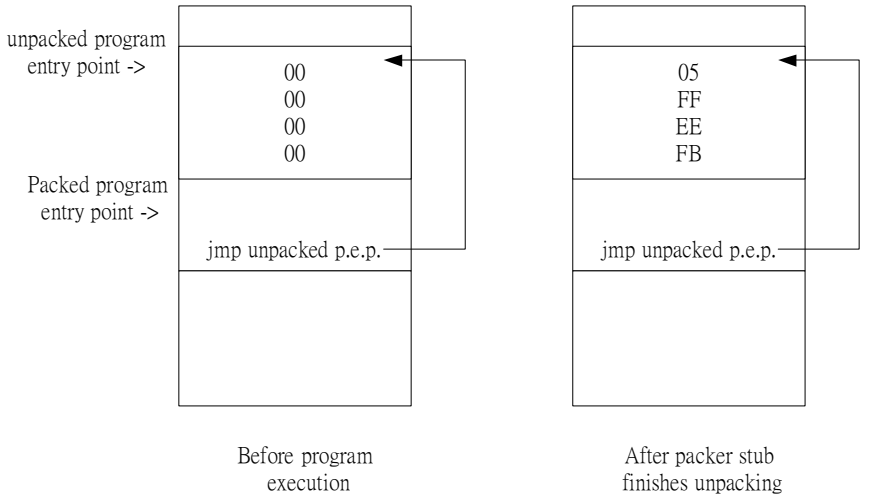

Figure 5 How packer works

Using Visual Studio Debugger, we verify our theory by looking at the end of the topatch.exe. The end of the packer's routine is a jump to 004A038E, which is an uninitialized location before the packer's routine is executed.

```
005D8CB5 89 03              mov dword ptr [ebx],eax
005D8CB7 83 C3 04           add ebx,4
005D8CBA EB E1              jmp 005D8C9D
005D8CBC FF 96 88 E6 1D 00  call dword ptr [esi+1DE688h]
005D8CC2 61                 popad
005D8CC3 E9 C6 76 EC FF     jmp 004A038E
005D8CC8 00 00              add byte ptr [eax],al
005D8CCA 00 00              add byte ptr [eax],al
```


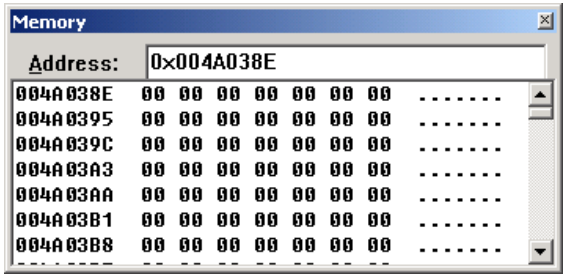

Figure 6 Memory at 0x004A38E

Running topatch.exe, setting breakpoint at GetTickCount, we found that everything is exactly the same as Textpad.exe during runtime. The one-byte patch is still at 00404ED0. In fact, this is intuitive as the operation of the packer should be transparent to the runtime program.

We need to patch topatch.exe so that the program will jump to our code after unpacking. Our code should modify the run-time memory (1 byte patch) and then performs another jump to normal unpacked program execution. See Figure 7.

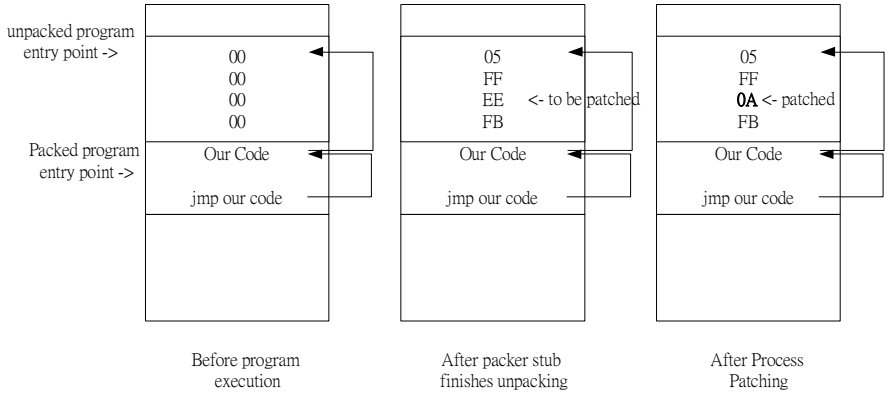

Figure 7 Procedures for Process Patching

We need to find space in topatch.exe to insert to our code. File offset at 0xb2100 contains some spaces (lots of zeros) for our code.

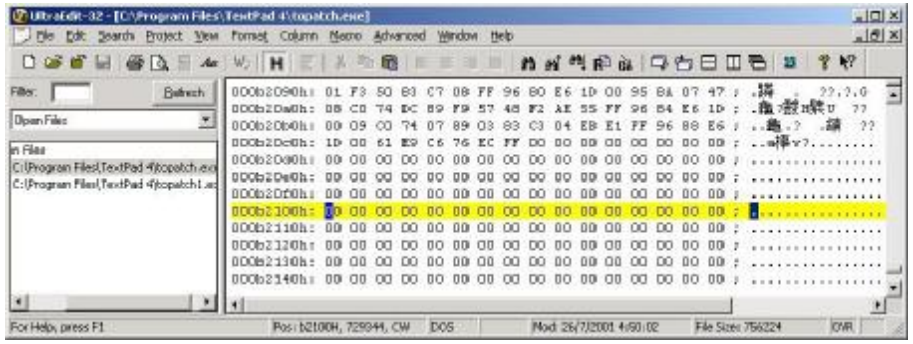

Figure 8 File Offset at 0xb2100

The assembly below is our run-time memory patching code. Remember in case 1, we need to patch TextPad at memory location 00404ED0 with EB.

```
push eax
mov eax, 00404ED0
mov byte ptr [eax], EB   ß 1 byte patch
pop eax
jmp 004A038E             ß unpack program entry point
```

We then need to modify the original **jmp 004A038E (E9 C6 76 EC FF)** at file-offset 0xb20c3 to jump to our code at offset 0xb2100.

But what is the relation between memory address and file offset? We need to know about Win32 PE file format and its memory organization.

PE (Portable Executable) is the native file format for Win32. There are very good references on PE file format [17,22,23]. PE files, as depicted in Figure 9, first come with a header, containing important information for the file.

Most file contents are stored into blocks called ‘sections’. A section is a block of data with common attributes. The whole executable will be mapped into memory, during runtime.

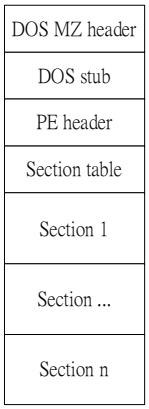

Figure 9 PE Format

Here are some important facts that we should be aware of:

1. Codes in the executable reference each other using relative addressing. The address is called 'Relative Virtual Offset' or 'RVO' for short.
2. Every process has its own 4GB address space.
3. When mapping the whole executable into memory, mapping start at address "Image Base", thus during run-time, code's address = Image Base + RVO.
4. Each program starts execution on its first statement at an address called "Program Entry Point".
5. During the mapping process, the size of sections in file MAY NOT BE EQUAL to that in memory. This is determined by "Raw Size" and "Virtual Size".

Figure 10 and 11 shows the information of topatch.exe displayed by PE Editor bundled with ProcDump, a widely used unpacking tool.

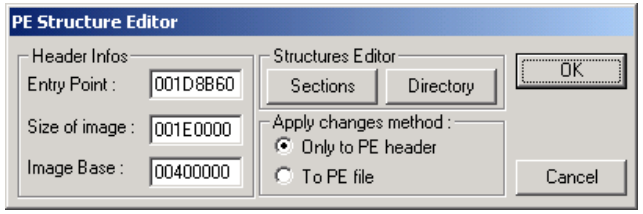

Figure 10 PE Header Information

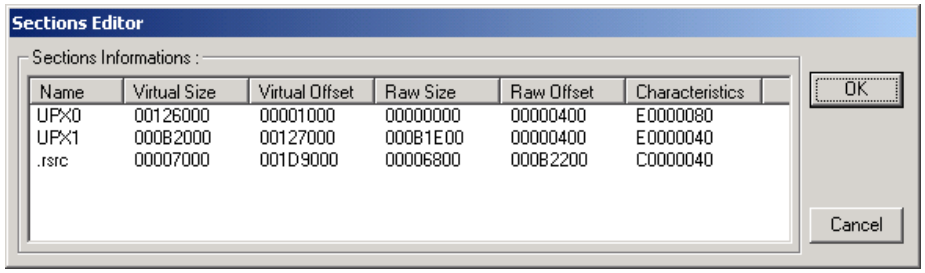

Figure 11 Section Information

Notice the important statistics of sections in Figure 11. Figure 12 shows the mapping of sections in file during runtime.

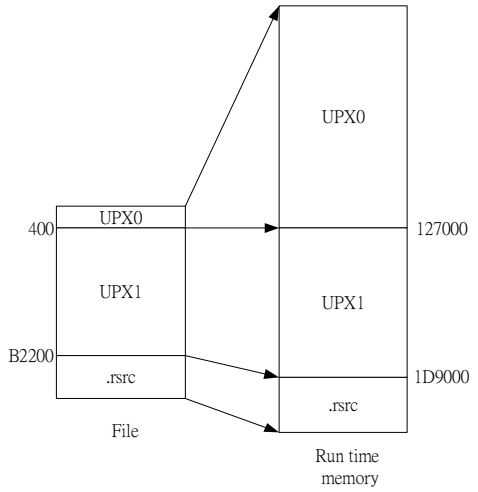

Figure 12 Executable Mapping in Runtime

Therefore offset 0xb2100 in UPX1 in the file corresponds to this address in memory
= Image Base + 0x127000 - 0x400 + 0xb2100
= 0x400000 + 0x1d8d00
= 0x5d8d00

Recalled that in the packer's routine:

```
005D8CC2 61                popad
005D8CC3 E9 C6 76 EC FF    jmp 004A038E
005D8CC8 00 00             add byte ptr [eax],al
```

The jump at 005D8CC3 can be changed in this way:
E9 *cd* JMP *rel32*

E9 is a near relative jump. The parameter cd is the displacement of the destination relative to next instruction at current position. Displacement calculation is always dictated by "Destination address – Source address", therefore our parameter cd is:

0x5d8d00-0x5d8cc8=0x38

Because Intel uses little-endian notation, i.e. lower byte to lower memory location. Therefore our final code = **E9 38 00 00 00** at file offset **0xb20c3**.

For our code's jump statement, the calculation for the cd parameter is similar:
0x4a038e – 0x5d8d10 = 0xffec767e. Therefore the op code for our jump statement is:
E9 7E 76 EC FF

```
005D8D00   50              push eax
005D8D01   B8 D0 4E 40 00  mov eax,404ED0h
005D8D06   66 C6 00 EB     mov byte ptr [eax],0EBh
005D8D0A   58              pop eax
005D8D0B   E9 7E 76 EC FF  jmp 004A038E
005D8D10 00 00             add byte ptr [eax],al
```

Add our run-time memory patch code at 0xb2100 and modify the file offset at 0xb20c3 to jump to our patch code. We have finished **patching the packer's routine** to do run-time tampering for us.

7.4 Discussions

I have illustrated how to patch and thus crack the packed program during runtime in memory. This is called 'process patching', which allows us to do run-time tampering. With this method, we can get around most of the issues arising from encryption/packing

because the program must be fully unpacked during execution time.

Our patch code should be in a section, which is executable in memory, and this must be true in UPX1, as this is the section where the packer's routine is located. This may not be the case in other sections such as '.rsrc'.

Since code and data can be in the same section [17], our patch code may fall on data (e.g. global variable) initially at zero. In this case, our code may be overwritten at runtime. We may need relocation (not at this time).

The 1-byte patch memory location should also be writable because we are doing tampering. This must be always true because it is in where the program is unpacked (written) at run-time.

7.5 Defeating Dynamic Decryption of Code

For programs that are protected by always maintaining most of its code encrypted in memory, with continuously encryption and decryption, they can be defeated in a similar way. These programs look like this:

```
Packer's routine:  For (every encrypted routine segment i)
                       decrypt i
                       jump  i   ß execute segment i
                       encrypt i
```

Since there must be a segment of code unencrypted, we can dump this segment from memory. By **exercising all the different functions of the software**, we can gather all the unencrypted contents. What crackers need to do is to patch the jump statement:

```
Packer's routine: For (every encrypted routine segment i)
                       decrypt i
                       jump   cracker's code ß cracker patch this to jump to do dumping/patching
                       encrypt i
```


Cracker's code of dumping:	Cracker's code of patching:
Dump all unencrypted bytes in i jump I	Check for signatures in the segment If (signature is in this segment) then patch it jump i

8. Case Study 3 – Dreamweaver

Version: 4.0, by Macromedia
Price: US $299.00
Website: http://www.macromedia.com/software/dreamweaver/
Trial:

- 30 days
- Filename: dreamweaver4tbyb.exe
- File size: 24.1 MB

8.1 Preliminary Investigation

Below is the screen shown on running Dreamweaver.

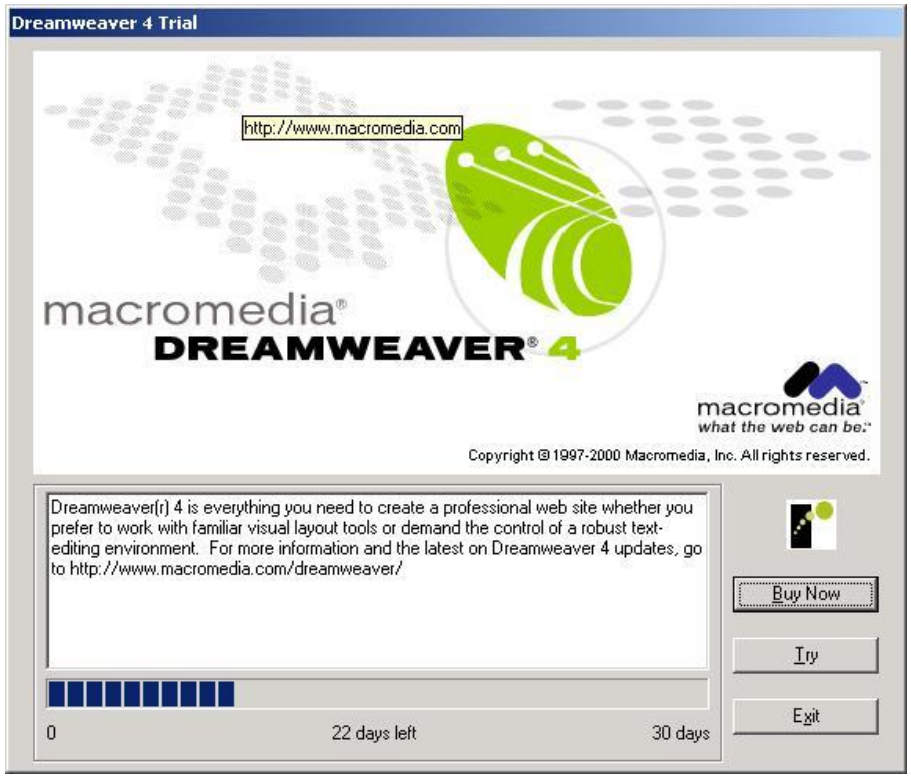

Figure 13 Running Dreamweaver

Changing system date results in security tampering and the trial is disabled.

Figure 14 Tamper warning

Click "Buy Now". We see a screen prompting up for user registration. Note the logo of "**releasenow.com**".

Figure 15 User Registration

When we go through the registration process and reach "Select Payment Method", we choose "Go Offline". This allows us to do transaction without typing in credit card information. We continue through the process. Then we reach "Select Communication Method". Let's choose to do it by phone.

Finally, a screen pops up and asks us to enter the unlock code. The unlock code will be given if we have completed the transaction on phone.

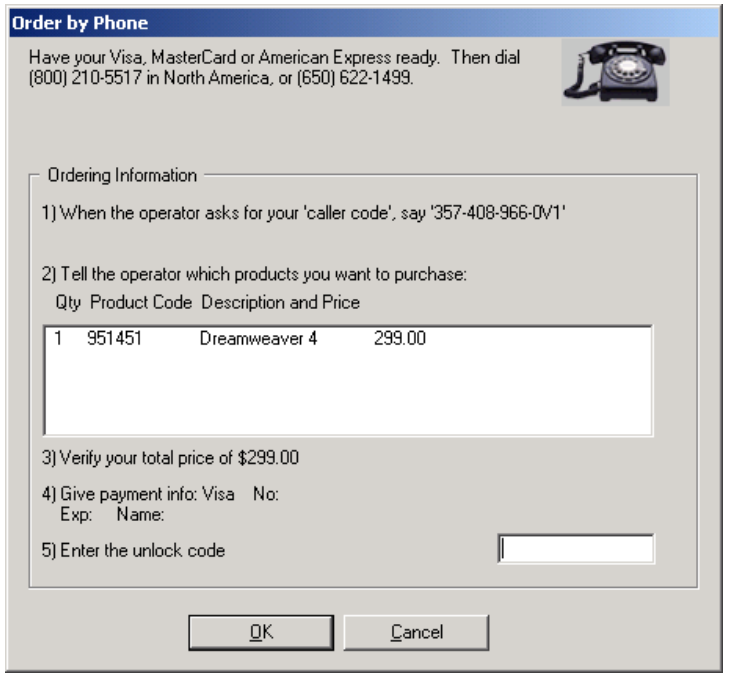

Figure 16 Ordering Dreamweaver by Phone

At this moment, we have the following clues:

1. What is ReleaseNow.com?
2. We have a textbox entering the unlock code. This means there must be a verification mechanism inside Dreamweaver to verify our code. The program will be "unlocked" if the code is correct.

