



2009

ARTeam eZine Issue IV



Editor: Shub-Nigurrath



ARTeam

3/25/2009

ARTEAM

E-ZINE 4

PRACTICAL RCE : REVERSE ENGINEERING MAGAZINE

Volume 1, Issue 4

REVERSING BINARY 500

BY EXTERNALIST

SEVERAL
VIDEO TUTORIALS
INCLUDED AS
SUPPLEMENTS

HANDY PRIMER ON LINUX REVERSING

BY GUNTHER

USING .NET PROFILING API FOR A CUSTOM .NET PROTECTION

BY KURAPICA

PRIMER ON REVERSING PALMOS APPLICATIONS EXTENDED EDITION

BY WAST3D_BYTES, SUNTZU

REVERSING THE PROTECTION'S SCHEME OF ALEXEY PAJITNOV'S GAME TWICE

BY GYVER75

LIVE DEBUGGING SYMBIAN APPLICATIONS USING OR NOT USING IDA

BY ARGV

Special Issue on Non-Windows Reversing

PLUS

**Interview With
Shub-Nigurrath**

\x52\x43\x45

®
RCE

CONTENT RATED BY
ARTEAM

**TABLE OF CONTENTS**

FOREWORDS	6
Disclaimer/License	8
Supplements	8
Verification	8
1 REVERSING BINARY 500 BY EXTERNALIST	9
1.1 Introduction	9
1.2 Tools needed	9
1.3 Exploring the Binary	9
1.4 Reversing the Binary	16
1.5 Conclusions	47
1.6 References	48
1.7 Greetings	48
2 HANDY PRIMER ON LINUX REVERSING BY GUNTHER	49
2.1 Forewords	49
2.2 Abstract	49
2.3 Target	50
2.4 Examining our target	50
2.5 Searching for more clue	56
2.6 Analysing the contents of specific parts of the file	56
2.7 Retrieving Symbol table	57
2.8 Reversing the program	58
2.9 Reverse Engineering	58
2.10 Conclusions	65

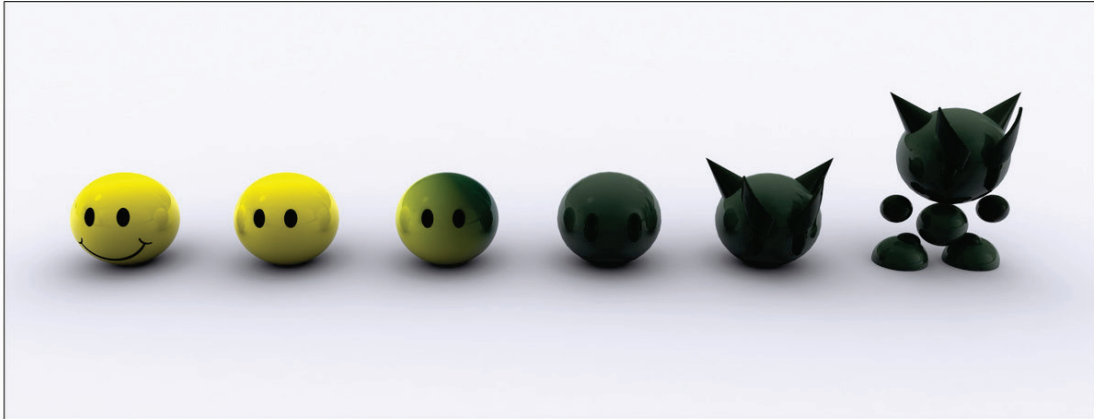


2.11	Greetings	65
3	USING .NET PROFILING API FOR A CUSTOM .NET PROTECTION BY KURAPICA	66
3.1	What is profiling?	66
3.2	How does the protection work?	67
3.3	The Profiling APIs	68
3.4	The workflow	69
3.5	Implementation	70
3.6	Preparing the Assembly:	76
3.7	Conclusion	79
3.8	References	79
3.9	Greetings	79
4	PRIMER ON REVERSING PALMOS APPLICATIONS EXTENDED EDITION BY, WAST3D_BYTES, SUNTZU	80
4.1	Forewords	80
4.2	Few words on Palm OS	80
4.3	Filling our Reversing Laboratory	82
4.4	Reversing with PRCEXplorer and PRCEdit	86
4.5	Reversing with POSE and SouthDebugger	94
4.6	Advanced Reversing	104
4.7	Conclusions and Further Readings	115
4.8	Greetings	116
5	REVERSING THE PROTECTION'S SCHEME OF ALEXEY PAJITNOV'S GAME DWICE BY GYVER75	117
5.1	Introduction	117
5.2	Target and tools used to reverse it	117



5.3	Analysis	118
5.4	Identification of Check's routines	121
5.5	Suggestions to program a Keygen	149
5.6	Addendum – Exercise	150
5.7	References	152
5.8	Greetings	152
6	LIVE DEBUGGING SYMBIAN APPLICATIONS USING OR NOT USING IDA BY ARGV	153
6.1	Some FAQ	154
7	INTERVIEW WITH SHUB BY GUNTHER	155
	ARTEAM EZINE #5 CALL FOR PAPERS	160

FOREWORDS



. E V O L U T I O N .

Hi all,

It's a long time since I promised the new ARTeam issue. As usual I had to postpone this new issue for a long time due to a lot of things happened in the meantime. As anyone following us probably knows we had to change hosting, rebuild the database of the forums and re-create the web site. All those stuffs have not been a snap and I must excuse with authors who sent to me their contributions some months ago. Anyway we are back now and hopefully we are here to stay. **Now we are hosted at www.accessroot.com write it down!**

This issue is then dedicated to the «EVOLUTION». Evolution ... means a lot of things indeed..

First of all, evolution of the reverse engineering world, this is evolving under the pressure of different issues. One is the appearance of new post-pc devices and the increasing interest in reversing non Windows worlds, the second is the augmented number of professionals involved in this area, on both sides of the barricades. The result is that the reversing skills required to stay at the edge are increasing each and every day. It's not as simple to be original and to release new things, to follow all the possible directions. To stay on top requires constant study and updating and this can be done only in two situations: when you are a kid without a work, or a student, when you are paid by someone (either from black or white reversing worlds). Moreover the economical crisis affecting most parts of the world is not helping! So what to happen? As with all things, we also have to evolve somehow...

The scene is rapidly changing; everyone would really have to agree that cracking is slowly fading away, but not away to oblivion, away to something else. The amount of protections or cool ideas is becoming less frequent, yet on the other hand the protections are becoming more and more complex.

A few years ago, the software industry created new official competences: the Secure Software Development Lifecycle developer, the Security Architect and the Professional Reverse Engineer. This was due to the fact that finally reversing for security needs was recognized as a necessity and has been regulated by laws (since year

The reverse engineering world, is evolving under the pressure of different issues. One is the appearance of new post-pc devices, the second is the augmented number of professionals involved in this area, on both sides of the barricades



2000), so it's now possible to build a career based solely upon reversing. This was not possible at the times of +Fravia or +HCU, so the industry suffered with a lack of professionals figures for reversing/protecting applications. The effect of this change of scenario is under everyone eyes: a big and constant rising of the bar under economic pressure. New protections (Themida, SecuROM,...) and malware witness a change of attitude (you may think that they are complex or not, but they are surely a breaking evolution compared to older things). How many crackers can handle them fully? Not many and those who are keep their secrets for their own clubs.

Which is the solution, I am not sure. I do know what it should be: share and work together, adopting the same methods used by the "official" scientific reversing community. I'll call it scientific reversing, it publishes papers, ezines, forums, conferences... but most of all collaboration is the keyword!

Collaboration is the keyword: software and protection industry collaborates; malware industry collaborates (on both sides). A friend wrote little time ago *"reversers should be more of a team, rather than acting like a loosely knitted group of individuals. That is not to say that we shouldn't have our own individual projects/desires, but there isn't really a sense of direction or purpose, other than to share what we find on our journeys... Sometimes we are blindsided by our own singlemindedness, and fail to see new opportunities that are there...At the same time as achieving this, you then get an improvement in everybodys skills and knowledge, because ALL get taken along together."* Ask yourself, what made +HCU what it was? I think that were two things: a truly sense of wonder and a team thinking (all together aiming at a final result).

For these reasons I want to raise my hand and call for quality contributors and reversers willing to share original new tutorials and experiences. Sharing for what? To keep this experience alive. ARTeam is not our child, is an instrument you may use or not.

So if you want to contribute with tutorials about reversing **anything** or describing original attacks and advanced approaches you are invited to contact us/me.



Coming to this issue, **it is focused on non-windows reversing, or better on non-win32 reversing.** There are insights into the Linux world (Externalist, Gunther) and the Palm (wast3d_bytes has released independently another Palm issue, which has been extended exclusively for this eZine, I also added an interesting video tutorial from Suntzu). There are also two interesting contributions into .NET advanced concepts and one about classical reversing from Gyver75, which I added for the passion for reversing it clearly shows, besides quality of the work. This walkthrough is completed by a series of video tutorials prepared by argv about live debugging Symbian systems, not only using IDA. The non-

win32 tutorials are just a completion of the activity we had already started a long time ago, investigating new reversing worlds like the already known Symbian, iPhone, .NET and Mac. Finally, I decided to add an interview Gunther prepared for me. It was requested a long time ago, you may skip it ;-)

Anyway as you can see the result is another extremely long issue, probably the longest one till now. All the times it happens to write such a big issue I ask myself: how many of you will read or appreciate it, how many will appreciate these "forewords" –hehe-. I really don't know. **One thing I must say is that each chapter can be printed separately, which is especially true for this issue, because each one belongs to completely different worlds.**

Your Favourite Neighbourhood Shubby



DISCLAIMER/LICENSE

All code included with this tutorial is free to use and modify; we only ask that you mention where you found it. This eZine is also free to distribute in its current unaltered form, with all the included supplements.

We have potentially illegal stuff inside. All the commercial programs used within our tutorials have been used only for the purpose of demonstrating the theories and methods described. These documents are released under the license of not using the information inside them to attack systems of programs for piracy. If you do it will be against our rules. No distribution of patched applications has been done under any media or host. The applications used were most of the times already been patched by other fellows, and cracked versions were available since a lot of time. ARTeam or the authors of the papers shouldn't be considered responsible for damages to the companies holding rights on those programs. The scope of this document as well as any other ARTeam tutorial is of sharing knowledge and teaching how to patch applications, how to bypass protections and generally speaking how to improve the RCE art. We are not releasing any cracked application. We are not at all encouraging people to release cracked applications; damages if there will be any have to be claimed to persons badly using information, not under our license.

This disclaimer applies to all ARTeam releases and tutorials!

SUPPLEMENTS

This eZine is distributed with Supplements for each paper; the supplements are stored in folders with the same title of the paper. Almost all the papers have supplements, check it.

VERIFICATION

ARTeam.esfv can be opened in the ARTeamESFVChecker to verify all files have been released by ARTeam and are unaltered. The ARTeamESFVChecker can be obtained in the release section of the ARTeam site: <http://releases.accessroot.com>



1 REVERSING BINARY 500 BY EXTERNALIST

1.1 INTRODUCTION

This tutorial is about reversing [Binary500](#) from DEFCON CTF Pre-quals 2008. Binary500 runs on a FreeBSD server and has a keyfile associated with it. A keyfile is simply a text file that contains a secret password. The goal of this challenge is to somehow retrieve the contents of this keyfile (which would have earned you an extra 500 points if it were during the contest). Nobody knows how the binary interacts with the keyfile, so dynamic or static reverse engineering of the binary might be required, and considering that this is the most difficult out of all 5 binary challenges, it could be hard (or easy for some of you people :P) and require a lot of work. I will try to thoroughly explain through the steps, but there is still some knowledge required for the reader to get everything out of this tutorial. The reader is recommended to

- be familiar with IDA
- know how to use IDA Python
- have basic knowledge of the common reversing tools on a non Windows platform
- be familiar with the shell commands used on linux, freebsd, etc.
- know how to program

If you have never done any reversing on a non-windows platform and get stuck somewhere, always do a search on google instead of just giving up. I myself haven't had that many reversing sessions on FreeBSD and google happened to be my best friend. This tutorial is targeted to people who aren't that confident in reversing in FreeBSD so the content might seem a little over explained in some parts. If you feel so, you can just skip the parts on the basic usage of tools for fast reading.

1.2 TOOLS NEEDED

- IDA pro
- IDA python
- A FreeBSD machine
- GNU binutils

1.3 EXPLORING THE BINARY

You might think reversing a binary on a non-windows platform is somewhat alien, but the fundamentals don't really differ greatly. It's just as simple as following certain known procedures like one would do on a windows binary, which is what I'm going to demonstrate here.

First off, let's look at the text that comes with the binary.

```
500: We're running this. Ask it for the key *very* nicely and it may give it to you ;):  
ihatedns.allyourboxarebelongto.us:2600 file
```



It seems like the binary is running on the server and we have to connect to it. Since the binary is no longer available on the server, it would be necessary to set up a custom FreeBSD server and run the binary there. If you don't already have a FreeBSD machine installed, then you could easily get a vmware image from [here](#).

After you're done downloading and booting the FreeBSD machine, get the binary(which I renamed as binary500) using the wget command, and have it running on the machine. You would have to supply a text file as an argument, which I named 'keyfile'. The keyfile will contain some sort of password that we must retrieve. It would be better to run the binary in the background by entering 'binary500 keyfile &' so we could later on analyze the binary without having to kill the binary500 process all the time.

Next, we want to know the IP address of the FreeBSD machine we're using so type 'ifconfig'. Mine turned out as 192.168.1.131 so that's what's going to be used throughout the tutorial.

Ok, first off, use nmap to scan for open ports on the server. 2600 turns out to be opened as tcp. Now let's try to simply connect to the server to see what happens. Just type as the following:

```
externalist@Externalist:~$ nmap 192.168.1.131
Starting Nmap 4.53 ( http://insecure.org ) at 2008-07-10 00:16 KST
Interesting ports on 192.168.1.131:
Not shown: 1713 closed ports
PORT      STATE SERVICE
2600/tcp  open  zebrasrv
Nmap done: 1 IP address (1 host up) scanned in 10.290 seconds
externalist@Externalist:~$ nc 192.168.1.131 2600
◆#◆M#◆B#
```

We get sent some weird hex bytes that obviously don't look like ascii characters. Let's dump it into a file and use the hexdump utility to dump the contents.

```
externalist@Externalist:~$ nc 192.168.1.131 2600 > temp
externalist@Externalist:~$ hexdump -C temp
00000000 fe 00 00 00 d2 00 00 00 21 01 00 00 96 03 00 00 |.....!.....|
00000010 59 02 00 00                                     |Y...|
00000014
```

We see that a total amount of 20 bytes were sent from the server just by simply connecting to it. If repeated, then do we get the same results?

```
externalist@Externalist:~$ nc 192.168.1.131 2600 > temp
externalist@Externalist:~$ hexdump -C temp
```




```
00000000 87 02 00 00 f4 01 00 00 b3 01 00 00 49 03 00 00 |.....!...|
00000010 52 01 00 00                                     |R...|
00000014
```

No. And by the looks of it, the server sends 5 Words expanded to Dwords. The meaning of this data is still unclear. After receiving 20 bytes, the server is waiting for the client to send some data. I just randomly typed anything and found out that the connection closes. I got curious and did the same thing, but this time typed in only a few characters, and the server still waited to receive more data. This was a clue that the server may be expecting only a certain amount of bytes to be received, so I just basically kept typing a couple of characters to find out how many bytes the server was expecting.

```
externalist@Externalist:~$ nc 192.168.1.131 2600
#]### #asdf
asdlfkjaklsdjflkasdf
externalist@Externalist:~$ nc 192.168.1.131 2600
p#7#w# #adskfjaskdfjasldkfjllkasjdfiasf
externalist@Externalist:~$ nc 192.168.1.131 2600
X#(# #s#1234567890
qqqqqqqqqqqq
externalist@Externalist:~$ nc 192.168.1.131 2600
#2##12345678901
```

For brevity, some of the repetitive parts were deleted. Sending only a few characters doesn't seem to do anything, but if you send exactly 12 bytes to the server, then the server breaks the connection. So that must be one condition of this binary. The user must initially send no more or less than 12 bytes.

What next? When reversing in Windows, one would open up PEID or any other Packer Identifier and determine the packer first. But on Linux or FreeBSD, it is very hard to see a file wrapped up with a packer/protector when most of the distributed software is Open Source. There is simply no need. One way to test if a program is packed is to compress the file and compare it with the original file. If the file was packed, then the file size will almost be the same.

```
-rwxr-xr-x 1 externalist externalist 26300 Jun 30 11:30 binary500
-rw-r--r-- 1 externalist externalist 19751 Jun 30 12:33 binary500.tar.gz
-rw-r--r-- 1 externalist externalist 16 Jun 30 11:32 keyfile
-rw-r--r-- 1 externalist externalist 0 Jun 30 12:33 temp
-rw-r--r-- 1 externalist externalist 17 Jun 30 12:29 test
```



The compression rate is hard to tell whether it's packed or not, but we could probably safely assume that the file is not packed since it's not a Windows PE. The next thing to do is see the characteristics of the file, what function it uses, and what strings are contained.

```
$ file binary500

binary500: ELF 32-bit LSB executable, Intel 80386, version 1 (FreeBSD), for FreeBSD 6.3, dynamically
linked (uses shared libs), stripped

$ objdump -R binary500

binary500:      file format elf32-i386-freebsd

DYNAMIC RELOCATION RECORDS

OFFSET      TYPE          VALUE
0804e0c8    R_386_JUMP_SLOT  waitpid
0804e0cc    R_386_JUMP_SLOT  getgid
0804e0d0    R_386_JUMP_SLOT  printf
0804e0d4    R_386_JUMP_SLOT  random
0804e0d8    R_386_JUMP_SLOT  recv
0804e0dc    R_386_JUMP_SLOT  geteuid
0804e0e0    R_386_JUMP_SLOT  getegid
0804e0e4    R_386_JUMP_SLOT  usleep
0804e0e8    R_386_JUMP_SLOT  memcpy
0804e0ec    R_386_JUMP_SLOT  perror
0804e0f0    R_386_JUMP_SLOT  getuid
0804e0f4    R_386_JUMP_SLOT  socket
0804e0f8    R_386_JUMP_SLOT  send
0804e0fc    R_386_JUMP_SLOT  accept
0804e100    R_386_JUMP_SLOT  calloc
0804e104    R_386_JUMP_SLOT  write
0804e108    R_386_JUMP_SLOT  bind
0804e10c    R_386_JUMP_SLOT  chdir
0804e110    R_386_JUMP_SLOT  initgroups
0804e114    R_386_JUMP_SLOT  setsockopt
0804e118    R_386_JUMP_SLOT  setgid
0804e11c    R_386_JUMP_SLOT  signal
0804e120    R_386_JUMP_SLOT  read
```



```
0804e124 R_386_JUMP_SLOT memcmp
0804e128 R_386_JUMP_SLOT listen
0804e12c R_386_JUMP_SLOT fork
0804e130 R_386_JUMP_SLOT setresuid
0804e134 R_386_JUMP_SLOT memset
0804e138 R_386_JUMP_SLOT err
0804e13c R_386_JUMP_SLOT _init_tls
0804e140 R_386_JUMP_SLOT seteuid
0804e144 R_386_JUMP_SLOT getpwnam
0804e148 R_386_JUMP_SLOT atexit
0804e14c R_386_JUMP_SLOT setresgid
0804e150 R_386_JUMP_SLOT exit
0804e154 R_386_JUMP_SLOT strlen
0804e158 R_386_JUMP_SLOT open
0804e15c R_386_JUMP_SLOT setegid
0804e160 R_386_JUMP_SLOT srandomdev
0804e164 R_386_JUMP_SLOT vasprintf
0804e168 R_386_JUMP_SLOT setuid
0804e16c R_386_JUMP_SLOT close
0804e170 R_386_JUMP_SLOT free
```

```
$ strings binary500
```

```
/libexec/ld-elf.so.1
```

```
FreeBSD
```

```
libc.so.6
```

```
fabs
```

```
waitpid
```

```
getgid
```

```
random
```

```
recv
```

```
geteuid
```

```
_DYNAMIC
```

```
getegid
```

```
usleep
```




```
memcpy
perror
getuid
socket
send
_init
accept
calloc
write
environ
bind
__deregister_frame_info
chdir
initgroups
setsockopt
__progname
setgid
signal
read
memcmp
listen
fork
setresuid
memset
_init_tls
seteuid
getpwnam
_fini
atexit
setresgid
_GLOBAL_OFFSET_TABLE_
strlen
open
setegid
```



```
_Jv_RegisterClasses
srandomdev
vasprintf
setuid
__register_frame_info
close
free
_edata
__bss_start
_end

$FreeBSD: src/lib/csu/i386-elf/crti.S,v 1.7 2005/05/19 07:31:06 dfr Exp $
Unable to set SIGCHLD handler
Unable to create socket
Unable to set reuse
Unable to bind socket
Unable to listen on socket
Failed to find user %s
drop_privs failed!
setgid current gid: %d target gid: %d
setuid current uid: %d target uid: %d
open for save failed
Usage: ./MathIsHarD <keyfile>
N@$FreeBSD: src/lib/csu/i386-elf/crtn.S,v 1.6 2005/05/19 07:31:06 dfr Exp $
```

The first result indicates that the file has no symbols but was dynamically linked, therefore we will be able to see the function names when debugging/disassembling. The second results shows all the functions used in the program, and `random/srandomdev` sort of explains why the 20 bytes sent was always different. Strings doesn't show anything really useful other than the fact that the original file name might have been `MathIsHarD`. Looks like we're going to have some headaches solving math problems later on. :)

Next, we want to see what functions are being called, and `ltrace` will be used for this.

```
$ ltrace -f ./binary500 test
```

```
atexit(0x28053b88)          = 0
```



```
atexit(0x804dc28)          = 0
open("test", 0, 05001245611)    = 3
read(3, "abcdefghijklmnop\n", 1024)  = 17
signal(20, 0x804b10f)          = NULL
socket(2, 1, 0)              = 4
setsockopt(4, 65535, 4, 0xbfbfecb8, 4)  = 0
bind(4, 0xbfbfeca0, 16, 0, 0x280a0200)  = 0
listen(4, 20, 16, 531, 0x280a0200)    = 0
accept(4, 0xbfbfeca0, 0xbfbfecb8, 0x804b56c, 4) = 5
fork()                      = 885
[pid 884] close(5)          = 0
[pid 884] accept(4, 0xbfbfeca0, 0xbfbfecb8, 0x804b56c, 4 <unfinished ...>
[pid 885] +++ exited (status 255) +++
--- SIGTSTP (Child exited: 20) ---
waitpid(-1, 0xbfbfe940, 1)    = 885
waitpid(-1, 0xbfbfe940, 1)    = -1
```

The program reads the keyfile(test is just a temporarily created one) and probably stores the contents in memory. Then it gets ready to accept connections from clients. When a client connects, it forks and waits for more connections. Nothing really interesting.

It looks like we won't get any further by dynamic analysis, so now is the time to fire up IDA and do some real reversing. :)

1.4 REVERSING THE BINARY

This section focuses more on the analysis of the binary, than the reverse engineering process itself. Explaining every little detail on how the reversing process is done is out of the scope of this tutorial, so if you're not that confident in reversing, then I highly recommend to read [this book](#) before going any further. The analysis will start from the Entrypoint.

```
; int __fastcall start(int, void (__cdecl *>(), int, char)
public start
start proc near

arg_0= dword ptr 8
arg_4= byte ptr 0Ch

push    ebp
mov     ebp, esp
push    edi
push    esi
push    ebx
sub     esp, 0Ch
and     esp, 0FFFFFFF0h
mov     ebx, [ebp+4]
mov     edi, edx
lea     esi, [ebp+ebx*4+arg_4]
test    ebx, ebx
mov     ds:environ, esi
jle     short loc_8048C77

mov     eax, [ebp+arg_0]
test    eax, eax
jz      short loc_8048C77
```

Although I'm writing this document in linux, I tend to use IDA in windows cause the graphical view facilitates reversing in many situations(It saves a few seconds. :P). The picture above shows the entry point of the binary, which obviously looks like some kind of Startup code. We immediately skip to the main function below.

```
loc_8048C8C:
sub     esp, 0Ch
push    offset _term_proc ; void (__cdecl *]()
call    _atexit
call    _init_proc
push    eax
push    esi
lea     eax, [ebp+arg_0]
push    eax
push    ebx
call    sub_8049891
add     esp, 14h
push    eax ; int
call    _exit
```

This is the main(argc, argv) function after the startup code. When you enter the function, you will see something like this.

```

sub_8049891 proc near
push    offset loc_804B8EE
retn
sub_8049891 endp

```

```

loc_804B8EE:                                ; CODE XREF:
                                           ; DATA XREF:
push    ebp
pushf
stc
jb     loc_804AE16
leave
outsb
std
xor    [esi+23h], esp
sub    al, 0EDh
jno   short near ptr byte_804B933
xchg  bh, [eax-4737CAF5h]
xchg  eax, esi
; START OF FUNCTION CHUNK FOR sub_8049AA2

```

```

loc_804AE16:                                ; CODE XREF:
popf
mov    ebp, esp
sub    esp, 8
and    esp, 0FFFFFF0h
mov    eax, 0
add    eax, 0Fh
push  offset loc_804D756
retn

```

```

loc_804D756:                                ; CODE XREF:
                                           ; DATA XREF:
add    eax, 0Fh
shr    eax, 4
shl    eax, 4
sub    esp, eax
mov    dword ptr [ebp-4], 0
jmp   loc_80495FD

```

Looks like the entry point is obfuscated to scare away newbie reversers. This doesn't seem to be a big problem at first, but if you keep following the jumps/rets until you get tired, then you will notice that it's not just the



entrypoint that's obfuscated, but the whole program. And what's even worse is the program is broken up into tiny pieces and scattered all over the place that it makes it impossible to create a graphical view of the binary.

Graphical view is indeed a great concept and speeds up the reversing process, but it is not essential; one could still live without it. However, there are some obstacles to overcome in order to successfully reverse with minimum amount of fuss. For instance, in this particular binary, IDA has failed to define functions, and stack variables depend on functions in IDA. You could try to manually define a function but IDA will get confused very easily because of the way the binary is constructed, which is exactly what the obfuscation scheme was implemented for; Anti-Disassembling. However, it is still possible to define stack variables with other means, such as structure offsets. The reason why this is possible is because the binary is just shattered, and nothing more. It still uses the traditional ebp based stack frame scheme, which is noticeable in the 3rd picture above.

This is exactly what I did when I first started analyzing this piece of binary. Jumping back and forth wasn't such a big deal for a long time, and things were running very smoothly, but by the time the binary was analyzed halfway, some unexpected problems started to arise. Everything was fine when there were a couple of jcc branches, and only one loop, but later on loops/breaks/returns within loops started to emerge, and memorizing all those variants became very tiring and writing them all down while referencing them one by one was also a very tedious task. That's when I decided to make some sort of tool that would glue all those broken parts into one piece.