8.2 ReleaseNow.com

The result below is extracted from the address:
http://www.ealaddin.com/partners/software_valueadd.asp?cf=tl
Company: ReleaseNow
URL: http://www.releasesoft.com
Description: ReleaseNow is the leading commerce service provider (CSP) for vendors of digital goods…ReleaseNow offers the essential building blocks of e-commerce, including online store creation, transaction processing and customer support, as well as functions specific to the e-commerce of digital goods, such as **electronic packaging, digital delivery and real-time fraud detection…**

The link to www.releasesoft.com is **DEAD**. Later, I was confirmed that another security company called "Aladdin Alliance" acquired it.

Information extracted from the Macromedia Dreamweaver Trial FAQ
http://www.macromedia.com/software/dreamweaver/trial/trial_faq.html:
The copyright protection scheme **created by ReleaseNow.com** for Macromedia ESD trial software is highly sensitive to changes and to attempts to change the system clock. The copyright protection scheme is also highly sensitive to **modification or deletion of its "secret" security files…**ReleaseNow.com builds Macromedia's ESD technology with high security and **places security information in the registry as well as other places**.

8.3 Imagined Scenario

Based on the above information, below is the imagined scenario for Dreamweaver:

- Macromedia design and final code Dreamweaver. Then it 'outsources' electronic distribution, protection and commerce stuffs to ReleaseNow.
- As the "time left counting and warning" appears in the screen with "Buy now" option as well as with the ReleaseNow logo, it is good to assume that Dreamweaver itself originally does NOT have any protection at all. ReleaseNow, acts as the SECURITY ENVELOPE, provides all the protections for Dreamweaver.
- ReleaseSoft protects Dreamweaver by storing security information (e.g. installation time) into places including **"secret files"** as well as **"registry"**.
- Every time ReleaseSoft runs, it checks against information stored in these places. If they appear invalid or contradicting, program will expire itself immediately.
- ReleaseSoft also provides electronic transaction, allowing software-buying offline on phone through the use of "unlock code". Possibly, after transaction confirmation, user gets the unlocking code from the sales agent.
- The unlocked trial Dreamweaver will probably treat itself as "FULL" version, as all the functions are included in the trial.
- ReleaseSoft security envelope may be stripped out after unlocking (just guessing).

8.4 Cracking Approaches

At this stage, these are the possible approaches:

1. By tracing through instructions for processing "unlock code", the correct "unlock code" can be dumped or we can modify the checking routines/results **à** Thus we can unlock the program and convert it to FULL.
2. We can also trace through the security information checking routines, by monitoring APIs such as GetSystemTime **à** we can get virtually UNLIMITED TRIAL.
3. By trashing all the information stored in these secure places, we can trick Dreamweaver into thinking that it is a FRESH FIRST TIME INSTALLATION.

8.5 First Attempt

Input arbitrary unlock code and hope it says "Invalid Number" by calling MessageBox API, then we can trace using "String Data References" or setting a breakpoint at "MessageBoxA".

Nothing happens after you input the unlock code and press "OK". ReleaseSoft deliberately eliminates ACKs to avoid tracing.

Then we use a must-use API. Type "bpx GetDlgItemTextA" in SoftICE. SoftICE results in two different breaks. There are two GetDlgItemTextA calls in the program to get the unlock code. Obviously, it is used to trick crackers, as GetDlgItemTextA one time is enough.

We first break at rsagnt32!.text+9e99. Rsagnt32 suggests that we are in another module, other than dreamweaver.exe.


```
* Reference To: USER32.GetDlgItemTextA, Ord:0104h
                                  |
:1000AE99 FF1564420210          Call dword ptr [10024264]
:1000AE9F E8EC67FFFF            call 10001690 ß Press F12 and we are here
:1000AEA4 6824630410            push 10046324
:1000AEA9 6828730410            push 10047328
* Reference To: KERNEL32.lstrcmpiA, Ord:02FFh
                                  |
:1000AEAE FF1550410210          Call dword ptr [10024150]
:1000AEB4 C3                    ret
```

Second break is at rsagnt32!.text+99cc.

```
* Reference To: USER32.GetDlgItemTextA, Ord:0104h
                                  |
:1000A9CC FF1564420210          Call dword ptr [10024264]
:1000A9D2 8D8D7CDEFFFF          lea ecx, dword ptr [ebp+FFFFDE7C] ß Press F12 and we are here
:1000A9D8 51                    push ecx
:1000A9D9 8D9574DEFFFF          lea edx, dword ptr [ebp+FFFFDE74]
:1000A9DF 52                    push edx
:1000A9E0 E82BC80000            call 10017210
:1000A9E5 83C408                add esp, 00000008
:1000A9E8 8D8574DEFFFF          lea eax, dword ptr [ebp+FFFFDE74]
:1000A9EE 50                    push eax
:1000A9EF 8D8D68DEFFFF          lea ecx, dword ptr [ebp+FFFFDE68]
:1000A9F5 51                    push ecx
:1000A9F6 B9D00B0410            mov ecx, 10040BD0
:1000A9FB E840CD0000            call 10017740
```

Initial attempt was made to reverse engineer the unlock code checking routine. However after ten days of reverse engineering, twenty pages of hand-written routines were drafted but still no useful conclusion was deduced because:

1. There is no acknowledgement to user saying "input valid" so we can't trace from the back to the final comparison statement. It is difficult to determine the end of routine.
2. The two GetDlgItemTextA are followed by deeply nested CALLs, unconditional jumps (JMP) and conditional jumps (JA/JB/JBE/JE/JG/JGE/JL/JLE/JNE/JNS/JS). Jump and Call sections are inter-referencing each other without structure - deliberately scrambled and obfuscated to trick crackers.
3. The inputted unlock code, instead of being checked by high-level routines like lstrcmp, is treated in bits unit and checked through low level instructions like cmp/test/xor/and, etc.

Clearly, It is a trap to crackers. Although reversing the scrambled codes is just a matter of time, I decided to give up and use other possibly smarter approaches.

Here are the files in the Dreamweaver directory:

Figure 17 Files in Dreamweaver 4 Directory

Because we break on Rsagnt32 in SoftICE, it suggests that we are in another module

(either a DLL or EXE file). Rsagnt32.tty should be a module in PE format, but was made to have file extension .tty – a trick.

Use Procdump's PE Editor, I have checked every .tty files against PE specification. All of them (except dreamtky.tty) are in valid PE format. This means they can be run (for EXE) or be linked (for DLL).

8.6 Second Attempt

At first I want to try approach 2 by setting breakpoints on file and registry operation APIs, but I may get overwhelmed with results because the Dreamweaver, apart from checking security information, also opens many files and registry entries for uses. It is better to find out where the security information is first. Let's try Approach 3. This can be greatly facilitated by the two monitoring tools:

1. FileMon
2. RegMon

The approach:

1. First set our system time to make Dreamweaver expired.
2. Then start filemon and regmon to LOG every file/registry read operations and filter out suspect entries.
3. Try deleting those suspect entries to see if Dreamweaver "refreshes".

You may want to limit the results to dreamweaver process by entering "dream" in the monitoring filter criteria.

These are the results of successful file reading operations from FileMon:

C:\5d0jawja.sys ß suspicious
C:\PROGRA~1\Logitech\MOUSEW~1\SYSTEM\ccmsghk.dll
C:\Program Files\Macromedia\Dreamweaver 4\Dreampop.tty
C:\Program Files\Macromedia\Dreamweaver 4\Dreamtky.tty
C:\Program Files\Macromedia\Dreamweaver 4\Dreamweaver.exe
C:\Program Files\Macromedia\Dreamweaver 4\language.tty
C:\Program Files\Macromedia\Dreamweaver 4\rsagnt32.tty

C:\WINNT\system32\config\software.LOG
C:\WINNT\System32\IMM32.DLL
C:\WINNT\System32\INDICDLL.dll
C:\WINNT\System32\NVDESK32.DLL
C:\WINNT\System32\RICHED20.dll
C:\WINNT\System32\RICHED32.DLL
C:\WINNT\win.ini

Security information would not save under Dreamweaver directory because deleting it will refresh its 'memory'. The DLLs accessed are well known system DLLs. This can be verified on the Internet. The win.ini haven't been modified.

These are the results of successful registry reading operations from RegMon:

HKCR\ultxfile\Format\MSHO0TO0\write ß suspicious
HKCR\ultxfile\Format\MSHO0TO0\open ß suspicious
HKCR\ultxfile\Format\MSHO0TO0\xlate ß suspicious
HKCU\CLSID
HKCU\Control Panel\Desktop
HKCU\Control Panel\Desktop\SmoothScroll
HKCU\Software\Microsoft\Windows NT\CurrentVersion\Windows
HKLM\Software\Microsoft\Windows NT\CurrentVersion\Compatibility2
HKLM\Software\Microsoft\Windows NT\CurrentVersion\Compatibility32
HKLM\Software\Microsoft\Windows NT\CurrentVersion\IME Compatibility
HKLM\Software\Microsoft\Windows NT\CurrentVersion\Windows
HKLM\Software\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon
HKLM\Software\Microsoft\Windows\CurrentVersion\App Paths\Dreamweaver.exe
HKLM\Software\Microsoft\Windows\CurrentVersion\App Paths\Dreamweaver.exe\PATH
HKLM\Software\Microsoft\Windows\CurrentVersion\Explorer
HKLM\SOFTWARE\RS_NT5
HKLM\System\CurrentControlSet\Control\Session Manager

Let's delete the registry entry "HKCR\ultxfile\" and the file "C:\5d0jawja.sys" and re-run Dreamweaver again. We get a warning:

Figure 18 Tamper Warning

Re-run FileMon. This time, we find one more suspicious file – C:\WINNT\System32\e81htwwt.dll

Delete it and Dreamweaver is now "refreshed".

Why there is e81htwwt.dll? This may due to how ReleaseNow handle lockout. Obviously, if information in the .sys and registry expires, the system doesn't need to check e81htwwt.dll and can disable the trial to the user. That's why we don't see e81htwwt.dll in the first file monitoring process.

To conclude, ReleaseNow in Dreamweaver uses the followings to store secure information:

1. HKCR\ultxfile\ (registry)
2. C:\5d0jawja.sys (file)
3. C:\WINNT\System32\e81htwwt.dll (file)

We are done – somehow, although not 'user friendly': the user is required to manually delete these entries when system expires. But at least, our approach 3 works.

Is there any other smarter method, which for example, patch the program checking permanently or "unlock it" into FULL version?

YES! See next attempt for a new approach to the problem.

8.7 Final Attempt

In case study 2, we have already come across programs protected by packing. They come with a small loader, which decrypts the packed content (may be stored with the loader executable or external) in real time and jump to it.

Observations:

1. Dreamweaver.exe is 244KB in size (too small for such a program).
2. Recall that all the .tty file (except dreamtky.tty) is a valid PE.
3. Dreamweaver.tty is 6332KB in size (reasonably large to be the actual Dreamweaver executable, may be it is even packed!).

So is Dreamweaver.exe a loader?

8.7.1 Dreamweaver.exe as a loader

Yes. Run Dreamweaver.exe and stop at the "Buy, Try, Exit" screen. Press Ctrl-Alt-Del, we see that there is only one dreamweaver.exe process in memory.

Now, press "Try" and get into Dreamweaver editor. When we look at the process list again, there are two: Dreamweaver.exe and Dreamw~1.tty.

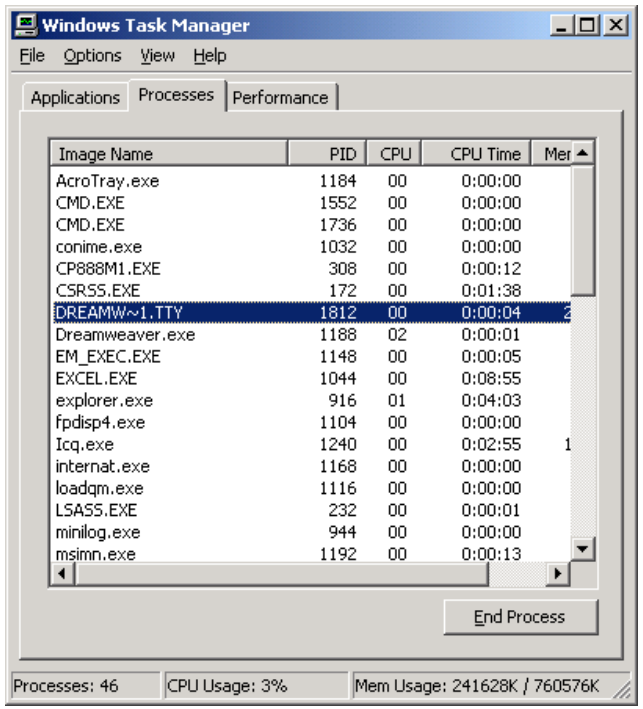

Figure 19 Windows Process List

This suggests that Dreamweaver.tty is the real executable and Dreamweaver.exe is merely its loader, enforcing security check.

I first renamed dreamweaver.tty into tty.exe, and an icon appeared before its filename, the same icon as the one in dreamweaver.exe. It is a good sign.

Then I run it, and got an exception error.

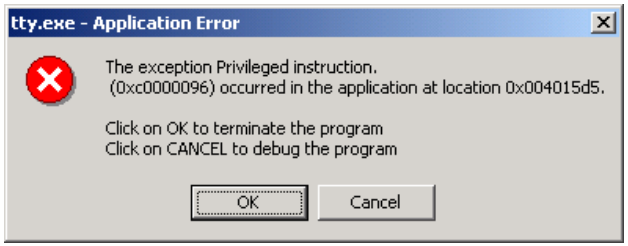

Figure 20 Execution Exception

Disassembly the file, the contents seem to be good (not packed/encrypted) and the import table (see Case Study 4) is intact. No sign shows that it is protected by a packer/encrypter.

This result suggests that Dreamweaver.exe also acts as a patcher!

8.7.2 Dreamweaver.exe as a patcher

Dreamweaver.tty should be the original exe of dreamweaver, but parts of its content are scrambled (at least those at the front because we get an exception from the very beginning). Whether Dreamweaver.exe patch it or not will depend on if we can pass the security check.

Patching can be done on the disk image before loading into memory, or in memory patching. Patching to disk image first before loading is silly because any abnormal program termination would let the "fixed" copy of executable image retained in the disk.

Having the experience of process patching in previous section, I would guess the dreamweaver.tty would be patched at runtime, after we have passed the security check. In this way, we are FORCED to go through the security checking in the loader because only it can patch the tty file.

8.7.3 Annihilating Dreamweaver

Creating a new process in a process needs the API "CreateProcessA" to be used. Type "bpx CreateProcessA" in SoftICE. The detailed disassembly text after break is in the Appendix B.

```
:00401AA6 52                    push edx
:00401AA7 53                    push ebx
* Reference To: KERNEL32.CreateProcessA, Ord:0044h
                                 |
:00401AA8 FF15C0504300          Call dword ptr [004350C0]
:00401AAE 85C0                  test eax, eax
:00401AB0 751F                  jne 00401AD1
:00401AB2 53                    push ebx
```



```
* Possible StringData Ref from Data Obj ->"Error"
:00401AB3 68CC914300          push 004391CC

* Possible StringData Ref from Data Obj ->"Error loading process"
:00401AB8 68B4914300          push 004391B4
:00401ABD 53                  push ebx

* Reference To: USER32.MessageBoxA, Ord:01BEh
:00401ABE FF1530534300        Call dword ptr [00435330]
```

If we put a breakpoint at 00401AA6 and dump edx's content on break, it shows "C:\PROGRA~1\MACROM~1\DREAMW~1\DREAMW~1.TTY". This suggests that we are guessing right.

The above disassembly code means: if the CreateProcess success, we go to 00401AD1, otherwise, an error MessageBox was created.

Win32 Debug API

After 00401AD1, we got two new APIs, the WaitForDebugEvent and ContinueDebugEvent. What are they? According to [31], Win32 has several APIs that allow programmers to use some of the powers of a debugger. They are called Win32 Debug APIs or primitives. With them, you can:

1. Load a program or attach to a running program for debugging.
2. Obtain low-level information about the program you're debugging, such as process ID, address of entry point, image base and so on.
3. Be notified of debugging-related events such as when a process/thread starts/exits, DLLs are loaded/unloaded etc.
4. **Modify the process/thread being debugged ß Process Patching!**

Therefore, with the Win32 Debug API, everyone can code a Debugger! WaitForDebugEvent and ContinueDebugEvent are two of these APIs.

The Debugging Concept

These are the steps used in debugging a process, called the debuggee:

1. Create a process or attach your program to a running process.
2. Wait for debugging events.
3. Do whatever your program want to do in response to the debug event.
4. Let the debuggee continues execution.
5. Continue this cycle in an infinite loop until the debuggee process exits.

The **WaitForDebugEvent** function waits for a debugging event to occur in a process being debugged. The **ContinueDebugEvent** function enables a debugger to continue a thread that previously reported a debugging event. The **DEBUG_EVENT** structure describes a debugging event. Please refer to Appendix A for details.

Of particular interest is the DebugEventCode. Below is its possible values and its meaning. They are extracted from [31]. This is necessary for successfully reversing Dreamweaver.

HEX VALUE	VALUE	MEANINGS
0x3	CREATE_PROCESS_DEBUG_EVENT	A process is created. This event will be sent when the debuggee process is just created (and not yet running) or when your program just attaches itself to a running process with DebugActiveProcess. This is the first event your program will receive.
0x5	EXIT_PROCESS_DEBUG_EVENT	A process exits.
0x2	CREATE_THEAD_DEBUG_EVENT	A new thread is created in the debuggee process or when your program first attaches itself to a running process. Note that you'll not receive this notification when the primary thread of the debuggee is created.

0x4	EXIT_THREAD_DEBUG_EVENT	A thread in the debuggee process exits. Your program will not receive this event for the primary thread. In short, you can think of the primary thread of the debuggee as the equivalent of the debuggee process itself. Thus, when your program sees CREATE_PROCESS_DEBUG_EVENT, it's actually the CREATE_THREAD_DEBUG_EVENT for the primary thread.
0x6	LOAD_DLL_DEBUG_EVENT	The debuggee loads a DLL. You'll receive this event when the PE loader first resolves the links to DLLs (you call CreateProcess to load the debuggee) and when the debuggee calls LoadLibrary.
0x7	UNLOAD_DLL_DEBUG_EVENT	A DLL is unloaded from the debuggee process.
0x1	EXCEPTION_DEBUG_EVENT	An exception occurs in the debuggee process. Important: This event will occur once just before the debuggee starts executing its first instruction. The exception is actually a debug break (int 3h). When you want to resume the debuggee, call ContinueDebugEvent with DBG_CONTINUE flag. Don't use DBG_EXCEPTION_NOT_HANDLED flag else the debuggee will refuse to run under NT (on Win98, it works fine).
0x8	OUTPUT_DEBUG_STRING_EVENT	This event is generated when the debuggee calls DebugOutputString function to send a message string to your program.
0x9	RIP_EVENT	System debugging error occurs

Figure 21 Debug Event Code

Important Constants

Because disassembling shows only the raw HEX values of constants, we may want to know its meaning by referring to its symbol, this can be done by referencing the "include files" in C++ compiler:

- INFINITE equ –1 (0xFFFFFFFF)
- DBG_CONTINUE equ 00010002h
- STATUS_BREAKPOINT equ 80000003h
- STATUS_SINGLE_STEP equ 80000004h
- STATUS_INVALID_HANDLE equ 0C0000008h
- DBG_EXCEPTION_NOT_HANDLED equ 80010001h
- EXCEPTION_BREAKPOINT equ STATUS_BREAKPOINT
- EXCEPTION_SINGLE_STEP equ STATUS_SINGLE_STEP

With these symbols, it will be easier for us to grasp the meaning in the disassembly code.

The Debugging Scenario

1. When the parent process creates the debuggee, the debuggee's primary thread is suspended until the parent calls "WaitForDebugEvent".
2. The WaitForDebugEvent will cause the calling thread to be blocked until the debug event occurs and sent by Windows. The calling thread can specify the time it want to wait (during blocking) in the dwTimeout parameter.
3. The first event to be received is CREATE_PROCESS_DEBUG_EVENT, which is signaled when the debuggee process is just created.
4. Next the Windows Loader will help the debuggee to load the DLLs it need to use, as specified in the import table. This causes LOAD_DLL_DEBUG_EVENT to be signaled.
5. Before the debuggee starts its very first instruction, an exception will occur in the debuggee. It is a breakpoint exception. This cause EXCEPTION_DEBUG_EVENT to be signaled and the exception code should be EXCEPTION_BREAKPOINT.
6. The debuggee will then start its normal execution.
7. In either case, once a debug event is signaled, the debuggee is suspended until the process debugging it calls further ContinueDebugEvent.

Reversing Dreamweaver

401AD1

```
* Referenced by a (U)nconditional or (C)onditional Jump at Addresses:
|:00401AB0(C), :00401AE5(C), :00401CBB(U)
|
:00401AD1 8D442430          lea eax, dword ptr [esp+30]
:00401AD5 6AFF              push FFFFFFFF
:00401AD7 50                push eax
:00401AD8 BD02000100        mov ebp, 00010002
* Reference To: KERNEL32.WaitForDebugEvent, Ord:02CBh
                                    |
:00401ADD FF1540514300      Call dword ptr [00435140]
:00401AE3 85C0              test eax, eax
:00401AE5 74EA              je 00401AD1
:00401AE7 8B542434          mov edx, dword ptr [esp+34]
:00401AEB A1E8AF4500        mov eax, dword ptr [0045AFE8]
:00401AF0 3BD0              cmp edx, eax
:00401AF2 0F85B2010000      jne 00401CAA
:00401AF8 8B4C2430          mov ecx, dword ptr [esp+30]
:00401AFC 8D41FF            lea eax, dword ptr [ecx-01]
:00401AFF 83F807            cmp eax, 00000007
:00401B02 0F87A2010000      ja 00401CAA
```

401CAA

```
:00401CAA 8B542438          mov edx, dword ptr [esp+38]
:00401CAE 8B442434          mov eax, dword ptr [esp+34]
:00401CB2 55                push ebp
:00401CB3 52                push edx
:00401CB4 50                push eax
* Reference To: KERNEL32.ContinueDebugEvent, Ord:0025h
                                    |
:00401CB5 FF154C514300      Call dword ptr [0043514C]
:00401CBB E911FEFFFF        jmp 00401AD1
```


We have a WHILE loop. The starting of while loop calls WaitForDebugEvent (Debug_Event, INFINITE) storing the Debug_Event to ESP+30. 401CAA will call the ContinueDebugEvent (Debuggee_PID, Debuggee_TID, EBP). EBP is default to store DBG_CONTINUE, in rare cases, changed to DBG_EXCEPTION_NOT_HANDLED.

Below is the reversed pseudo code of Dreamweaver's while loop process in Appendix B.

```
WHILE TRUE
{
    DebugEvent ptr ESP+30; ESP=DBG_CONTINUE;
    RESULT=WaitForDebugEvent (DebugEvent,INFINITE);
    IF RESULT==FALSE
        CONTINUE
    IF (PID_FROM_CREATEPROCESS!==D_PID) // D_PID=Debuggee Process ID
        ContinueDebugEvent (D_PID,D_TID,DBG_CONTINUE); // D_TID==Debuggee Thread ID
    ELSE
        IF EVENTCODE>8
            ContinueDebugEvent (D_PID,D_TID,DBG_CONTINUE);
        ELSEIF EVENTCODE==CREATE_PROCESS_DEBUG_EVENT
            Copy CREATE_PROCESS_DEBUG_EVENT STRUCTURE obtained to 0043BD40
            mov ecx,handle of thread
            mov [esp+18],handle of thread
            ContinueDebugEvent (D_PID,D_TID,DBG_CONTINUE);
        ELSEIF EVENTCODE==LOAD_DLL_DEBUG_EVENT
            ContinueDebugEvent (D_PID,D_TID,DBG_CONTINUE);
        ELSEIF EVENTCODE==EXCEPTION_DEBUG_EVENT
            IF EXCEPTION_CODE==EXCEPTION_BREAKPOINT
                {Process Patching}
            ELSEIF EXCEPTION_CODE ==  EXCEPTION_SINGLE_STEP
                                      ||STATUS_INVALID_HANDLE
                ContinueDebugEvent (D_PID,D_TID,DBG_CONTINUE);
            ELSE
                ContinueDebugEvent (D_PID,D_TID,DBG_EXCEPTION_NOT_HANDLED);
}
```

It can be seen that after the security envelope creates dreamweaver.tty as a debuggee process, for all debug events it received EXCEPT ONE, it simply does nothing and resumes the debuggee process by ContinueDebugEvent with DBG_CONTINUE.

The only exception is the debug event – EXCEPTION_DEBUG_EVENT, in particular, with exception code – EXCEPTION_BREAK_POINT.

Recalled that after the debuggee process is created, its imported DLLs are loaded by the loader, and just before its first instruction to be executed, a break point exception is generated.

This is the time the security envelope acts. In the disassembly code, the security envelope does many things, including spawning a new thread running in loop. In particular to our interest, the envelope will **generate the patch data in run time** in its own memory space and then will **"inject" them** into the dreamweaver.tty process (run-time patching).

Because the dreamweaver.tty process is patched by valid code before its first instruction is to be executed, it can be started normally, without problems.

How do I know it? Just set a breakpoint on WriteProcessMemory after the breakpoint exception is received. We will break at here:

```
:00401368 51                    push ecx
:00401369 8B0DE0AF4500          mov ecx, dword ptr [0045AFE0]
:0040136F 52                    push edx
:00401370 55                    push ebp
:00401371 50                    push eax
:00401372 51                    push ecx

* Reference To: KERNEL32.WriteProcessMemory, Ord:02E9h
                                 |
:00401373 FF153C514300          Call dword ptr [0043513C]
:00401379 8BF0                  mov esi, eax
```

The **WriteProcessMemory** function writes memory in a specified process. The entire

area to be written to must be accessible, or the operation fails.

Set a breakpoint at
:00401372 51 push ecx

Then we can see all the parameter contents to WriteProcessMemory:
ECX=84 (hProcess)
EAX=401000 (lpBaseAddress)
EBP=8E6E38 (lpBuffer)
EDX=001000 (nSize)
ECX=F4C7A0 (lpNumberOfBytesRead)

Therefore, the envelope will inject 0x1000 (4096) bytes into dreamweaver.tty process at location 401000. The patch data is stored at 8E6E38.

By comparing memory contents at 8E6E38 ("db 8E6E38" in SoftICE) just before this WriteProcessMemory and before breakpoint exception handling concludes that this patch data is generated runtime.

Dump out the 4096 bytes of patch data at 8E6E38 into hard disk. We can patch this data directly and permanently into dreamweaver.tty file, thereby get rid of the protection.

Dumping the patch data

SoftICE doesn't support memory dumping to disk file unless it is patched (added functionality) by other reverse engineering add-ons.

Here my SoftICE in DriverStudio 2.5 final is patched by NTICEDUMP version 1.13.
http://icedump.tsx.org/

Make sure we have path expert mode off at SoftICE by toggling "PAGEIN D".

To dump the patch code, type "PAGEIN D 8E6E38 1000 C:\DW.BIN" just before WriteProcessMemory is to be executed. The patch data generated by the envelope is now stored in C:\DW.BIN.

Patching .tty file manually

The data should be patched into memory location 401000. Using the PE Editor inside ProcDump, the Image Base is 400000.

Virtual Offset = Memory Location – Image Base = 401000 – 400000 = 1000.

Click "Sections" to open Section Editor, we see the Virtual Offset 1000 corresponds to File Offset 1000.

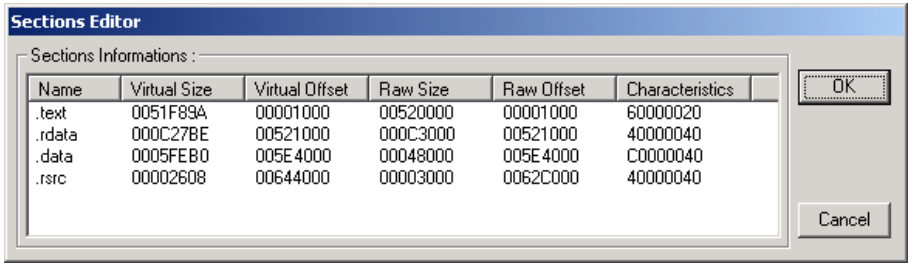

Figure 22 Section information of dreamweaver.tty

Open a HEX Editor, replace the hex data in dreamweaver.tty at offset 0x1000 with the data in dw.bin. The total patch size should be 4096 (0x1000) bytes. Rename the .tty file to .exe.

Dreamweaver will now run with its protection removed.

8.8 Discussions

This ReleaseNow security software is named "SalesAgent" and the past versions of SalesAgent's protection is very stupid, e.g. it displays an ERROR dialog box after user has input the wrong unlocking code [24,25,26,27,28,29].

ReleaseNow is a COMMERCIAL software protection company. SalesAgent is a dedicated software protection envelope, wrapping up the client programs and gives protections and e-commerce abilities.

After reversing the protection method used by ReleaseNow and cracking it, it is disappointed to see how commercial software is being protected by these **so-called commercial protection systems.**

Moreover, defeating SalesAgent implies that all programs protected by it are **immediately threatened**. According to [29], some software from other blue-chip companies are also protected by it:

1. Macromedia: Dreamweaver, Director, Fireworks, Flash
2. Adobe: ImageReady, ImageStyler
3. Symantec: Norton Utilities

ReleaseNow case is a typical example of achieving protection through obscurity because:

1. It obfuscates the codes.
2. It stores secret files in "secret" places.
3. It renames valid PE files to .tty format so as to distract the attention from crackers.

ReleaseNow protection can somehow give reasonable protection against cracker beginners – random walkers; those **randomly** search through the codes, disabling call or inversing jump conditions with little reasoning. Its obfuscation poses difficulties in reversing.

The golden adage - "the security is only as good as its weakest link" still applies here. No matter how good the scrambling is, the protection is easily defeated through the use of file/registry monitors. Using these tools, people without cracking knowledge can still defeat the protection.

For a pricey and popular money making program such as Dreamweaver (US $299.00), protection given by SalesAgent is definitely NOT STRONG enough.

"That's why ReleaseNow disappeared in the market…"

8.9 Suggestions

1. To protect better, SalesAgent can employ anti-monitoring and anti-debugging techniques. In case, it detects a running monitoring tools/debugger, it refuses to run.
2. ReleaseNow should not put its logo onto the screen because this gives clues to

crackers of which external protections the software is using.

3. Instead of merely renaming the .exe to .tty, the .exe file can be encrypted/packed. This makes it more difficult to discover the file as a valid PE.
4. Advance packing technique with Import Table manipulations to the .tty file can be used. Using this technique, merely dumping of the entire decrypted process will not work without Import Table reconstruction. See case study 4.

9. Case Study 4 – Smart Saver Pro

Version: 3.0, by Ulead
Price: US $59.95
Website: http://www.ulead.com/ssp/runme.htm
Trial:

- 15 days
- Filename: Ussp30to.exe
- File size: 6.39 MB

9.1 Preliminary Investigation

First run the program. This dialog box is similar to Dreamweaver. Again, we see a logo called ‘Vbox’ at the bottom left corner.