If you jump back and forth the broken pieces for quite a while, you will notice that a certain pattern exists on the code where once piece jumps to another.

```

sub_8049891 proc near
push    offset loc_804B8EE
retn
sub_8049891 endp

```

```

pushf
stc
jb     loc_804AE16

```

```

push    ecx
mov     ecx, offset sub_8049917
call   ecx ; sub_8049917

```

Those are the only 3 patterns that exist in this binary. If there was a way to manually add cross references for all those 3 patterns, then it would be possible to define functions, and also navigate through a whole function in a graph, instead of mindlessly jumping back and forth basic blocks in the text view. For this, I used the internal functions in IDA with IDA Python. The following Python script will work as a 'gluer' which combines all the broken pieces into one nice function. This is only a 'Semi Automatic' version. I tried to construct one that does everything automatically, but failed gracefully. :) For that reason, I admire the guys at Sexy Pandas team who were able to make an automatic deobfuscator, which doesn't only attach all the pieces together, but also takes out all the irrelevant instructions, then reconstructs the whole binary in one perfect piece! After that, Hex-Rays can do the rest of the job. That is just truly amazing, I must say. :D



Anyways, back to the python script. I tried to comment everything thoroughly so it would be easier to understand. However, please forgive me if the code looks very messy and lame. I only just started learning python recently btw. :P

```
# - Header Note -
# If you see any functions you are unsure of, then refer to the headers files in IDA sdk,
# specifically bytes.hpp and
# functions.hpp. Most of the functions used here belong to those two header files. And also, some
# IDC functions
# are used so the internal IDA help file may also be useful.

# This function takes care of re-attaching the blocks when they break apart for some reason.
# Since the blocks are forced to be attached, they sometimes break apart because of their nature of
# being
# seperated. That's when this function will be used.
def Reanalyze_Cross_References():
    func_iter = func_tail_iterator_t(get_func(ScreenEA()))
# Initializes a function tail iterator class. With a function iterator, one could iterate through
# function tails(basic blocks that belong to a function)
# that belong to a particular function.
    status = func_iter.main()
# Status is initialized. Status is 0 when there are no more functions to iterate.
    while status:
        chunk = func_iter.chunk()
# Fetching the next function chunk
        status = func_iter.next()
        code = Heads(chunk.startEA, chunk.endEA)
# Generating a list(array) of all the address of the codes that belong to the chunk
        last_instruction = code[len(code)-1]
# getting the address of the last instruction
        next_instruction = last_instruction + 5
# getting the address of the instruction after the last instruction
        if (GetMnem(last_instruction) == 'mov') and (GetOpnd(last_instruction,0) == 'ebx')\
        and ((GetOperandValue(last_instruction,1) & 0xFF000000) == 0x8000000)\
        and (GetMnem(next_instruction) == 'retn'):
# if the code at the last instruction matches (mov ebx, 0x8*****; retn), then proceed
            AddressFrom = last_instruction
            AddressTo = GetOperandValue(last_instruction,1)
            AddCodeXref(AddressFrom,AddressTo,f1_JF)
            SetManualInsn(last_instruction,'jump' + ' loc_%x'
%(GetOperandValue(last_instruction,1)))
            MakeComm(last_instruction,"")
# Add a cross reference from the source to the destination, and overwrite the irrelevant
# instruction with
# a more meaningful manual instruction
            if (GetMnem(last_instruction) == 'mov') and (GetMnem(last_instruction-1) ==
'push')\
            and ((GetOperandValue(last_instruction,1) & 0xFF000000) == 0x8000000)\
            and (GetMnem(next_instruction) == 'call'):
# if the code at the last instruction matches (push reg; mov reg,0x8*****; call const), then
# proceed
                AddressFrom = last_instruction
                AddressTo = GetOperandValue(last_instruction,1)
                AddCodeXref(AddressFrom,AddressTo,f1_JF)
                SetManualInsn(last_instruction,'jump' + ' loc_%x'
%(GetOperandValue(last_instruction,1)))
                SetManualInsn(last_instruction-1,'nop')
                DestAddress = GetOperandValue(last_instruction,1)
                SetManualInsn(DestAddress,'nop')
                SetManualInsn(DestAddress+1,'nop')
                MakeComm(last_instruction,"")
# Add a cross reference from the source to the destination, and overwrite the irrelevant
# instruction with
# a more meaningful manual instruction

Choice = AskLong(5,'1 : reanalyze, 2 : remove tail, 3 : delete function\n\
4 : append tail generic, 5: append tail custom, 6 : Add Cross Reference\n\')
```



```

7 : Add Cross Reference custom, 8 : testing functions')
# Ask the user which option he/she wants to choose

if Choice == 1:
    reanalyze_function(get_func(ScreenEA()))
# reanalyzing a function would sometimes(I don't know why not all the time but that's how it is :/
) make IDA recognize
# local variables. It requires a function to be already constructed.

if Choice == 2:
    remove_func_tail(get_func(ScreenEA()),ScreenEA())
# Removing function tails that obviously don't belong to a function will result in less confusion.

if Choice == 3:
    del_func(ScreenEA())
# Delete a function only when you are sure about it. Deleting a function will make all the function
tails get deleted also.
# If you have put together many function tails, and accidentally delete a function, then all your
work will be lost.

if Choice == 4:
    TailStart = AskAddr(0,'Enter the tail start :');
    TailEnd = AskAddr(0,'Enter the tail end :');
    append_func_tail(get_func(ScreenEA()),TailStart,TailEnd)
# Manually add a function tail by entering the tail start and end addresses.

if Choice == 5:
# This is the default choice, and the most used one. Putting the cursor on a correct line, this
function will
# attach the basic blocks that are not already attached to each other. You will have to use this
function numerous times
# because of the reason stated above; this is only a 'Semi automatic' script. :P Don't blame me
for not completing it.
# I was simply too tired, lazy. :P
    OriginalPosition = ScreenEA()
    AddressFrom = ScreenEA()
    AddressTo = GetOperandValue(ScreenEA(),0)
    if AddressTo == 0:
        AddressTo = GetOperandValue(ScreenEA(),1)
# Retrieving the source/destination addresses to use when adding cross references.

    Previous_Mnemonic = GetMnem(ScreenEA()-1)
    Current_Mnemonic = GetMnem(ScreenEA())
    Next_Mnemonic = GetMnem(ScreenEA()+5)
# Retrieving mnemonics to use in the comparison.
    if (((Current_Mnemonic == 'push') and ((GetOperandValue(ScreenEA(),0) & 0xFF000000) ==
0x80000000))\
or ((Current_Mnemonic == 'mov') and ((GetOperandValue(ScreenEA(),1) & 0xFF000000) ==
0x80000000))\
and (Next_Mnemonic == 'retn')):
# Does the instruction under the cursor match (push/mov const; retn) ?
        func_setend(ScreenEA(),ScreenEA()+5)
        PatchByte(ScreenEA(),0xBB)
# Then change the function end, and patch the 'push' to 'mov'. Why patch? Cause if you leave it as
push, then IDA
# will have trouble adding cross references.
        if (Current_Mnemonic == 'mov') and (Previous_Mnemonic == 'push')\
and ((GetOperandValue(ScreenEA(),1) & 0xFF000000) == 0x80000000) and (Next_Mnemonic ==
'call'):
# Does the instruction under the cursor match (push reg; mov 0x8*****; call const) ?
            func_setend(ScreenEA(),ScreenEA()+5)
# Then change the function end.
            autoWait()
# Wait till the analysis finishes, so the script won't screw up.
            if (get_func_num(AddressTo) != get_func_num(AddressFrom)) and (get_func_num(AddressFrom) !=
-1):
# If the source function matches the destination function, then proceed.
                temp = get_item_end(AddressTo)
                temp = prev_head(temp,temp-1000)
# If the current address is in the middle of an item(code or data), then Make**** functions will

```



```

fail.
# This was not documented, hence, caused a lot of headaches. The above code sets the current
address to the beginning
# of the item. Why didn't I use get_item_start? Because sadly, IDA Python doesn't import that
function.
    if temp != AddressTo:
        MakeUnkn(temp,0)
        autoWait()
        MakeCode(AddressTo)
        autoWait()
# Sometimes, the destination address will be right in the middle of an instruction or data. That's
because of the
# nature of this kind of obfuscation. IDA confuses code from data, and data from code. The above
code will define
# code at the location even if it's in the middle of an item. It will make it possible to define a
function on that location.
    if get_func_num(AddressTo) == -1:
# If no function exists in the destination location
        add_func(AddressTo,AddressTo+10)
        temp = get_item_end(AddressTo+10)
        temp = prev_head(temp,temp-1000)
        MakeComm(temp,'Warning! This might not\ne the end of the block\nUse "e" to
set end')
# Arbitrarily add a function to the destination location, and add a comment there so the user can
fix it later.
        add_func(AddressTo,BADADDR)
        fchunk = get_fchunk(AddressTo)
        StartAddress = fchunk.startEA
        EndAddress = fchunk.endEA
        del_func(get_func(AddressTo).startEA)
        append_func_tail(get_func(ScreenEA()),StartAddress,EndAddress)
# Create a function tail and add it to the main function.
        autoWait()
        AddCodeXref(AddressFrom,AddressTo,f1_JF)
# Add a cross reference to it, so it will be connected when viewed in graph mode.
        autoWait()
        Reanalyze_Cross_References()
        Jump(OriginalPosition)
# Blocks will break apart after adding cross references, because IDA forces a reanalysis.
# Using the Reanalyze_Cross_References() to put them back together.

if Choice == 6:
    AddressFrom = ScreenEA()
    AddressTo = GetOperandValue(ScreenEA(),1)
    AddCodeXref(AddressFrom,AddressTo,f1_JF)
# Automatically add a cross reference below the cursor that needs to be added.

if Choice == 7:
    OriginalPosition = ScreenEA()
    Reanalyze_Cross_References()
    Jump(OriginalPosition)
# This function will be used to unite the separated blocks.

if Choice == 8:
    func_setend(ScreenEA(),ScreenEA())
# Testing function. Currently used to set a function end.

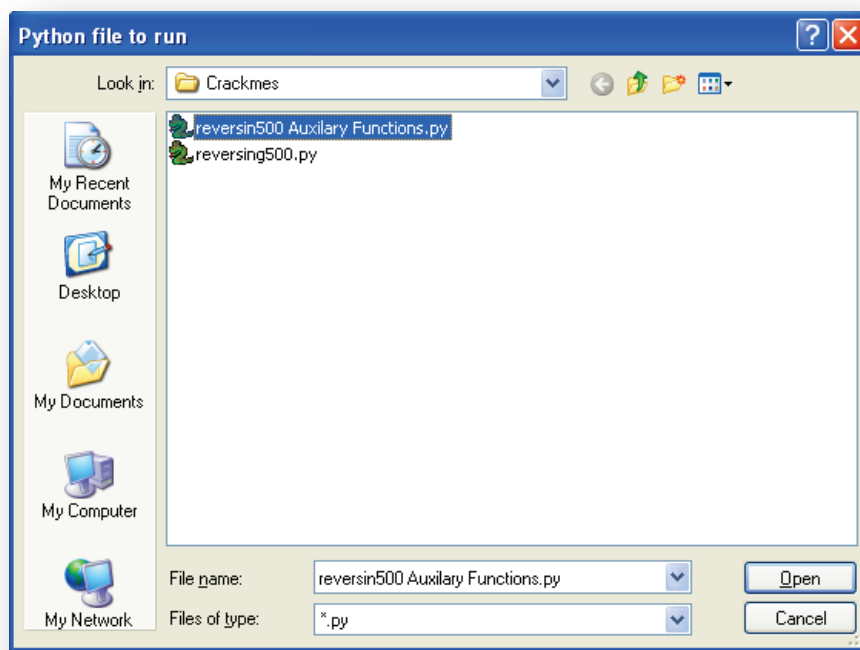
```

Pretty crappy piece of code, isn't it? :P If you have read through and understood the code above, then you will know how to use it. Nevertheless, I will explain just in case someone didn't feel like reading through all those messy codes and comments.

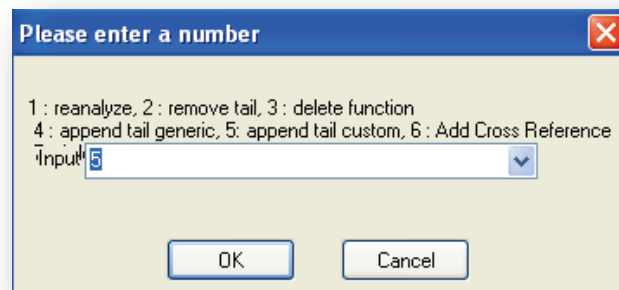
Let's start from the main function.

```
sub_8049891 proc near
push    | offset loc_804B8EE
retn
sub_8049891 endp
```

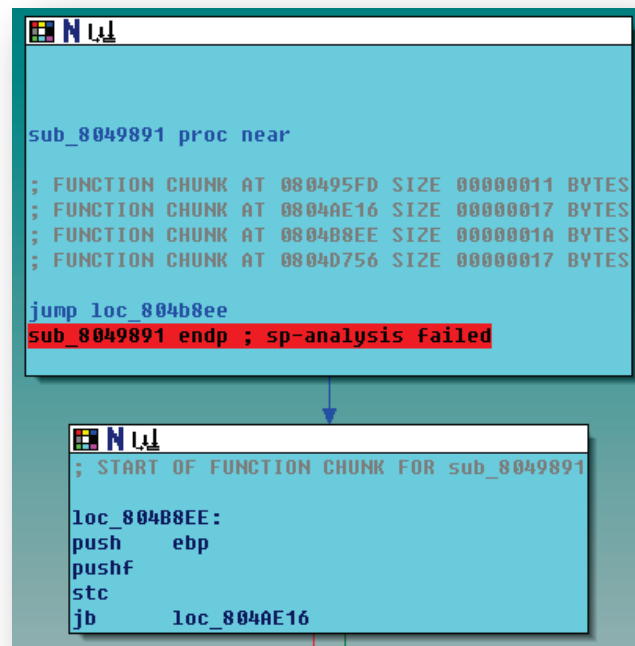
The code beside is one of the obfuscation patterns. Put your cursor on the push instruction and load the python script.



The default will be set to 5, which acts as the attaching function.



Press OK and the source should be attached with the destination.



```
sub_8049891 proc near
; FUNCTION CHUNK AT 080495FD SIZE 00000011 BYTES
; FUNCTION CHUNK AT 0804AE16 SIZE 00000017 BYTES
; FUNCTION CHUNK AT 0804B8EE SIZE 0000001A BYTES
; FUNCTION CHUNK AT 0804D756 SIZE 00000017 BYTES

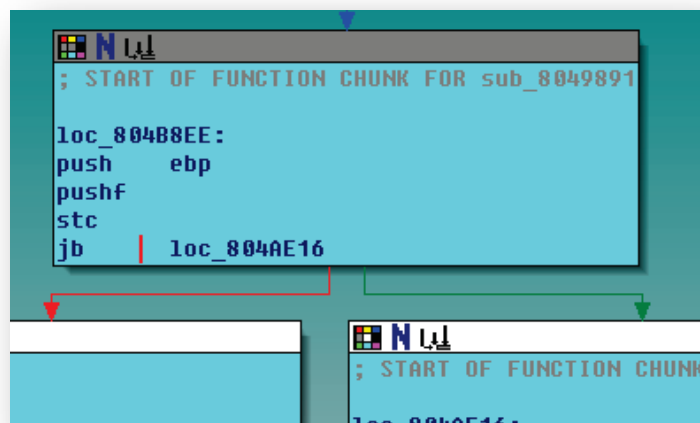
jump loc_804b8ee
sub_8049891 endp ; sp-analysis failed
```

↓

```
; START OF FUNCTION CHUNK FOR sub_8049891

loc_804B8EE:
push    ebp
pushf
stc
jb     loc_804AE16
```

This only works one at a time as mentioned earlier. If you want to fully automatize it, feel free to edit the code.



```
; START OF FUNCTION CHUNK FOR sub_8049891

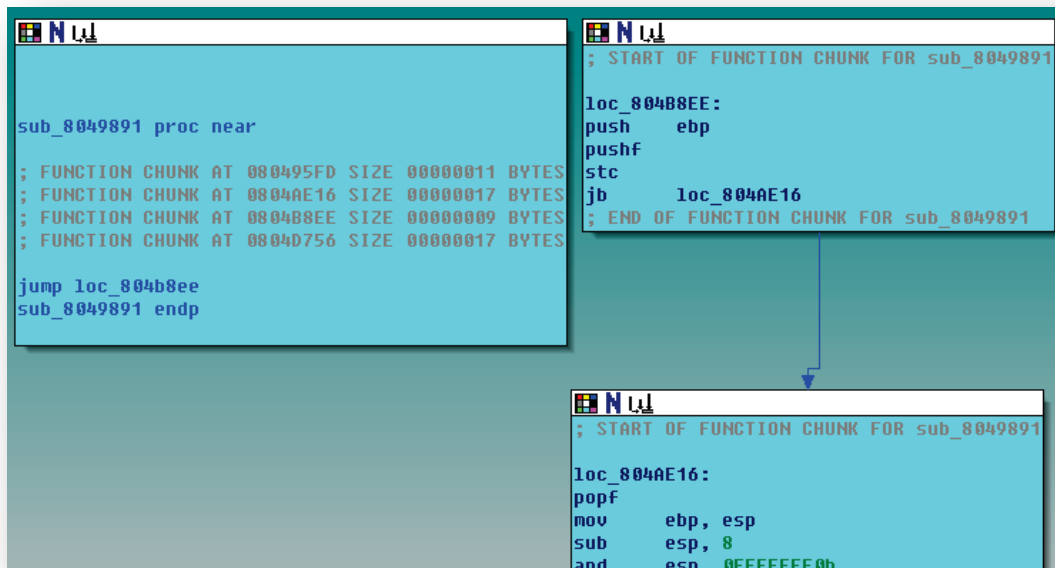
loc_804B8EE:
push    ebp
pushf
stc
jb     loc_804AE16
```

↓

```
; START OF FUNCTION CHUNK FOR sub_8049891

loc_804AE16:
```

In the above code, IDA thinks that `jb` will lead to both paths, when it always goes to one path. Therefore, you must set the function end with the 'e' key at `jb`, and IDA will fix the cross references so it will only lead to one path.



```

sub_8049891 proc near
; FUNCTION CHUNK AT 080495FD SIZE 00000011 BYTES
; FUNCTION CHUNK AT 0804AE16 SIZE 00000017 BYTES
; FUNCTION CHUNK AT 0804B8EE SIZE 00000009 BYTES
; FUNCTION CHUNK AT 0804D756 SIZE 00000017 BYTES

jump loc_804b8ee
sub_8049891 endp

; START OF FUNCTION CHUNK FOR sub_8049891

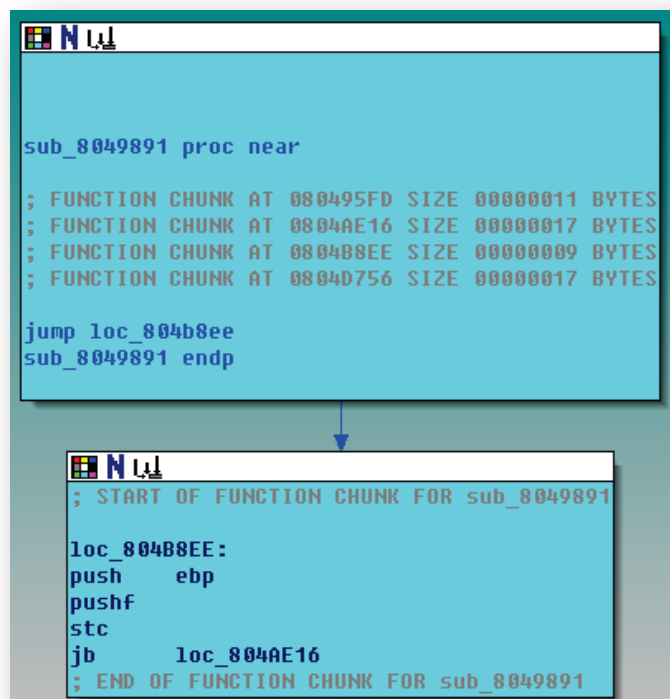
loc_804b8ee:
push    ebp
pushf
stc
jb     loc_804ae16
; END OF FUNCTION CHUNK FOR sub_8049891

; START OF FUNCTION CHUNK FOR sub_8049891

loc_804ae16:
popf
mov     ebp, esp
sub     esp, 8
and     esp, 0FFFFFFF0h

```

However, the graph will break apart cause IDA forces a function reanalysis. Reunite it with the 7th option in the python script.



```

sub_8049891 proc near
; FUNCTION CHUNK AT 080495FD SIZE 00000011 BYTES
; FUNCTION CHUNK AT 0804AE16 SIZE 00000017 BYTES
; FUNCTION CHUNK AT 0804B8EE SIZE 00000009 BYTES
; FUNCTION CHUNK AT 0804D756 SIZE 00000017 BYTES

jump loc_804b8ee
sub_8049891 endp

; START OF FUNCTION CHUNK FOR sub_8049891

loc_804b8ee:
push    ebp
pushf
stc
jb     loc_804ae16
; END OF FUNCTION CHUNK FOR sub_8049891

```

Now if you ever see the (push const; retn) or (pushf; stc; jb const)sequence, then you could always fix it with the above method.

```

loc_804BFD0:
pushf
stc
jb     loc_804AA43

fadd  st, st(6)
db    65h
add   ch, al
mov   dl, 0F4h
and   ah, [ebx+7C343E0Fh]
mov   al, 0Ah
push  ebx
fmul  qword ptr [ecx]
not   byte ptr [ecx+6Eh]
; END OF FUNCTION CHUNK FOR sub_8049891
  
```

While analyzing, if you see any function tails that obviously don't belong to a function like above, then you could delete it with the second option in the python script. It's not necessary, but it makes the graph look a little cleaner.

```

loc_804C396:
jmp   sub_80498AE
; END OF FUNCTION CHUNK FOR sub_8049891
  
```

```

inc   ecx
pop   es
jz    short loc_804A051

wait
shr   edx, cl
das
  
```

If you see any Jmps or Jccs that IDA didn't correctly resolve, you could also manually resolve them with the default 5th option in the script.

```

push  offset aUsage_Mathisha ; "U
push  ecx
mov    ecx, offset sub_8049917
call  ecx ; sub_8049917
inc   ecx
pop   es
jz    short loc_804A051

```

This is another kind of pattern. Put the cursor on the mov instruction above the call and Alt + F7(Python script shortcut), Enter, Enter(Default selection 5), and it should properly resolve the cross references.

```

sub    esp, 0Ch
push  offset aUsage_Mathisha ; "Us
nop
jump  loc_8049917
; END OF FUNCTION CHUNK FOR sub_804B

```

↓

```

; START OF FUNCTION CHU
loc_8049917:
nop
nop
call  _printf
add   esp, 10h

```

Sometimes you will see that IDA has mis-analyzed the end of a function chunk like below.

```

; START OF FUNCTION CHUNK FOR sub_8049891
loc_804C572:                                ; CODE XREF
    popf
    push  0A28h
    call  sub_804C3D0
; END OF FUNCTION CHUNK FOR sub_8049891
; -----
    add   esp, 10h
    mov   [ebp-4], eax
    sub   esp, 8
    pushf
    stc
    jb   near ptr dword_804A500+77h

```

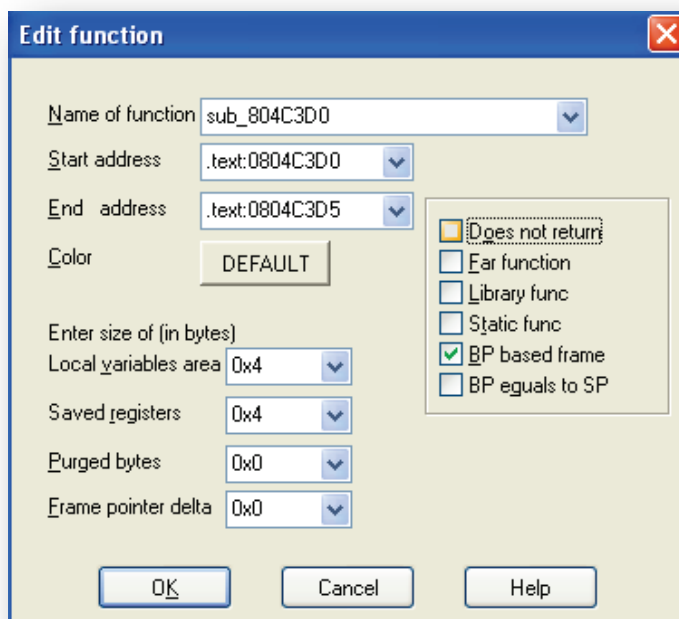
We already know that 'e' will renew the function end, but in some occasions, just setting the function end won't solve the problem.

```

; START OF FUNCTION CHUNK FOR sub_8049891
loc_804C572:                                ; CO
        popf
        push    0A28h
        call   sub_804C3D0
; -----
        add    esp, 10h
        mov    [ebp-4], eax
        sub    esp, 8
        pushf
        stc
        jb    near ptr dword_804A5
; END OF FUNCTION CHUNK FOR sub_8049891

```

In this example, the function end was set to a correct address, but IDA split the chunk into two pieces. This is because the function `sub_804C3D0` is marked as 'doesn't return'. This could be fixed by unchecking the 'function doesn't return' checkbox in the function properties dialogbox.



IDA analyzes the function as 'does not return' because it contains an `_exit()` function. After unchecking the option, the chunk will now be in one piece now.

Okay, before this tutorial turns into a 'How to use IDA' tutorial, I'll stop talking about IDA and start focusing on the binary itself. Understanding everything explained so far, and with a decent knowledge of using IDA, you will be able to successfully construct a full set of functions out of all those broken/shuffled basic blocks. I know it's quite a boring task to do all that work by hand, but like I said, feel free to update the code to a fully automated version. And hey, at least it's better than nothing. :D

We have two options now. You could either

1. Add the cross references along the way, while reversing at the same time. However, stack variables cannot be used, so structure offsets will have to be used instead.
2. Add cross references until a full function is made, and after that, stack variables will be accessible. Loops will also be easily recognizable.

What I did is use 1 on functions that seemed to be long, and use 2 on the rest. It would be preferable to use the later if you have more patience so everything will be in your sight, instead of not knowing what will come up next. Don't forget to save frequently because the script sometimes screws up(!), and when that happens, all your work will be gone. :o

Having this all in mind, let's start to analyze the program from beginning to end. If you feel hard to follow, then you can use the idb file that comes with this tutorial.

The screenshot shows a debugger window with assembly code and a 'Graph overview' window. The assembly code is as follows:

```

; FUNCTION CHUNK AT 0804C7BB SIZE 00000008 BYTES
; FUNCTION CHUNK AT 0804D756 SIZE 00000017 BYTES

jump loc_804b8ee
Main endp

; START OF FUNCTION CHUNK FOR realmain
realmain:
push    ebp
pushf
stc
jb     loc_804AE16
; END OF FUNCTION CHUNK FOR realmain

; START OF FUNCTION CHUNK FOR Main
loc_804AE16:
popf
mov     ebp, esp
sub     esp, 8           ; size_t
and     esp, 0FFFFFF0h

```

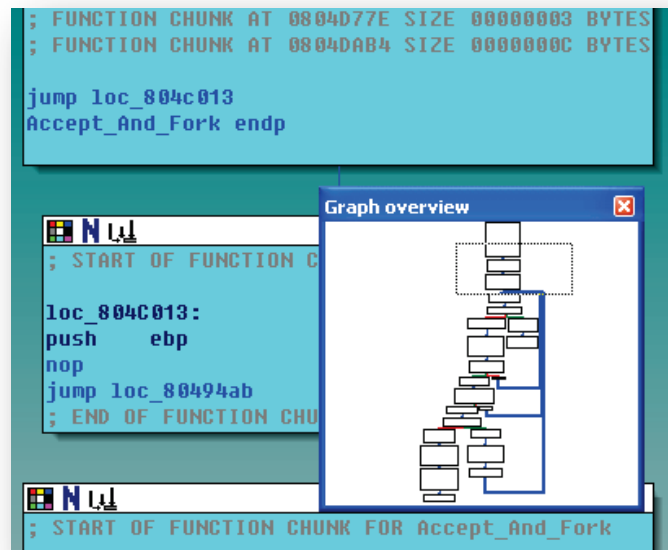
The 'Graph overview' window shows a control flow graph with a vertical sequence of nodes and a dashed box around the top two nodes.

This is the main function. Since the functions are all broken into pieces, I couldn't copy/paste the text of the whole function, neither make a screenshot of the entire function, so I will only write down the summary of what each functions do. But as mentioned before, you could open up the idb file supplied with this tutorial if you feel hard to follow up. The following is the summary of the main things this function does.

1. It checks if there is an Argument supplied. If there is no argument, it calls a function [NeedKeyFile]. NeedKeyFile function just prints out the string "Usage: ./MathIsHarD <keyfile>\n" and exits.
2. Then it calls [ReadKeyFile]. As the name implies, the function treats the supplied argument as a text file name and reads it's contents, then stores it in a global variable. If there are any issues, such as the keyfile can't be opened, then [NeedKeyFile] is called again.

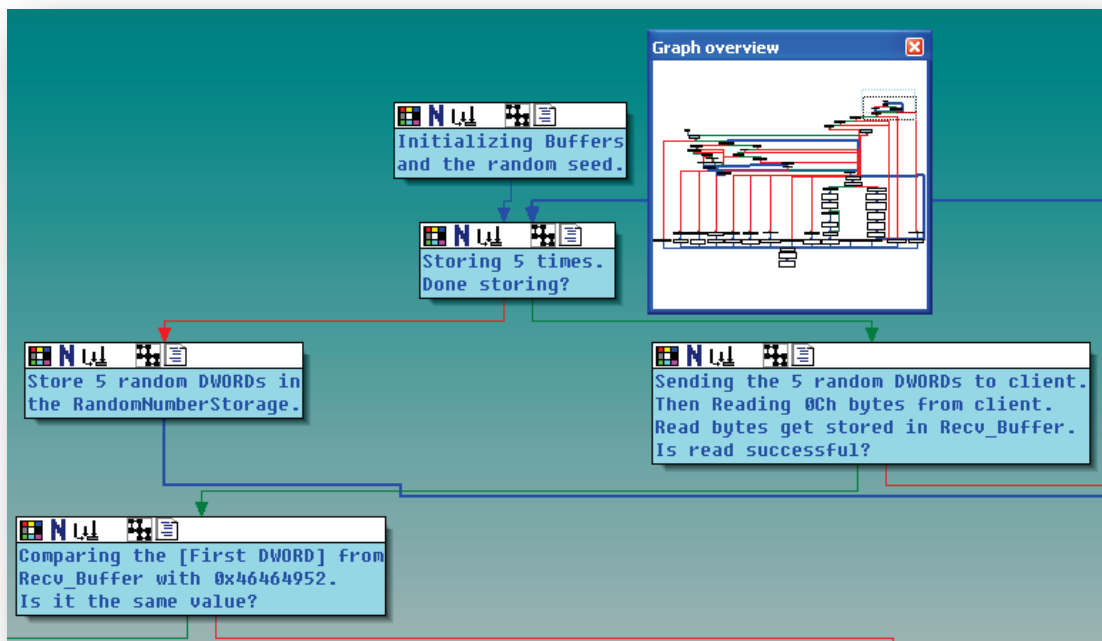
3. If all goes well, then the [InitializeNetwork] function initializes the network variables and calls network functions so it can later accept connections from clients. Nothing important in this function.
4. If the network is all set up properly, it calls the [AcceptAndFork] function to accept incoming connections.

Now let's see what this AcceptAndFork function does.



As you can see in the picture, there are 3 loops in this function. Right in the middle of the first loop, there is an `accept()` function which waits for connections from clients. If a client connects, but the `accept` function fails for some reason, then it loops back and waits for other connections. If a socket is successfully created, then it enters the second loop, where the program `forks` (Creates a copy of the process), and checks if the `fork()` function succeeded. If so, then it goes into the third loop. Otherwise, it loops back and waits for other connections. In the third loop, it checks if the current process executing the loop is the parent process, or the forked process. To explain `fork` for people coming from Windows, when `fork` is executed, both the parent process and the child process resume execution from the code right after `fork()`, with the child process having the exact same context of the parent process. One thing different is that the child process will have a return value of 0, while the parent process has a value of the child process ID. (Btw, If you ever get lost not knowing a meaning of a certain function, use [this site](#) as a reference.)

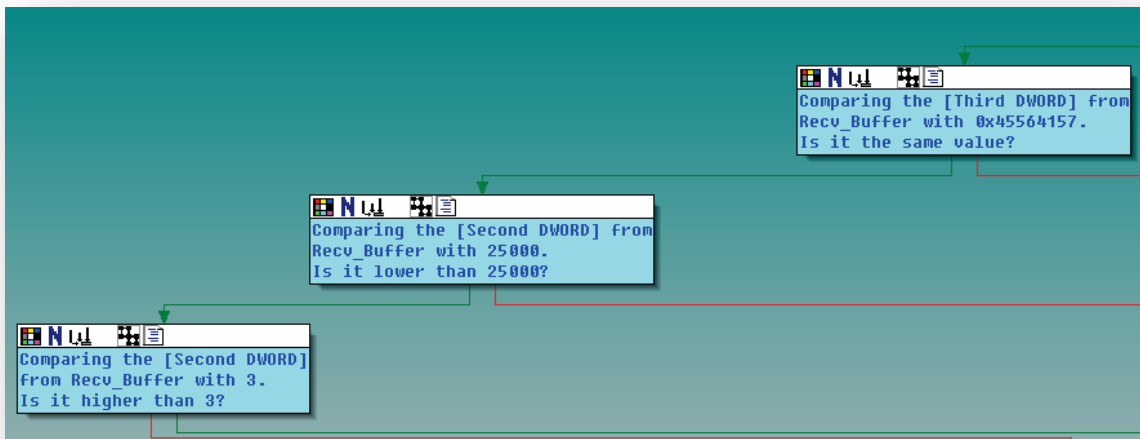
If the case is a parent process, then the execution will loop back and accept connections again. But in the client process case, the execution will fall to the left side, and call some kind of function that was supplied as an argument. After that function is executed, the program will close the connection and exit. This is only the child process being exited, so the parent will still be in the loop waiting for more connections. Let's see what that unknown function does.



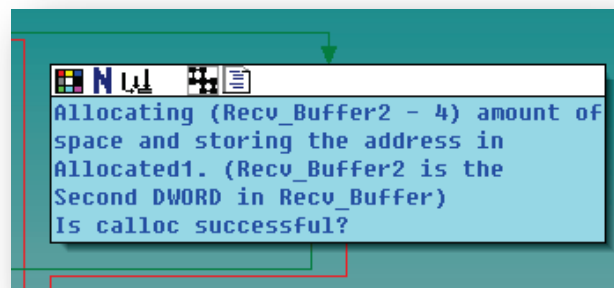
Although this function doesn't look that complicated at first sight, if you unhide all the grouped nodes, then the function will turn huge. I had to group most of the nodes little bits at a time to not get easily drowned in the ocean of basic blocks.

The part of the function shown above is where the function generates 5 random Dwords ranging from 0xC8 to 0x3E8 using a random seed generated by `srandomdev()`, and then sends those 5 Dwords to the client. This explains why we received 5 randomized Dwords that looked like Words expanded to Dwords.

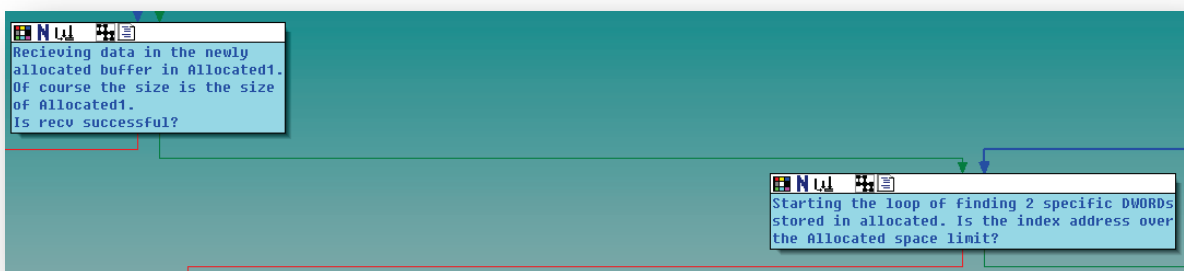
After that, it receives exactly 12 bytes, or more like 3 Dwords from the client, and compares the first Dword with the value 0x46464952. From here on now, most of the red arrow paths will lead to `(return -1)`, which is the path we don't want to follow. We want the first Dword to exactly match 0x46464952 so the function will go to the left path. Let's see the code that follows.



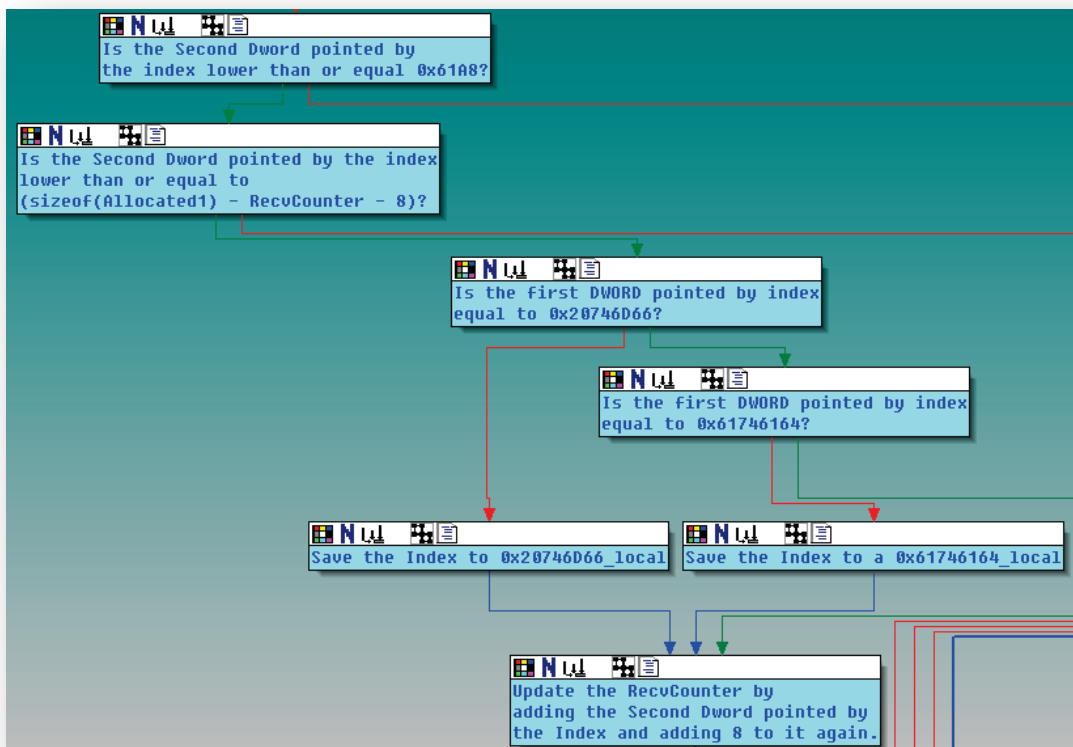
You can see that the 3rd Dword must equal 0x45564157, otherwise the function will exit and nothing will happen. The second Dword only has to be within a specific range ($3 < \text{Dword} < 25000$). With all those conditions satisfied, we can move on to the next code.



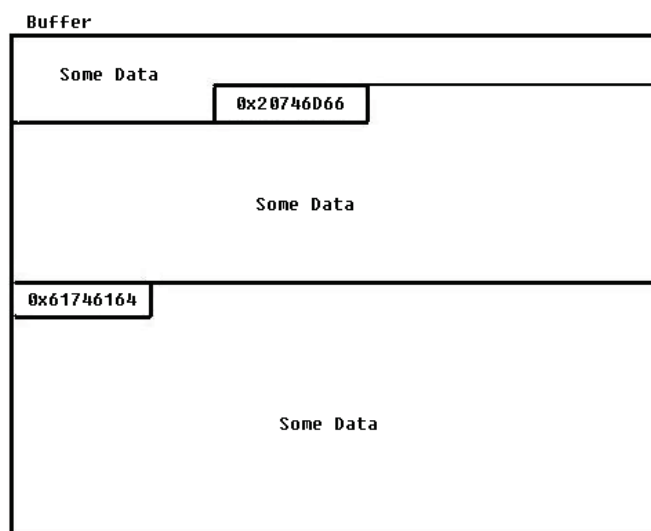
This indicates that the Second Dword in Recv_Buffer is the size of Recv_Buffer2. Allocated1 is a local variable. The rest should be self explanatory.



The program receives data from the client in the newly allocated buffer called 'Allocated1'. Then it searches for 2 specific Dwords from the received data.



This is the searching loop. The search starts from the beginning of Allocated1 which gets updated in each loop iteration. I will call this index 'IndexAddress'(dword ptr). The loop first checks if (IndexAddress[1] < 25000). Then it checks if (IndexAddress[1] < remaining bytes to search – 8). If the 2nd condition is satisfied, then the 1st condition is automatically satisfied, so the 1st condition could be considered redundant. Then it is searching for 2 specific Dwords 0x20746D66 and 0x61746164. The data layout would look something like this.

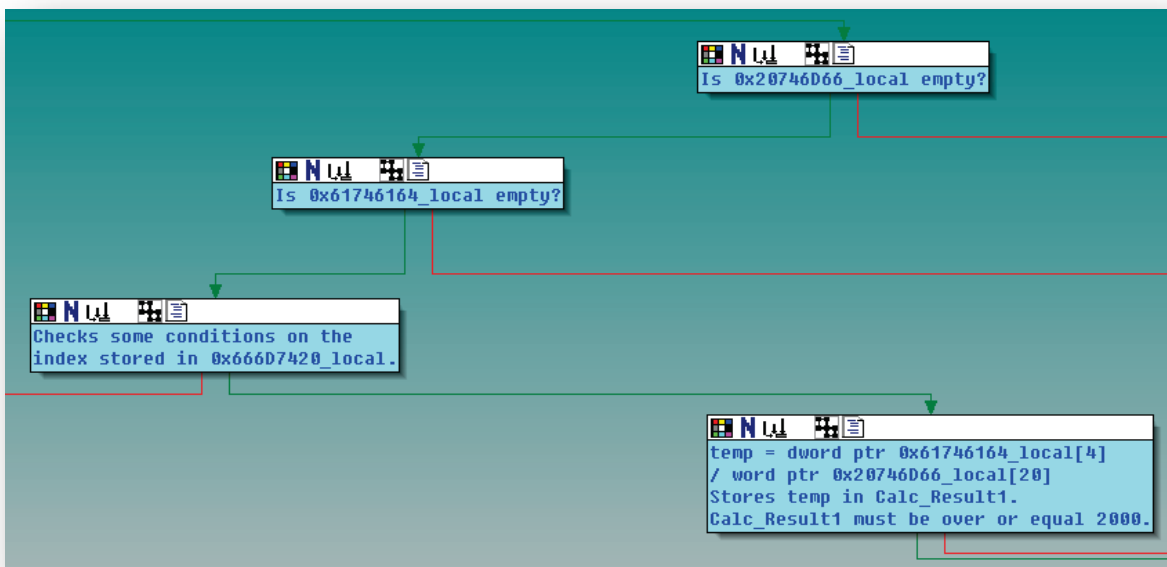


If it successfully finds those two Dwords, then it stores the address where the Dwords are located in 2 local variables for later use. After each loop operation is done, IndexAddress is updated by (IndexAddress[1]+8). The

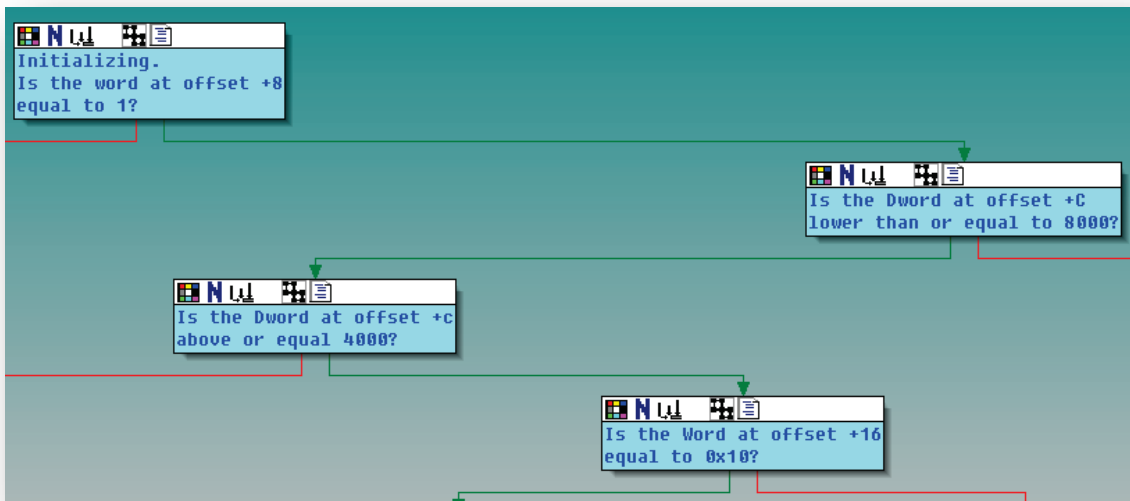
loop stops when IndexAddress passes or equals the buffer end. One thing we learned from this function is that the function treats AddressIndex as a structure pointer. The structure would look something like this:

Struct MagicStructure

```
{
    long MagicDword; // could be either 0x20746D66 or 0x61746164
    long size;
};
```



This is the code after the loop. It checks if the two locals that store addresses are empty, then calls a function called CheckCondition1. CheckCondition1 checks some conditions on the structure pointed by 0x20746D66_local. Just to refresh your memory, 0x20746D66_local points to the structure that has 0x20746D66 as the first Dword. I will call this structure 'SpecialStruct1', and the 0x61746164 struct 'SpecialStruct2' from now on. We already know that the second Dword is some kind of size, but if you analyze CheckCondition1, then you'll know that some other members exist in this structure.



Several conditions exist in this function.

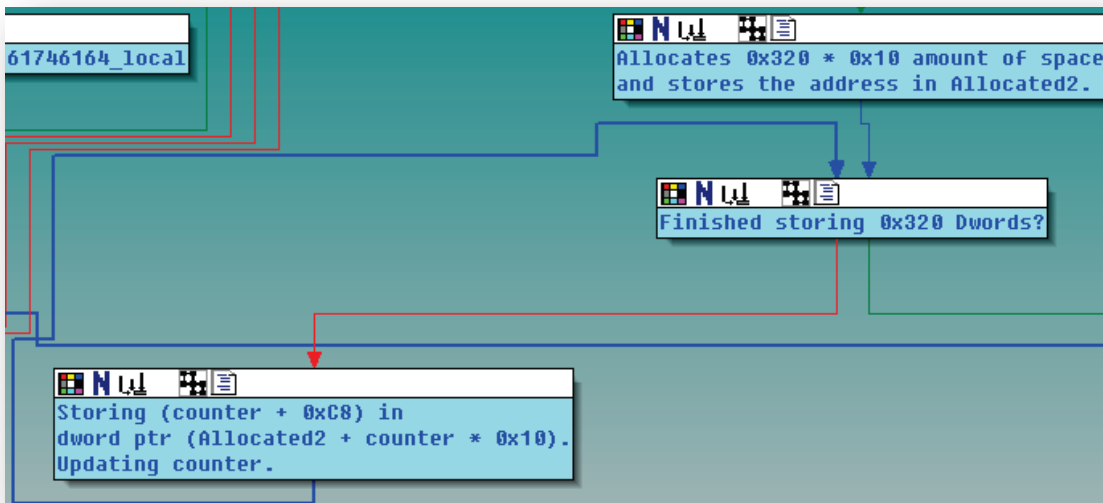
1. `word ptr [SpecialStruct1+8] == 1`
2. `4000 <= dword ptr [SpecialStruct1+12] <= 8000`
3. `word ptr [SpecialStruct1+22] == 0x10`

These conditions must be satisfied. We can expand Structure1 to have some new members now.

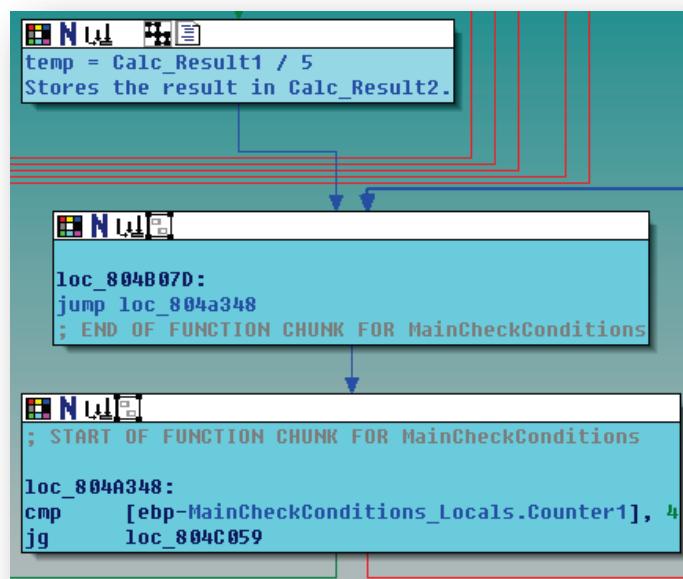
Struct MagicStructure

```
{
    long MagicDword; // could be either 0x20746D66 or 0x61746164
    long size;
    short const1 // must be 1
    short const2 // must be 0x10
    short const3 // must be between 4000 and 8000
};
```

After returning from CheckCondition1, it enters a function called Calculate1. Calculate1 simply calculates `(SpecialStruct2.size/word ptr [SpecialStruct+20])` and returns it. Then it stores it in a local 'Calc_Result1' for later use. The condition `(Calc_Result1 >= 2000)` must satisfy, so we have one more condition.



'counter' is the loop counter. It ranges from 0~0x320. Everything else should be self explanatory. 'Allocated2' will be used quite often, so it should be noted here.



Calc_Result1 is divided by 5 and stored in Calc_Result2. One thing peculiar here is that the code uses multiplication and shr operations for division instead of directly using idiv/div. This is because the mul operation takes less amount of cpu cycles to execute than idiv/div. If you've been reversing for a while, then you would easily recognize this type of compiler optimization.

Next, it enters a loop.

```

mov     eax, [ebp-MainCheckConditions_Locals.Calc_Result2]
pushf
stc
jb      loc_804D141
; END OF FUNCTION CHUNK FOR MainCheckConditions

; START OF FUNCTION CHUNK FOR MainCheckConditions

loc_804D141:
popf
imul   eax, [ebp-MainCheckConditions_Locals.Counter1]
mov     [ebp-MainCheckConditions_Locals.DWORD.Calc_Result3], eax
sub     esp, 0Ch
push   [ebp-MainCheckConditions_Locals.Calc_Result2]
push   [ebp-MainCheckConditions_Locals.DWORD.Calc_Result3]
jump   loc_804bb77
; END OF FUNCTION CHUNK FOR MainCheckConditions

; START OF FUNCTION CHUNK FOR MainCheckConditions

loc_804BB77:
push   320h
push   [ebp-MainCheckConditions_Locals.Allocated2]
lea    eax, [ebp-MainCheckConditions_Locals.SpecialStruct1]
push   eax
call   Fill_Allocated2
jmp    loc_8049501
; END OF FUNCTION CHUNK FOR MainCheckConditions

; START OF FUNCTION CHUNK FOR MainCheckConditions

loc_8049501:
add     esp, 20h
mov     ebx, [ebp-MainCheckConditions_Locals.Counter1]
sub     esp, 8
push   320h
push   [ebp-MainCheckConditions_Locals.Allocated2]
pushf
stc
jb      loc_804D729
; END OF FUNCTION CHUNK FOR MainCheckConditions

; START OF FUNCTION CHUNK FOR MainCheckConditions

loc_804D729:
popf
call   GetBiggest_FaddResult_Index
add     esp, 10h
mov     edx, eax
shl     edx, 4
mov     eax, [ebp-MainCheckConditions_Locals.Allocated2]
jmp    loc_804C045
; END OF FUNCTION CHUNK FOR MainCheckConditions

; START OF FUNCTION CHUNK FOR MainCheckConditions

loc_804C045:
mov     eax, [edx+eax]
mov     ds:FinalCompareBuffer[ebx*4], eax
lea    eax, [ebp-MainCheckConditions_Locals.Counter1]
inc     dword ptr [eax]
jmp    loc_804B07D

```



On each loop iteration, Calc_Result2 is multiplied with the loop counter, and stored to Calc_Result3. Two functions are called after that.

```
Fill_Allocated2(Allocated2,0x320,Calc_Result3,Calc_Result2);
```

```
index = GetBiggest_FaddResult_Index(0x320,Allocated2);
```

And finally, index(Dword) is stored in a 20 byte buffer(which I named FinalCompareBuffer) 5 times in each loop iteration. That buffer is going to be later compared with the 5 random Dwords(RandomNumberStorage) that were initially generated and sent to the client. Do you get the picture now? :P

If they both turn out as the same 20 bytes, the keyfile contents will be sent to the client. Now we have to figure out how we're going to have complete control over the contents of FinalCompareBuffer. This will require analysis of both functions.

The first function is kind of large and a bit confusing, because it uses a lot of fpu instructions. I actually made a C version of the code to better understand the big picture instead of the small details. Instead of going through every aspect of the assembly instructions, I will present here the C code for better readability and understanding of the code logic.

```
int GetBiggest_FaddResult_Index(Allocated2_Struct* Allocated2Array);
void FillAllocated2(SpecialStructPointers SSP,Allocated2_Struct* Allocated2Array,DWORD Calc3,DWORD Calc2);
double CalcHash(Allocated2_Struct Allocated2,SpecialStructPointers SSP,DWORD Calc3,DWORD Calc2);
SpecialStructPointers* AllocateAndCopy(SpecialStructPointers SSP);
void FillSS2(SpecialStructPointers* Allocated3,DWORD Allocated2_Counter_Plus_C8);
int ReturnIndex(SpecialStructPointers* Allocated3);

struct SpecialStruct1
{
    DWORD   StructSize;
    WORD    Counter_Updater;
    DWORD   fdiv_divisor;
    WORD    Calc1_Div_Const;
};

struct SpecialStruct2
{
    DWORD   ArraySize;
    WORD   Array[?];
    //Array is a variable size array with the size of ArraySize
};

struct SpecialStructPointers
{
    SpecialStruct1* SpecialStruct1Pointer;
    SpecialStruct2* SpecialStruct2Pointer;
};

struct Allocated2_Struct
{
    int Counter_Plus_C8;
    double fadd_result;
    SpecialStructPointers* Allocated3;
};

extern SpecialStruct1 SS1;           // We have complete control over SpecialStruct1 Members
extern SpecialStruct2 SS2;         // We also have complete control over SpecialStruct2
DWORD Calc1 = SS2.ArraySize / SS1.Calc1_Div_Const;
// Some conditions exist.
// 1 : Calc1 >= 2000
// 2 : SS2.Arraysize <= 25000
```

```

// 3 : 4000 <= SS1.fdiv_divisor <= 8000

void main()
{
    int MainCounter = 0;
    DWORD FinalCompareBuffer[5];
    DWORD Calc2 = Calc1 / 20;
    Allocated2_Struct Allocated2Array[0x320];
    SpecialStructPointers SSP;
    SSP.SpecialStruct1Pointer = &SS1;
    SSP.SpecialStruct2Pointer = &SS2;

    for(int i=0; i<0x320; i++)
    {
        Allocated2Array[i].Counter_Plus_C8 = i+0xC8;
    }

    while(MainCounter < 5)
    {
        int Counter1 = 0;
        int index;
        DWORD Calc3 = Calc2 * MainCounter;

        FillAllocated2(SSP, Allocated2Array, Calc3, Calc2);
        index = GetBiggest_FaddResult_Index(Allocated2Array);

        FinalCompareBuffer[MainCounter] =
            Allocated2Array[index].Counter_Plus_C8;
    }

    // At this point, FinalCompareBuffer is compared with
    // RandomNumberStorage, and if both are the same, then the
    // keyfile will be sent.
}

// Returns the index of the biggest Fadd_Result among the 0x320 stored
// Fadd_Results.
int GetBiggest_FaddResult_Index(Allocated2_Struct* Allocated2Array)
{
    int counter = 0, index = 0;
    double biggest_fadd_result = Allocated2Array[0].fadd_result;

    while(counter < 0x320)
    {
        Allocated2_Struct Allocated2 = Allocated2Array[counter];
        if(Allocated2.fadd_result > biggest_fadd_result)
        {
            biggest_fadd_result = Allocated2.fadd_result;
            index = counter;
        }
        counter++;
    }
    return index;
}

// Fill the Allocated2 Array.
void FillAllocated2(SpecialStructPointers SSP, Allocated2_Struct* Allocated2Array, DWORD Calc3, DWORD
Calc2)
{
    int counter = 0;
    double current_fadd_result = 0;
    double smallest_fadd_result = 0;

    while(counter < 0x320)
    {
        current_fadd_result =
            CalcHash(&Allocated2Array[counter], SSP, Calc3, Calc2);
    }
}

```




```

// Calculating the fadd_result member of Allocated2Array.
Allocated2Array[counter].fadd_result = current_fadd_result;
// Storing 0x320 fadd_result member values.

if(counter = 0)
    smallest_fadd_result = current_fadd_result;
if(smallest_fadd_result > current_fadd_result)
    smallest_fadd_result = current_fadd_result;

    counter++;
}

counter = 0;
while(counter < 0x320)
{
    Allocated2Array[counter].fadd_result -= smallest_fadd_result;
    // This doesn't really effect the final results.
    counter++;
}
}

// Calculate Fadd_Result values to be stored in Allocated2.
double CalcHash(Allocated2_Struct& Allocated2,SpecialStructPointers SSP,DWORD Calc3,DWORD Calc2)
{
    SpecialStructPointers* Allocated3;
    double fadd_result = 0; // QWORD
    int Index = 0;

    if(Calc2 == 0) // Calc2 is obviously not 0
        Calc2 = Calc1;

    if(Allocated2->Allocated3 == 0)
    {
        Allocated3 = AllocateAndCopy(SSP);
        // Allocating, and filling in data of Allocated3.
        Allocated2->Allocated3 = Allocated3;
        // Saving the Allocated3 member of Allocated2Array.
        FillSS2(Allocated3,Allocated2->Counter_Plus_C8);
        // Filling in the newly allocated SpecialStruct2 Array
        // belonging to Allocated3.
    }

    SpecialStruct1 SS1 = *(Allocated3->SpecialStruct1Pointer);
    SpecialStruct2 Current_SS2 = *(Allocated3->SpecialStruct2Pointer);
    SpecialStruct2 Original_SS2 = SSP.SpecialStruct2Pointer;
    Index = ReturnIndex(Allocated3);
    // Returns a strange index. 99.99% of the time, this index is a
    // non-zero value.

    if(Index != 0)
    {
        int counter;
        DWORD imul_result = 0;

        while(counter < Calc2)
        {
            imul_result = (DWORD)Current_SS2.Array[counter] *
(DWORD)Original_SS2.Array[Calc3 + counter];
            // takes only eax from edx:eax, but edx is always 0 anyway.
            fadd_result += (double)imul_result;
            // Multiplies the user supplied input with the currently generated
discretional sin wave, and adds all the values.
            // Looks like it's calculating some integral value.
            counter += SS1.Counter_Updater;
            // This counter is later on revealed as 1.
        }
        if(fadd_result < 0)
            fadd_result = fadd_result * (double)-1;
            // making fadd_result a positive value.
    }
}

```



```

else
{
    fadd_result = 0;
}

return fadd_result;
// The returned values will be stored in the 0x320 fadd_result members of Allocated2.
// Since this value will directly affect the FinalCompareBuffer, we must have complete
control over this value.
}

// Allocates some space for Allocated3. Allocate space for SpecialStruct1, SpecialStruct2, copy the
contents, and store the pointers in Allocated3.
SpecialStructPointers* AllocateAndCopy(SpecialStructPointers SSP)
{
    SpecialStructPointers* Allocated3 = calloc(1,8);
    Allocated3->SpecialStruct1Pointer = calloc(1,0x18);
    Allocated3->SpecialStruct2Pointer = calloc(1,SSP.SpecialStruct2Pointer->ArraySize + 8);
    // Storing the pointers of the newly allocated SpecialStruct1, SpecialStruct2
    memcpy(Allocated3->SpecialStruct1Pointer,SSP.SpecialStruct1Pointer,0x18);
    memcpy(Allocated3-
>SpecialStruct2Pointer,SSP.SpecialStruct2Pointer,SSP.SpecialStruct2Pointer->ArraySize + 8);
    // Copying the original SpecialStruct1, SpecialStruct2 data into
    // the allocated one.
    return Allocated3;    // This will later be stored in Allocated2.
}

// Fill the SpecialStruct2 Array.
void FillSS2(SpecialStructPointers* Allocated3,DWORD Allocated2_Counter_Plus_C8)
{
    int counter = 0;
    SpecialStruct1 SS1 = *(Allocated3->SpecialStruct1Pointer);
    SpecialStruct2 SS2 = *(Allocated3->SpecialStruct2Pointer);
    double fmul_result = (double)Allocated2_Counter_Plus_C8 * 6.283185307179586; // = 2 * PI

    while(counter < Calc1);
    {
        double fdiv_result = (double)counter / (double)SS1.fdiv_divisor;
        double fsin_result = sin(fmul_result * fdiv_result);
        SS2.Array[counter] = (int)(fsin_result * (double)32767);
        // Creating a discretionary sin wave and storing it into the
        // allocated SpecialStruct2.
        //The range of a signed short value = -32767 ~ 32768, and considering
        //the result of the sin operation is between -1/+1,
        //this operation is to generate an integer that fits in a WORD.
        counter += SS1.Counter_Updater;
    }
    return;
}

// Return the index of the 3rd value with the same sign as the initial value. Dunno the exact
meaning of this function. ./
int ReturnIndex(SpecialStructPointers* Allocated3)
{
    int counter = 1;
    int old_counter = 0;
    int SignEqualCounter = 0;
    SpecialStruct1 SS1 = *(Allocated3->SpecialStruct1Pointer);
    SpecialStruct2 SS2 = *(Allocated3->SpecialStruct2Pointer);

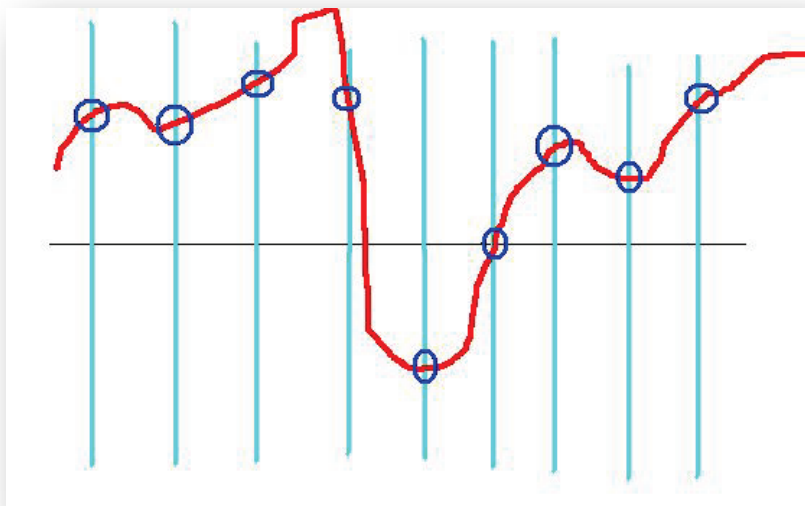
    while(counter < Calc1)
    {
        if(SS2.Array[counter] & 0x8000 == SS2.Array[old_counter] & 0x8000)
        // Check if both words have the same sign.
        {
            SignEqualCounter++;
            old_counter = counter;
        }
        if(SignEqualCounter == 2)
            break;
    }
}

```

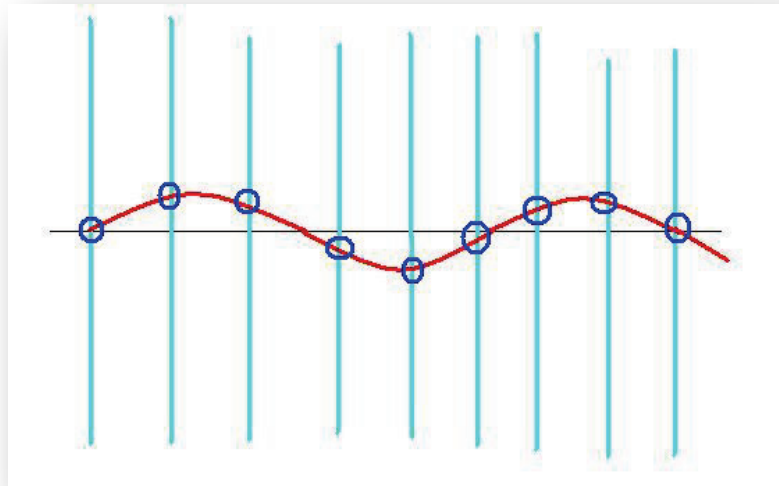
```
counter += SS1.Counter_Updater;
// counter started out as 1, and old_counter as 0, so this indicates
// that Counter_Updater must be 1.
// Otherwise, there will be no value to compare to.
}
if(counter != Calc1)
    return counter;
return 0;
}
```

This code is obviously not compilable (is there even such word as compilable? :P), but you could understand what it's trying to do by reading the code. If you want to later on analyze the assembly code, then always have the IA-32 Architecture Software Developer's Manual by your side so you could easily decipher the somewhat less used fpu instructions quickly.