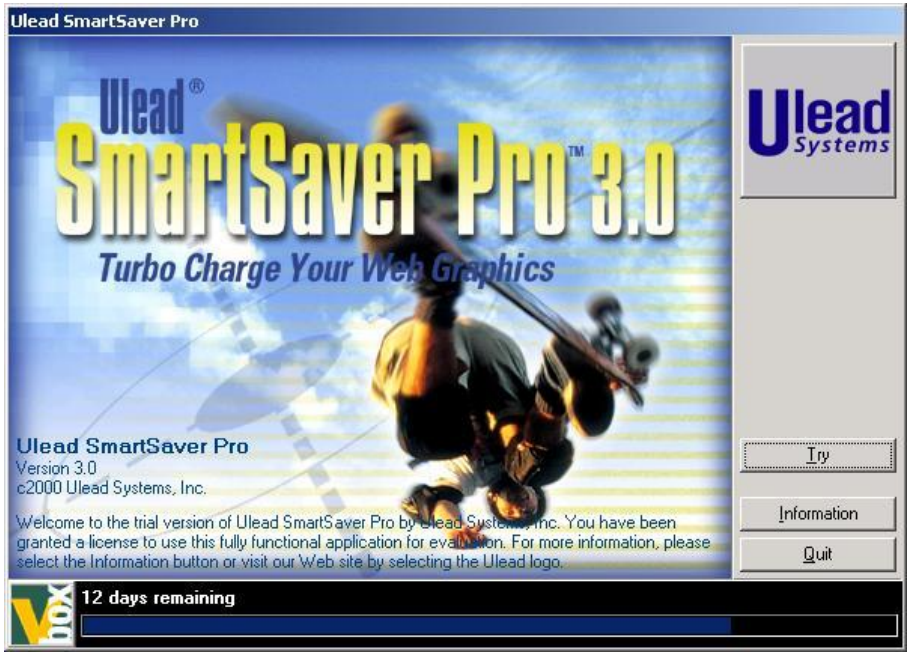

Figure 23 Running SmartSaver Pro

Clicking ‘Information’ pops up the next dialog box, saying that this trial version has been “Vboxed” by Preview Software

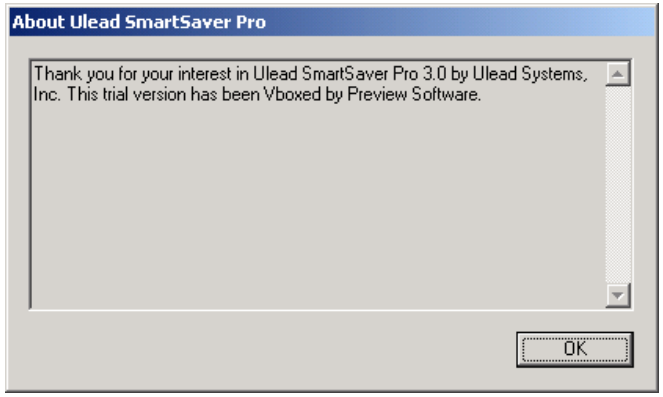

Figure 24 Vbox

9.2 Preview Systems

http://siliconvalley.internet.com/news/article/0,2198,3531_768811,00.html states:
"Caving in to the stringent technological demands of crafting digital products, software maker Preview Systems Inc. called it quits Friday and sold its electronic software distribution (ESD) business to Chicago's Aladdin Knowledge Systems for $5 million in cash."

Same fate as ReleaseNow – it was acquired by Aladdin Knowledge Systems.

9.2.1 Understanding Vbox

The best way to understand how Vbox works is referring to Preview System's description of Vbox. Since Preview System's web site is removed, I can only obtain the information below in Google's web cache:

- Secure Virtual Packaging: Vbox Builder electronically packages a software product securely for distribution via the web, DVD, CD-ROM, OEM hard disk or any other digital medium, using **export-approved RSA encryption**.
- Build Trial-Enabled Applications Quickly and Easily.
- Additional Customization with Vbox SDK API.
- Commerce-Enabling Made Easy

From the same document – "*Preview Systems' **Vbox** Builder allows publishers such as Adobe and Symantec to build robust, trial-enabled versions of their software applications quickly and easily.*"

Okay. Let's try the minimalist's approach. Fire up filemon and regmon. Both programs are able to log down every file/registry access used by Vbox. Therefore, by using just only the monitors, we can defeat its trial protection. It is too trivial to be mentioned any more.

For interest, in case it uses the same approach as Dreamweaver, I look at the process list in Windows and see if there will spawn two processes if I run SmartSaver Pro, i.e. one for Vbox and one for SmartSaver.

NO. Only one.

9.2.2 Cracking Strategy

These are the information we currently know:

- Preview Systems' Vbox protects SmartSaver Pro by a security envelope.
- The security envelope "trial-made" the software, e-commerce abilities are also provided as well.
- It is vulnerable to system monitoring.
- Both Adobe and Symantec are Preview Systems' clients.
- Vbox, decrypts the encrypted code at runtime (encrypted by RSA), in its own memory space and then later transfer the control to SmartSaver.
- Usspro.exe is the program executable.

Since SmartSaver is protected by the so-called "export-approved RSA encryption", and encryption is the key-feature of Vbox, so I decided not to try to reverse the decrypted routine nor try to decrypt the data by myself.

I try to use the technique "Manual Unpacking". The primary idea here is that since SmartSaver must be decrypted completely before its execution, if I can dump this piece of memory content into the disk, I can get an unencrypted version of SmartSaver. By changing the Program Entry Point from the start of Vbox routine to the "Original" Program Entry Point of SmartSaver, the security protection can be stripped off.

In words, it is that easy. But practically, there are many concerns and it requires a deep knowledge of Windows Architecture as well as PE specification.

9.3 Manual Unpacking

References [32,33,34,35,37] provide some basic discussions on Manual Unpacking. However, they are not comprehensive and they simply ignore the implications of architecture differences between Win 95/98/ME and NT/2000. They are already selected quality paper from the cracking world.

In general, to have the manually unpacked program workable, the following procedures should be followed:

1. Locate the Original Program Entry Point (not the entry point for packer's routine)
2. Dumping the memory into disk
3. Fixing the Section Information
4. Regenerate missing information (e.g. Import Table)
5. Affix the regenerated information to the dump file
6. Update Entry Point and necessary PE header information

9.3.1 Locate the Original Program Entry Point

All Windows OS since 95 uses a flat memory model, having a 32-bit linear address that gives 4GB of virtual address space. The virtual addresses used by a process do not represent the actual physical location of an object in memory. Instead, Windows will translate those virtual addresses internally into corresponding physical addresses.

Most Windows programs have their base addresses starting at 0x400000 (4194304) bytes because:

1. For Windows 95/98/ME, the first 4MB of virtual address space is reserved by the system for use by 16-bit and MS-DOS software for compatibility.
2. However, NT/2000/XP are truly 32-bit OS and don't have this restriction, therefore, programs can in theory start at address 0x0.
3. But in order to let programs run on both Windows systems, and to maximize compatibility, most programs start at 0x400000.

This base address is specified as 'Image Base' field in the PE file. However, this Image Base address is only the preferable address specified by the program, it is the Windows loader's decision to map the executable to that preferred address. In case for some reasons, the loader maps it to a different base address then relocation is needed. Although relocation is minimized through the use of relative addressing, there are some cases that relocation cannot be done automatically by the loader (e.g. de-referencing a memory pointer to a memory location). In this case, the executable needs to tell the loader information about these fix-ups in the loaded image. This data is stored in the .reloc section in the PE file.

Since the first module to be loaded (i.e.the program itself) normally will not be relocated (must have no conflicts), it may assume that it must start at its preferred base address and doesn't have the .reloc section.

When the executable is being run, the loader will map the whole executable file into the memory space starting at Image Base. In view of the packer, since it needs to minimize the intervention to the protected executables and to avoid problems, most packers don't modify the base address of these executables. Therefore, in case the packer's routine and the packed content are stored in the same executable, the protected program must be unpacked to 0x400000. And since that program occupies size, this will force the packer's routine to be stored BELOW the image of the unpacked program (i.e. higher memory address). The executable's entry point needs to first point to the packer's routine, so, will have Entry Point VERY FAR (e.g. 0x701000) away from the Image Base. See Figure 5.

This property is drastically different from a normal unpacked program and therefore is a **tried-and-true** way to identify packed executables.

The step of locating the EXACT original entry point is very IMPORTANT because:

1. If we choose those instructions executed before the original entry point (i.e. in the packer's routine), we would get a partially decrypted/unpacked dumped program.
2. If we choose those instructions executed after the original entry point, we can't execute the dumped program correctly because it hasn't been initialized properly (routines between the original entry point and the wrong one are not executed!)

Locating the correct entry point can be very difficult. However, a good sign is for a JUMP

or CALL at very high address to a very low address. Therefore if I see a JUMP/CALL 0x401000 instruction at 0x705500 in the debugger, I would guess that 0x401000 is the original entry point.

However, packer/encrypter aiming at protection will put many traps to trick crackers. They will obfuscate the codes, so that the Original Entry Point cannot be seen easily OR they may explicitly put some calls/jumps to very low address before jumping to the real one.

Back to our SmartSaver, now we put a breakpoint in SoftICE to make it break before it jumps to the original entry point. Here I choose GetProcAddress. Later you will see why GetProcAddress is the more appropriate one.

Make sure you are in "Try-Information-Quit" screen, and "bpx GetProcAddress" in SoftICE. If your SoftICE has other breakpoints on memory access/executions, you may get a warning dialog box after clicking the "Try" button. Vbox does some anti-debugging.

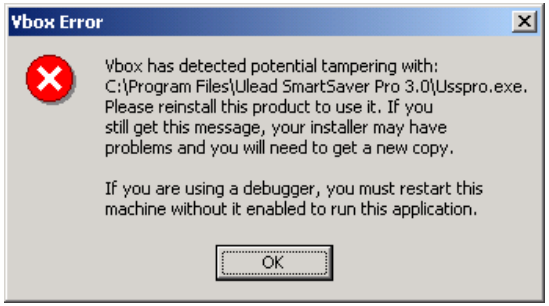

Figure 25 Vbox Tampering Warning

However, if you only have API breakpoints on DLL functions, such as GetProcAddress, you can bypass this anti-debugging check. I just don't know why Vbox doesn't block them as well.

Once SoftICE breaks, "bd *" and Press F12 to return from call. You will notice that we are now in module VboxT410. Press F10 to step over the instructions. Notice any suspect CALL to low address.

Here I use F10 to step over calls because I want to experiment with the first layer of CALLs first. Obviously, Vbox can mask the 'CALL to low address' inside deeply nested CALLs to hide the original entry point (e.g. CALL 0765550 **à** CALL 0675000 **à** … **à** CALL 401000).

Press F10 until you see:

PUSH FFFFFFFF
CALL EAX **ß** This is the first call to EAX, and EAX=4CC1E2

Here we are at USSPRO!PREVIEW+00136020.

It is not that difficult to spot not only because it is the first call to EAX, but it also follows by lots of null instructions (shown below), indicating that we are at the end of the unpacker routine:

0000 ADD [EAX], AL
0000 ADD [EAX], AL

Press F8 and trace into the call **à** we are now at the unpacked SmartSaver's Original Entry Point.

Sometimes, it may be too time consuming in tracing step-by-step manually. There exists tools that allow you to specify the range at where when executions fall in (say 0x400000 to 0x500000) and you will be given a prompt:

- The "tracex" command when SoftICE is patched by ICEDUMP
- The Tracer function in a tool called "Revirgin"

9.3.2 Dumping the memory into disk

004CC1E2 55	**push ebp ß we are here**
004CC1E3 8B EC	mov ebp,esp
004CC1E5 6A FF	push 0FFh
004CC1E7 68 B8 4D 4E 00	push 4E4DB8h
004CC1EC 68 40 C3 4C 00	push 4CC340h

Before dumping the unpacked program, use the ProcDump's PE Structure Editor to get the Image Base and the Size of Image from the header of Usspro.exe. This sets the range that we should dump.

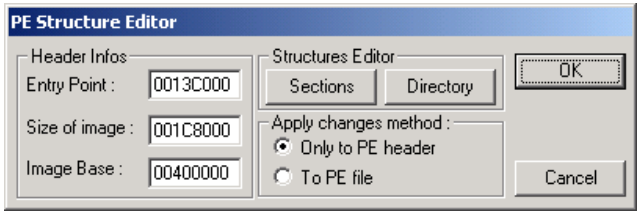

Figure 26 PE header of usspro.exe

Make sure we have path expert mode off at SoftICE by toggling "PAGEIN D". Dump the program by "PAGEIN D 400000 13C800 C:\ss.exe".

9.3.3 Fixing the Section Information

Copy the ss.exe to the SmartSaver directory. Use whatever PE Editor (here PEditor v1.7) to look at the section information.

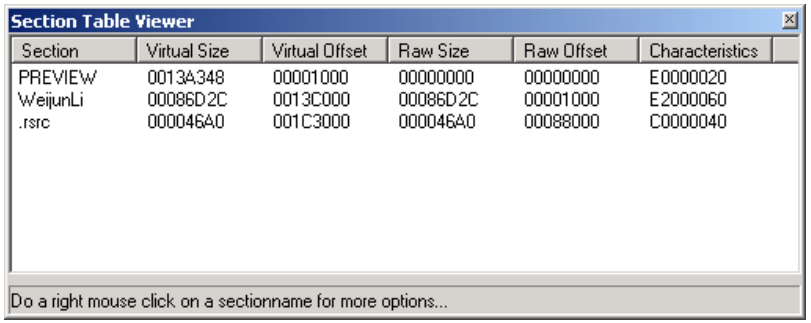

Figure 27 Dump File with wrong section information

Observe that the Vbox's routine start at WeijunLi section. Unencrypted SmartSaver stores at PREVIEW section and .rsrc stores resource information such as icons.

Also notice that ss.exe is a direct image dumped from memory. Because memory at Image Base is dumped to file offset 0, therefore,

- Raw Size should be equal to Virtual Size
- Raw Offset should be equal to Virtual Offset

The current section information is INVALID and should be fixed. Modify the section information to equalize them.

At this point, an icon should display correctly for ss.exe (the same as usspro.exe) because Windows Explorer can locate the correct .rsrc data.

9.3.4 Regenerate missing information

One may wonder now if the Entry Point in the header is now modified to 4CC1E2, will ss.exe be run correctly? The answer is 'DEPENDS'. Windows 95/98/ME have a MUCH HIGHER chance for running correctly than NT/2000/XP. In fact, this is the point that most crackers overlook.

The problem is mainly due to Import Table, rarely among others - a thing that is too difficult to be explained in short. We first need to understand what happen during compilation.

During Compilation

When you write a program in Windows, calling Win32 APIs, say "MessageBox", the assembly code generated would be like this:

```
Push param4
Push param3
Push param2
Push param1
Call addr 1
```

As first discussed by PIETREK in [40], cited by [22,35], "the CALL instruction emitted by the compiler doesn't transfer control directly to the function in the DLL. Instead, the call instruction transfers control to a JMP DWORD PTR [XXXXXXXX] instruction that's also in the .text section. The JMP instruction jumps to an address stored in a DWORD in the .idata section. This .idata section DWORD contains the real address of

the operating system function entry point". "In Visual C++ 2.0, the operating system function prototypes in the system DLLs include a __declspec(dllimport) as part of their definition. The __declspec(dllimport) turns out to have quite a useful effect when calling imported functions. When you call an imported function prototyped with __ declspec(dllimport), the compiler doesn't generate a call to a JMP DWORD PTR [XXXXXXXX] instruction elsewhere in the module. Instead, the compiler generates the function call as CALL DWORD PTR [XXXXXXXX]. The [XXXXXXXX] address is in the .idata section. It's the same address that would have been used had the old JMP DWORD PTR [XXXXXXXX] form been used."

In either case, there must be at least an indirect call and it is through the address XXXXX. XXXXX stores a DWORD value referring to the actual address of the API function of the DLL in the memory, in this case, the MessageBox. See Figure 28.

The region of memory storing a group of DWORD value is called **Import Address Table (IAT)**. Each DWORD value stores the address of a particular function exported by one DLL. Therefore, if a program uses 5 DLLs, there will be 5 different IATs, each storing the addresses of exported functions needed only by the program.

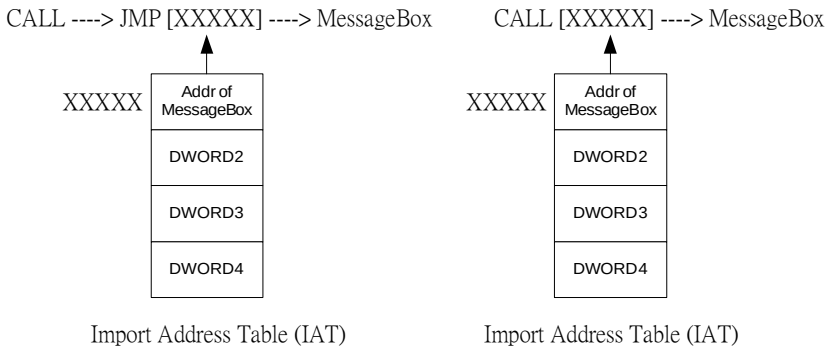

Figure 28 CALL to IAT

In practice, these different IATs will stack together into one single chunk of memory, and can be treated as one big IAT.

The IAT

The reasons for having this indirect call/jump and IAT structure is best again described by PIETREK [40], "After contemplating this for awhile, I came to understand why calls to DLLs are implemented this way. By **funneling** all calls to a given DLL function through one location, there's no longer any need for the loader to patch every instruction that calls a DLL. All the PE loader has to do is put the correct address of the target function into the DWORD in the .idata section. **No CALL instructions need to be patched.**"

Therefore the IAT is where Windows PE loader will fix during runtime, storing addresses of the required DLL functions. So here comes another problem, how can the loader know which functions in which DLLs are required by the program?

The answer lies in the use of **Import Directory Table** and **Import Lookup Table**. Note that the naming here differs in different sources. Import Lookup Table is called "OriginalFirstThunk" and IAT is called "FirstThunk" in [17]. Here I will use the notation and naming as specified in [23].