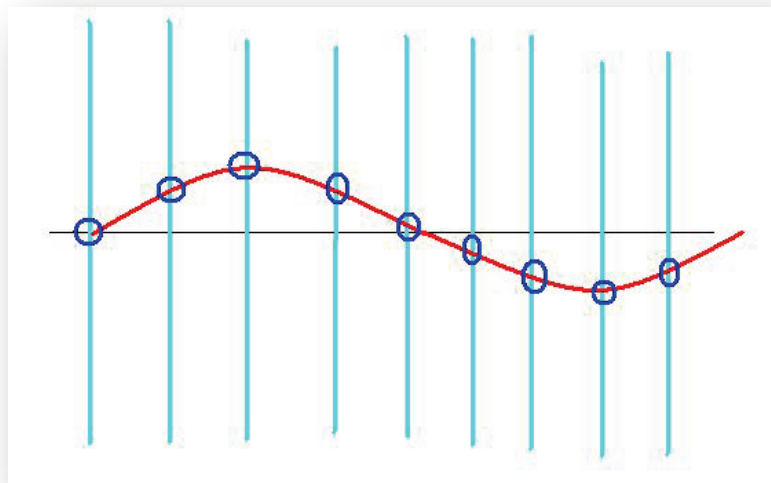
The two pictures below illustrate an example of a user supplied input, and two of the generated discretional sin wave out of the 0x320 sin waves.



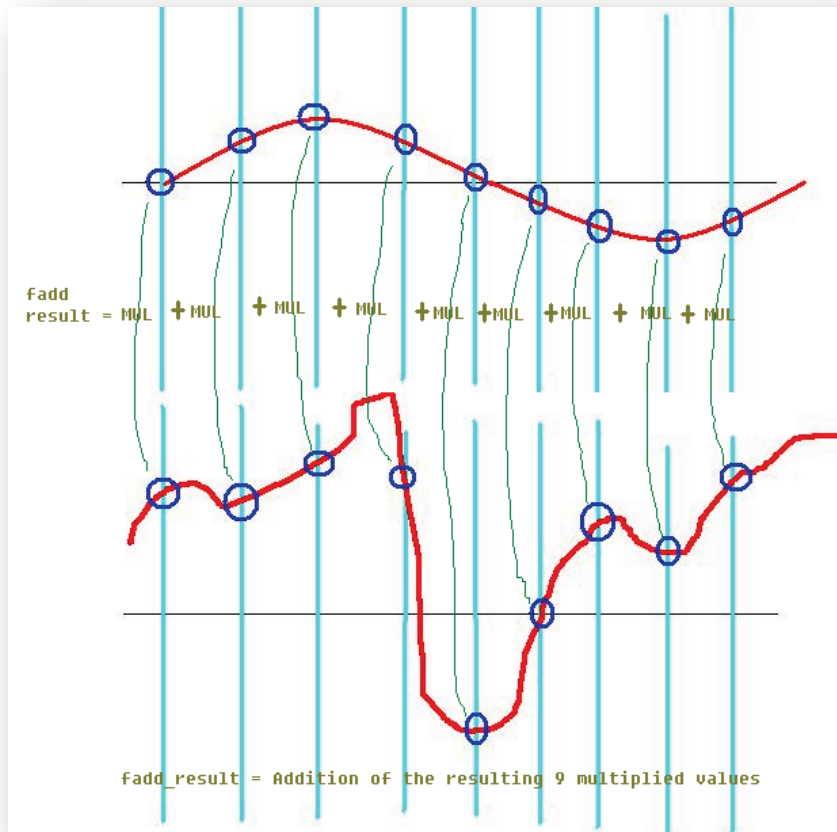
[User supplied Data]



[Generated sin wave 1 : Const = 0x1D8]



[Generated sin wave 2 : Const = 0x1C8]



[How fadd_result is calculated]

Let's try to ignore the crappy artwork. :D Now if you understood the C code, you would already know that fadd_result is calculated by multiplying the user supplied graph with one of the generated sin waves, then adding up all the discretionary values of the resulting graph like the picture above. The value that gets stored in FinalCompareBuffer is the constant(index+0xC8) associated with the sin wave with the highest fadd_result.

For instance, if we receive a random number 0x1C8(the first random Dword) from the server, we want to supply a user generated graph that would make the fadd_result of sin wave2(from above pictures) the biggest value out of all 0x320 fadd_results. We can only supply the user input once, and all the 0x320 sin waves will be multiplied to that input, and added to create 0x320 fadd_results. So the question is, what kind of input would make the fadd_result of the 0x1C8 sin wave the biggest, and the other 0x31F fadd_results always smaller?

The answer would be a sin wave that has the exact same cycle of the 0x1C8 sin wave. While a user supplied sin wave(with the same cycle of the 2nd 0x1C8 wave) multiplied with the 2nd wave would result in a wave with all positive values, the same user supplied sin wave multiplied with the 1st 0x1D8 wave would result in a wave with positive and negative values, and the negative values would contribute in making the fadd_result smaller. So what we want to do is supply 5 sin waves that have exactly the same cycles as the 5 sin waves corresponding to the 5 random(0xC8~0x3E8) Dwords we received from the server. That would be the last condition and if that condition is satisfied, then FinalCompareBuffer will have the exact same content as RandomNumberStorage and we will have the keyfile contents in our hands. :)



The following code attack.c does exactly the thing just described.

```
#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <math.h>

#define BUFSIZE 100
#define PI 3.14159265

void DieWithError(char* errorMessage);
// You would notice I've been studying with this book :
// "The pocket guide to TCP/IP sockets" :P

typedef struct MagicBytes_{
    long magic1;           // Must be 0x46464952
    long size;            // sizeof(SpecialStruct1) + sizeof(SpecialStruct2)+4.
                        // 4 is because the size is later subtracted with 4 when
allocated.
    long magic2;          // Must be 0x45564157
}MagicBytes;

typedef struct SpecialStruct1_{
    long magic;           // Must be 0x20746d66
    long structsize;     // Must be 16
    short const1;        // Must be 1
    short CounterUpdater; // Must be 1
    long FdivDividor;    // Must be between 4000 ~ 8000
    long unused;
    short Calc1DivConst; // Must be 2 or higher
    short const2;        // Must be 0x10
}SpecialStruct1;

typedef struct SpecialStruct2_{
    long magic;           // 0x61746164
    long structsize;     // size of the Array
    short SinWave[2500]; // this is where the 5 sin waves go
}SpecialStruct2;

void DieWithError(char* errorMessage)
{
    perror(errorMessage);
    exit(1);
}

int main(int argc, char* argv[])
{
    int sock;
    struct sockaddr_in ServAddr;
    unsigned short ServPort;
    char* servIP;
    char StrRecv[BUFSIZE];

    int bytesRcvd, totalBytesRcvd;
    int mulconst, i, j;
    MagicBytes MB;
    SpecialStruct1 SS1;
    SpecialStruct2 SS2;

    servIP = "192.168.1.131";
    ServPort = 2600;

    if((sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
        DieWithError("socket() failed");
```



```

memset(&ServAddr,0,sizeof(ServAddr));
ServAddr.sin_family = AF_INET;
ServAddr.sin_addr.s_addr = inet_addr(servIP);
ServAddr.sin_port = htons(ServPort);

if(connect(sock,(struct sockaddr*)&ServAddr,sizeof(ServAddr)) < 0)
    DieWithError("connect() failed");

if((bytesRcvd = recv(sock,StrRecv,BUFSIZE-1,0)) <= 0)
    DieWithError("recv failed 1");
StrRecv[bytesRcvd] = '\0';
// Receiving the 5 random Dwords to create the sin waves

MB.magic1 = 0x46464952;
MB.size = sizeof(SpecialStruct1) + sizeof(SpecialStruct2)+4;
MB.magic2 = 0x45564157;
// filling MB structure with the appropriate values

SS1.magic = 0x20746d66;
SS1.structsize = 16;
SS1.const1 = 1;
SS1.CounterUpdater = 1;
SS1.FdivDividor = 4000;
// Could be any value between 4000 ~ 8000
SS1.Calc1DivConst =2;
// Must be 2, cause the program reads/stores the Array values in WORD size.
SS1.const2 = 0x10;

SS2.magic = 0x61746164;
SS2.structsize = 5000;
// Must be over 4000, cause SS2.structsize/SS1.Calc1DivConst must be over 2000.
// The lower the value, the faster the calculation.
for(i=0; i<5; i++){
    for(j=0; j<500; j++){
        mulconst = *((long*)StrRecv+i);
        SS2.SinWave[i*500+j] = 32767 * sin(2 * PI * mulconst * j / SS1.FdivDividor);
        // Generating 5 kinds of discretional sinwaves,
        // each corresponding
        // to each of the received 5 random Dwords.
        // 32767 exists so the result would be an integer that
        // fits in a WORD.
    }
}

if(send(sock,(char*)&MB,sizeof(MagicBytes),0) != sizeof(MagicBytes))
    DieWithError("send() failed 1\n");
if(send(sock,(char*)&SS1,sizeof(SpecialStruct1),0) != sizeof(SpecialStruct1))
    DieWithError("send() failed 2\n");
if(send(sock,(char*)&SS2,sizeof(SpecialStruct2),0) != sizeof(SpecialStruct2))
    DieWithError("send() failed 3\n");
// Sending the structures along with the sin waves

if((bytesRcvd = recv(sock,StrRecv,BUFSIZE-1,0)) <= 0)
    DieWithError("recv failed 2");
printf(StrRecv);
// If all goes well, then the keyfile will be printed out

close(sock);
return 0;
}

```




And the result.

```
externalist@Externalist:~/Documents/Reverse Engineering/Binary500$ ./attack  
ARTeam Rocks!!! :)
```

1.5 CONCLUSIONS

This concludes the tutorial on reversing this year's DEFCON Binary500 challenge. Personally, I think all of the last year and this year's binary challenge(except Binary300. Will someone please give me a reason for inet_aton??) were excellent, and I'm looking forward for next year's challenge. :P I must say I've learned lots about reversing on non-windows platforms from reversing those binaries, and I would give a 5 out of 5 for this year's Binary500. :P

Later on, when the walkthroughs came up, I read that the user input was supposed to be a wave file, and indeed when I did a quick search on the constants 0x46464952/0x45564157, google pointed to many references to source codes related to wave files. This taught me an important lesson. 'Always search for constants that look suspicious before doing anything serious because it can provide some important clues that reduce the amount of work and time spent'.

To sum up the main features of Binary500, first, Binary500 uses an obfuscation scheme that splits up the file into many pieces to make static analysis more challenging, while it scatters a lot of trash bytes everywhere in the binary. Second, it sends some data to the client and expects the client to send a mass of data that meets certain conditions. The client program must deduce the correct form of data to send by using the initial data sent from the server. Third, it does some mathematical operations, hence, requires the reverser to be capable of accurately analyzing math functions.

I hope this tutorial was useful for some readers and sorry if my English was a bit sloppy or hard to read, it's not my mother language btw. ./ Most of the non-Windows platform reversers would have little or no problems reading this tutorial, but if you're from Windows and feel like you can't even understand 20% of this tutorial, don't worry. Just keep studying, and time will solve the problems for you. :P

Anyway, thanks for taking your time reading this tutorial and Happy Reversing! :)



1.6 REFERENCES

- <http://www.nixdoc.net>
- <http://www.intel.com/products/processor/manuals/>
- <http://www.hex-rays.com/idapro/idadoc/707.htm>
- http://www.openrce.org/articles/full_view/11
- http://www.rexx.com/~dkuhlman/python_101/python_101.html

1.7 GREETINGS

- To ARTeam members for creating awesome tutorials and always having the mind of sharing knowledge. I wouldn't have gone this far if it weren't for ARTeam.
- To Teddy Rogers for maintaining a huge database of Windows reversing tools, documents, information. You are the man!! :D
- To Lena and Tiga for creating the well known tutorial series. I simply would have dropped out quickly if it weren't for those video tutorial series. :)
- To deLTA for being my first overseas friend, and for creating/maintaining the great [CRCETL](#) (and also the recently founded [CISTL](#)).
- To all beistlab members and especially beist for inviting me to their team in the DEFCON CTF Pre-quals, and also for kicking ass all the time. :D
- To graylynx for guiding me to the security world.
- To phin3h45 for being a good friend.
- To all team members who once belonged to a now unexisting team formerly known as unkn0wn. I hope you guys are all doing well.
- To all my friends on my nateon friend list (You know who you are! :)

[In the Supplements folder "0.1 Externalist" you can find also:

- Analysis, idb ida file
- Binary, the original file
- Pics, pictures of this tutorial
- Source, sources used for this tutorial]



2 HANDY PRIMER ON LINUX REVERSING BY GUNTHER

2.1 FOREWORDS

The time has come to write a tutorial on Reversing Engineering on Linux. Reversing Engineering on Linux will surely be welcome among the rapidly growing community of Linux users. Important applications which appeared on Linux had slowly started to be protected with important tricks and some tools appeared which can handle them.

After a conversation with Shub, I have decided to write a primer on this, examining which tools we can use to disassemble the Linux applications, how to approach to them and what generally we can do. The tutorial will cover different issues:

- What instruments we have and what to use and customize them.
- Practical examples of real applications

I also included a long list of references and further readings, as usual. The applications which I have used are selected based on two criteria:

- being educative for my purposes
- being already cracked by someone else, so as to not create problems on my own

I hope that this could begin a new chapter in the ongoing ARTeam series of Reverse Engineering tutorials. Today's topic will go over Reverse Engineering on Linux. This topic has been hardly touched upon, thus giving me room to add some information to the reader.

This tutorial is for anyone running Linux with gcc and who knows a bit of C. This tutorial does not claim to be complete, exclusivity and is geared to beginners.

2.2 ABSTRACT

In this tutorial we are going to learn the ELF file format and learn how to conduct reverse engineering on an ELF binary.

There has been a trend for growing commercial applications under Linux. Thus, there is a need for protective mechanisms in the new environment. In this primer, I will try to fill in required information about tools which Reverse Engineers used in Linux.

Let's start getting our hands dirty...

2.3 TARGET

In this primer, I will use the following crackme to conduct the tutorial.

Here is the link to the crackme:

http://www.crackmes.de/users/damo2k/damos_crackme_1_for_linux/

2.4 EXAMINING OUR TARGET

2.4.1 STARTING

Before we try to break the program, it is often necessary to analyze it. For this purpose, we can use the following programs to do so:

file - Displays information about the file type, architecture, the use of shared libraries.

```
root@...:~/Desktop
File Edit View Terminal Tabs Help
[root@... Desktop]# file crkme1-linux32
crkme1-linux32: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.4.1, not stripped
[root@... Desktop]#
```

Figure 1 -file command displaying information of the target.

nm - List information about symbolic references (symbols) in object files.

```
File Edit View Terminal Tabs Help
[root@ Desktop]# nm crkme1-linux32
08049740 D __DYNAMIC
0804980c D __GLOBAL_OFFSET_TABLE__
080486f0 R __IO_stdin_used
w __Jv_RegisterClasses
08049730 d __CTOR_END__
0804972c d __CTOR_LIST__
08049738 d __DTOR_END__
08049734 d __DTOR_LIST__
08048728 r __FRAME_END__
0804973c d __JCR_END__
0804973c d __JCR_LIST__
08049830 A __bss_start__
08049824 D __data_start__
080486a0 t __do_global_ctors_aux
08048350 t __do_global_dtors_aux
08049828 D __dso_handle
0804972c A __fini_array_end__
0804972c A __fini_array_start__
w __gmon_start__
08048690 T __i686.get_pc_thunk.bx
0804972c A __init_array_end__
0804972c A __init_array_start__
08048640 T __libc_csu_fini
```

Figure 2 - nm command listing about symbolic references.

size - Displays of all sections' sizes and total size of the application.

```
File Edit View Terminal Tabs Help
[root@ Desktop]# size crkme1-linux32
text data bss dec hex filename
1543 260 4 1807 70f crkme1-linux32
[root@ Desktop]#
```

Figure 3 - size command displaying all sizes of the target.

strings - Outputs all text strings contained in the file. It's very helpful when seeking registration codes stored in the clear.

```
File Edit View Terminal Tabs Help
[root@ ~ Desktop]# strings crkme1-linux32
/lib/ld-linux.so.2
_Jv_RegisterClasses
_gmon_start
libc.so.6
printf
_IO_stdin_used
__libc_start_main
strlen
GLIBC 2.0
PTRh@
8`~1
[^]
Bad Serial!
Good Serial!
Usage: %s <serial>
[root@ ~ Desktop]#
```

Figure 4 - strings command displaying all text strings found in the target.

ldd - Shows dependency programs from dynamic libraries. Programmers benefit more than crackers.

```
File Edit View Terminal Tabs Help
[root@ ~ Desktop]# ldd crkme1-linux32
linux-gate.so.1 => (0x00b09000)
libc.so.6 => /lib/libc.so.6 (0x43b3a000)
/lib/ld-linux.so.2 (0x43169000)
[root@ ~ Desktop]#
```

Figure 5 - ldd command displaying dependency of the target.

readelf - Explore file. Displays lots more information than the command **file**. Shows file type, architecture, point of entry and other equally important data for reverse engineers and crackers.

```
File Edit View Terminal Tabs Help
[root@kali ~]# readelf -a crkme1-linux32 | more
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                               ELF32
  Data:                                   2's complement, little endian
  Version:                               1 (current)
  OS/ABI:                                UNIX - System V
  ABI Version:                           0
  Type:                                  EXEC (Executable file)
  Machine:                                Intel 80386
  Version:                                0x1
  Entry point address:                   0x8048300
  Start of program headers:              52 (bytes into file)
  Start of section headers:              7716 (bytes into file)
  Flags:                                  0x0
  Size of this header:                   52 (bytes)
  Size of program headers:               32 (bytes)
  Number of program headers:              7
  Size of section headers:               40 (bytes)
  Number of section headers:              34
  Section header string table index:     31

Section Headers:
  [Nr] Name              Type              Addr             Off             Size             ES Flg Lk  Inf Al
```

Figure 6 - readelf command displaying the all the information of the target.

2.4.2 INITIAL ANALYSIS

Where should we start from? We need to collect some basic information about the object. From the output given by the **file** command, it has indicated that the object is an ELF executable compiled for Intel x86 architecture (Intel 80386, 32-bit, LSB – least significant byte).

```
file crkme1-linux32
```

It also reveals that the object has been linked dynamically and not stripped. If the ELF header of the binary was corrupted in any way, the file command would report that as well.

2.4.3 ELF FILE FORMAT

ELF stands for Executable and Linking Format and the file format used (with some exceptions) on the Linux system for relocatable, executable, and shared binary files that do not need any other hardware than CPU to run, unlike Java and .NET.

- Relocatable objects (*.o) are linked with other objects in order to build an executable file or a shared library – these are produced by compilers and assemblers.
- Executable objects are files that are ready to be executed, already relocated and with symbols resolved (excluding those that refer to shared libraries, resolved at runtime).
- Shared objects (*.so) contain code and data which can be used for linking in two different ways. They can be linked with relocatable or shared objects to produce another object. They can also be linked with executable code by the system dynamic linker/loader to create a process image in memory.

The basic component of an ELF file is its header (see [Figure 7](#)). The header is located at the beginning of the file and serves as a sort of a map of its remaining parts. It contains information such as the location of the program header and section header relative to the beginning of the file. The memory location where control is to be passed to when the program is launched (the so-called entrypoint), as well as some platform-independent information that determines how the file content is to be interpreted.

To keep the ELF format as flexible as possible, 2 parallel views were introduced: linking view and execution view (see [Figure 8](#)). When the object is being built, the compiler, assembler or linker treats the ELF file as a collection of sections described by the section header with an optional program header (see [Figure 9](#)). However, the system linker/loader treats the file as a collection of segments described by the program header with an optional section header. The link view is not required for running executable code.

Browsing and examining the internals of ELF files can be accomplished with the help of the objdump program, elfsh utility, ht program.

```

File Edit View Terminal Tabs Help
File Edit Windows Help 12:46 14.07.2008
[x] /root/Desktop/crkme1-linux32 2
* ELF header at offset 0x00000000
ident
magic          7f 45 4c 46 = ?ELF
class          01 (32-bit objects)
data           01 (LSB encoding)
version        01
OS ABI         00 (System V)
version        00
reserved       00 00 00 00 00 00 00
type           0002 (executable file)
machine        0003 (Intel 80386)
version        00000001
entrypoint     08048300
program header offset 00000034
section header offset 00001e24
flags          00000000
elf header size 0034
program header entry size 0020
program header count 0007
section header entry size 0028
help save open edit mode viewin. quit

```

Figure 7 - Elf header view with ht editor.

```

File Edit View Terminal Tabs Help
File Edit Windows Help 12:46 14.07.2008
[x] /root/Desktop/crkme1-linux32
* ELF program headers at offset 00000034
[-] entry 0 (phdr)
  type                00000006 phdr)
  offset              00000034
  virtual address     08048034
  physical address    08048034
  in file size        000000e0
  in memory size      000000e0
  flags               00000005 details
  alignment           00000004
[-] entry 1 (interp)
  type                00000003 interp)
  offset              00000114
  virtual address     08048114
  physical address    08048114
  in file size        00000013
  in memory size      00000013
  flags               00000004 details
  alignment           00000001
[+] entry 2 (load)
1help 2save 3open 4edit 5 6mode 7 8 9viewin.0quit

```

Figure 8 - ELF program header view with ht editor.

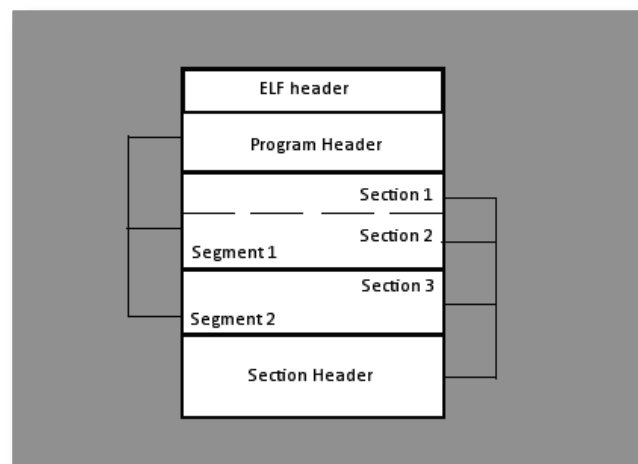


Figure 9 - ELF format outline.

2.4.4 DISASSEMBLER AND HEX EDITING TOOLS

The following are some of the Disassembler and Hex editing tools which you can use:

Disassembler:

- **Objdump** - More than just a disassembler. a program for displaying various information about object files.
 - ✓ Section file (-x)
 - ✓ Disassemble file (-D).
 - ✓ It displays the content of sections in hexadecimal notation (-s)



- **Dissy** - a disassembler for Linux and UNIX which supports multiple architectures and allows easy navigation through the code. It is implemented as a graphical frontend to objdump. It allows fast navigation through the disassembled code and easy searching for addresses and symbols.
- **Lida (Linux Interactive DisAssembler)** - Quite interesting disassembler, which includes modules like cryptanalysis, the possibility of placing bookmarks, etc.
- **LDasm (Linux Disassembler)** - Another shell for objdump/binutils.
- **Bastard** - Disassembler under Linux & FreeBSD, understands ELF/PE/bin- formats.

Hexadecimal editors:

- **Bless** - Written in mono hex editor. It supports the tab.
- **Biew** - Console editor. Allows you to view and edit files in text, hex, assembler form.
- **Radare** – An advanced cmdline hexadecimal editor.

2.5 SEARCHING FOR MORE CLUE

Where we could start from? A good starting point is trying to find out if the analysed binary file contains any interesting character strings from our next stage of investigation. This way we can gather some information about the platform used to build the binary and get an overall idea of the potential actions that the program could take. In every reversing engineering task which we handle, we should note that even trivial details could be useful to our analysis.

Searching for character strings will be accomplished with the help of the indispensable *strings* utility. It examines the contents of a given file and prints out all sequences of 4 or more printable (ASCII) characters (the default length of 4 can be changed with the *-n* option). By default, it scans only the initialised and loaded sections of an ELF file. To display all strings, we use the *-a* option.

In some cases, *strings* can reveal interesting information such as the operating systems used to compile the program and the compiler itself. Using *strings* with dynamically linked binaries produces more detailed output. Besides the strings found in the program code, it also shows a list of symbol names corresponding to the called shared library functions.

2.6 ANALYSING THE CONTENTS OF SPECIFIC PARTS OF THE FILE

Another way to search for interesting character strings in a file is to look through specific sections of the analysed program (see ELF file section header structure) that usually contain character strings.

We are going to look through the *.comment*, *.strtab*, *.dynstr*, *.note* or *.rodata* sections. The location of any section within the object is determined by the offset value in the section header. The header itself can be retrieved using *elfsh*, *ht* editor or *objdump* with the *-h* option. A fragment of the section header is shown in Figure 9.



2.6.1 ELF FILE SECTION HEADER STRUCTURE

The file's section header holds information about ELF object sections. A single segment may consist of one or more sections – for example, the PT_LOAD segment with permissions to read and execute might contain the .text, .init, .fini and .plt sections. Each section is described in the header with its type, name, size and memory location where the section is to be placed. The section header is required only for compiling the program (during the linking stage) and is ignored when the program is being executed. Each section contains information of a specific kind:

- *.init, fini* – the code responsible for starting and exiting the process.
- *.text* – the actual program code.
- *.data* – initialised data.
- *.bss* – uninitialized data (initialised to zero when the program is loaded).
- *.dynamic* – information used for dynamic linking.
- *.symtab* – symbol table.
- *.dynsym* – dynamic linking symbol table.
- *.strtab* – string table.
- *.dynstr* – dynamic linking string table.
- *.debug* – debugging information.
- *.rodata* – read-only data.
- *.rel** – relocation tables.
- *.ctors, dtors* – constructor and destructor tables.
- *.hash* – hash table.
- *.got* – global offset table.
- *.plt* – procedure linkage table.

2.7 RETRIEVING SYMBOL TABLE

The symbol table (see Frame Symbol Table) improves the readability of the program code. The table makes it possible to link function references using their names and it also defines the boundaries of each part of the program. To get a list of the symbols, we can use the `nm` command. Using the `-D` option shows the list of dynamic symbols, the `-g` option shows global symbols and the `-a` option shows all symbols.

Removal of the .symbol table destroys the obvious evidence of specific functions being used in the program. Using the `strip` command on a dynamically linked program wipes out all local symbols. Whereas, using it on a statically linked program deletes all contents of the symbol table.

The symbol table is not the only part of an executable that can be removed. Executable objects contain a few other optional sections such as `.debug` and `.comment`. Another removable part is the section header, which is required for link view, but not needed for execution view of an ELF object.

An example of a tool that strips the object of all unnecessary parts including the section header is `sstrip` from the ELF Kickers package (<http://www.muppetlabs.com/~breadbox/software/elfkickers.html>). The effect of using it apart from destroying the links between function calls and names is that it obliterates section boundaries. Currently, the ELF Kickers package is at version 2.0a.

Removing the section header has a side effect of preventing the analysis of the object using many utilities that make use of the `bfd` library (GNU Binary File Descriptor), which relies on the section header being present. An example of such an utility is `objdump`.



2.7.1 THE SYMBOL TABLE

When the final program is built, references among objects are managed through the so-called symbolic references. The linker or system linker/loader resolves these symbols and modifies the parts of the code that refer to them so that they point to the actual locations.

Symbols are structures that contain the names of objects (encoded as indexes to a table of character strings) and symbol values. Each symbol may be local, global or weak. Local symbols are available only within a single object, while global ones are accessible to other objects as well. Weak symbols are considered global until a global symbol with the same name is encountered.

A statically linked binary contains the `.symtab` symbol table, whereas a dynamically linked binary contains two tables: `.symtab` and `.dynsym`. The `.dynsym` table holds only those symbolic references which are needed for dynamic linking.

Statically linked binaries have all references already resolved so the symbol table is not longer required and can be removed. The removal is accomplished by stripping the ELF file (using the `strip` command). It is a simple method of making the analysis of a binary file more difficult.

2.8 REVERSING THE PROGRAM

Now, let's get our hands dirty by reverse engineering the program. This time round, I shall use `radare` to assist us. But you can use `ht editor` or other tools too.

2.9 REVERSE ENGINEERING

Issuing the following command will give us something like [Listing 1](#).

```
[root@home Desktop]# radare crkme1-Linux32
open ro crkme1-linux32
Message of the day:
  I like to suck nibbles and make hex
Automagically flagging crkme1-linux32
15 symbols added.
17 strings added.
0 syscalls added.
[0x00000000]> s sym_main
[0x000003C4]> pD
0x000003C4, sym_main: 55          push ebp
0x000003C5          89e5          ebp = esp
0x000003C7          83ec18       esp -= 0x18 ; 24 ' '
0x000003CA          83e4f0       esp &= 0xf0 ; 240 ' '
0x000003CD          b800000000  eax = 0x0
0x000003D2          29c4         esp -= eax
0x000003D4          c745fcf4860408 dword [ebp-0x4] = 0x80486f4
0x000003DB          c745f800870408 dword [ebp-0x8] = 0x8048700
0x000003E2          c745f000000000 dword [ebp-0x10] = 0x0
0x000003E9          c745f400000000 dword [ebp-0xc] = 0x0
0x000003F0          837d0801     cmp dword [ebp+0x8], 0x1
0x000003F4          0f8ec3010000 ^ jle dword 0x5BD 1 = sym_main+0x1f9
0x000003FA          8b450c       eax = [ebp+0xc]
```



```

0x000003FD      83c004      eax += 0x4 ; 4 ' '
0x00000400      83ec0c      esp -= 0xc ; 12 ' '
0x00000403      ff30        push dword [eax]
0x00000405      e8befeffff  ^ call 0x2C8 ; 2 = sym_strlen
0x0000040A      83c410      esp += 0x10 ; 16 ' '
0x0000040D      83f80a      cmp eax, 0xa
0x00000410      741f        v jz 0x431 ; 3 = sym_main+0x6d
0x00000412      83ec08      esp -= 0x8 ; 8 ' '
0x00000415      ff75fc      push dword [ebp-0x4]
0x00000418      680d870408 push dword 0x804870d ; "GoodSerial!"+0
0x0000041D      e8c6feffff  ^ call 0x2E8 ; 4 = sym_printf
0x00000422      83c410      esp += 0x10 ; 16 ' '
0x00000425      c745ec00000000 dword [ebp-0x14] = 0x0
0x0000042C      e9a8010000  ^ goto 0x5D9 ; 5 = sym_main+0x215
0x00000431      90          nop

```

Listing 1

From the above listing, we can roughly conclude that the serial MUST be of length 10 otherwise it will print "BadSerial!"

Now let's attach a debugger to it using the following commandline and give the program a serial, abcdefghij, of a length of 10:

```
[root@home Desktop]# radare dbg://"crkme1-linux32 abcdefghij"
```

```

argv = 'crkme1-linux32', 'abcdefghij', ]
Program 'crkme1-linux32 abcdefghij'
open debugger ro crkme1-linux32 abcdefghij
Message of the day:
  Find hexpairs with '/x a0 cc 33'
Automagically flagging crkme1-linux32
15 symbols added.
17 strings added.
15 syscalls added.
flag 'entry' at 0x08048300 and size 00
[0x43169810]>

```

What we should do next is to set a breakpoint at the address where the address of nop starts; in this case it is 0x8048431.

```
[0x43169810]> !bp 0x08048431
new breakpoint at 0x8048431
```

```
[0x43169810]> !cont
cont: breakpoint stop (0x8048431)
```

```
[0x43169810]> V
```

Press 'V' without the quotes and you will get something like below.

```

gunther@Gunther: ~/Desktop
File Edit View Terminal Tabs Help
[ 0xb7f20810 (inc=16, bs=100 sz=0 mark=0x0) hexb ] oeip
[. . . # . . . .]
offset  0 1 2 3 4 5 6 7 8 9 A B C D E F 0 1 0123456789ABCDEF01
0xB7F20810, 89e0 e849 0200 0089 c7e8 e2ff ffff 81c3 d6a7 ...I.....
0xB7F20822  0100 8b83 00ff ffff 5a8d 2484 29c2 528b 832c .....Z.$.)R..
0xB7F20834, 0000 008d 7494 088d 4c24 0489 e583 e4f0 5050 ....t...L$.....PP
0xB7F20846  5556 31ed e8f1 d100 008d 93cc 2dff ff8b 2424 UV1.....-...$$
0xB7F20858, ffe7 8db6 0000 0000 e842 4f01 0081 c18f a701 .....B0.....
0xB7F2086A  0055 89e5 5d8d 81cc 0500 .....U..].....

```

Pressing 'p' to change the print view to debugger mode.

```

gunther@Gunther: ~/Desktop
File Edit View Terminal Tabs Help
[ 0xb7f20810 (inc=18, bs=100 sz=0 mark=0x0) visual ] oeip
[. . . # . . . .]
Stack:
offset  0 1 2 3 4 5 6 7 8 9 A B C D E F 0 1 0123456789ABCDEF01
0xBF8930D0, c0dd f2b7 e085 0408 0000 0000 0000 0000 0087 .....
0xBF8930E2  0408 f486 0408 4831 89bf 5094 ddb7 0200 0000 .....H1..P.....
0xBF8930F4, 7431 89bf 8031 89bf 0083 0408 0000 0000 0200 t1...1.....
0xBF893106  0000 1c8b 7f03 f4af f3b7 f4df .....
Registers:
eax 0x0000000a   esi 0xb7f3ace0   eip 0x08048432
ebx 0xb7f0dff4   edi 0x00000000   oeax 0xffffffff
ecx 0x4554006a   esp 0xbf8930d0   eflags 0x200346
edx 0xffff0000   ebp 0xbf8930e8   cPaZsTIdor0 (PZTI)
Disassembly:
No line specified
                                ; Get arg12
0x08048432   eip: 8b450c           eax = [ebp+0xc]           ; o
eax+0xffffffff
0x08048435           83c004           eax += 0x4 ; 4 ' '
0x08048438,         8b55f0           edx = [ebp-0x10]
0x0804843B           8b00           eax = [eax]
0x0804843D           01d0           eax += edx
0x0804843F           803800           cmp byte [eax], 0x0
.==< 0x08048442       7502           v jnz 0x8048446           ;

```

Now, scroll down the dead listing, we can see that the program is checking to see if the current char is between 0x60 and 0x7A (which is 'a' to 'z').


```

root@Gunther: /home/gunther/Desktop
File Edit View Terminal Tabs Help
[ 0x804843d (inc=2, bs=100 sz=0 mark=0x0) disasm ] eip+0xb
[. #]
0x0804843D      01d0      eax += edx
0x0804843F      803800      cmp byte [eax], 0x0
0x08048442      7502      v jnz 0x8048446 ; 1 = eip+0x14
0x08048444,    eb4a      v goto 0x8048490 ; 2 = eip+0x5e
0x08048446      8b450c      eax = [ebp+0xc] ; oeax+0xfffffffff5
0x08048449      83c004      eax += 0x4 ; 4 ' '
0x0804844C,    8b55f0      edx = [ebp-0x10]
0x0804844F      8b00      eax = [eax]
0x08048451      01d0      eax += edx
0x08048453      803860      cmp byte [eax], 0x60 ←
0x08048456      7e31      v jle 0x8048489 ; 3 = eip+0x57
0x08048458,    8b450c      eax = [ebp+0xc] ; oeax+0xfffffffff5
0x0804845B      83c004      eax += 0x4 ; 4 ' '
0x0804845E,    8b55f0      edx = [ebp-0x10]
0x08048461      8b00      eax = [eax]
0x08048463      01d0      eax += edx
0x08048465      80387a      cmp byte [eax], 0x7a ←
0x08048468,    7f1f      jg 0x8048489 ; 4 = eip+0x57
0x0804846A      8b550c      edx = [ebp+0xc]
0x0804846D      83c204      edx += 0x4 ; 4 ' '

```

As we go through the disassembled code properly, we can see that each character is being converted to uppercase by subtracting 0x20.

```

gunther@Gunther: ~/Desktop
File Edit View Terminal Tabs Help
[ 0x804845e (inc=2, bs=100 sz=0 mark=0x0) disasm ] sym_main+0x9a
[. #]
0x0804845E      8b55f0      edx = [ebp-0x10]
0x08048461      8b00      eax = [eax]
0x08048463      01d0      eax += edx
0x08048465      80387a      cmp byte [eax], 0x7a
0x08048468,    7f1f      jg 0x8048489 ; 1 = sym_main+0xc5
0x0804846A      8b550c      edx = [ebp+0xc]
0x0804846D      83c204      edx += 0x4 ; 4 ' '
0x08048470,    8b45f0      eax = [ebp-0x10] ; oeax+0x5
0x08048473,    8b0a      ecx = [edx]
0x08048475      01c1      ecx += eax
0x08048477      8b550c      edx = [ebp+0xc]
0x0804847A      83c204      edx += 0x4 ; 4 ' '
0x0804847D      8b45f0      eax = [ebp-0x10] ; oeax+0x5
0x08048480,    0302      eax += [edx]
0x08048482      8a00      al = [eax]
0x08048484,    83e820      eax -= 0x20 ; 32 ' ' ← Subtract 0x20
0x08048487      8801      [ecx] = al
0x08048489      8d45f0      eax (lea) = [ebp-0x10]
0x0804848C,    ff00      dword [eax]++
0x0804848E      eba2      ^ goto 0x8048432 ; 2 = sym_main+0x6e
↑ Simple loop until end of string.

```

When we have reached the end of the string, we can find that it is doing another check by taking the first and last byte of the serial and store in edx and eax respectively. After that, it subtracts 0x3 from the last byte and check whether it's the same as the first byte.

```

gunther@Gunther: ~/Desktop
File Edit View Terminal Tabs Help
[ 0x8048483 (inc=2, bs=100 sz=0 mark=0x0) disasm ] sym_main+0xbf
[. #]
0x08048483 0083e8208801 [ebx+0x18820e8] += al
0x08048489 8d45f0 eax (lea)= [ebp-0x10]
0x0804848c, ff00 dword [eax]++
0x0804848e eba2 ^ goto 0x8048432 ; 1 = sym_main+0x6e
0x08048490, 8b450c eax = [ebp+0xc] ;
0x08048493 83c004 eax += 0x4 ; 4 ' '
0x08048496 8b00 eax = [eax]
0x08048498, 0fbe10 movsx edx, byte [eax] ← First Byte
0x0804849b 8b450c eax = [ebp+0xc] ;
0x0804849e 83c004 eax += 0x4 ; 4 ' '
0x080484a1 8b00 eax = [eax]
0x080484a3 83c009 eax += 0x9 ; 9 ' '
0x080484a6 0fbe00 movsx eax, byte [eax] ← Last Byte
0x080484a9 83e803 eax -= 0x3 ; 3 ' ' ← Subtract 0x3 from the
0x080484ac, 39c2 cmp edx, eax ← last byte and check if
0x080484ae 740c v jz 0x80484bc ; 2 = sym_main+0xf8 ← it's same as first byte.
0x080484b0, c745f40000000000 dword [ebp-0xc] = 0x0 ; oeax+0x1
0x080484b7 e9d1000000 ^ goto 0x804858d ; 3 = sym_main+0x1c9
0x080484bc, 8b450c eax = [ebp+0xc] ;
0x080484bf 83c004 eax += 0x4 ; 4 ' '

```

After we bypass the first check, we encountered another check by the application again. This time round, it retrieves the 2nd and 9th Character and stored it in edx and eax respectively. Next, it add 0xE to the 9th character to compare whether it is the same as the 2nd character. If it is, the application continues.

```

gunther@Gunther: ~/Desktop
File Edit View Terminal Tabs Help
[ 0x80484a9 (inc=3, bs=100 sz=0 mark=0x0) disasm ] sym_main+0xe5
[. #]
0x080484a9 83e803 eax -= 0x3 ; 3 ' '
0x080484ac, 39c2 cmp edx, eax
0x080484ae 740c v jz 0x80484bc ; 1 = sym_main+0xf8
0x080484b0, c745f40000000000 dword [ebp-0xc] = 0x0 ; oeax+0x1
0x080484b7 e9d1000000 ^ goto 0x804858d ; 2 = sym_main+0x1c9
0x080484bc, 8b450c eax = [ebp+0xc] ;
0x080484bf 83c004 eax += 0x4 ; 4 ' '
0x080484c2 8b00 eax = [eax]
0x080484c4, 40 eax++
0x080484c5 0fbe10 movsx edx, byte [eax] ← Get 2nd Char.
0x080484c8, 8b450c eax = [ebp+0xc] ;
0x080484cb 83c004 eax += 0x4 ; 4 ' '
0x080484ce 8b00 eax = [eax]
0x080484d0, 83c008 eax += 0x8 ; 8 ' '
0x080484d3 0fbe00 movsx eax, byte [eax] ← Get 9th Char.
0x080484d6 83c00e eax += 0xe ; 14 ' ' ← Add 0xE from 9th Char
0x080484d9 39c2 cmp edx, eax ← and check if it's same
0x080484db 740c v jz 0x80484e9 ; 3 = sym_main+0x125 ← as the 2nd Char
0x080484dd c745f40000000000 dword [ebp-0xc] = 0x0 ; oeax+0x1
0x080484e4, e9a4000000 ^ goto 0x804858d ; 4 = sym_main+0x1c9

```

Just as we thought that we had gone through the checks implemented in this application, there is another checking being done again. But this time, it retrieves the 3rd and 8th character and subtracting 0x14 from the

8th character. After that, it did a compare with the new value with the 3rd character. If it passes that, it proceeds.

```

gunther@Gunther: ~/Desktop
File Edit View Terminal Tabs Help
[ 0x80484dd (inc=2, bs=100 sz=0 mark=0x0) disasm ] sym_main+0x119
[ . # ]
0x080484DD      c745f400000000    dword [ebp-0xc] = 0x0 ; oeax+0x1
0x080484E4,    e9a4000000      ^ goto 0x804858D      ; 1 = sym_main+0x1c9
0x080484E9      8b450c          eax = [ebp+0xc] ;
0x080484EC,    83c004          eax += 0x4 ; 4 ' '
0x080484EF      8b00           eax = [eax]
0x080484F1      83c002          eax += 0x2 ; 2 ' '
0x080484F4,    0fbe10          movsx edx, byte [eax] ← Get 3rd Character.
0x080484F7      8b450c          eax = [ebp+0xc] ;
0x080484FA      83c004          eax += 0x4 ; 4 ' '
0x080484FD      8b00           eax = [eax]
0x080484FF      83c007          eax += 0x7 ; 7 ' '
0x08048502,    0fbe00          movsx eax, byte [eax] ← Get 8th Character.
0x08048505      83e814          eax -= 0x14 ; 20 ' ' ← Subtract 0x14 from 8th Char
0x08048508,    39c2           cmp edx, eax          ← and check if it's same
0x0804850A,    7409           jz 0x8048515          as the 3rd Char
0x0804850C,    c745f400000000    dword [ebp-0xc] = 0x0 ; oeax+0x1
0x08048513,    eb78           v goto 0x804858D      ; 2 = sym_main+0x151
0x08048515      8b450c          eax = [ebp+0xc] ;
0x08048518,    83c004          eax += 0x4 ; 4 ' '
0x0804851B      8b00           eax = [eax]

```

And just as I thought, there is more checking being done after the previous one. It retrieves the 4th and 7th character and added 0x6 to the 7th character and did a compare.

```

gunther@Gunther: ~/Desktop
File Edit View Terminal Tabs Help
[ 0x8048505 (inc=3, bs=100 sz=0 mark=0x0) disasm ] sym_main+0x141
[ . # ]
0x08048505      83e814          eax -= 0x14 ; 20 ' '
0x08048508,    39c2           cmp edx, eax
0x0804850A,    7409           jz 0x8048515          ; 1 = sym_main+0x151
0x0804850C,    c745f400000000    dword [ebp-0xc] = 0x0 ; oeax+0x1
0x08048513,    eb78           v goto 0x804858D      ; 2 = sym_main+0x1c9
0x08048515      8b450c          eax = [ebp+0xc] ;
0x08048518,    83c004          eax += 0x4 ; 4 ' '
0x0804851B      8b00           eax = [eax]
0x0804851D      83c003          eax += 0x3 ; 3 ' '
0x08048520,    0fbe10          movsx edx, byte [eax] ← Get 4th Character.
0x08048523,    8b450c          eax = [ebp+0xc] ;
0x08048526,    83c004          eax += 0x4 ; 4 ' '
0x08048529,    8b00           eax = [eax]
0x0804852B      83c006          eax += 0x6 ; 6 ' '
0x0804852E,    0fbe00          movsx eax, byte [eax] ← Get 7th Character.
0x08048531,    83c006          eax += 0x6 ; 6 ' ' ← Add 0x6 to 7th Char
0x08048534,    39c2           cmp edx, eax          ← and check if it's same
0x08048536,    7409           jz 0x8048541          as the 4th Char
0x08048538,    c745f400000000    dword [ebp-0xc] = 0x0 ; oeax+0x1
0x0804853F      eb4c           v goto 0x804858D      ; 3 = sym_main+0x17d

```

Now, for the final 2 characters. After retrieving both of them, it added them together and did a arithmetic shift to the right (which is dividing the result by 2).

```

gunther@Gunther: ~/Desktop
File Edit View Terminal Tabs Help
[ 0x8048541 (inc=3, bs=100 sz=0 mark=0x0) disasm ] sym_main+0x17d
[ . # ]
0x08048541      8b450c      eax = [ebp+0xc] ;
0x08048544,    83c004      eax += 0x4 ; 4 ' '
0x08048547      8b00        eax = [eax]
0x08048549      83c004      eax += 0x4 ; 4 ' '
0x0804854C,    0fbe10      movsx edx, byte [eax] ← Get 5th Character.
0x0804854F      8b450c      eax = [ebp+0xc] ;
0x08048552      83c004      eax += 0x4 ; 4 ' '
0x08048555      8b00        eax = [eax]
0x08048557      83c005      eax += 0x5 ; 5 ' '
0x0804855A      0fbe00      movsx eax, byte [eax] ← Get 6th Character.
0x0804855D      01c2        edx += eax
0x0804855F      89d0        eax = edx ← Add the 2
                                                characters together.
0x08048561      clf81f      sar eax, 0x1f
0x08048564,    c1e81f      eax >>= 0x1f ; 31 ' ' (zerofill)
0x08048567      8d0410      eax (lea) = [eax+edx]
0x0804856A      89c2        edx = eax
0x0804856C,    d1fa        sar edx, 1 ← Arithmetic shift to the right.
0x0804856E      8b450c      eax = [ebp+0xc] ;
0x08048571      83c004      eax += 0x4 ; 4 ' '
0x08048574,    8b00        eax = [eax]

```

Then we take the 1st character and did a compare with our last result which we stored in edx. Once we have bypass that test, you can see that it printf "Good Serial".

```

gunther@Gunther: ~/Desktop
File Edit View Terminal Tabs Help
[ 0x804856c (inc=2, bs=100 sz=0 mark=0x0) disasm ] sym_main+0x1a8
[ . # ]
0x0804856C,    d1fa        sar edx, 1
0x0804856E      8b450c      eax = [ebp+0xc] ;
0x08048571      83c004      eax += 0x4 ; 4 ' '
0x08048574,    8b00        eax = [eax]
0x08048576      0fbe00      movsx eax, byte [eax] ← Get 1st Character.
0x08048579      39c2        cmp edx, eax ← Compare with the previous result we got.
0x0804857B      7409        jz 0x8048586 ← Jump if zero.
0x0804857D      c745f400000000 dword [ebp-0xc] = 0x0 ; oeax+0x1
0x08048584,    eb07        v goto 0x804858D ; 2 = sym_main+0x1c9
0x08048586      c745f401000000 dword [ebp-0xc] = 0x1 ; oeax+0x1
0x0804858D      837df401    cmp dword [ebp-0xc], 0x1
0x08048591      7515        v jnz 0x80485A8 ; 3 = sym_main+0x1e4
0x08048593      83ec08      esp -= 0x8 ; 8 ' ' ;
0x08048596      ff75f8      push dword [ebp-0x8] ;
0x08048599      680d870408 push dword 0x804870d ; str_Good_Serial_+0xd
0x0804859E      e845fdffff ^ call 0x80482E8 ; 4 = imp_printf
0x080485A3      83c410      esp += 0x10 ; 16 ' '
0x080485A6      eb2a        v goto 0x80485D2 ; 5 = sym_main+0x20e
0x080485A8,    83ec08      esp -= 0x8 ; 8 ' ' ;
0x080485AB      ff75fc      push dword [ebp-0x4] ;

```



2.10 CONCLUSIONS

Now, it's time for trying to reverse other applications with we have seen so far.

Using the above analysis, you can build your own key generator to the above crackmes or any other Linux applications which require serial numbers.

If some of you are intimidated by the CLI (Command Line Interface), you can try the gradare, GUI front-end, of radare, It'll provides decent graphs like IDA Pro.

All the stuff explained in this tutorial has been tested.

I hope that this document has provided some new ways of reversing knowledge in Linux, and when we face similar disassembled codes it wouldn't intimidate you in any way and you could assume the challenge of researching and researching...in the ARTeam spirits.

2.11 GREETINGS

This tutorial is dedicated to all the components of ARTeam and Shub for editing it. Big thanks to Shub.

And of course, to you, who decided to read this document, because without your support and contribution this task wouldn't be worth to be done.

[No supplements for this tutorial.]



3 USING .NET PROFILING API FOR A CUSTOM .NET PROTECTION BY KURAPICA

In this paper we will discuss the .NET Profiling APIs that ship with the .NET framework and see how we can use them to implement a tight protection scheme to hide the MSIL code in the final assembly, hidden MSIL code means that tools like reflector won't be able to produce source code from our compiled assembly and this can increase the security and protection against reverse engineering.

Before we start exploring the ins and outs of the protection scheme we will discuss the Profiling APIs briefly and see the events and methods that are important in our protection.

3.1 WHAT IS PROFILING?

Part of the .NET framework SDK speaks on an API that allows you to find out lots of information about the behavior of the application coded with .NET technology, The API allows us to get many different types of information during run time, you can also use the API to code tracing tools, exception analysis tools and memory usage analysis tools.

This paper discusses the Profiling API, a part of .NET that has not yet received the attention it really deserves; you can code a profiler that can be used against any .NET application without that application knowing it is being profiled. The profiler works on an event system and as interesting things happen the profiler is immediately notified, and can therefore build up real-time analysis of the profilee (the application being profiled)

The Profiling API is all based around COM interfaces, .NET profilers must be built as in-process COM servers. This means we can write .NET profilers in any COM-capable unmanaged language such as Delphi or C++.

The profiler must implement an interface defined by the Profiling API. This interface has a variety of methods that are triggered like events when interesting things happen. There are many of these events in the interface, and the profiler specifies which groups of events it is interested in receiving. Doing this avoids having every event repeatedly triggered during the profiling session, which may heavily impact the application's performance.

When managed applications are subsequently run the CLR loads the profiler COM server into the managed application's address space and told which events to fire and proceeds to do so throughout the rest of that application's lifetime.

The callback interface [ICorProfilerCallback](#) which must be implemented by each profiler. This interface contains 69 methods and each of which is an event, although 3 of the methods will never be called in .NET 1.x (they are there for use in future versions of .NET). Two of these events are special and are always called; the rest are called only if requested (and if the event actually occurs in the managed application), in .NET framework 2.0 the interface was expanded and new events were added, the special event methods are [Initialize](#) and [Shutdown](#). [Initialize](#) is called as soon as the CLR has initialized and loaded up the profiler, and it is where you request the event categories you wish to be triggered throughout the program run. [Shutdown](#) is called as the CLR is closing down during application termination.



Initialize is passed an **IUnknown** reference to another interface as a parameter, this one being **ICorProfilerInfo** (also expanded to **ICorProfilerInfo2** in .NET framework 2.0). This interface (like all of them except the callback interface) is implemented in the CLR and is there to help you get more information when various events trigger, offering you 33 methods to choose from. You should save this reference away so it can be used in the event methods that are subsequently called. For example:

```
procedure TSomeProfiler.Initialize(const pICorProfilerInfoUnk: IUnknown);
begin
```

```
    CorProfilerInfo := pICorProfilerInfoUnk as ICorProfilerInfo;
    CorProfilerInfo.SetEventMask(COR_PRF_MONITOR_JIT_COMPILATION);
```

```
end;
```

You initially request the appropriate event categories in **ICorProfilerCallback.Initialize** by passing the required flags along to the **ICorProfilerInfo.SetEventMask** method.

Linking the profiler COM object to the protected assembly is easy and all you have to do is to create a process object and set the environment variables for this process like this:

```
'Create Process
```

```
Dim PSInfo As New ProcessStartInfo(TextBox1.Text)
```

```
'Initialize Profiler variables
```

```
PSInfo.EnvironmentVariables.Add("COR_ENABLE_PROFILING", "1")
```

```
PSInfo.EnvironmentVariables.Add("COR_PROFILER", "{34F20DB8-FA3C-4356-945D-5D7819E81C8B}")
```

```
'Start the protected assembly
```

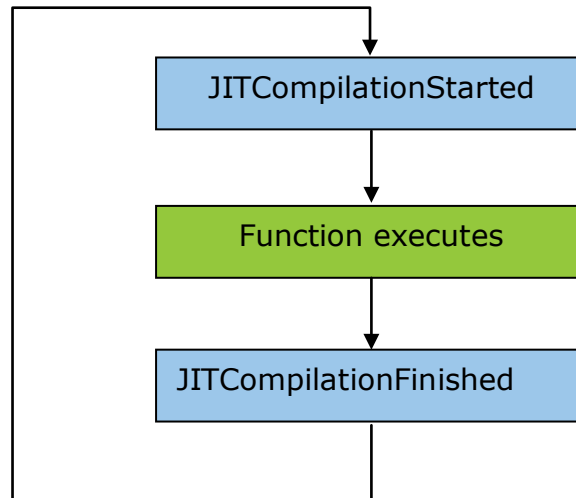
```
Dim Pr As Process = Process.Start(PSInfo)
```

These lines of code are enough to start the profiling process, the first environment variable tells the CLR to enable profiling for the process and the second variable is the globally unique identifier of this profiler COM object in windows registry, so it guides the CLR to the correct profiler COM object to load and initialize.

3.2 HOW DOES THE PROTECTION WORK?

The basic idea of this protection is using two important events that the CLR notifies the profiler of, these two events are:

- **JITCompilationStarted:** The CLR calls JITCompilationStarted to notify the code profiler that the JIT compiler is starting to compile a function but the important point here is that the function didn't start execution yet.
- **JITCompilationFinished:** The CLR calls JITCompilationFinished to notify the code profiler that the JIT compiler has finished compiling a function; here the function has finished execution.



Basically this protection scheme consists of 3 files:

- 1- The Loader: it can be a simple console executable which job is to create the profilee process and prepare the environment variables.
- 2- The Profiler DLL: which is an implementation of the interfaces I described earlier, you can use any COM-enabled language like Delphi or C++ to write this DLL.
- 3- The Protected assembly: This is the assembly that we need to hide its MSIL to prevent decompilers and disassemblers from reproducing the MSIL code in any higher level language like VB.net or C# or even IL.

The beauty of this protection lies in the fact that the protected assembly can't execute on its own unless loaded by the Loader, simply because the protected assembly code is nopped, all the MSIL code bytes are set to 00 which means that nothing is executed, the task of the Profiler DLL is to interfere in the right time to provide the original MSIL code to the CLR and then execution can go normally.