The Import Directory Table + Import Lookup Table + Import Address Table + others minor fields together form the so-called "Import Table" or ".idata" section.

The Import Table (.idata)

According to [23], the import table has the structure like this:

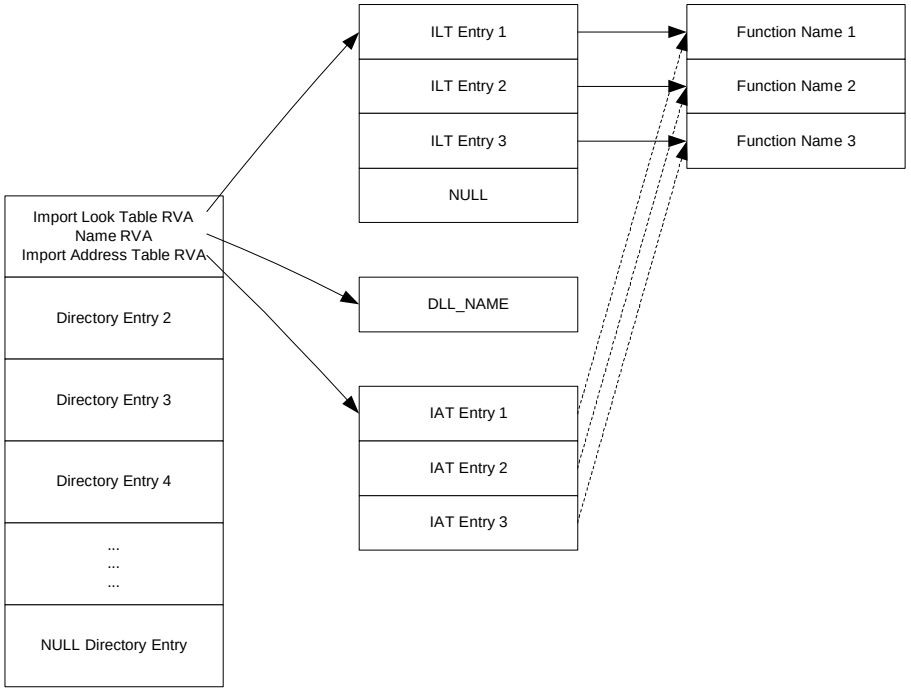

Figure 29 Structure of Import Table

The import table begins with Import Directory Table, containing directory entries. Each entry corresponds to a DLL required by the program. The executable specifies the name of the DLL it requires in the Name RVA field of the directory entries. If the program imports the function in that DLL through function name, the address of the function name is stored in the Import Lookup Table entry. If it imports the function through index (called ordinal), the Import Lookup Table entry will contain the ordinal number.

The function names required in the DLL is stacked up together to form a Name Table. Usually, individual small name tables stack up together to form one SINGLE big name table for the whole .idata section.

Before the program is executed, the structure and content of the IAT are identical to that of the ILT, therefore, it is also pointing to the Name Table initially.

Binding

When the executable runs, the Windows loader will fetch the information stored inside the IDT and ILT to see what DLLs and what functions inside the DLL the program needs, then the loader will load those DLLs into the address space of its process and then fix the addresses of respective functions in the IAT entries - a process known as “binding”.

Therefore, after binding, the IAT will be fixed with the address of the DLL functions. As a result, the indirect jumps in the program code now become VALID.

The address of the whole Import Table structure is specified inside a particular field inside the PE header. Usually, the Import Table occupies its own section (with name commonly as .idata), but it can technically reside in other sections (e.g. code) as well.

Relations to Unpacking

If we look at the import table of usspro.exe (e.g. using disassembly program or PE Editor), we can see that, sensibly, the import DLLs and their functions are too **little** for SmartSaver Pro 3 (see Appendix C).

The implication of this is that when the Windows loader loads the packed executable, the loader only loads the DLLs required for the **packer’s routine** and fixes only its IAT. Therefore, indirect jumps/calls belonging to the packer’s routine are able to call into the correct functions.

However, this is not true for the IAT of the protected routine, as the IAT, among with its codes/data, are only treated as encrypted data belonging to the packer’s routine. Since the Windows loader only acts according to the PE header, and the Import Table pointed to by the PE header is the packer’s. Therefore, Windows loader cannot fix the IAT of the protected routine anyway, and even if the protected routines is fully decrypted, the IAT is invalid and therefore, any indirect jumps/calls to these invalid addresses will **generate an excpetion**.

To fix this and allow the unpacked program to run properly, **the packer’s routine would do the job of fixing the IAT, as if it is the OS loader**. As a result, at the time the control is transferred permanently to the protected program, everything is decrypted and all the

necessary things (most importantly the IAT) will be fixed.

The packer can fix this IAT by calling two WIN32 APIs – LoadLibrary and GetProcAddress. The LoadLibrary function maps the specified executable module (e.g. DLL) into the address space of the calling process. The GetProcAddress function returns the address of the specified exported DLL function. Using these two APIs, the packer's routine can emulate the work done by the OS loader.

Now we have the dump file "ss.exe" and has its section fixed. Let's briefly summarize what is containing in the file. The corresponding file is shown in Figure 30.

1. A PE file header from the usspro.exe. Of particular interest is the Import Table RVA field pointing to the Import Table structure used by usspro.exe, containing DLLs and functions used by the packer's routine.
2. Unpacked program
3. A Fixed IAT for use by the unpacked SmartSaver Pro

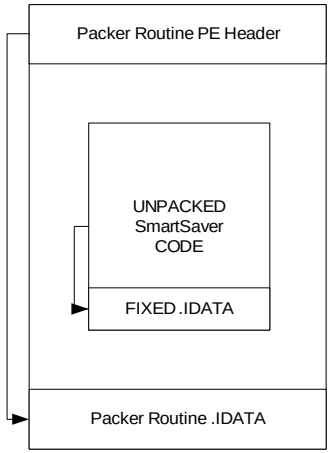

Figure 30 Unpacked SmartSaver inside the old PE header

The Subtle Point

Before, it is said that the dumped image (unpacked program + fixed IAT) + correct entry point won't guarantee you can run the dump file without problems and cases are different on NT/2000/XP, here referred to as "NT series" and 95/98/ME, here referred to as "95

series".

To begin our discussion, we first run the dump "ss.exe" file. We got an application error. Then use Visual Debugger and try to debug ss.exe by "step into" it. I got another dialog box error and no instruction (not even the first one) has been executed.

Because the first instruction at the entry point doesn't have a chance to be executed, there must be a problem in the PE loading process. As the PE loader works according to the PE header, this suggests that there may have some problems in the PE header.

But we got our header from "usspro.exe", why "usspro.exe" can run smoothly but "ss.exe" not? This is one of the tricks used by Vbox.

Use PEditor, open "usspro.exe" and look at its import table:

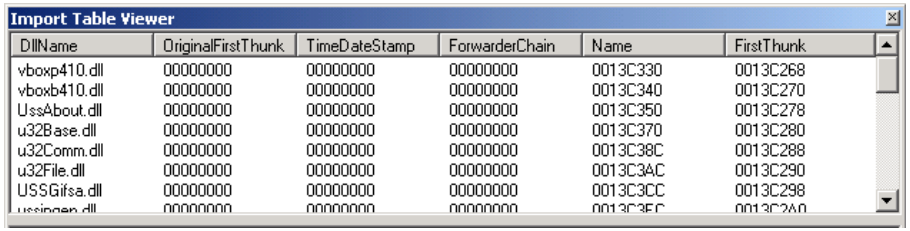

Figure 31 Import Table of usspro.exe

As said earlier, the OriginalFirstThunk (Import Lookup Table) should contain pointers to function names. Here it is 00000000. Therefore, it is not a fully Microsoft specification complaint PE header.

However, this executable can still be able to run up smoothly. This is because the Windows loader can use the information stored in the IAT entries. Recalled that the IAT initially (before binding) should have the same content as the ILT, the loader of Windows, instead of denying loading the malformed PE header, will treat the data in IAT as if it is in ILT. Therefore, the program is allowed to start. This NULL ILT issue has been discussed in [40] as a bug in Borland C++ compiler.

The implication of this is that after binding, the IAT is modified, and is no more pointing

to the function name. Since our "ss.exe" is dumped after IAT is modified, therefore, our import table is CORRUPTED – the ILT is NULL and the IAT is not pointing to a valid function name array.

This explains why the loader fails and it is the trick used by Vbox to stop people from dumping the process.

We can restore this IAT to point back to function names by:

- Using an editor to open usspro.exe, go to first FirstThunk entry in the file, the offset should be 0x13c268-0x13c000+0x1000=0x1268 because RVO 0x13c000 corresponds to 0x1000 in the file.
- Copy the data from 0x1268 down to 0x1330. We stop at 0x1330 because the next byte is the beginning of the string "vboxp410.dll". The total data is 200 bytes long.
- Patch this data into "ss.exe" starting at offset 0x13c268 because the file offset and the VO are equal in this file.

Now we can run "ss.exe" without any seeming problem, and the Vbox protection has been bypassed. But this is not a stable unpacked file. The unpacked executable may not run on other computers. Why?

For NT Series OS, the first 2GB (0-2GB) address space is private for the process. The upper 2GB (2-4GB) is reserved for the system.

For 95 Series OS, the 4MB-2GB area is private for the process. In the 2GB-3GB area, it is a shared area, readable and writable by all processes. A number of system DLLs and other data are loaded into this space. The 3GB-4GB area is system memory, readable or writable by any process.

According to [39], under Windows 95/98, the operating system DLLs, such as KERNEL32, USER32, and GDI32, reside in the **shared** address space. Therefore, everyone own the same copy of these DLLs. Hence, it is possible for one application to interfere with the working of another application. It is also possible to load other dynamic link librarys in the shared address space as well. These DLLs again may get interfered if the DLL is used by multiple applications in the system.

Windows NT, on the other hand, loads all the system DLLs, such as KERNEL32, USER32, and GDI32, in the **private** address space. Therefore, everyone's view on these DLLs is different. As a result, it is never possible for one application to interfere with the other applications in the system without intending to do so. If one application accidentally overwrites these DLLs, it will affect only that application. Other applications will continue to run without any problems.

Let's take a look at our "ss.exe" in Figure 30. Here, our dump file implicitly assumes that the packer routine's .IDATA section will contain all the DLL information required by SmartSaver and let the loader loads the DLL into the process. Although, the Vbox here contains all the required DLLs information in its .IDATA section, **the packer can specify only those DLLs it needs** (but not for the protected program). This is because the packer can emulate the loader by using the previously discussed APIs and explicitly loads those required DLLs for the program during unpacking.

Therefore, our dump file **will not work** under Windows NT series if the packer's .IDATA section doesn't include the DLLs required by the protected program, as the loader would not load the required DLLs into the private address space of the process.

However, our unpack file **may work** under Windows 95 series, because major DLLs (KERNEL32, USER32 and GDI32), and also possibly others are in the shared address space. As long as one other process in Windows loads a DLL, the DLL will be in memory, accessible by every other process. In this case, although the loader won't load the required DLLs, these DLLs may be already existed in memory.

This explains why these dump files (fixed IAT + unpack code) + original program entry point may sometimes work in Windows 95 series but not Windows NT.

Another point that needs to be mentioned is that even if the dump file's .IDATA section contains all the DLL information the program needs, the program may still fail during execution. This is because the **pre-fixed IAT is vulnerable to address relocation**. If the loader loads a DLL and finds that the DLL needs to be relocated from an address that is used at the time the file is dumped, all functions that are exported by this DLL need to be relocated. But the loader is not able to pass this information to the pre-fixed IAT used by the program. It can only fix the IAT used by the packer's routine.

If the program is executed on another computer using some DLLs of different versions, the program may not be able to start up properly, as the address of the functions can be different.

In any cases, **our previous approach is NOT a good one**, as the dump file generated by that method can only be able to run on specific Windows at specific computer occasionally.

To deal with the problem, a COMPLETE reconstruction of .IDATA section is needed, containing all the DLLs and all the functions required by the program. Instead of relying on a pre-fixed IAT table, we should rely on an IAT that will be fixed by the loader every time the program runs.

.idata Reconstruction

To completely reconstruct the import table, we must be able to:

1. Reconstruct completely the IDT, containing entries for every DLLs required by the program.
2. Reconstruct completely the ILT for each IDT entry, containing entries for every required function for the particular DLL.
3. Modify the IAT RVA for each IDT entry to point to the IAT location that is actually used by the program.

For 1 & 2, it can be difficult to be achieved because normally packer's routine, would manipulate (e.g. trash, encrypt or create fake data) the .idata section after using it, leaving only a fixed IAT required by the program. This makes us difficult to reconstruct the .idata section. The program will not be affected by the trashed data because the program only needs the correct IAT for execution.

For 3, it is not difficult to locate the IAT used by the program, as the CALL to API inside the code is implemented as CALL [XXXX] or CALL **à** JUMP [XXXX]. XXXX is the location of an IAT entry (see Figure 28).

In the past, doing 1 & 2 is a hard work and requires many human interventions. E.g. the cracker needs to set a breakpoint on LoadLibrayA or GetProcAddress and then dump out the parameter to these APIs to get the name of the functions/DLLs. Or the cracker needs

to patch the unpacking routine so that those naming information will be written onto disk.

A piece of software may call many API functions in different DLLs, it is not uncommon to have a program that uses over hundreds of functions. Manually reconstructing the import table is too tedious and may pose technical challenges for crackers.

Therefore, **Packing with Import Table Manipulation** is provided as common features in commercial software protection schemes.

In our target Vbox, the Import Table is being manipulated. This is verified by setting a breakpoint at GetTextColor, SoftICE breaks when you play around the software, however, the ss.exe doesn't contain the plain text "GetTextColor". If the program uses GetTextColor, it should exist in the ILT in its .idata section.

Import Table Reconstruction is now made easy though the use of a program called "Revirgin":

- http://tsehp.cjb.net/
- Version 1.31

Its principle is very simple. First it needs to locate the IAT location of the program. Then, for every IAT entry, Revirgin **compares** them to ALL possible API's export values. E.g. GetTickCount are exported at 0x77E839AD. **For Revirgin to work, the API addresses in the IAT should not have relocated**, i.e. at well-known standard exported addresses.

It also supports redirection – a technique used by packer to confuse crackers by making the IAT to point to some more nested layer of calls instead of the function address.

Let's rebuild our Import Table:

1. Execute the protected usspro.exe, click 'Trial' to run the program.
2. Fire up Revirgin.
3. Select usspro.exe as the process to be attached.
4. Revirgin will then examine the PE header and prompt you a dialog box, saying import is corrupted.
5. Enter 4CC1E2 as the OEP and click <Fetch IAT>.
6. The RVA of the IAT is detected at D9000, with length FF4.

7. Make sure <Show IAT referers> is checked, Click <IAT Resolver> to resolve the IAT entries.
8. Click <Resolve again> to resolve unresolved redirected entries.
9. At this moment, Revirgin should resolve most IAT entries, unless for those under encryption (not in our case).
10. Notice any unresolved entries. Note their reference counts. If those unresolved are not referenced by others. We don't need to fix it.

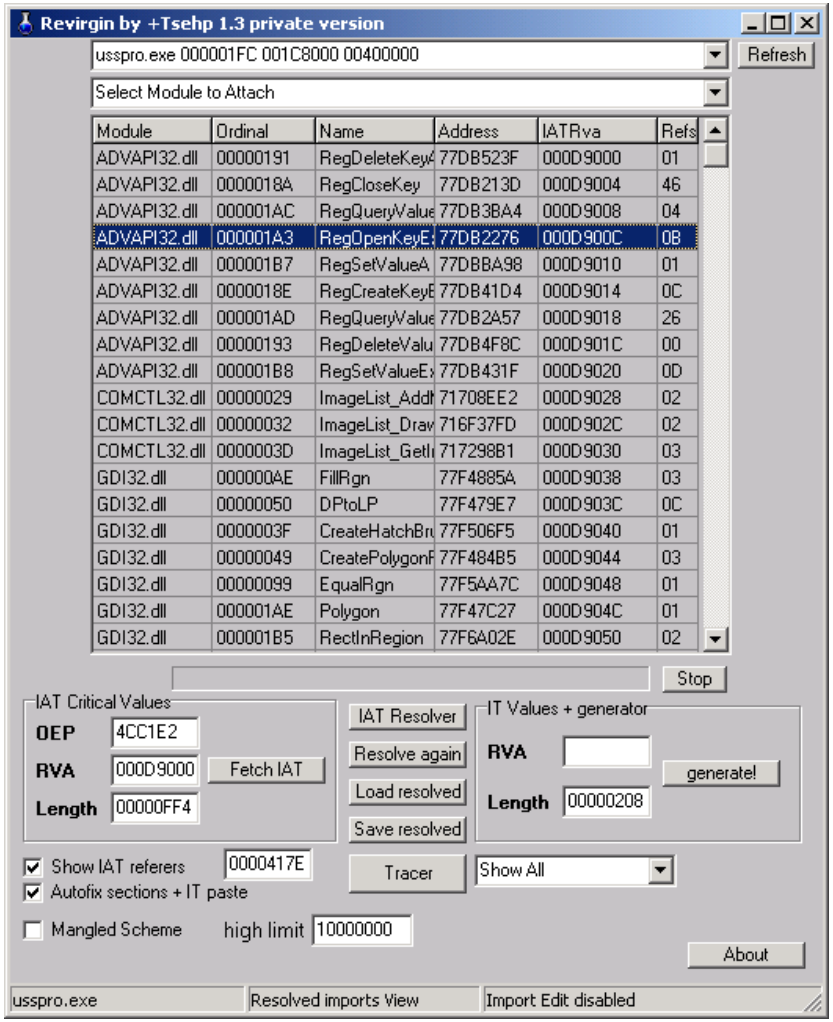

Figure 32 Revirgin in operation

11. Finally, we need to generate a new Import Table based on these results, and paste it into the end of "ss.exe". Check <AutoFix Sections + IT paste>, we want to let Revirgin to fix the IT automatically.
12. Since Revirgin has a bug on working at the boundary of the file, we first need to add an empty section at the end of ss.exe, using PEditor (e.g. name=.temp, VS=0000, VO=1C8000, RS=0000, RO=1C8000). Note that the size of the image of "ss.exe" as seen from the PE header is 1C8000.
13. In IT generator, put the value "1C8000" in the RVA field. Click <generate> and choose to patch "ss.exe". It will save the generated IT for you as a separate file as well.
14. A dialog box will prompt up saying that some entries have not been resolved, we can ignore it because those unresolved has zero reference count.
15. Revirgin will then append the generated IT to the end of the file as a new section, fixing the PE header (size of the Image, the Import Table RVA, its size and the new section characteristics) for you automatically.
16. Finally, delete the added ".temp" section.

Now if you examine the new .idata section of "ss.exe", you will find that all the information about DLLs and functions used by SmartSaver Pro have been added. Therefore, the IAT of the new ss.exe is no more a static fixed one, but the one that is generated dynamically by the OS loader. This implies that we can run the ss.exe on 95/NT series OS and on different computers without problems.

Other possible missing information

Apart from the import table, in very rare cases, we need to fix other things as well, as the packer may deliberately manipulate them. However, there are not many things that packer can manipulate WITHOUT affecting normal program executions and the restrictions imposed by the OS.

It is possible for the packer to manipulate (e.g. trash) the relocation information (.reloc section) in the executable, without affecting its execution, because in most cases, the executable is mapped to its preferred starting point and doesn't need to use the .reloc

section.

As a result, our dump file will lose the .reloc section, therefore vulnerable to address relocation.

Note that in this case, the packer's routine needs to handle relocation by itself as well, working everything the loader should do. This will complicate the packer (making it virtually a loader). This method is rarely used.

Import Table manipulation, remains the most dominant technique to avoid unpacking.

9.3.5 Final fix ups

The last step is to ensure everything is all right:
- Affix the regenerated information to the dump file.
- Update Entry Point and necessary PE header information.

Done! We now have a workable ss.exe with the Vbox security envelope removed.

9.4 Discussions

Ulead designed and coded SmartSaver Pro. Then it added electronic distribution, protection stuffs to its executable by using Vbox from Preview Systems. Vbox, works like Sales Agent by wrapping up the protected executables, acting as a security envelope. However, unlike Sales Agent, the protection routine and the executable, are stored in the same file.

Vbox provides some levels of anti-debugging, it also encrypts the executable by RSA, thereby protecting SmartSaver from cracking and more importantly reverse engineering. In this way, it is more sophisticated than SalesAgent. But, it is still vulnerable to file/registry monitoring.

For crackers, they can avoid working out the decryptions by unpacking. However, unpacking is not an easy stuff (esp. for most lame crackers), as we need to deal with lots of fix-ups. Vbox further protects the software from dumping by using Import Table manipulation. Two approaches have been discussed to fix the dump file (one bad one and

one good one).

Both ReleaseNow and Preview Systems are acquired by Aladdin. We also saw that some big software firms (e.g. Adobe, Symantec, Macromedia) are using their products. It should be stressed that once these security envelopes are defeated; all those products using them are immediately threatened.

9.5 Suggestions

Vbox can do these measures to protect better:
1. Employ anti-monitoring and stricter anti-debugger measures.
2. The IAT can be further protected as well (thus avoiding the use of automatic tools like Revirgin). IAT entries, instead of storing direct addresses to the DLL functions, can be redirected into nested calls or decryption routines to decrypt the DLL function addresses at runtime and call into it.
3. It is good to use several kinds of IAT manipulations at the same time, e.g. some IAT entry contains direct entry to DLL function, some uses 1-level redirection, some use two. This non-uniformity will make tracing more difficult.
4. Also, the packer should not include all the required DLL information in its .idata section, therefore cracker can use our first method to crack it (easier approach).
5. Instead, the packer should only include the DLL information required by the packer routine only, and it should fully emulate a loader, use LoadLibrary in its runtime unpacking routine to explicitly load the required DLL into the process memory. Then it should use GetProcAddress to fix the IAT. (In Vbox, it relies on the OS to load the DLLs and use only the GetProcAddress to fix the IAT).
6. Since locating the program entry point is the first and the most crucial step in successful unpacking. Therefore, this address should be masked in a more sophisticated way. The OEP, here is stored in a CALL EAX, is so easy to be spotted because it is the first appeared call that CALL into a register and it contains very low memory address (at 4xxxxx). This CALL is at the outermost program layer as well. Instead, this CALL to OEP can be put into a location under nested calls/jumps (not at least the first outermost one). Before calling into OEP, there should be some fake calls into low memory address. The first call to low memory address is the first address that will be tried by the cracker! Also, it is better to try using a non-uniformity of patterns of CALLs, comprising different call methods. E.g. CALL EAX, CALL EBX, CALL 4xxxxx, CALL [7xxxxx].

10. Future of software protections

10.1 Code Partitioning

In underground community, crackers believe that "whenever the program run on their computers, the program can be cracked". This is very true. From the case studies, we can see that encryption can be defeated by unpacking or process patching at the time after the program is fully unpacked. Dynamic decryption of code can also be defeat as well. Code obfuscation just renders the program more difficult to be cracked, but not impossible.

The final solution may be to relegate the part of the program that needs to be protected to a third location.

10.1.2 Relegating through networks

If it is a third location over the network, it can be the software vendor itself or a trusted third party. The implication of this is that whenever the user needs to use the program, it sends the requests and gets the results back through the network. The protected code is never exposed.

However, code partitioning through the network can pose the following issues:

1. We need to assume the underlying network is secure, which is not true over the public Internet.
2. The performance of the program is now also a function of network performance. So it is critical to determine how much codes should be relegated on the remote server. If it is too small, the crackers can guess and emulate the server code by patching around the program. If it is too much, the program performance will be fluctuating, depending on network traffic.
3. There is mobility issue as well. The host running the program needs to maintain a constant network access, which is not possible for notebook, modem, or PDA users. Also, even if the wireless Internet access becomes more and more popular (e.g. through 3G), an ongoing connection will be broken under the current Internet Protocol (IPv4) when the host is changing access (e.g. from LAN to Wireless). The program needs to implement fault tolerance mechanisms in order to avoid service interruption. This problem cannot be solved unless Mobile IP is adopted. The mobility issue is discussed in my previous work [41].
4. People may concern about their privacy. Not only they worry about their private data sent over the network for computation, they may also worry about their usage patterns such as time and frequency being tracked by the server.

Because of these restrictions, current implementations of code partitioning over the network are mostly restricted to remote authentication. However, crackers can easily disable this remote authentication by patching.

10.1.3 Relegating to a co-processor

The previous remote partitioning scheme is controversial. Another possible solution is local partitioning scheme. Under this scheme, the protected code is stored on a trusted local device, executing on a trusted co-processor. It is similar to hardware "dongles" but with the added instruction execution capability on a separate co-processor.

This can be made possible with the use of smart-card technology because some smart cards are come with programmable co-processors. More importantly, they are tamper-resistant. Sensors are contained to destroy the chip or the memory content in case an intrusion is detected.

The use of smart cards greatly increases the hardware asset costs required by the cracker for a successful break-in. This would be in most cases much greater than the advantage gained from it. Of course with the advance in technology, the smart-card hardware needs to be updated for some years.

This scheme will not be widely used until smart card reader becomes a common computer accessory. It will cause problems if more than one program requires smart card access. So this protection may only be used to protect very important software, such as OS.

10.2 Watermarking

Acting as an anti-piracy technique, a watermark may be added into the software before distributing to individual customers. This watermark is used to identify individual customers and therefore, make it possible to trace from the pirated copy back to the

original customer.

According to [02], software watermarking should be stealthy and resilient. It should be stealth so that it is difficult for the intruder to find it. It should be resilient so that the program will be damaged if the watermark is removed.

There are two kinds of watermarking techniques: static and dynamic [02]. Static watermarks are stored in the application executable itself, while the dynamic one is generated at runtime as dynamic states of the program.

Watermarking as anti-piracy measures, however, may not work in some countries where piracy is rampant and the intellectual property law is not mature or loose. Pirates can always find some 'innocents' (e.g. children) as the one to buy the first copy of the software and then mass-produce it.

10.3 Secure Software Engineering

Companies tend to think their programs can be better protected through commercial protection packages. In fact, this is not true. The case studies of ReleaseNow and Preview Systems clearly show the problem.

Because of the "publicity" of these commercial protection packages, they are also the aim of crackers. Because many programs are protected by these packages, the advantages yielded from defeating them are tremendous. Therefore, they are famous target for crackers. By cracking these packages, all the products protected by them are immediately threatened.

The solution is to do secure software engineering. Instead of adding security at the final stage of the product development, security should always be in mind in **every stage** of the product's lifecycle. In this way, security engineering will be fully integrated into software engineering.

This can be done by establishing security policy in the software design process, emphasizing security as the 'non-functional requirement' of the program, together with performance, reliance, resiliency, etc.

10.4 Adversary Economics

It is unintuitive to protect a low-priced program with very secure but expensive protection techniques or an expensive program is protected by lame schemes. Since at today's state-of-art technology, there is no "uncrackable" scheme, the point is how to choose a cost-effective protection scheme that matches the target to be protected. Ideally, a good scheme should protect your program without being cracked BEFORE your program stop making money.

Devanbu and Stubblebine [03] have suggested an economic model that relates the cost of buying the program (C_b), the first hack of the protection mechanism (C_h), making n copies (each C_c), the risk of being caught (P_{11}) and the cost after caught (C_{11}). For an effective software protection:

$$C_h + n^* C_c + P_{11}(n) * C_{11}(n) >> n^* C_b$$

Currently, the parameters here are mostly subjective. Research into how to calculate the C_h of the different protection schemes should be of paramount importance. With this, we can compare the effectiveness of schemes and apply them suitably on the target.

11. Conclusions

In this report, I have discussed various software protection techniques and their vulnerabilities. I tried to look at the problem from the worldviews of crackers, software industry, and researchers. I first started with the simple threat model, the cracking tools and then protection schemes.

Many programs apply the simple protection model, with its guard module unprotected. It is shown that 1-byte modification can be sufficient to crack these programs. To better secure our programs, encryption, packing and obfuscation should also be used.

However, obfuscation cannot make our programs uncrackable. For encryption and packing, no matter how strong they are, crackers can get around it by unpacking at the time when the program is decrypted.

The final way to overcome crackers can be code partitioning, in particular through the use of local co-processor such as smart cards. There are still a lot of issues that need to be addressed, and currently it is not widely accepted by end-users.

The mean-time solution, therefore, remains through the use of heavy obfuscation, good encryption, and backup by anti-debugging routines. Although they are not very foolproof, a combination of them already highly raises the difficulty to the crackers.

Beside technical means, we should also adopt secure software engineering practices, treating security as the non-functional requirement throughout the life cycle of the product. The study of adversary economics gives the measures to choose the most cost-effective schemes to protect our systems. Making quality software pricing at the right range with satisfactory customer support continues to be the basic formula to combat piracy.

Software security benefits nothing if it is just only the topic in academic papers. In order to avoid becoming ‘too academic’ and fills the void between research and practicality, three commercial applications in the market are selected to investigate in depth. This resulted in four different case studies. From the case studies, I have analyzed the principles behind the attacks, the investigating psychology, how the exploits are constructed, and what can be done to prevent the problems.

The three programs that have been selected are TextPad, SmartSaver Pro and Dreamweaver. They represent different market segments in the industry: TextPad (US $27:cheap), SmartSaver Pro (US $59.95:medium) and Dreamweaver (US $299:expensive).

The case study results ring the alarms in the software industry. It is surprising to see that our daily-used commercial software that is protected by commercial security packages is **too easily to be defeated**. The protection to Dreamweaver is given by ReleaseNow’s SalesAgent and to SmartSaver Pro is given by Preview System’s Vbox.

Comparing the protection provided by SalesAgent and Vbox, it can be concluded that Vbox provides better protection over SalesAgent as it provides encryption to avoid reverse engineering. SalesAgent just modifies 4096 bytes of the Dreamweaver executable file, leaving most content intact. Cracking Vbox requires more sophisticated OS knowledge, including loading process, PE execution format, etc. SalesAgent can be easily defeated by spotting it is a loader for the dreamweaver.tty process. It is immediately overcome if someone can extract the 4096 bytes of patch codes and the injection offset.

The result also shows that because of the lack of use of adversary economics, protection schemes mismatch with the protected programs. It is ridiculous that a much more expensive (US $299) software is protected by a weaker scheme than the much cheaper one (US $ 59.95).

May be all those practical security implementation weaknesses are rooted from the fact that the way of cracking, the tricks and traps are in long-term being underground stuffs. Therefore, proper programmers receiving proper trainings writing proper programs cannot be aware of these improper attacks. This report tries to be an **awareness paper** to the software industry and universities.

Finally, the protection schemes are presented in such a way that follows historical evolution, from old-days manual lookups to today’s widely adopted techniques such as packing. The future of protections is also conjectured and investigated. Throughout the evolution, we can see how technological developments and innovations contributed and will continue to contribute to protection methods and attackers.

Last but not least, the combat between attackers and defenders will never ends. Cracking, although at most of the time being undervalued by others, will nevertheless continue to exist and leads to better-protected software.

References

[01] R.Bjones, S.Hoeben. Vulnerabilities in pure software security systems, Utimaco Software AG, 2000

[02] C.S.Collberg, C.Thomborson. Watermarking, Tamper-Proofing, and Obfuscation

[03] P.T.Devanbu, S.Stubblebine. Software Engineering for Security: a Roadmap, ICSE 2000

[04] R.Mester. All About Copy Protection

[05] Compuware. **http://www.compuware.com/products/driverstudio/ds/softice.htm**

[06] Sysinternal Filemon. **http://www.sysinternals.com/ntw2k/source/filemon.shtml**

[07] Sysinternals Regmon. **http://www.sysinternals.com/ntw2k/source/regmon.shtml**

[08] Crashtest's tutorial #1, 2nd version, 1998

[09] Bullet.Very Easy Cracking Tutorial, 1999

[10] Basdog22. The Ultimate Begginer Cracker's Book v1.0 – v1.5

[11] B.Brey. 8086/8088, 80286, 80386 and 80486 Assembly Language Programming, Merril, 1994

[12] Intel. IA-32 Intel Architecture Software Developer's Manual Volume 1-3, 2001

[13] A.K.M. Lo. Buffer Overflow Attack – Design and Implementation for Microsoft Windows Media Player, Thesis Report, The University of Hong Kong, 2001

[14] Compuware. SoftICE Command Reference Release 2.5, 2001

[15] Microsoft Win32 Programmer's Reference, 1996

[16] Iczelion's Win32 Assembly Tutorial (Set 1-22,24)

[17] Iczelion's PE Tutorial (Set 1-7)

[18] Iceman. Tweaking with memory in Window95 – An API approach

[19] Stone. In memory patching: three approaches (how to introduce breakpoints in an automated debugger and other marvels), 1997

[20] ShADë. Patching in a Patcher, 2000

[21] UPX (The Ultimate Packer for eXecutables) Software Manual

[22] Luevelsmeyer. The PE File Format v1.9, 1999

[23] Microsoft Portable Executable and Common Object File Format Specification Revision 6.0, Microsoft Corporation, 1999

[24] Commercial Protection Systems: SalesAgent, cRACKER's nOTES

[25] Freddy K. Dreamweaver 3 Trial/Rsagent v3.12, 2000

[26] Pincopall. SalesAgent defeating, 2002

[27] Capac. How to completely remove a SalesAgent protection. Bye Bye SalesAgent, 2000

[28] EtErNaL_L0ser. Sales Agent Generic Cracking, 2001

[29] Christal. An addition on a "Ready Made Protection": Sales Agent

[30] Viktor Toth. Visual C++ 4 Unleashed, Chapter 16 – The Registry

[31] Icezlion's Win32 Assembly (Set 28-30: Win32 Debug API)

[32] Tsehp, Manually unpacking Asprotect version 1.0 – The encrypted import table, 2000

[33] Tsehp, Manually unpacking Asprotect version 1.05 – Building a fake import table, 2000

[34] Predator, Unpacking: a generic approach, including IT rebuilding, 2001

[35] Sandman, Manual Unpacking Project, 1999

[36] BlackB, Unpacking asprotected programs – PicView v1.32, 2000

[37] BlackB, Cracking Iris v2.0, 2001

[38] Tsehp, Revirgin 1.2 readme, 2001

[39] P.Dabak; M.Borate; S.Phadke, Undocumented Windows NT, M&T Books, 1999

[40] M. Pietrek, Windows 95 System Programming Secrets, IDG, 1995

[41] A.K.M.Lo. Future Mobile Internet – Mobile IP support in Third Generation Mobile Systems, Technical Report, The University of Birmingham, 2002

[42] I.Raz. Anti Debugging Tricks Release Number 5

[43] D.Vekhter, J.Peng. Software Piracy. http://cse.stanford.edu/class/cs201/projects-99-00/software-piracy/mainframe.html

Appendix A – Selected Win32 API

WaitForDebugEvent

The **WaitForDebugEvent** function waits for a debugging event to occur in a process being debugged.

```
BOOL WaitForDebugEvent(
  LPDEBUG_EVENT lpDebugEvent,    // debug event information
  DWORD dwMilliseconds           // time-out value
);
```

Parameters

lpDebugEvent

[out] Pointer to a DEBUG_EVENT structure that receives information about the debugging event.

dwMilliseconds

[in] Specifies the number of milliseconds to wait for a debugging event. If this parameter is zero, the function tests for a debugging event and returns immediately. If the parameter is INFINITE, the function does not return until a debugging event has occurred.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

Only the thread that created the process being debugged can call **WaitForDebugEvent**.