3.3 THE PROFILING APIS

The profiler DLL is made up of two main objects, the **ICorProfilerCallback** which is responsible for the events triggering, i.e. it tells us when something interesting happens like when some method is about to execute or when some exception is raised and many other events, the second object is the **ICorProfilerInfo** which is used by a code profiler to communicate with the CLR to control event monitoring and request information

We are mainly interested in two API which are:

- **Procedure** GetILFunctionBody(moduleId: **ULONG**; methodId: **ULONG**; out ppMethodHeader: **PByte**; out pcbMethodSize: **Ulong**); **safecall**;

Retrieves a pointer to the body of a method starting at its header, a method is copied by the module it lives in, because this function is designed to give a tool access to MSIL before it has been loaded by the Runtime, it



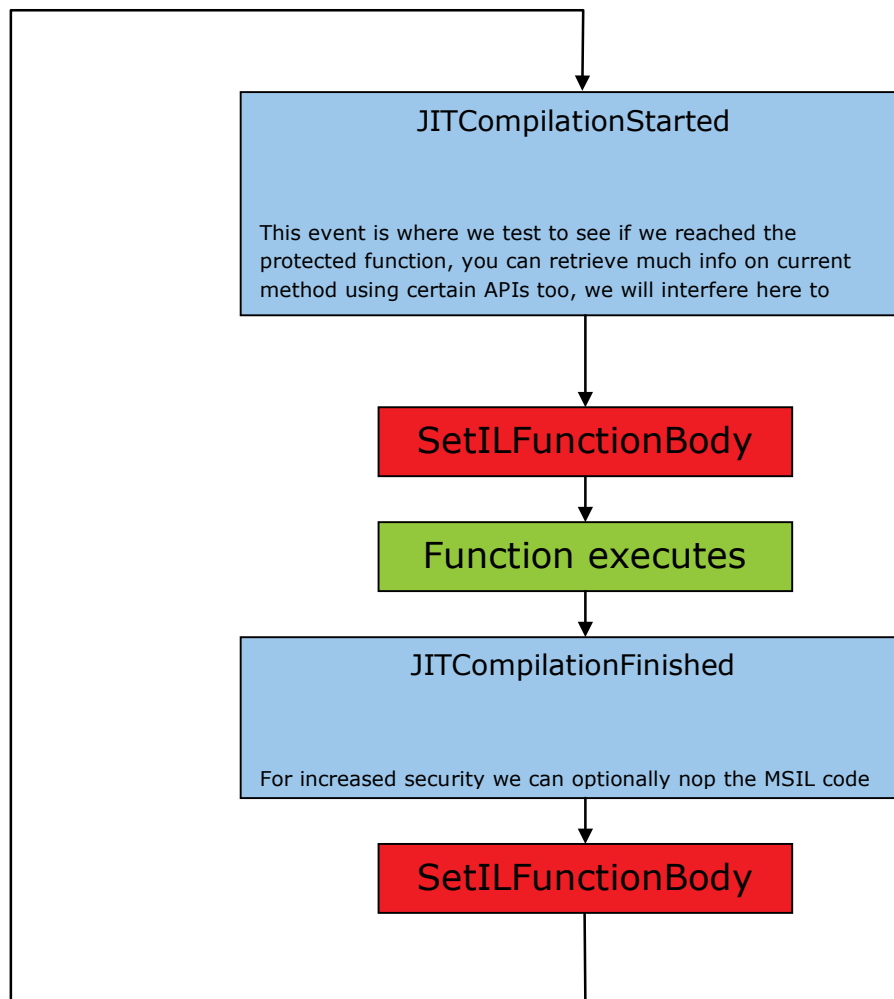
uses the metadata token of the method to find the instance desired, this API is not actually used in our protection but I mentioned here for relevance.

- **Procedure** SetILFunctionBody(moduleId: **ULONG**; methodId: **ULONG**; var pbNewILMethodHeader: **Byte**); safecall;

Replaces the method body for a function in a module, This will replace the RVA of the method in the metadata to point to this new method body and adjusts any internal data structures as required, This function can only be called on those methods which have never been compiled by a JITTER, this is the API that we will use to replace the nopped MSIL code with the original code so that execution continues normally.

3.4 THE WORKFLOW

Clearly our job is to provide the original MSIL code just before the protected function is executed and if you look at this figure you will understand when we have to do this task.





3.5 IMPLEMENTATION

- **The Loader** : implemented in VB.NET here but you can use C# or Delphi.NET

```
Imports System.Runtime.InteropServices.Marshal
```

```
Imports System.Runtime.InteropServices
```

```
Module Main
```

```
    Sub Main()
```

```
        Try
```

```
            'Register Profiler Library Classes in windows registry
```

```
            RegisterDLL()
```

```
            'Extract Path of protected assembly which is supposed to be placed  
            in same folder
```

```
            Dim xPath As New
```

```
                IO.FileInfo(Reflection.Assembly.GetExecutingAssembly.Location)
```

```
            'Create Process
```

```
            Dim PSInfo As New ProcessStartInfo(xPath.DirectoryName &  
                "\ProtectedAssembly.exe")
```

```
            'Initialize Profiler variables
```

```
            ProcessStartInfo.EnvironmentVariables.Add("COR_ENABLE_PROFILING", "1")
```

```
            ProcessStartInfo.EnvironmentVariables.Add("COR_PROFILER", "{C827957B-  
A8EB-4CD9-BD38-F0CC9B1DB1E5}")
```

```
            'Start Process
```

```
            PSInfo.UseShellExecute = False
```

```
            Dim Pr As Process = Process.Start(PSInfo)
```

```
        Catch ex As Exception
```

```
            MsgBox(ex.Message, MsgBoxStyle.Critical)
```

```
        End Try
```



End Sub

```
#Region "Register Profiler DLL"
```

```
<Runtime.InteropServices.DllImport("System.dll")> _
```

```
Private Function DllRegisterServer() As Integer
```

```
End Function
```

```
Private registered As Boolean
```

```
Friend Function RegisterDLL() As Boolean
```

```
    If Not registered Then
```

```
        If (DllRegisterServer() <> 0) Then
```

```
            Throw New Exception("Couldn't register System.dll")
```

```
        End If
```

```
        registered = True
```

```
    End If
```

```
    Return registered
```

```
End Function
```

```
#End Region
```

```
End Module
```



- **The Protection DLL** : It's implemented in Delphi

```

unit SampleProfiler;

interface

uses

    Windows, ActiveX, ComObj, BaseProfiler, CorProf, ComServ, Classes, Sysutils,
    StrUtils;

type

    TSampleProfiler = class(TBaseProfiler, ICorProfilerCallback2)
    protected

        procedure OnInitialize; override;

        // COR_PRF_MONITOR_JIT_COMPILATION

        procedure JITCompilationStarted(functionId: ULONG; fIsSafeToBlock: Integer);
        safecall;

    end;

const

    // Update these constants for each new profiler

    ProfilerCoClassName = 'Kurapica_Profiler';

    ProfilerDescription = 'Protection Profiler for .NET assemblies';

    ProfilerGUID: TGUID = '{C827957B-A8EB-4CD9-BD38-F0CC9B1DB1E5}';

    ProfilerClass: TComClass = TSampleProfiler;

implementation

var

    // These are the original MSIL code bytes that we want to hide

    ProtectedMSIL: array[0..201] of byte = (
        $1B,$30,$04,$00,$BE,$00,$00,$00,$1E,$00,$00,$11,$02,$6F,$39,$00,
        $00,$06,$6F,$62,$00,$00,$0A,$6F,$63,$00,$00,$0A,$6F,$64,$00,$00,
        $0A,$16,$2E,$18,$02,$6F,$37,$00,$00,$06,$6F,$62,$00,$00,$0A,$6F,

```



```

$63,$00,$00,$0A,$6F,$64,$00,$00,$0A,$16,$33,$02,$16,$2A,$73,$65,
$00,$00,$0A,$13,$04,$11,$04,$72,$AE,$08,$00,$70,$6F,$66,$00,$00,
$0A,$73,$67,$00,$00,$0A,$0B,$07,$28,$68,$00,$00,$0A,$02,$6F,$39,
$00,$00,$06,$6F,$62,$00,$00,$0A,$6F,$63,$00,$00,$0A,$6F,$69,$00,
$00,$0A,$6F,$6A,$00,$00,$0A,$0C,$02,$6F,$37,$00,$00,$06,$6F,$62,
$00,$00,$0A,$6F,$63,$00,$00,$0A,$28,$6B,$00,$00,$0A,$13,$05,$11,
$04,$08,$72,$D7,$0F,$00,$70,$11,$05,$6F,$6C,$00,$00,$0A,$0D,$09,
$45,$02,$00,$00,$00,$06,$00,$00,$00,$02,$00,$00,$00,$DE,$19,$17,
$0A,$DE,$15,$16,$0A,$DE,$11,$25,$28,$1D,$00,$00,$0A,$13,$06,$16,
$0A,$28,$21,$00,$00,$0A,$DE,$00,$06,$2A
);
//=====
// Start
//=====
procedure TSampleProfiler.OnInitialize;
begin
    {Set events mask}
    EventMask := COR_PRF_MONITOR_JIT_COMPILATION;
end;

//=====
// Method is about to be executed
//=====
procedure TSampleProfiler.JITCompilationStarted(functionId: ULONG;
    fIsSafeToBlock: Integer);
var
    // Variables to extract current method name and see
    // if it's the protected method

```



```
ClassName, MethodName: WideString;

// Variables necessary to extract info about current method using the
// GetFunctionInfo API
ClassId,ModuleID,IToken:ULONG;

// Object to allocate memory for the original MSIL code
Ialloc : IMethodMalloc ;

// Pointer to ProtectedMSIL array bytes, used for copying memory
PprotectedMSIL : Pbyte ;

// Pointer to original MSIL location where we will copy the ProtectedMSIL
// to memory block
PtempMSIL : PByte;

begin
SyncEnter;

try
{Initialize variables}
ClassId:=0;
ModuleID:=0;
IToken:=0;

// Retrieve information about current function
CorProfilerInfo.GetFunctionInfo(functionId,ClassId,ModuleID,IToken);
{Get function name}
if GetClassAndMethodFromFunctionId(functionID, ClassName, MethodName) then
begin
// Check if we are at "CheckLicense" method
If MethodName = 'CheckLicense' then
begin
//Alloc memory for new ILs
Ialloc := CorProfilerInfo.GetILFunctionBodyAllocator(ModuleId);
```

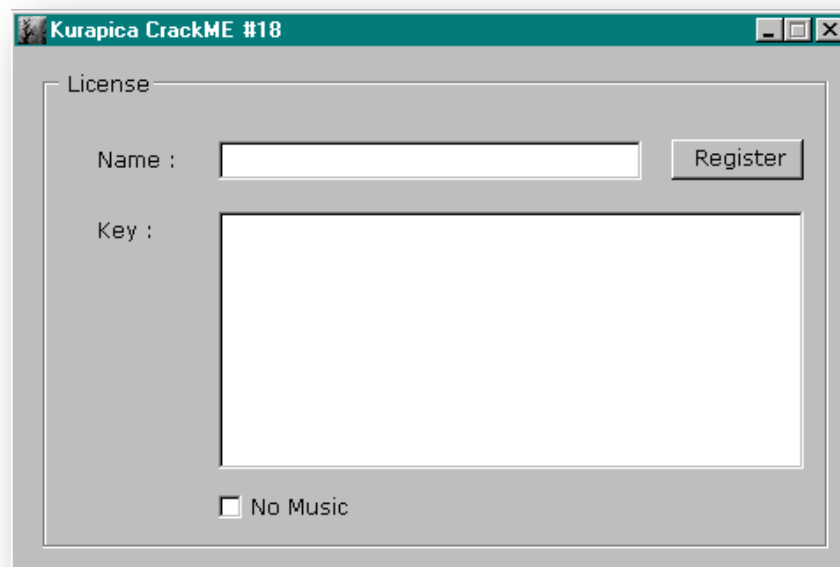



```
PtempMSIL:=Ialloc.Alloc(202);  
  
//Copy original MSIL to memory  
  
PprotectedMSIL:=@ProtectedMSIL;  
  
CopyMemory(PtempMSIL,PprotectedMSIL,202);  
  
// Set new ILs  
  
CorProfilerInfo.SetILFunctionBody(ModuleID,IToken,PtempMSIL^);  
  
end  
  
end  
  
except  
  
on E: Exception do  
  
log(E.message);  
  
end;  
  
SyncExit  
  
end;  
  
initialization  
  
TComObjectFactory.Create(ComServer, ProfilerClass, ProfilerGUID,  
    ProfilerCoClassName, ProfilerDescription, ciMultiInstance, tmFree);  
  
end.
```

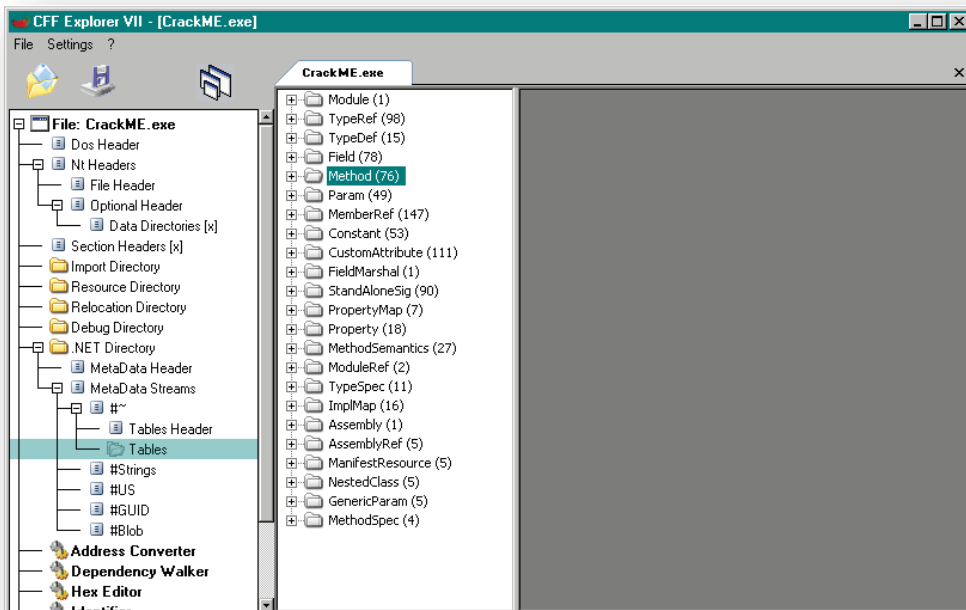
3.6 PREPARING THE ASSEMBLY:

The assembly that you want to protect can only be of type executable, all you have to do is to replace the MSIL code in every important function with 00 bytes so that reflector or other tools are useless, here is an example that I posted earlier in Crackme #18 which implements this protection effectively to protect one function called "**CheckLicense**", the entire key generation process lies within this function so It's our target.

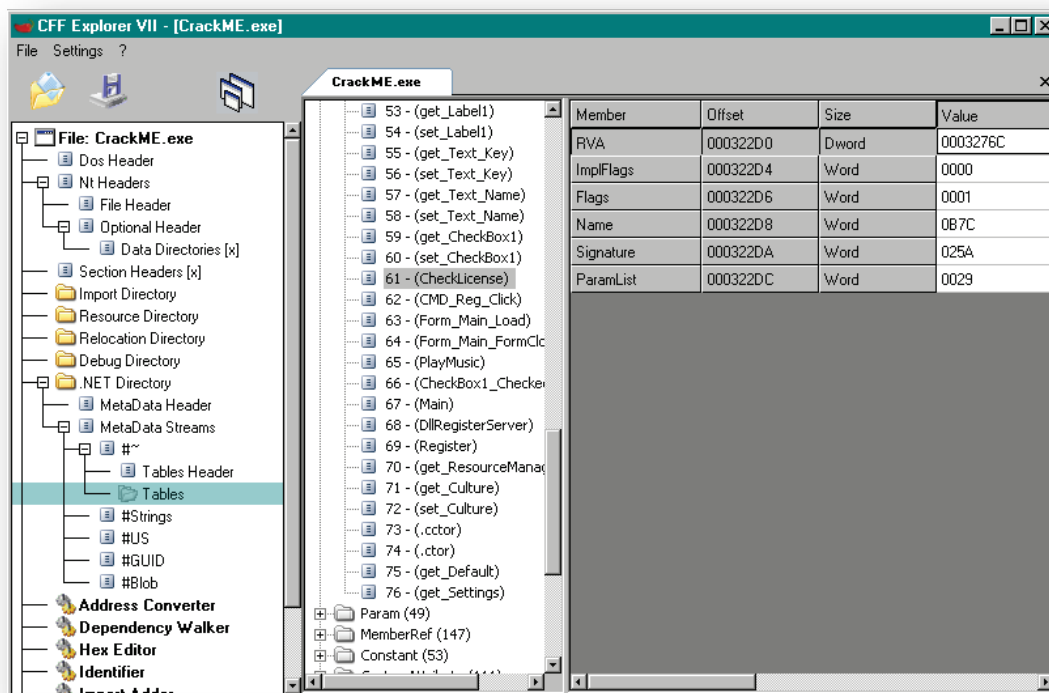
1. Build your Assembly using visual studio and keep a list of functions' addresses that you want to protect.



2. Next step is locating these functions in WinHex to nop their bytes, this entire procedure can be automated using simple coding but I will explain my manual method here.



3. You can also use the classic method by finding the correct node of the protected function in **ILdsam** then you can read the RVA value and convert it to a file offset like we did in most previous tutors, but **CFF explorer** provides us with the RVA value directly by expanding the "method" table, now find the "CheckLicense" node and read its RVA value.



- Use the address converter function to convert this value to a file offset and open WinHex

The 12 shaded bytes are the method's header and should be left intact

00031760	06 73 2F 00 00 0A 6F 61	00 00 0A 2A 1B 30 04 00	.s/...oa...*.0..
00031770	BE 00 00 00 1E 00 00 11	02 6F 39 00 00 06 6F 62	%.....o9...ob
00031780	00 00 0A 6F 63 00 00 0A	6F 64 00 00 0A 16 2E 18	...oc...od.....
00031790	02 6F 37 00 00 06 6F 62	00 00 0A 6F 63 00 00 0A	.o7...ob...oc...
000317A0	6F 64 00 00 0A 16 33 02	16 2A 73 65 00 00 0A 13	od....3..*se....
000317B0	04 11 04 72 AE 08 00 70	6F 66 00 00 0A 73 67 00	...r@..pof...sg.
000317C0	00 0A 0B 07 28 68 00 00	0A 02 6F 39 00 00 06 6F(h....o9...o
000317D0	62 00 00 0A 6F 63 00 00	0A 6F 69 00 00 0A 6F 6A	b...oc...oi...oj
000317E0	00 00 0A 0C 02 6F 37 00	00 06 6F 62 00 00 0A 6Fo7...ob...o
000317F0	63 00 00 0A 28 6B 00 00	0A 13 05 11 04 08 72 D7	c...(k.....r×
00031800	0F 00 70 11 05 6F 6C 00	00 0A 0D 09 45 02 00 00	..p..ol.....E...
00031810	00 06 00 00 00 02 00 00	00 DE 19 17 0A DE 15 16ق...ق..
00031820	0A DE 11 25 28 1D 00 00	0A 13 06 16 0A 28 21 00	.ق.%(...(.
00031830	00 0A DE 00 06 2A 00 00	01 0C 00 00 00 00 32 00	..2.....*..ق.

- Select the method's MSIL code bytes and nop them

00031770	BE 00 00 00 1E 00 00 11	02 6F 39 00 00 06 6F 62	%.....o9...ob
00031780	00 00 0A 6F 63 00 00 0A	6F 64 00 00 0A 16 2E 18	...oc...od.....
00031790	02 6F 37 00 00 06 6F 62	00 00 0A 6F 63 00 00 0A	.o7...ob...oc...
000317A0	6F 64 00 00 0A 16 33 02	16 2A 73 65 00 00 0A 13	od....3..*se....
000317B0	04 11 04 72 AE 08 00 70	6F 66 00 00 0A 73 67 00	...r@..pof...sg.
000317C0	00 0A 0B 07 28 68 00 00	0A 02 6F 39 00 00 06 6F(h....o9...o
000317D0	62 00 00 0A 6F 63 00 00	0A 6F 69 00 00 0A 6F 6A	b...oc...oi...oj
000317E0	00 00 0A 0C 02 6F 37 00	00 06 6F 62 00 00 0A 6Fo7...ob...o
000317F0	63 00 00 0A 28 6B 00 00	0A 13 05 11 04 08 72 D7	c...(k.....r×
00031800	0F 00 70 11 05 6F 6C 00	00 0A 0D 09 45 02 00 00	..p..ol.....E...
00031810	00 06 00 00 00 02 00 00	00 DE 19 17 0A DE 15 16ق...ق..
00031820	0A DE 11 25 28 1D 00 00	0A 13 06 16 0A 28 21 00	.ق.%(...(.
00031830	00 0A DE 00 06 2A 00 00	01 0C 00 00 00 00 32 00	..ق..*.....2.

- There is a feature in WinHex that helps you rip the bytes to a Pascal or C source so you can paste the original MSIL code directly as an array in your protective DLL, now overwrite these selected bytes with 00 byte and you are done.
- After you nop the bytes you can sign your assembly with a strong name signature to add another layer of protection.



3.7 CONCLUSION

Now we will talk about the Pros and Cons of this protection technique.

As you can see, using the profiling APIs which are part of the .NET framework itself can ensure stability more than other code hiding techniques used in commercial protectors like **Maxtocode** and **CLI-Secure**, you can use this technique for all .net versions as long as you implement the profiler's COM interfaces correctly, another point of strength is that If you use other supporting techniques like names obfuscation and control-flow obfuscation then you will have a stronger protection in total.

The weakness of this protection is that it can impact the performance, the hook in the profiler can slow the performance a bit since it has to check for every method being compiled, I used an explicit string comparison to see if this is the right function to fill, in bigger application and when you have many protected functions you can use a hash table to increase performance, but this is theoretically unnoticeable to humans, besides most machines run on high clock rates nowadays.

Defeating this protection requires hooking the JITTER to dump the MSIL code just before it gets executed so it's possible but it can be a hectic job if you add control-flow obfuscation, UFO-PU55Y's ILLY plug-in is a good example for this.

3.8 REFERENCES

- [1] The .NET Profiling API and the DNProfiler Tool, Matt Pietrek, MSDN Magazine December 2001, <http://msdn.microsoft.com/msdnmag/issues/01/12/hood/default.aspx>
- [2] .NET Internals: The Profiling API, Brian Long
- [3] The .NET Framework SDK Tool Developer Guide

3.9 GREETINGS

I want to send a special greeting to my friend in crime UFO-PU55Y for all the help and support he provided and to all the following people with no particular order

LibX – Devine9 – revert – Apakekdah – Lena151 – Ntoskrnl – Shub-Niqurrath

And the following teams

RET – SnD – ARTeam – Bl@ck Storm – AT4RE – AORE – CiM

[In the Supplements folder “0.3 Kurapica” you can find also the full crackme18 used here.]



4 PRIMER ON REVERSING PALMOS APPLICATIONS EXTENDED EDITION BY, WAST3D_BYTES, SUNTZU

4.1 FOREWORDS

I believe time has come for a new tutorial. This time we will leave the Windows and we will be transferred in a new field.

Palm OS scene is active since a long time, releasing a lot of applications and with some important groups. Applications are found a lot in the market. Some of them have been protected well and with nice tricks and that will make our study more interesting.

For the Palm OS scene I have found a few tutorials the market is not so prolific. The majority (btw you can consider all of them except 4) are not written in English. Most of these are simple so they will help us get the basic idea and some of them are for advanced reversing.

I decided to take a trip to this world, by searching the tools which will help disassemble the programs, what approaches we should follow and what we can do to create some patches so we can distribute them.

I started from scratch since I haven't seen any discussion on the forums (or I have not seem/found them). This is a special issue which includes various tutorials I wrote with different targets and different difficulty levels. I know that some people who have better knowledge from me may laugh because of the simple approaches.

The tutorial will cover different issues:

- Some words about Palm OS
- The laboratory with our tools
- Examples on real applications

Suntzu is author of a video tutorial in the supplements folder

4.2 FEW WORDS ON PALM OS

Since the introduction of the first Palm Pilot in 1996, the Palm OS platform has defined the trends and expectations for mobile computing - from the way people use handhelds as personal organizers to the use of mobile information devices as essential business tools, and even the ability to access the Internet or a central corporate database via a wireless connection.

Palm OS 5, which has been available to customers for years, supports ARM®-compliant processors. [Palm OS Garnet](#) is an enhanced version of Palm OS 5 and provides features such as dynamic input area, improved network communication, and support for a broad range of screen resolutions including QVGA.

[Palm OS Cobalt 6.1](#) is the next generation of Palm OS. It will enable the creation of new categories of devices for the communications, enterprise, education and entertainment markets. Palm OS Cobalt 6.1 provides integrated telephony features, support for WiFi and Bluetooth, and enhancements to the user interface.



As with previous versions of Palm OS, Palm OS Garnet and Palm OS Cobalt retain application compatibility with existing 68K-based applications.

Palm OS was originally developed by [Jeff Hawkins](#) for use on the original [Pilot PDA](#) by [US Robotics](#). Version 1.0 was present on the original Pilot 1000 and 5000 and version 2.0 was introduced with the PalmPilot Personal and Professional.

With the launch of the Palm III series, version 3.0 of the OS was introduced. Incremental upgrades occurred with the release of versions 3.1, 3.3, and 3.5, adding support for color, multiple expansion ports, new processors, and other various additions.

Version 4.0 was released with the m500 series, and later made available as an upgrade for older devices. This added a standard interface for external [filesystem](#) access (such as SD cards) and improved [telephony](#) libraries, security, and the UI. Version 4.1 included a series of bug fixes.

Palm OS 1.0-4.1 were based on top of a small kernel licensed from Kadak. While these versions are technically capable of multitasking, the "terms and conditions of that license specifically state that Palm may not expose the API for creating/manipulating tasks within the OS.

Version 5.0 was introduced with the [Tungsten T](#) and was the first version released to support [ARM](#) devices. Described as a stepping stone to full ARM support, Palm apps are run in an emulated environment called the [Palm Application Compatibility Environment](#) (PACE), making the device capable of running software written for older versions. Even with the additional overhead of PACE, Palm applications usually run faster on ARM devices than on previous generation hardware. New software can take advantage of the ARM processors with PNO (PACE Native objects), small units of ARM code, these are also sometimes referred to as 'ARMlets'. It was also around this time when Palm began to separate its hardware and OS efforts, eventually becoming two companies, [PalmSource, Inc.](#) (OS) and palmOne (hardware, now named Palm, Inc.). Further releases of Palm OS 5 have seen a standardised API for hi-res and dynamic input areas, along with a number of more minor improvements.

Palm OS 5.2 and 4.1.2 (and later) also feature [Graffiti 2](#). This is based on Jot by CIC.

Version 5.4 added the NVFS (Non-Volatile File System), and used Flash for storage instead of DRAM, preventing data-loss in the event of battery drain. However, this fundamentally changed the way programs were executed from the Execute-in-Place system that PalmOS traditionally used, and has been the source of many compatibility problems, requiring many applications to have explicit NVFS support added for them to be stable.

In December 2006, Palm (Hardware) paid \$44 million to ACCESS for the rights to the source code for Palm OS Garnet. With this arrangement, a single company is again developing palm hardware and software. Palm can modify the licensed software as needed and it need not pay royalties to ACCESS over future years.



4.3 FILLING OUR REVERSING LABORATORY

Fortunately for us, some tools are in existence (debugger ☺) which will make our life easier. We have the option to crack the application with dead listing or with the live approach. Also we have in our hands an emulator and we will load our programs.

1. Here is the approach that we will follow:
2. Choosing which PalmOS we will use
3. Install the software on our emulator
4. Wait till finishes
5. Find the correct patch
6. Change correctly the bytcodes, later more info
7. Test the application by loading into the emulator(or real Palm)

Repack the new .prc file to distribute our patched application

4.3.1 EXISTING TOOLS TO HANDLE PRC FILES

SouthDebugger: This debugger is Java-based and specifically designed for the Palm OS. This program has a memory dump, trap breakpoints and a notes section. You can log addresses and register values. It will become our Olly for Palm.

- PRCEdit: PRCEdit is a graphical frontend for pilotdis and splitprc.
- PRCEdit has an own splitprc built in but it still can use the original program.
- PilotDis: This Disassembler was originally created to overcome the "Crash" problem that "Pildis" had when disassembling certain .prc files.
- Debuffer: We can analyze the alive code. Something like Numega Softice, but freeware edition.
- Palm OS emulator: We will load the applications to test like being our real palm.



4.3.2 PRC FILE STRUCTURE

Pocket Smalltalk generates .PRC (Pilot resource database) files which can be transferred to a PalmPilot for execution. Resource databases consist of a list of arbitrary-length chunks of binary data, with each chunk being tagged by a **resource type** and a **resource ID**. This document describes the resource types used to encode the different parts of a Pocket Smalltalk program.

Since the compiled .PRC file contains everything needed for execution of the Pocket Smalltalk program, there will be many resource records present in addition to those representing actual compiled Smalltalk source code. These records include the machine language virtual machine, the icon used in the application launcher, some PalmOS GUI resources, and so on. These resources will not be described here.

At the time of this writing, the Smalltalk portion of the compiled .PRC file will consist of one or more of each of the following resource record types (some types are optional):

- **ClsN** - class name data [optional]
- **ClsO** - class offsets
- **ClsD** - compiled class and method data
- **ObjD** - statically defined objects
- **SelD** - symbol text data [optional]
- **SysP** - system properties

The resource record type is simply a 4-byte integer which can be conveniently represented as an ASCII string. The resource record abstraction is supported by the PalmOS firmware.

Because of the possibility of memory fragmentation in the PalmPilot, each record is kept to a small size (usually under 8k). If there is more data of a particular type than can fit in a single record, the data is broken up into chunks and each chunk is assigned a consecutive resource ID. This is handled transparently by the virtual machine.

4.3.3 THE CLSD RECORD

The most important segment type is the **ClsD** type. It contains the compiled form of classes and methods and will usually make up the bulk of a compiled program. It has a somewhat complex layout, but it is logical and efficient. At the highest level, it consists of a header followed by a body. The header has the following structure:

- A 2-byte integer with the total number of compiled classes.
- For each compiled class, a 2-byte integer with the number of bytes taken by the compiled form of each class.

Following this header are the compiled forms of each class. Note: if the **ClsD** segment needs to be split to avoid fragmentation problems, the split will always occur between the end of one compiled class and the beginning of another.

The compiled form of a class is, basically, a method dictionary. It encodes the selector/method associations of a class. First are two bytes with the number of selectors defined in the class (not counting selectors defined in



any superclasses). After this are pairs of 16-bit integers, with the first integer in each pair containing a selector defined in the class, and the second integer in each pair containing the byte offset to the method for that selector (the offset is from the start of the compiled class data, i.e., from the number-of-selectors word above).

The selectors are sorted in ascending order. When looking up a method at runtime, a binary search is performed (in contrast to the hash table mechanism used by other Smalltalks). In general, the binary search is just as fast as a hash table (and faster for classes with large numbers of methods), and it saves space because there are no unused slots.

After the selector/offset table come the instructions of the methods, one after the other. The methods are completely linear; in other words, no "literal table" is needed. Therefore compiled methods need no extra structure and need not be scanned by the garbage collector at runtime.

4.3.4 PALM OS OPERATING SYSTEM

In the world of win32 reversing on x86 architectures we have registers like EDX,EAX,ESP etc. Palm have 8 data registers, these are: D0, D1, D2, D3, D4, D5, D6 and D7. We also have 8 address registers: A0, A1, A2, A3, A4, A5, A6 and A7, where A7 (USP/SSP) works as a stack pointer. We are dealing with 68k assembly which has a different instruction set, and the syntax is opposite of windows.

Examples for further comprehension:

```
MOV destination, source (Intel) corresponds to MOV source, destination
```

```
MOV EDX, ECX (Intel) corresponds to MOV ECX, EDX
```

Another thing you should be aware of is that on Palm the code "branches", while on windows it "jumps". The only real difference is the name of the instructions.

4.3.5 OTHER TOOLS

PRCExplorer: Using this tool you can open any PRC file and navigate into it. You can see individual resources, graphically display Forms and Alerts, Bitmaps, Fonts, ... This is useful to debug and develop your application as well as analyzing PRCs.

Palm Debugger: Our second debugger. Manage all PDB data and Resources on-line from Palm device. Ideal for debugging, analyze and fix problems. Get information about: Device resources: view statistics, software versions, etc. DataBases: view, Sort and Print all data, size, version, and backup and records information.

You can Add, Delete, Modify or Search specific data. Applications: analyze structure, size, version, related databases, date info and much more!

X-Master: X-Master is an extension manager similar to HackMaster, but it is designed to offer various technical and user-interface improvements. It is fully compatible with HackMaster, all existing Hacks are supported, and

switching sets of active extensions is supported. Version 1.5 adds an improved interoperability with locking apps like OnlyMe and more compatibility fixes. It will help a lot our prime debugger.

4.3.6 INSTALLING THE TOOLS

4.3.6.1 INSTALLING EMULATOR

We extract emulator from the zip/rar archive and we open Emulator_Profile.exe.

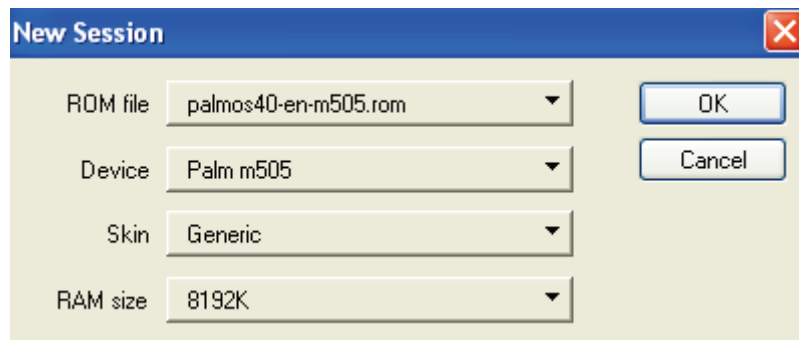


Figure 10 - Start Screen of emulator

Unfortunately, the package does not include the ROM file which you should download also. After pressing ok we see how our palm is.

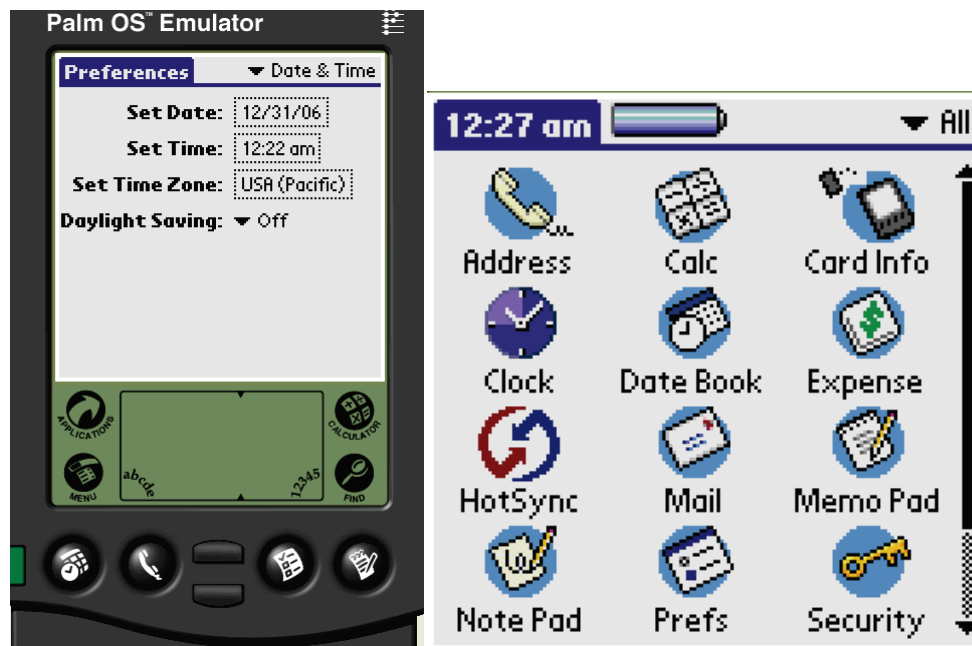




Figure 11 - The emulator and the menu

By pressing the applications button we see the menu (Figure 11). The menu only contains the starting edition applications. The installation of the application now is simple. You just drag and drop them from your HDD.

4.3.6.2 INSTALLING PRC EDIT

We want to have a powerful Disassembler in our hands and in order to we need few additional things. Open it go to options and you choose Disassembler options. Be careful with the installation of PilotDis because it can cause some problems with long directories (see my configuration in Figure 12).

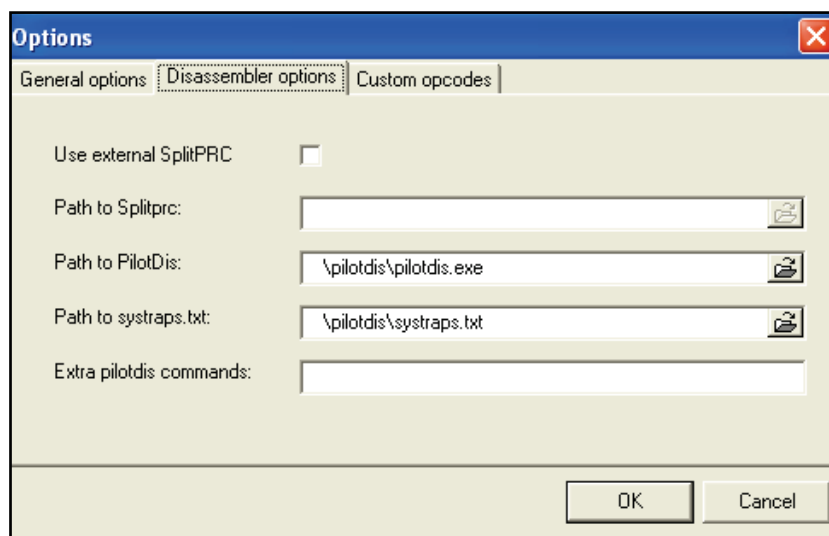


Figure 12 - Installation of pilotdis to prcredit

As you see I have extracted somewhere on my HDD and i have make a link to them. Now our Disassembler is ok so let's start reversing...

4.4 REVERSING WITH PRCEXPLORER AND PRREDIT

The moment has come to reverse a real application so we can have a better knowledge of these 2 great tools. Basically the approach we will follow is the "dead listing" approach (we are going to use Disassembler). The first application is a commercial one that has an easy protection. Moreover you are protected from viruses ;)

4.4.1 REVERSING THE FIRST APPLICATION: F-SECURE ANTIVIRUS 2.00

I have opened my Palm emulator and I have installed F-Secure antivirus (installation is done with by dragging and dropping the .prc to the emulator screen).



Figure 13 - the installed Antivirus

I open the application and I take a nag screen (see Figure 13) that my program expires in 14 days.



Figure 14 - the limitation

Unfortunately there is no registration button. So we are going to remove the nag screen with the trial period. Our next move is to open PRCEXplorer so we can check what information we can obtain. After we had opened file we find ourselves here:

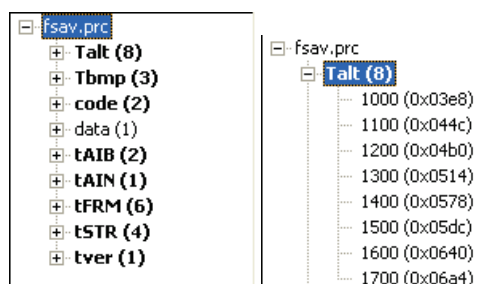


Figure 15 - The contents of the file

We are currently interested in the Talt section so click on that (Talt are resources which give important information about target). Hopefully we can see it has only 8 resources which is good for us. A bit of searching gives us an important resource

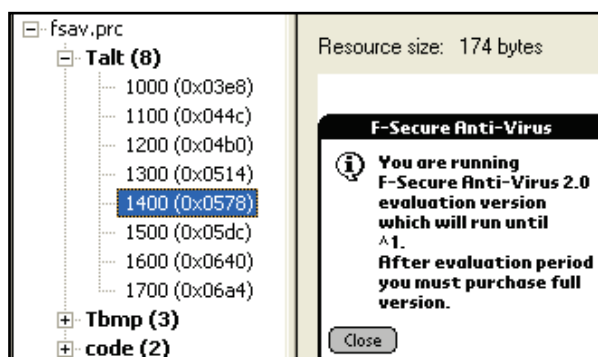


Figure 16 - Talt 1400 period of evaluation nag

We continue our search and we find another important resource

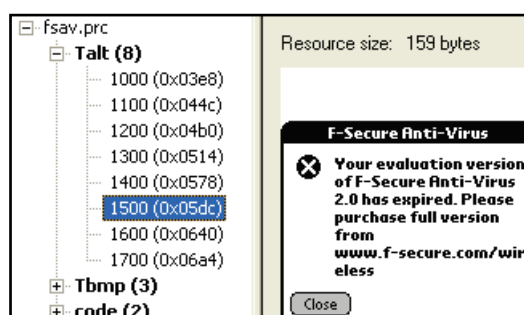


Figure 17 - Talt 1500 nag application expired

So what is happening here? As we see in resource 1400 we see the starting nag screen about the trial/evaluation period. In resource 1500 we find the expired message nag. We have obtained all our valuable information so we can close PRCE Explorer and we can go on with our Disassembler PRCEdit. Sometime it will complain about opening other files. You have to press “yes to all” and we find ourselves here:



Figure 18 - What I'm seeing in our Disassembler

Now we have to go to alerts and search for resource 1400 and 1500

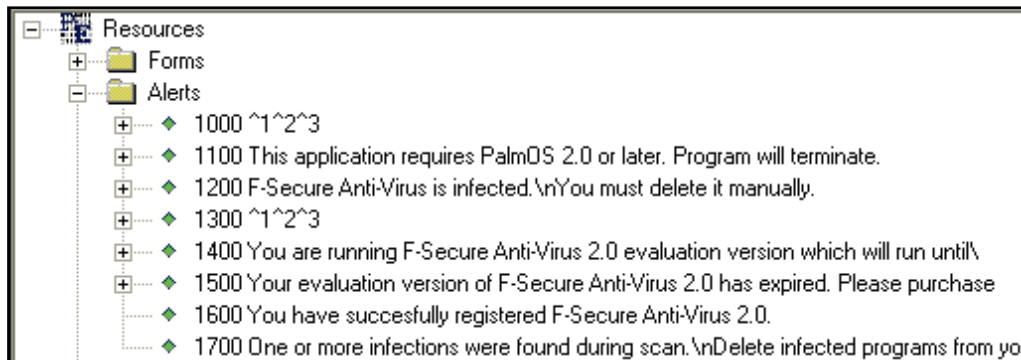


Figure 19 - Talt 1400 and 1500

As we can see all the nags/alerts are here. At first we will go to talt 1400. We see we can extend the tree (if you cannot you have not done correctly the configuration of PilotDis)

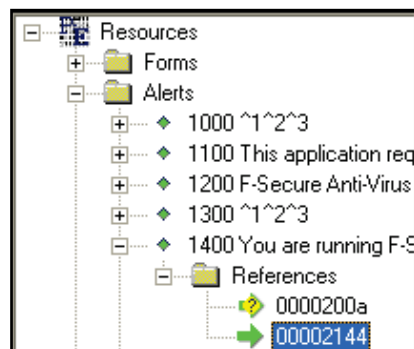


Figure 20 - The important reference

Remember that for this program we will need the references only with the green arrow. We can see the right window with the disassembled code. I am completely sure you do not understand anything

```

00002144  3f3c0578          MOVE.W  #1400!$578,-(A7)
00002148  4e4fa194          sysTrapFrmCustomAlert
0000214c  4fef001e          LEA    30(A7),A7
00002150  4227             L282      CLR.B  -(A7)
00002152  48780008          PEA    $0008.W
00002156  486de998          PEA    -5736(A5)
0000215a  4e4fa027          sysTrapMemSet

```

Figure 21 - The nag

The highlighted command is the point where our nag is appeared (the starting nag). We have to disappear it. But how we are going to achieve that? I have already made a list with valuable opcodes.(it is included with the 68k in the supplements of this package). I'm going to explain the code but you need to study the commands and the opcodes.

We have to remember the opcode the nag appears:

OPCODE	HEX VALUE
MOVE.W #1400!\$578,-(A7)	3F3C0578
sysTrapFrmCustomAlert	4E4FA194

Now we continue with the expired nag which has 3 references. We follow the reference 000020ea. If we think logically we can understand that something decides if that nag will be executed or not. That must be a jump (here branch). So we have to scroll up a little.

```

000020de  670a          BEQ    L279
000020e0  5340          SUBQ.W #1,D0
000020e2  676c          BEQ    L282
000020e4  5340          SUBQ.W #1,D0
000020e6  6712          BEQ    L280
000020e8  6066          BRA    L282
000020ea  3f3c05dc     L279    MOVE.W #1500!$5dc,-(A7)

```

Figure 22 - Important code

If you get familiarized with the code you can see that also Figure 22 shows us the jump. It is L279.

```

000020de  670a          BEQ    L279
000020e0  5340          SUBQ.W #1,D0
000020e2  676c          BEQ    L282
000020e4  5340          SUBQ.W #1,D0
000020e6  6712          BEQ    L280
000020e8  6066          BRA    L282
000020ea  3f3c05dc     L279    MOVE.W #1500!$5dc,-(A7)

```

Figure 23 - The jump that executes the command

The BEQ= Branch if Equal and it works like JE in Win32 platform. I am sure that you have thought already a few ways of solving it but we are going to just keep it

OPCODE	HEX VALUE
BEQ L279	670a

And we continue with the next reference which sends us here

00002104	544f	ADDQ.W #2,A7
00002106	6710	BEQ L281
00002108	3f3c05dc	MOVE.W #1500!\$5dc,-(A7)
0000210c	4e4fa192	sysTrapFrmAlert

Figure 24 - Third Important location

As we can see above this command there is a jump (branch) which decides if the nag appears. If we change it it will not affect us.

OPCODE	HEX VALUE
6710	BEQ L281

We have explained above how BEQ works so we need to change it to jump always. Ok we have finished with our nags. It's time to go and patch the program now.

4.4.1.1 PATCHING THE TARGET

It's time to make the changes to our target. Go to fsav.prc window

fsav.prc	fsav.rcp	fsav.bin.s	GCR 1	LCR 1	
0x0000:	6673	6176	0000	0000	fsav.....
0x0010:	0000	0000	0000	0000
0x0020:	0001	0002	B819	8227	...E.'E.'...
0x0030:	0000	0000	0000	0000appl
0x0040:	4653	4156	0000	0000	FSAV.....co

Figure 25 - The start of the code

And now we are going to find our opcodes. We need to search and find the first opcode so do a Ctrl+F and fill the opcode 3F 3C 05 78 4E 4F A1 94

```

0x2240: 3003 673C 5340 670A 5340 676C 5340 6712 0.g<S@g.S@g1S@g.
0x2250: 6066 3F3C 05DC 4E4F A192 303C 3039 544F `f?<.áNO~`0<09TO
0x2260: 606A 3F3C 000E 4EBA 0286 3600 544F 6710 `j?<..NI.¡6.TOg.
0x2270: 3F3C 05DC 4E4F A192 303C 1538 544F 604C ?<.áNO~`0<.8TO`L
0x2280: 486E FF88 4EBA 0392 1F3C 0007 2F2E FF8C Hn•|NI.´.<../.•|
0x2290: 1F3C 0019 486E FFE6 4EBA 0678 41FA 0034 .<..Hn•ζNI.xAv.4
0x22A0: 4850 41FA 0030 4850 486E FFE6 3F3C 0578 HPAv.OHPHn•ζ?<.x
0x22B0: 4E4F A194 4FEF 001E 4227 4878 0008 486D NO~|Oo..B'Hx..Hm

```

Figure 26 - Original code

We do not see any jump affecting us so the only thing we can do is to NOP the messagebox.

```

0x2240: 3003 673C 5340 670A 5340 676C 5340 6712 0.g<S@g.S@g1S@g.
0x2250: 6066 3F3C 05DC 4E4F A192 303C 3039 544F `f?<.áNO~`0<09TO
0x2260: 606A 3F3C 000E 4EBA 0286 3600 544F 6710 `j?<..NI.¡6.TOg.
0x2270: 3F3C 05DC 4E4F A192 303C 1538 544F 604C ?<.áNO~`0<.8TO`L
0x2280: 486E FF88 4EBA 0392 1F3C 0007 2F2E FF8C Hn•|NI.´.<../.•|
0x2290: 1F3C 0019 486E FFE6 4EBA 0678 41FA 0034 .<..Hn•ζNI.xAv.4
0x22A0: 4850 41FA 0030 4850 486E FFE6 4E71 4E71 HPAv.OHPHn•ζNqNq
0x22B0: 4E71 4E71 4FEF 001E 4227 4878 0008 486D NqNqOo..B'Hx..Hm

```

Figure 27 - Modified Code

We continue with the second location. This time search for 67 0A. We land on here

```

0x2240: 3003 673C 5340 670A 5340 676C 5340 6712 0.g<S@g.S@g1S@g.
0x2250: 6066 3F3C 05DC 4E4F A192 303C 3039 544F `f?<.áNO~`0<09TO
0x2260: 606A 3F3C 000E 4EBA 0286 3600 544F 6710 `j?<..NI.¡6.TOg.
0x2270: 3F3C 05DC 4E4F A192 303C 1538 544F 604C ?<.áNO~`0<.8TO`L
0x2280: 486E FF88 4EBA 0392 1F3C 0007 2F2E FF8C Hn•|NI.´.<../.•|
0x2290: 1F3C 0019 486E FFE6 4EBA 0678 41FA 0034 .<..Hn•ζNI.xAv.4
0x22A0: 4850 41FA 0030 4850 486E FFE6 4E71 4E71 HPAv.OHPHn•ζNqNq
0x22B0: 4E71 4E71 4FEF 001E 4227 4878 0008 486D NqNqOo..B'Hx..Hm

```

Figure 28 - Highlighted=original code

This jump needs to be noped because there are jumps after that they will not show us the nags.

```

0x2240: 3003 673C 5340 4E71 5340 676C 5340 6712 0.g<S@NqS@g1S@g.
0x2250: 6066 3F3C 05DC 4E4F A192 303C 3039 544F `f?<.áNO~'0<09TO
0x2260: 606A 3F3C 000E 4EBA 0286 3600 544F 6710 `j?<..NI.¡6.TOg.
0x2270: 3F3C 05DC 4E4F A192 303C 1538 544F 604C ?<.áNO~'0<.8TO`L
0x2280: 486E FF88 4EBA 0392 1F3C 0007 2F2E FF8C Hn•INI.'.<../.•I
0x2290: 1F3C 0019 486E FFE6 4EBA 0678 41FA 0034 .<..Hn•ζNI.xAv.4
0x22A0: 4850 41FA 0030 4850 486E FFE6 4E71 4E71 HPAv.OHPHn•ζNqNq
0x22B0: 4E71 4E71 4FEF 001E 4227 4878 0008 486D NqNqOo..B'Hx..Hm

```

Figure 29 - Second patch done

The only we need to do now is to find the third location to patch which is here

```

0x2240: 3003 673C 5340 4E71 5340 676C 5340 6712 0.g<S@NqS@g1S@g.
0x2250: 6066 3F3C 05DC 4E4F A192 303C 3039 544F `f?<.áNO~'0<09TO
0x2260: 606A 3F3C 000E 4EBA 0286 3600 544F 6710 `j?<..NI.¡6.TOg.
0x2270: 3F3C 05DC 4E4F A192 303C 1538 544F 604C ?<.áNO~'0<.8TO`L
0x2280: 486E FF88 4EBA 0392 1F3C 0007 2F2E FF8C Hn•INI.'.<../.•I
0x2290: 1F3C 0019 486E FFE6 4EBA 0678 41FA 0034 .<..Hn•ζNI.xAv.4
0x22A0: 4850 41FA 0030 4850 486E FFE6 4E71 4E71 HPAv.OHPHn•ζNqNq
0x22B0: 4E71 4E71 4FEF 001E 4227 4878 0008 486D NqNqOo..B'Hx..Hm

```

Figure 30 - The third location original code

We need to modify this jump from BEQ to BRA so it is always jumping..

```

0x2240: 3003 673C 5340 4E71 5340 676C 5340 6712 0.g<S@NqS@g1S@g.
0x2250: 6066 3F3C 05DC 4E4F A192 303C 3039 544F `f?<.áNO~'0<09TO
0x2260: 606A 3F3C 000E 4EBA 0286 3600 544F 6010 `j?<..NI.¡6.TOg.
0x2270: 3F3C 05DC 4E4F A192 303C 1538 544F 604C ?<.áNO~'0<.8TO`L
0x2280: 486E FF88 4EBA 0392 1F3C 0007 2F2E FF8C Hn•INI.'.<../.•I
0x2290: 1F3C 0019 486E FFE6 4EBA 0678 41FA 0034 .<..Hn•ζNI.xAv.4
0x22A0: 4850 41FA 0030 4850 486E FFE6 4E71 4E71 HPAv.OHPHn•ζNqNq
0x22B0: 4E71 4E71 4FEF 001E 4227 4878 0008 486D NqNqOo..B'Hx..Hm

```

Figure 31 - Fully patched code

Now save your file with the modifications. I saved it as fsav.patched.prc

4.4.1.2 TESTING THE TARGET

Load your palm and install the patched target.



Figure 32 - Patched target loaded

Now open so we can test our work..

We fully patched the application...Congratulations

4.5 REVERSING WITH POSE AND SOUTHDEBUGGER

This time we will reverse another commercial application but with a completely different way. We will use the live approach.

4.5.1 REVERSING THE SECOND APPLICATION: TEALDOC 6.77

Install application and see how it behaves

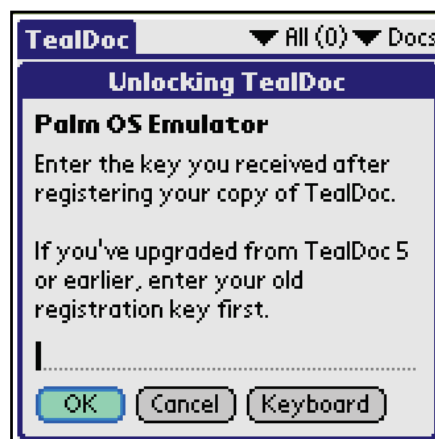


Figure 33 - Registration Box

To find the registration you have to click menu in your palm OS emulator when you have loaded the program and you will see the register option. Let's try registering it..

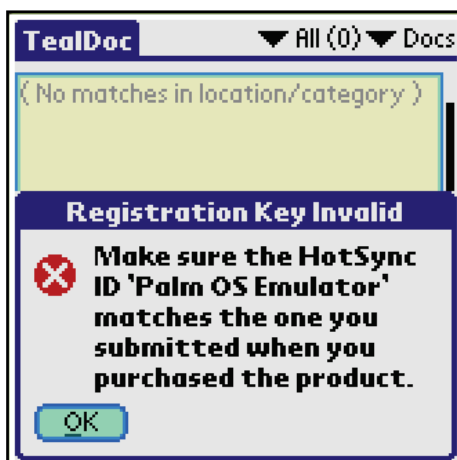


Figure 34 - Invalid registration

It's time to run our Debugger. His name is SouthDebugger and he is a shareware for unlimited time. It will replace Olly in Palm. To run South Debugger you need the java environment. SouthDebugger is being developed in java so you click on the jar file to run him.

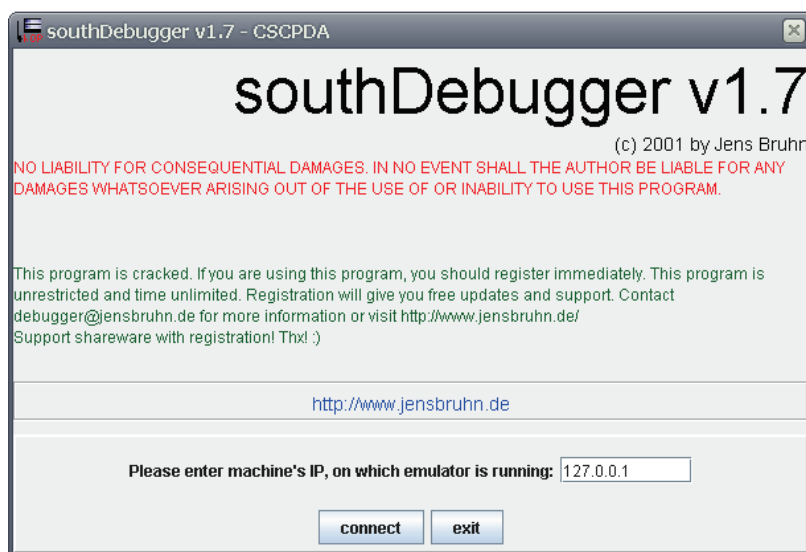


Figure 35 - SouthDebugger Start Screen

SouthDebugger has reached edition 1.7. It has an annoying nag and the “unregistered” word at the top bar. Well we could reverse it because I do not like unregistered tools but it is not in the purpose of this tutorial (moreover CSCPDA already patched it). SouthDebugger needs help from another program called X-Master. There are some events that are generated by the program. It is X-Master that generates these events and keeps POSE blocked. Think what happens to Windows and Olly during debug. Also SouthDebugger connects to the machine's IP. So press “connect” and goes to X-Master.



Figure 36 - X-Master Installed

So we have installed X-Master (I hope you remember that it is done by drag and drop). Load it in POSE and SouthDebugger becomes green. When it becomes green we execute POSE.

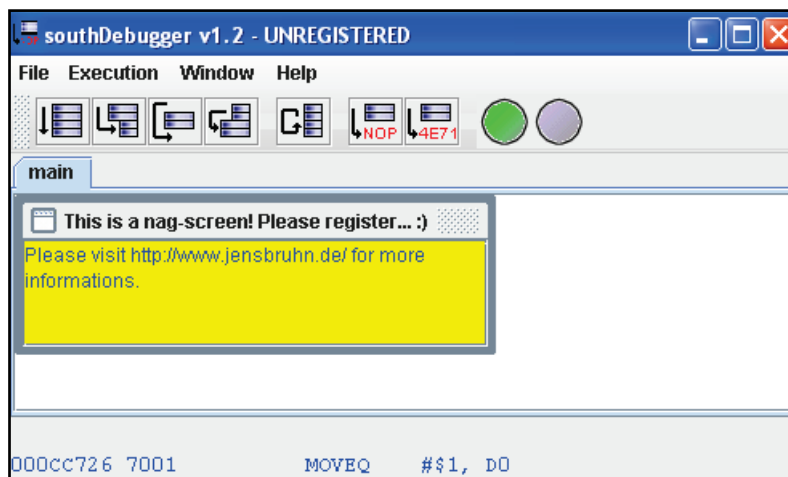


Figure 37 - SouthDebugger

Now we have to press the execute button. It is the F5

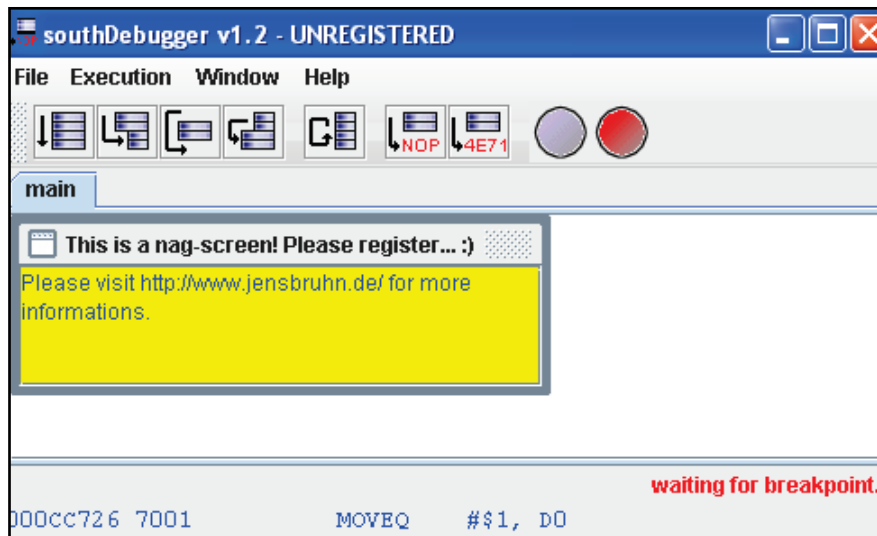


Figure 38 - SouthDebugger Executed

And we see in the POSE screen. The red light in the debugger means that the debugger is not in "contact" with the emulator, while the green one means we have "contact". As long as the green light is on we can't do anything in the emulator, we can set breakpoints in the debugger.