When a CREATE_PROCESS_DEBUG_EVENT occurs, the debugger application receives a handle to the image file of the process being debugged, a handle to the process being debugged, and a handle to the initial thread of the process being debugged in the DEBUG_EVENT structure. The DEBUG_EVENT members these handles are returned in are **u.CreateProcessInfo.hFile**, **u.CreateProcessInfo.hProcess**, and **u.CreateProcessInfo.hThread** respectively. The system will close these handles. The debugger should not close these handles.

Similarly, when a CREATE_THREAD_DEBUG_EVENT occurs, the debugger application receives a handle to the thread whose creation caused the debugging event in the **u.CreateThread.hThread** member of the **DEBUG_EVENT** structure. This handle should also not be closed by the debugger application, as it will be closed by the system.

Also, when a LOAD_DLL_DEBUG_EVENT occurs, the debugger application receives a handle to the loaded DLL in the **u.LoadDll.hFile** member of the **DEBUG_EVENT** structure. This handle should be closed by the debugger application by calling the CloseHandle function when the corresponding UNLOAD_DLL_DEBUG_EVENT occurs.

ContinueDebugEvent

The **ContinueDebugEvent** function enables a debugger to continue a thread that previously reported a debugging event.

BOOL ContinueDebugEvent(
  DWORD *dwProcessId*, // process to continue
  DWORD *dwThreadId*, // thread to continue
  DWORD *dwContinueStatus* // continuation status
);

Parameters

dwProcessId

[in] Handle to the process to continue.

dwThreadId

[in] Handle to the thread to continue. The combination of process identifier and thread identifier must identify a thread that has previously reported a debugging event.

dwContinueStatus

[in] Specifies how to continue the thread that reported the debugging event.

If the DBG_CONTINUE flag is specified for this parameter and the thread specified by the *dwThreadId* parameter previously reported an EXCEPTION_DEBUG_EVENT debugging event, the function stops all exception processing and continues the thread. For any other debugging event, this flag simply continues the thread.

If the DBG_EXCEPTION_NOT_HANDLED flag is specified for this parameter and the thread specified by *dwThreadId* previously reported an EXCEPTION_DEBUG_EVENT debugging event, the function continues exception processing. If this is a first-chance exception event, the search and dispatch logic of the structured exception handler is used; otherwise, the process is terminated. For any other debugging event, this flag simply continues the thread.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

Only the thread that created *dwProcessId* with the CreateProcess function can call **ContinueDebugEvent**.

After the **ContinueDebugEvent** function succeeds, the specified thread continues. Depending on the debugging event previously reported by the thread, different actions occur. If the continued thread previously reported an EXIT_THREAD_DEBUG_EVENT debugging event, **ContinueDebugEvent** closes the handle the debugger has to the thread. If the continued thread previously reported an EXIT_PROCESS_DEBUG_EVENT debugging event, **ContinueDebugEvent** closes the handles the

debugger has to the process and to the thread.

DEBUG_EVENT

The **DEBUG_EVENT** structure describes a debugging event.

```
typedef struct _DEBUG_EVENT {
  DWORD dwDebugEventCode;
  DWORD dwProcessId;
  DWORD dwThreadId;
  union {
      EXCEPTION_DEBUG_INFO Exception;
      CREATE_THREAD_DEBUG_INFO CreateThread;
      CREATE_PROCESS_DEBUG_INFO CreateProcessInfo;
      EXIT_THREAD_DEBUG_INFO ExitThread;
      EXIT_PROCESS_DEBUG_INFO ExitProcess;
      LOAD_DLL_DEBUG_INFO LoadDll;
      UNLOAD_DLL_DEBUG_INFO UnloadDll;
      OUTPUT_DEBUG_STRING_INFO DebugString;
      RIP_INFO RipInfo;
  } u;
} DEBUG_EVENT, *LPDEBUG_EVENT;
```

Members

dwDebugEventCode

Specifies a debugging event code that identifies the type of debugging event. This parameter can be one of the following values.

Value	Meaning
EXCEPTION_DEBUG_EVENT	Reports an exception debugging event. The value of **u.Exception** specifies an EXCEPTION_DEBUG_INFO structure.
CREATE_THREAD_DEBUG_EVENT	Reports a create-thread debugging event. The value of **u.CreateThread** specifies a CREATE_THREAD_DEBUG_INFO structure.
CREATE_PROCESS_DEBUG_EVENT	Reports a create-process debugging event. The value of **u.CreateProcessInfo** specifies a

	CREATE_PROCESS_DEBUG_INFO structure.
EXIT_THREAD_DEBUG_EVENT	Reports an exit-thread debugging event. The value of **u.ExitThread** specifies an EXIT_THREAD_DEBUG_INFO structure.
EXIT_PROCESS_DEBUG_EVENT	Reports an exit-process debugging event. The value of **u.ExitProcess** specifies an EXIT_PROCESS_DEBUG_INFO structure.
LOAD_DLL_DEBUG_EVENT	Reports a load-dynamic-link-library (DLL) debugging event. The value of **u.LoadDll** specifies a LOAD_DLL_DEBUG_INFO structure.
UNLOAD_DLL_DEBUG_EVENT	Reports an unload-DLL debugging event. The value of **u.UnloadDll** specifies an UNLOAD_DLL_DEBUG_INFO structure.
OUTPUT_DEBUG_STRING_EVENT	Reports an output-debugging-string debugging event. The value of **u.DebugString** specifies an OUTPUT_DEBUG_STRING_INFO structure.
RIP_EVENT	Reports a RIP-debugging event (system debugging error). The value of **u.RipInfo** specifies a RIP_INFO structure.

dwProcessId

Specifies the identifier of the process in which the debugging event occurred. A debugger uses this value to locate the debugger's per-process structure. These values are not necessarily small integers that can be used as table indices.

dwThreadId

Specifies the identifier of the thread in which the debugging event occurred. A debugger uses this value to locate the debugger's per-thread structure. These values are not necessarily small integers that can be used as table indices.

u

Specifies additional information relating to the debugging event. This union takes on the type and value appropriate to the type of debugging event, as described in the **dwDebugEventCode** member.

Remarks

If the WaitForDebugEvent function succeeds, it fills in the members of a **DEBUG_EVENT** structure.

CREATE_PROCESS_DEBUG_INFO

The **CREATE_PROCESS_DEBUG_INFO** structure contains process creation information that can be used by a debugger.

```
typedef struct _CREATE_PROCESS_DEBUG_INFO {
  HANDLE hFile;
  HANDLE hProcess;
  HANDLE hThread;
  LPVOID lpBaseOfImage;
  DWORD dwDebugInfoFileOffset;
  DWORD nDebugInfoSize;
  LPVOID lpThreadLocalBase;
  LPTHREAD_START_ROUTINE lpStartAddress;
  LPVOID lpImageName;
  WORD fUnicode;
} CREATE_PROCESS_DEBUG_INFO, *LPCREATE_PROCESS_DEBUG_INFO;
```

Members

hFile

Handle to the process's image file. If this member is NULL, the handle is not valid. Otherwise, the debugger can use the member to read from and write to the image file.

hProcess

Handle to the process. If this member is NULL, the handle is not valid. Otherwise, the debugger can use the member to read from and write to the process's memory.

hThread

Handle to the initial thread of the process identified by the **hProcess** member. If **hThread** is NULL, the handle is not valid. Otherwise, the debugger has THREAD_GET_CONTEXT, THREAD_SET_CONTEXT, and THREAD_SUSPEND_RESUME access to the thread, allowing the debugger to read from and write to the registers of the thread and to control execution of the thread.

lpBaseOfImage

Pointer to the base address of the executable image that the process is running.

dwDebugInfoFileOffset

Specifies the offset to the debugging information in the file identified by the **hFile** member. The system expects the debugging information to be in Microsoft® CodeView® version 4.0 format. This format is currently a derivative of COFF (Common Object File Format).

nDebugInfoSize

Specifies the size, in bytes, of the debugging information in the file. If this value is zero, there is no

debugging information.

lpThreadLocalBase

Pointer to a block of data. At offset 0x2C into this block is another pointer, called ThreadLocalStoragePointer, that points to an array of per-module thread local storage blocks. This gives a debugger access to per-thread data in the threads of the process being debugged using the same algorithms that a compiler would use.

lpStartAddress

Pointer to the starting address of the thread. This value may only be an approximation of the thread's starting address, because any application with appropriate access to the thread can change the thread's context by using the SetThreadContext function.

lpImageName

Pointer to the filename associated with the *hFile* parameter. This parameter may be NULL, or it may contain the address of a string pointer in the address space of the process being debugged. That address may, in turn, either be NULL or point to the actual filename. If **fUnicode** is a nonzero value, the name string is Unicode; otherwise, it is ANSI.

This member is strictly optional. Debuggers must be prepared to handle the case where **lpImageName** is NULL or ***lpImageName** (in the address space of the process being debugged) is NULL. Specifically, the system does not provide an image name for a create process event, and will not likely pass an image name for the first DLL event. The system also does not provide this information in the case of debug events that originate from a call to the **DebugActiveProcess** function.

fUnicode

Indicates whether a file name specified by the **lpImageName** member is Unicode or ANSI. A nonzero value indicates Unicode; zero indicates ANSI.

EXCEPTION_DEBUG_INFO

The **EXCEPTION_DEBUG_INFO** structure contains exception information that can be used by a debugger.

```
typedef struct _EXCEPTION_DEBUG_INFO {
  EXCEPTION_RECORD ExceptionRecord;
  DWORD dwFirstChance;
} EXCEPTION_DEBUG_INFO, *LPEXCEPTION_DEBUG_INFO;
```

Members

ExceptionRecord

Contains an EXCEPTION_RECORD structure with information specific to the exception. This includes the exception code, flags, address, a pointer to a related exception, extra parameters, and so on.

dwFirstChance

Indicates whether the debugger has previously encountered the exception specified by the **ExceptionRecord** member. If the **dwFirstChance** member is nonzero, this is the first time the debugger has encountered the exception. Debuggers typically handle breakpoint and single-step exceptions when they are first encountered. If this member is zero, the debugger has previously encountered the exception. This occurs only if, during the search for structured exception handlers, either no handler was found or the exception was continued.

EXCEPTION_RECORD

The **EXCEPTION_RECORD** structure describes an exception.

```
typedef struct _EXCEPTION_RECORD {
  DWORD ExceptionCode;
  DWORD ExceptionFlags;
  struct _EXCEPTION_RECORD *ExceptionRecord;
  PVOID ExceptionAddress;
  DWORD NumberParameters;
  ULONG_PTR ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
} EXCEPTION_RECORD, *PEXCEPTION_RECORD;
```

Members

ExceptionCode

Specifies the reason the exception occurred. This is the code generated by a hardware exception, or the code specified in the RaiseException function for a software-generated exception. The following tables describes the exception codes that are likely to occur due to common programming errors.

Value	Meaning
EXCEPTION_ACCESS_VIOLATION	The thread tried to read from or write to a virtual address for which it does not have the appropriate access.
EXCEPTION_ARRAY_BOUNDS_EXCEEDED	The thread tried to access an array element that is out of bounds and the underlying hardware supports bounds checking.
EXCEPTION_BREAKPOINT	A breakpoint was encountered.
EXCEPTION_DATATYPE_MISALIGNMENT	The thread tried to read or write data that is misaligned on hardware that does not

	provide alignment. For example, 16-bit values must be aligned on 2-byte boundaries; 32-bit values on 4-byte boundaries, and so on.
EXCEPTION_FLT_DENORMAL_OPERAND	One of the operands in a floating-point operation is denormal. A denormal value is one that is too small to represent as a standard floating-point value.
EXCEPTION_FLT_DIVIDE_BY_ZERO	The thread tried to divide a floating-point value by a floating-point divisor of zero.
EXCEPTION_FLT_INEXACT_RESULT	The result of a floating-point operation cannot be represented exactly as a decimal fraction.
EXCEPTION_FLT_INVALID_OPERATION	This exception represents any floating-point exception not included in this list.
EXCEPTION_FLT_OVERFLOW	The exponent of a floating-point operation is greater than the magnitude allowed by the corresponding type.
EXCEPTION_FLT_STACK_CHECK	The stack overflowed or underflowed as the result of a floating-point operation.
EXCEPTION_FLT_UNDERFLOW	The exponent of a floating-point operation is less than the magnitude allowed by the corresponding type.
EXCEPTION_ILLEGAL_INSTRUCTION	The thread tried to execute an invalid instruction.
EXCEPTION_IN_PAGE_ERROR	The thread tried to access a page that was not present, and the system was unable to load the page. For example, this exception might occur if a network connection is lost while running a program over the network.
EXCEPTION_INT_DIVIDE_BY_ZERO	The thread tried to divide an integer value by an integer divisor of zero.
EXCEPTION_INT_OVERFLOW	The result of an integer operation caused a carry out of the most significant bit of the result.
EXCEPTION_INVALID_DISPOSITION	An exception handler returned an invalid disposition to the exception dispatcher. Programmers using a high-level language such as C should never encounter this exception.
EXCEPTION_NONCONTINUABLE_EXCEPTION	The thread tried to continue execution after a noncontinuable exception occurred.
EXCEPTION_PRIV_INSTRUCTION	The thread tried to execute an instruction whose operation is not allowed in the current machine mode.
EXCEPTION_SINGLE_STEP	A trace trap or other single-instruction mechanism signaled that one instruction has been executed.
EXCEPTION_STACK_OVERFLOW	The thread used up its stack.

Another exception code is likely to occur when debugging console processes. It does not arise because of a programming error. The DBG_CONTROL_C exception code occurs when CTRL+C is input to a console process that handles CTRL+C signals and is being debugged. This exception code is not meant to be handled by applications. It is raised only for the benefit of the debugger, and is raised only when a debugger is attached to the console process.

ExceptionFlags

Specifies the exception flags. This member can be either zero, indicating a continuable exception, or EXCEPTION_NONCONTINUABLE indicating a noncontinuable exception. Any attempt to continue execution after a noncontinuable exception causes the EXCEPTION_NONCONTINUABLE_EXCEPTION exception.

ExceptionRecord

Pointer to an associated **EXCEPTION_RECORD** structure. Exception records can be chained together to provide additional information when nested exceptions occur.

ExceptionAddress

Specifies the address where the exception occurred.

NumberParameters

Specifies the number of parameters associated with the exception. This is the number of defined elements

in the **ExceptionInformation** array.

ExceptionInformation

Specifies an array of additional arguments that describe the exception. The RaiseException function can specify this array of arguments. For most exception codes, the array elements are undefined. The following table describes the exception codes whose array elements are defined.

Exception code	Array contents
EXCEPTION_ACCESS_VIOLATION	The first element of the array contains a read-write flag that indicates the type of operation that caused the access violation. If this value is zero, the thread attempted to read the inaccessible data. If this value is 1, the thread attempted to write to an inaccessible address. The second array element specifies the virtual address of the inaccessible data.

WriteProcessMemory

The **WriteProcessMemory** function writes data to an area of memory in a specified process. The entire area to be written to must be accessible, or the operation fails.

```
BOOL WriteProcessMemory(
  HANDLE hProcess,                  // handle to process
  LPVOID lpBaseAddress,             // base of memory area
  LPCVOID lpBuffer,                 // data buffer
  SIZE_T nSize,                     // count of bytes to write
  SIZE_T * lpNumberOfBytesWritten // count of bytes written
);
```

Parameters

hProcess

[in] Handle to the process whose memory is to be modified. The handle must have PROCESS_VM_WRITE and PROCESS_VM_OPERATION access to the process.

lpBaseAddress

[in] Pointer to the base address in the specified process to which data will be written. Before any data transfer occurs, the system verifies that all data in the base address and memory of the specified size is accessible for write access. If this is the case, the function proceeds; otherwise, the function fails.

lpBuffer

[in] Pointer to the buffer that contains data to be written into the address space of the specified process.

nSize

[in] Specifies the requested number of bytes to write into the specified process.

lpNumberOfBytesWritten

[out] Pointer to a variable that receives the number of bytes transferred into the specified process. This parameter is optional. If *lpNumberOfBytesWritten* is NULL, the parameter is ignored.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError. The function will fail if the requested write operation crosses into an area of the process that is inaccessible.

Remarks

WriteProcessMemory copies the data from the specified buffer in the current process to the address range of the specified process. Any process that has a handle with PROCESS_VM_WRITE and PROCESS_VM_OPERATION access to the process to be written to can call the function. The process whose address space is being written to is typically, but not necessarily, being debugged.

The entire area to be written to must be accessible. If it is not, the function fails as noted previously.

Appendix B – Partial Dreamweaver Disassembly

```
:00401A8B 8D8C2490000000        lea ecx, dword ptr [esp+00000090]
:00401A92 68E0AF4500            push 0045AFE0
:00401A97 51                    push ecx
:00401A98 53                    push ebx
:00401A99 53                    push ebx
:00401A9A 6A02                  push 00000002
:00401A9C 53                    push ebx
:00401A9D 53                    push ebx
:00401A9E 8D9424F0000000        lea edx, dword ptr [esp+000000F0]
:00401AA5 53                    push ebx
:00401AA6 52                    push edx
:00401AA7 53                    push ebx

* Reference To: KERNEL32.CreateProcessA, Ord:0044h
:00401AA8 FF15C0504300          Call dword ptr [004350C0]
:00401AAE 85C0                  test eax, eax
:00401AB0 751F                  jne 00401AD1
:00401AB2 53                    push ebx

* Possible StringData Ref from Data Obj ->"Error"
:00401AB3 68CC914300            push 004391CC

* Possible StringData Ref from Data Obj ->"Error loading process"
:00401AB8 68B4914300            push 004391B4
:00401ABD 53                    push ebx

* Reference To: USER32.MessageBoxA, Ord:01BEh
:00401ABE FF1530534300          Call dword ptr [00435330]
:00401AC4 33C0                  xor eax, eax
:00401AC6 5F                    pop edi
:00401AC7 5E                    pop esi
:00401AC8 5D                    pop ebp
:00401AC9 5B                    pop ebx
:00401ACA 81C49C260000          add esp, 0000269C
```



```
:00401AD0 C3                    ret

* Referenced by a (U)nconditional or (C)onditional Jump at Addresses:
|:00401AB0(C), :00401AE5(C), :00401CBB(U)
:00401AD1 8D442430              lea eax, dword ptr [esp+30]
:00401AD5 6AFF                  push FFFFFFFF
:00401AD7 50                    push eax
:00401AD8 BD02000100            mov ebp, 00010002

* Reference To: KERNEL32.WaitForDebugEvent, Ord:02CBh
:00401ADD FF1540514300          Call dword ptr [00435140]
:00401AE3 85C0                  test eax, eax
:00401AE5 74EA                  je 00401AD1
:00401AE7 8B542434              mov edx, dword ptr [esp+34]
:00401AEB A1E8AF4500            mov eax, dword ptr [0045AFE8]
:00401AF0 3BD0                  cmp edx, eax
:00401AF2 0F85B2010000          jne 00401CAA
:00401AF8 8B4C2430              mov ecx, dword ptr [esp+30]
:00401AFC 8D41FF                lea eax, dword ptr [ecx-01]
:00401AFF 83F807                cmp eax, 00000007
:00401B02 0F87A2010000          ja 00401CAA
:00401B08 FF2485E41C4000        jmp dword ptr [4*eax+00401CE4]
:00401B0F 8B44243C              mov eax, dword ptr [esp+3C]
:00401B13 3D03000080            cmp eax, 80000003
:00401B18 0F85C7000000          jne 00401BE5
:00401B1E 395C2420              cmp dword ptr [esp+20], ebx
:00401B22 7470                  je 00401B94
:00401B24 895C2420              mov dword ptr [esp+20], ebx
:00401B28 E8E3F4FFFF            call 00401010
:00401B2D 89442414              mov dword ptr [esp+14], eax
:00401B31 A15CE94400            mov eax, dword ptr [0044E95C]
:00401B36 3BC3                  cmp eax, ebx
:00401B38 0F849C000000          je 00401BDA
:00401B3E A1E0AF4500            mov eax, dword ptr [0045AFE0]
:00401B43 8D542410              lea edx, dword ptr [esp+10]
:00401B47 52                    push edx
```

```
:00401B48 53                    push ebx
:00401B49 50                    push eax
:00401B4A 68101D4000            push 00401D10
:00401B4F 53                    push ebx
:00401B50 53                    push ebx
:00401B51 895C2428              mov dword ptr [esp+28], ebx

* Reference To: KERNEL32.CreateThread, Ord:004Ah
:00401B55 FF1544514300          Call dword ptr [00435144]
:00401B5B 50                    push eax

* Reference To: KERNEL32.CloseHandle, Ord:001Bh
:00401B5C FF1530514300          Call dword ptr [00435130]
:00401B62 8D4C2424              lea ecx, dword ptr [esp+24]
:00401B66 8D542414              lea edx, dword ptr [esp+14]
:00401B6A 51                    push ecx
:00401B6B 8B4C2438              mov ecx, dword ptr [esp+38]
:00401B6F 8D8424E4030000        lea eax, dword ptr [esp+000003E4]
:00401B76 52                    push edx
:00401B77 8B542420              mov edx, dword ptr [esp+20]
:00401B7B 50                    push eax
:00401B7C 51                    push ecx
:00401B7D 52                    push edx
:00401B7E C744243001000000      mov [esp+30], 00000001
:00401B86 E895F9FFFF            call 00401520
:00401B8B 83C414                add esp, 00000014
:00401B8E 89442410              mov dword ptr [esp+10], eax
:00401B92 EB46                  jmp 00401BDA

* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:00401B22(C)
:00401B94 395C241C              cmp dword ptr [esp+1C], ebx
:00401B98 7469                  je 00401C03
:00401B9A 8B442424              mov eax, dword ptr [esp+24]
:00401B9E 8B4C2410              mov ecx, dword ptr [esp+10]
:00401BA2 50                    push eax
```



```
:00401BA3 8B442418              mov eax, dword ptr [esp+18]
:00401BA7 51                    push ecx
:00401BA8 50                    push eax
:00401BA9 81ECCC020000          sub esp, 000002CC
:00401BAF B9B3000000            mov ecx, 000000B3
:00401BB4 8DB424B8060000        lea esi, dword ptr [esp+000006B8]
:00401BBB 8BFC                  mov edi, esp
:00401BBD F3                    repz
:00401BBE A5                    movsd
:00401BBF 8B8C24F0020000        mov ecx, dword ptr [esp+000002F0]
:00401BC6 52                    push edx
:00401BC7 51                    push ecx
:00401BC8 899C24FC020000        mov dword ptr [esp+000002FC], ebx
:00401BCF E8BCF8FFFF            call 00401490
:00401BD4 81C4E0020000          add esp, 000002E0

* Referenced by a (U)nconditional or (C)onditional Jump at Addresses:
|:00401B38(C), :00401B92(U)
:00401BDA 8B44243C              mov eax, dword ptr [esp+3C]
:00401BDE 3D03000080            cmp eax, 80000003
:00401BE3 741E                  je 00401C03

* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:00401B18(C)
:00401BE5 3D04000080            cmp eax, 80000004
:00401BEA 740C                  je 00401BF8
:00401BEC 3D080000C0            cmp eax, C0000008
:00401BF1 7405                  je 00401BF8
:00401BF3 BD01000180            mov ebp, 80010001

* Referenced by a (U)nconditional or (C)onditional Jump at Addresses:
|:00401BEA(C), :00401BF1(C)
:00401BF8 3D03000080            cmp eax, 80000003
:00401BFD 0F85A7000000          jne 00401CAA

* Referenced by a (U)nconditional or (C)onditional Jump at Addresses:
```

```
|:00401B98(C), :00401BE3(C)
:00401C03 399C248C000000        cmp dword ptr [esp+0000008C], ebx
:00401C0A 0F859A000000          jne 00401CAA
:00401C10 8B442448              mov eax, dword ptr [esp+48]
:00401C14 8B4C2428              mov ecx, dword ptr [esp+28]
:00401C18 3BC8                  cmp ecx, eax
:00401C1A 752F                  jne 00401C4B
:00401C1C 50                    push eax
:00401C1D 8D9424B0060000        lea edx, dword ptr [esp+000006B0]

* Possible StringData Ref from Data Obj ->"Application error occurred at "
                                       ->"address 0x%x, from which it is "
                                       ->"unable to recover."
:00401C24 6864914300            push 00439164
:00401C29 52                    push edx

* Reference To: USER32.wsprintfA, Ord:02ACh
:00401C2A FF1538534300          Call dword ptr [00435338]
:00401C30 83C40C                add esp, 0000000C
:00401C33 8D8424AC060000        lea eax, dword ptr [esp+000006AC]
:00401C3A 6A10                  push 00000010

* Possible StringData Ref from Data Obj ->"Application Terminating."
:00401C3C 6848914300            push 00439148
:00401C41 50                    push eax
:00401C42 53                    push ebx

* Reference To: USER32.MessageBoxA, Ord:01BEh
:00401C43 FF1530534300          Call dword ptr [00435330]
:00401C49 EB5F                  jmp 00401CAA

* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:00401C1A(C)
:00401C4B 89442428              mov dword ptr [esp+28], eax
:00401C4F BD02000100            mov ebp, 00010002
:00401C54 EB54                  jmp 00401CAA
```



```
:00401C56 B90A000000            mov ecx, 0000000A
:00401C5B 8D74243C              lea esi, dword ptr [esp+3C]
:00401C5F BF40BD4300            mov edi, 0043BD40
:00401C64 F3                    repz
:00401C65 A5                    movsd
:00401C66 8B4C2444              mov ecx, dword ptr [esp+44]
:00401C6A 894C2418              mov dword ptr [esp+18], ecx
:00401C6E EB3A                  jmp 00401CAA
:00401C70 8B442442              mov eax, dword ptr [esp+42]
:00401C74 8D54242C              lea edx, dword ptr [esp+2C]
:00401C78 25FFFF0000            and eax, 0000FFFF
:00401C7D 52                    push edx
:00401C7E 8B542440              mov edx, dword ptr [esp+40]
:00401C82 8D8C24B0160000        lea ecx, dword ptr [esp+000016B0]
:00401C89 50                    push eax
:00401C8A A1E0AF4500            mov eax, dword ptr [0045AFE0]
:00401C8F 51                    push ecx
:00401C90 52                    push edx
:00401C91 50                    push eax

* Reference To: KERNEL32.ReadProcessMemory, Ord:021Ch
:00401C92 FF1534514300          Call dword ptr [00435134]
:00401C98 85C0                  test eax, eax
:00401C9A 740E                  je 00401CAA
:00401C9C 8D8C24AC160000        lea ecx, dword ptr [esp+000016AC]
:00401CA3 51                    push ecx

* Reference To: KERNEL32.OutputDebugStringA, Ord:01F5h
:00401CA4 FF1548514300          Call dword ptr [00435148]

* Referenced by a (U)nconditional or (C)onditional Jump at Addresses:
|:00401AF2(C), :00401B02(C), :00401BFD(C), :00401C0A(C), :00401C49(U)
|:00401C54(U), :00401C6E(U), :00401C9A(C)
:00401CAA 8B542438              mov edx, dword ptr [esp+38]
:00401CAE 8B442434              mov eax, dword ptr [esp+34]
:00401CB2 55                    push ebp
```

```
:00401CB3 52                    push edx
:00401CB4 50                    push eax

* Reference To: KERNEL32.ContinueDebugEvent, Ord:0025h
:00401CB5 FF154C514300          Call dword ptr [0043514C]
:00401CBB E911FEFFFF            jmp 00401AD1
:00401CC0 8B0D64F04300          mov ecx, dword ptr [0043F064]
:00401CC6 53                    push ebx
:00401CC7 53                    push ebx
:00401CC8 6A10                  push 00000010
:00401CCA 51                    push ecx

* Reference To: USER32.PostMessageA, Ord:01DEh
:00401CCB FF1580524300          Call dword ptr [00435280]
:00401CD1 5F                    pop edi
:00401CD2 5E                    pop esi
:00401CD3 5D                    pop ebp
:00401CD4 B801000000            mov eax, 00000001
:00401CD9 5B                    pop ebx
:00401CDA 81C49C260000          add esp, 0000269C
:00401CE0 C3                    ret

:00401CE1 8D4900                lea ecx, dword ptr [ecx+00]

:00401CE4 0F1B4000              DWORD 00401B0F
:00401CE8 AA1C4000              DWORD 00401CAA
:00401CEC 561C4000              DWORD 00401C56
:00401CF0 AA1C4000              DWORD 00401CAA
:00401CF4 C01C4000              DWORD 00401CC0
:00401CF8 AA1C4000              DWORD 00401CAA
:00401CFC AA1C4000              DWORD 00401CAA
:00401D00 701C4000              DWORD 00401C70
:00401D04 90                    nop
:00401D05 90                    nop
:00401D06 90                    nop
:00401D07 90                    nop
```


Appendix C – USSPRO.EXE Import Details

```
++++++++++++++++++ IMPORT MODULE DETAILS +++++++++++++++

   Import Module 001: vboxp410.dll
 Addr:80000001 hint(0001) Name:
   Import Module 002: vboxb410.dll
 Addr:80000001 hint(0001) Name:
   Import Module 003: UssAbout.dll
 Addr:0013C360 hint(0004) Name: IsFullVersion
   Import Module 004: u32Base.dll
 Addr:0013C37C hint(0020) Name: buf32MergeBuf
   Import Module 005: u32Comm.dll
 Addr:0013C398 hint(002C) Name: ufdSplitPathname
   Import Module 006: u32File.dll
 Addr:0013C3B8 hint(0018) Name: ufFileGetFileData
   Import Module 007: USSGifsa.dll
 Addr:0013C3DC hint(0000) Name: AniGifAction
   Import Module 008: ussjpgen.dll
 Addr:0013C3FC hint(0005) Name: _JpegSave@12
   Import Module 009: MPR.dll
 Addr:0013C414 hint(000A) Name: WNetAddConnectionA
   Import Module 010: WINMM.dll
 Addr:0013C438 hint(007C) Name: mmioCreateChunk
   Import Module 011: UssCvt.dll
 Addr:0013C458 hint(000C) Name: cvt32DIBToBuf
   Import Module 012: UssUtil.dll
 Addr:0013C474 hint(0000) Name: UssContextMenuState
   Import Module 013: u32Sel.dll
 Addr:0013C498 hint(0011) Name: sel32MagicWandMakeMask
   Import Module 014: Pngfio.dll
 Addr:0013C4C0 hint(0003) Name: Png_Write
   Import Module 015: u32sn.dll
 Addr:0013C4D8 hint(0001) Name: snGetPushURL
   Import Module 016: u32Cfg.dll
 Addr:80000002 hint(0002) Name: snGetPushURL
```

```
    Import Module 017: MFC42.DLL
  Addr:80001491 hint(1491) Name: snGetPushURL
    Import Module 018: MSVCRT.dll
  Addr:0013C50C hint(0298) Name: memmove
    Import Module 019: KERNEL32.dll
  Addr:0013C528 hint(011C) Name: GetLocaleInfoA
    Import Module 020: USER32.dll
  Addr:0013C548 hint(00E4) Name: GetCapture
    Import Module 021: GDI32.dll
  Addr:0013C564 hint(00A8) Name: FillRgn
    Import Module 022: ADVAPI32.dll
  Addr:0013C580 hint(0162) Name: RegDeleteKeyA
    Import Module 023: SHELL32.dll
  Addr:0013C59C hint(0050) Name: SHGetPathFromIDListA
    Import Module 024: COMCTL32.dll
  Addr:0013C5C4 hint(001E) Name: ImageList_AddMasked
```