Figure 39 - The reset

Before this you may see a notice for restarting X-Master. Just press ok.

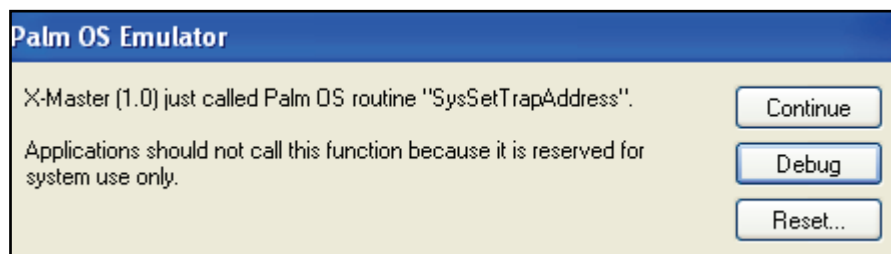


Figure 40 - The Error

This is an error created by application. The X-MASTER acceded to the SysSetTrapAddress function. Press continue.. And we continue reversing. What are we going to do? We will obtain a serial for our name. How is this done? This is done by some functions. In Win32 platform we have the api like GetDlgItemTextA, GetWindowTextA. That palm functions do not have names in windows but their names can describe well their

functions. A function that almost always usually works in PALM is sysTrapSrtCompare that as they will occur account serves to compare text chains. We go in SouthDebugger → Window → New window breakpoint.

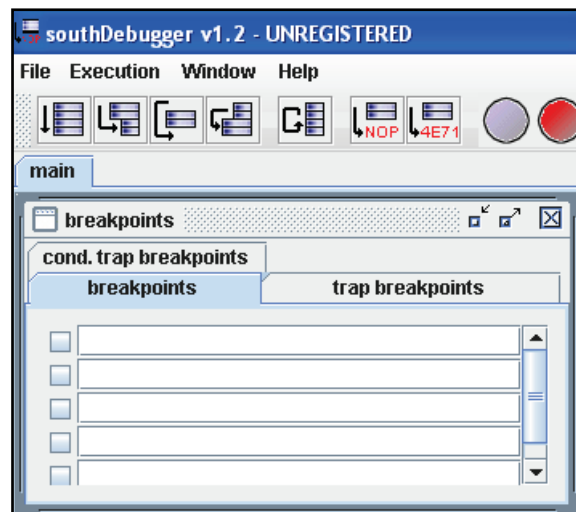


Figure 41 - Breakpoints Window

We are not interested breakpoints but we want trap breakpoints.

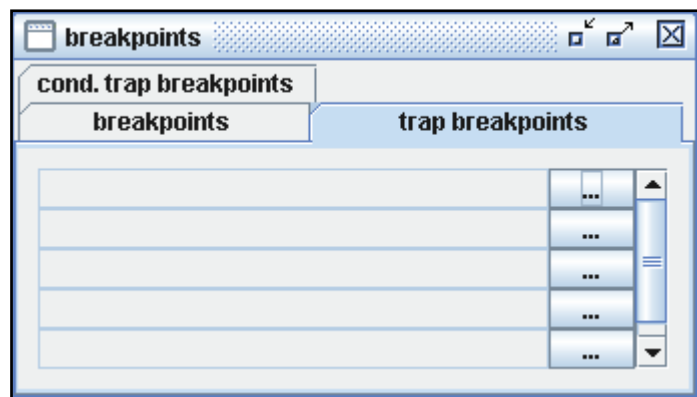


Figure 42 - Trap Breakpoints

By getting into to the X-Master the breakpoints are being “not set”.

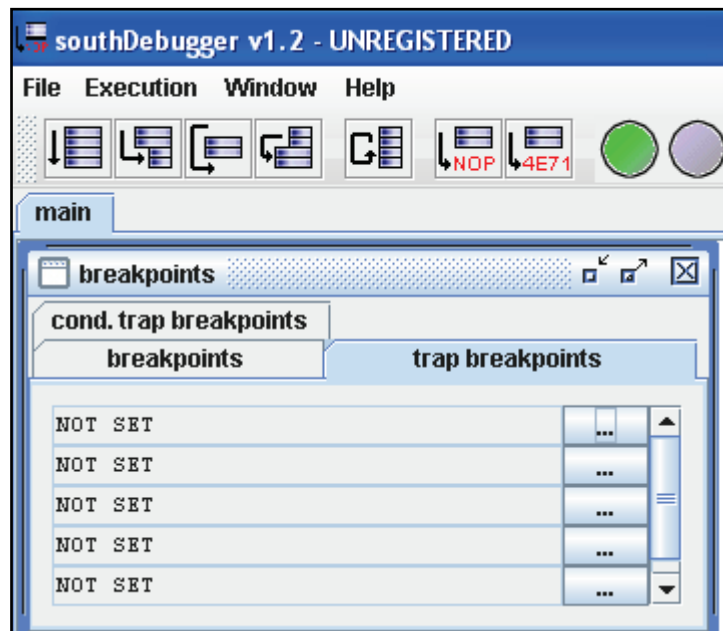


Figure 43 - "Not Set" condition

The trap breakpoints are a lot but for this one application we will use SysTrapStrCompare. I am quite sure that you will be asking yourselves why I used that. It is a common breakpoint in Palm that checks for the serial. Remember that the only way to learn the breakpoints is experience.

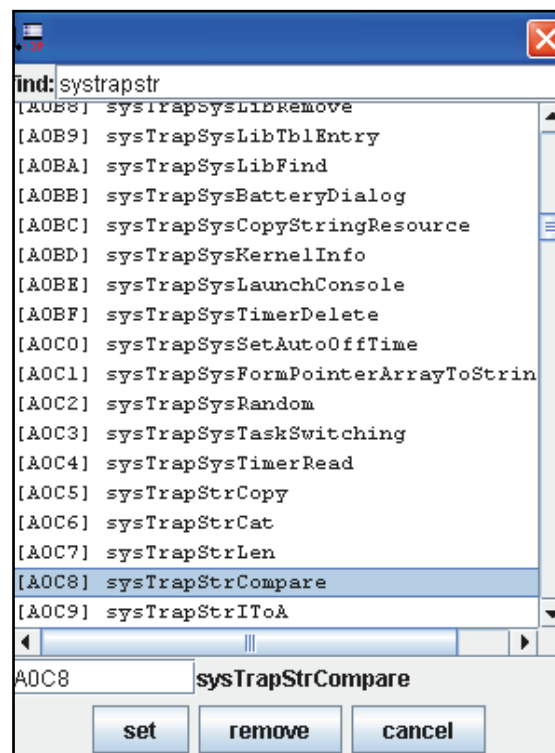


Figure 44 - Breakpoint List

Only we need to do is click set (it is like setting a breakpoint in Olly with command bar).

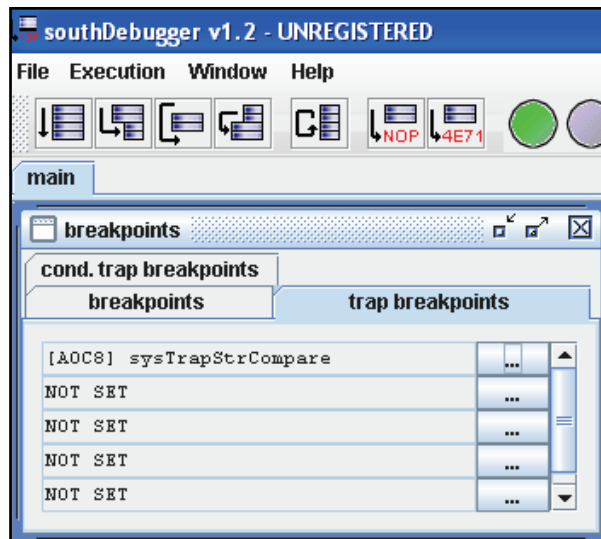


Figure 45 - Breakpoint is Set

We click F5 (execute in SouthDebugger) and we are in the window of X-Master. We return to the applications and we get into the TealDoc Options and we register again until it breaks on SysTrapStrCompare.

NOTE: If the SouthDebugger stops before arriving at the register window then you press F5 until you reach there.

In this program you have to press many times F5..

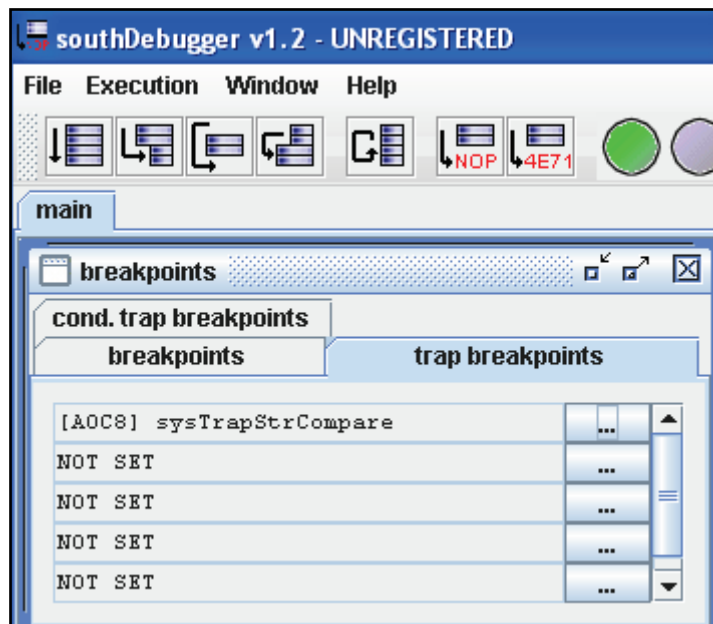


Figure 46 - Green Light

Finally we broke on registration. The bar of SouthDebugger informs us where the breakpoint has stopped.

```
0007AB0E 4E4FA0C8      TRAP      #15!$F, #41160!$A0C8 [sysTrapStrCompare]
```

Figure 47 - Our breakpoint

In our case the breakpoint has stopped in 7AB0E and the opcodes of this instruction are 4E4FA0C8. The last part is the instruction. Now we have to go to this place we click on the button with the icon of NOP.

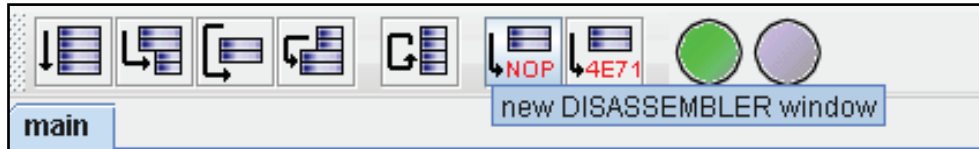


Figure 48 - Second button right to left

Now we can see the part of the comparison.

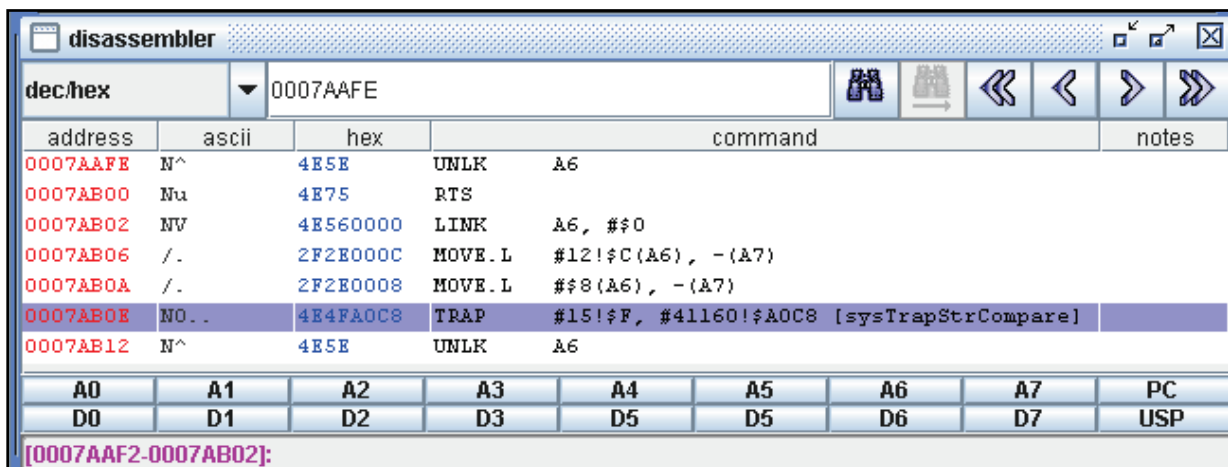


Figure 49 - Compare routine

Now we have to load trap stack which will give all the important information.

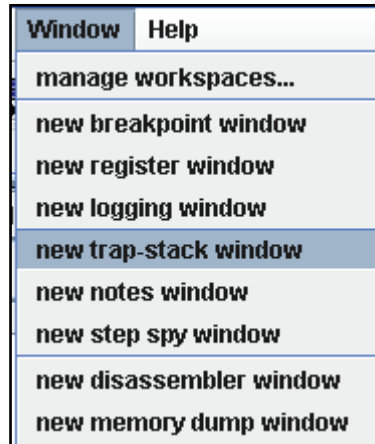


Figure 50 - Trap Stack Window

Now we click and we open trap stack window and we see that our serial is compared with the string debug (Figure 51). I have the sense that it is not what exactly we want.

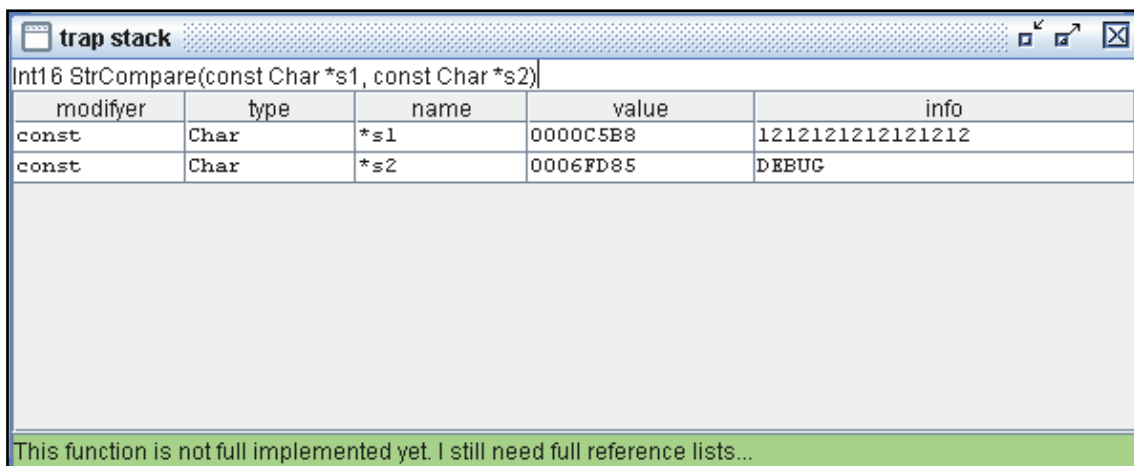


Figure 51 - Our serial compared

Well I do not believe that our correct serial may be the string DEBUG. So press once more F5.

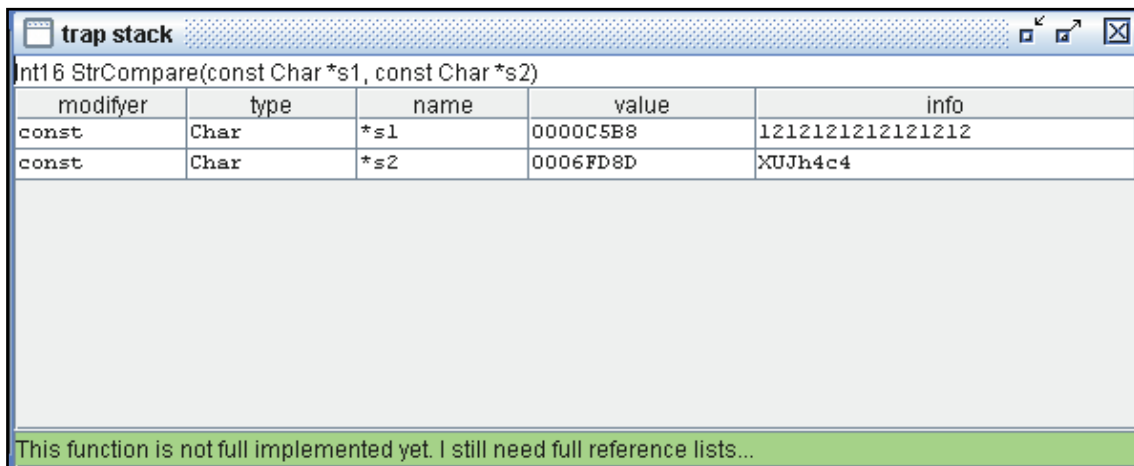


Figure 52 - Again our serial compared....

Now our serial is compared with the string XUJh4c4. Is it our real serial?

4.5.1.1 TESTING THE TARGET

Press F5 and go to the registration box..

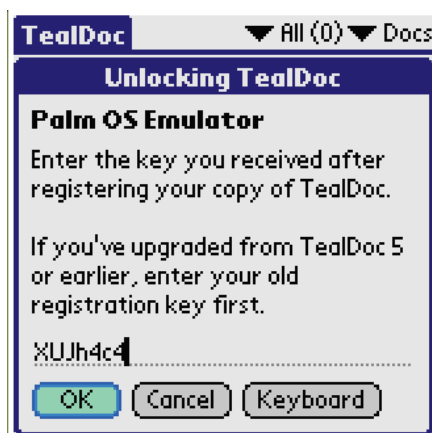


Figure 53 - Serial we found

Press ok and see

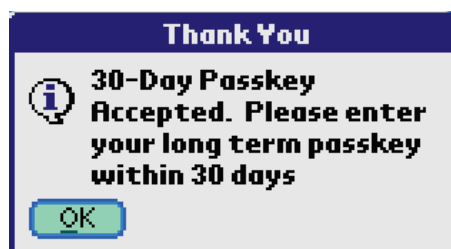


Figure 54 - Registered

Congratulations...We cracked that application and we can use it forever...

4.6 ADVANCED REVERSING

In this chapter of tutorial we will discuss applications which have a more complex protection system and they need more thinking from the other applications. In my opinion we will get a clear image of what is happening to Palm OS. We will use a combination of tools to reach in the final result. If you want to follow this tutorial be sure you have understand the approaches used in the previous chapters.

4.6.1 REVERSING THE THIRD APPLICATION: 300 BOWL

We examine the application to see the limitations

```
You are playing an unregistered
version of 300 Bowl. You will
be limited to 20 runs (exiting
to another palm application and
restarting 300 Bowl counts as
a run) and allows 5 frame
games, one player or versus
the computer.

You may play as many 5 frame
games as you like each run.
```

Figure 55 - Limitations

We fill the registration box



Figure 56 - Invalid Registration

We open PRCEXplorer and we look for the talt. Fortunately they are only 14..

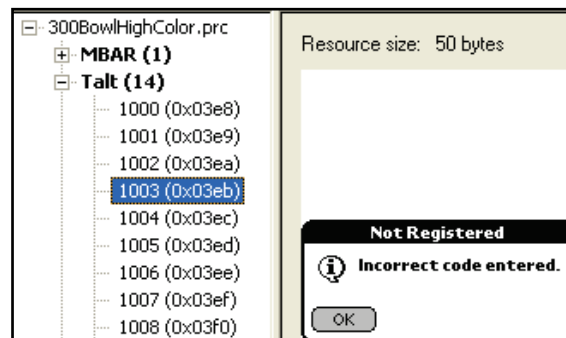


Figure 57 - Talt with invalid registration nag

So we close the PRCE Explorer and we open PRCEdit. Remember to say “yes to all” to Disassembler window.

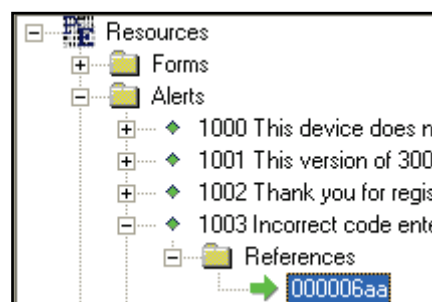


Figure 58 - The reference

We follow to the right window so we can check the code

00000684	4e90		JSR	(A0)	
00000686	4a2a000c		TST.B	12(A2)	
0000068a	671e		BEQ	L55	
0000068c	3f3c03ea		MOVE.W	#1002!\$3ea,-(A7)	
00000690	4e4fa192		sysTrapFrmAlert		
00000694	3f3c03e8		MOVE.W	#1000!\$3e8,-(A7)	
00000698	4e4fa19b		sysTrapFrmGotoForm		
0000069c	41edfb74		LEA	-1164(A5),A0	
000006a0	117c00010001		MOVE.B	#1,1(A0)	
000006a6	6000ff34		BRA	L52	
000006aa	3f3c03eb	L55	MOVE.W	#1003!\$3eb,-(A7)	
000006ae	4e4fa192		sysTrapFrmAlert		
000006b2	6000ff28		BRA	L52	
000006b6	4240	L56	CLR.W	D0	

Figure 59 - The code

That's all the important code (I'll copy and paste so I can comment on him)

```

00000684 4e90 JSR (A0) ;verification routine
00000686 4a2a000c TST.B 12(A2) ;registration ok?
0000068a 671e BEQ L55 ;not. follow L55
0000068c 3f3c03ea MOVE.W #1002!$3ea,-(A7) ;if registration ok go on

```

```

00000690 4e4fa192      sysTrapFrmAlert ;show talt window registration fine
00000694 3f3c03e8      MOVE.W #1000!$3e8,-(A7)
00000698 4e4fa19b      sysTrapFrmGotoForm
0000069c 41edfb74      LEA    -1164(A5),A0
000006a0 117c00010001  MOVE.B #1,1(A0)
000006a6 6000ff34      BRA    L52
000006aa 3f3c03eb      L55    MOVE.W #1003!$3eb,-(A7) ;if registration invalid go on
000006ae 4e4fa192      sysTrapFrmAlert ;show talt window registration invalid
000006b2 6000ff28      BRA    L52
000006b6 4240         L56    CLR.W  D0

TST.B 12(A2) ;tests byte to be A2+12
BEQ L55      ;it goes to L55 if test ok(A2+12=0)

```

We go to SouthDebugger and we press connect. Then we go to trap breakpoints and they are in “not set” condition. Remember the X-Master plugin we used and the whole approach to reach the “not set” condition.

It should be followed again. And you should reach here

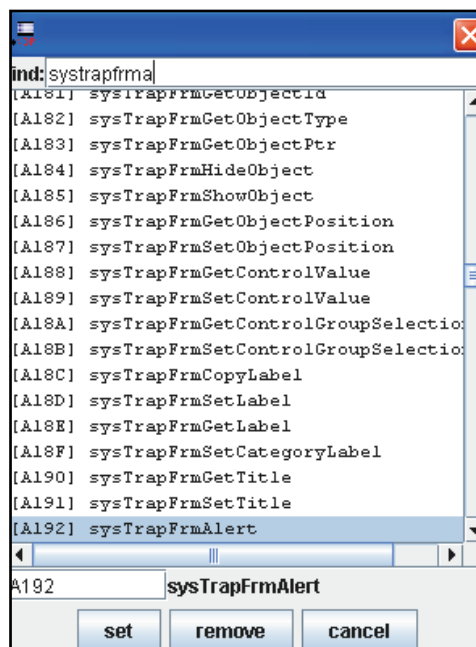


Figure 60 - Breakpoint List

We follow the previous method with the reset of the Palm OS and continuous breaks till we reach the registration button. We break here:

```
00082D96 4E4FA192 TRAP #15!$F, #41362!$A192 [sysTrapFrmAlert]
```

Figure 61 - Location we broke

00082D6C	N.	4E90	JSR	(A0)
00082D6E	J*	4A2A000C	TST.B	#12!\$C(A2)
00082D72	g.	671E	BEQ	#30!\$1E ;00082D92
00082D74	?<..	3F3C03EA	MOVE.W	#1002!\$3EA, -(A7)
00082D78	NO..	4E4FA192	TRAP	#15!\$F, #41362!\$A192 [sysTrapFrmAlert]
00082D7C	?<..	3F3C03E8	MOVE.W	#1000!\$3E8, -(A7)
00082D80	NO..	4E4FA19B	TRAP	#15!\$F, #41371!\$A19B [sysTrapFrmGotoForm]
00082D84	A..t	41EDFB74	LEA	#-1164!\$-48C(A5), A0
00082D88	-	117C00010001	MOVE.B	#\$1, #\$1(A0)
00082D8E	`	6000FF34	BRA	#-204!\$-CC ;00082CC4
00082D92	?<..	3F3C03EB	MOVE.W	#1003!\$3EB, -(A7)
00082D96	NO..	4E4FA192	TRAP	#15!\$F, #41362!\$A192 [sysTrapFrmAlert]

Figure 62 - The important location (see Figure 59)

Now you have to set the breakpoint to the JSR (= call in Win 32) command so right and activate breakpoint.

00082D6C	N.	4E90	JSR	(A0)
00082D6E	J*	4A2A000C	TST.B	#12!\$C(A2)
00082D72	g.	671E	BEQ	#30!\$1E ;00082D92
00082D74	?<..	3F3C03EA	MOVE.W	#1002!\$3EA, -(A7)
00082D78	NO..	4E4FA192	TRAP	#15!\$F, #41362!\$A192 [sysTrapFrmAlert]
00082D7C	?<..	3F3C03E8	MOVE.W	#1000!\$3E8, -(A7)
00082D80	NO..	4E4FA19B	TRAP	#15!\$F, #41371!\$A19B [sysTrapFrmGotoForm]
00082D84	A..t	41EDFB74	LEA	#-1164!\$-48C(A5), A0
00082D88	-	117C00010001	MOVE.B	#\$1, #\$1(A0)
00082D8E	`	6000FF34	BRA	#-204!\$-CC ;00082CC4
00082D92	?<..	3F3C03EB	MOVE.W	#1003!\$3EB, -(A7)
00082D96	NO..	4E4FA192	TRAP	#15!\$F, #41362!\$A192 [sysTrapFrmAlert]

Figure 63 - New Breakpoint location

F5 and click on the invalid registration nag. Then click again register and you land on the location as in the Figure 63. Then click F8.

```
00080E48 4E56FFAC LINK A6, #-84!$-54
```

Figure 64 - New location

Here is our new location 00080E48. We go back till we find that address.

00080E48	NV..	4E56FFAC	LINK	A6, #-84!\$-54
00080E4C	H..8	48E71C38	MOVEM.L	A4/A3/A2/D5/D4/D3, -(A7)
00080E50	G..h	47EDF168	LEA	#-3736!\$-E98(A5), A3
00080E54	J-..	4A2DC90C	TST.B	#-14068!\$-36F4(A5)
00080E58	g.	671E	BEQ	#30!\$1E ;00080E78
00080E5A	J+	4A2B000C	TST.B	#12!\$(A3)
00080E5E	g.	6718	BEQ	#24!\$18 ;00080E78
00080E60	J-..	4A2DC90D	TST.B	#-14067!\$-36F3(A5)
00080E64	g	67000388	BEQ	#904!\$388 ;000811EE
00080E68	.m..	206DE9A4	MOVEA.L	#-5724!\$-165C(A5), A0
00080E6C	..	D1FC00000056	ADDA.L	#86!\$56, A7
00080E72	N.	4E90	JSR	(A0)
00080E74	`	60000378	BRA	#888!\$378 ;000811EE
00080E78	B.	42A7	CLR.L	-(A7)
00080E7A	B.	42A7	CLR.L	-(A7)
00080E7C	E...	45EEFFB0	LEA	#-80!\$-50(A6), A2
00080E80	/.	2FOA	MOVE.L	A2, -(A7)
00080E82	B.	42A7	CLR.L	-(A7)
00080E84	B.	42A7	CLR.L	-(A7)
00080E86	B.	42A7	CLR.L	-(A7)
00080E88	NO..	4E4FA2A9	TRAP	#15!\$F, #41641!\$A2A9 [sysTrapDlkGetSyncInfo]

Figure 65 - The important subroutine

We observed the routine which is called by sysTrapDlkSyncInfo that returns the hotsyncname. Palm has something called HotSyncID, it is in a way the name of your Palm. All Palms have an ID like this. The majority of programs have a registration routine where the serial is generated from your HotSyncID, so HotSyncID is kind of like a username. That means a serial working for your HotSyncID won't work on other Palms than yours. It's time to get back to the PRCEdit without closing SouthDebugger (I had closed it and broke on systrapdlkgetsynInfo). And we have to search the trap. So search for Getsync and you will find it in code0002.bin.s

00000a68	4e4fa2a9		sysTrapDlkGetSyncInfo
00000a6c	4243		CLR.W D3
00000a6e	4242		CLR.W D2
00000a70	4fef0018		LEA 24(A7),A7
00000a74	49edc90d		LEA -14067(A5),A4
00000a78	4a12		TST.B (A2)
00000a7a	673e		BEQ L75
00000a7c	224a		MOVEA.L A2,A1
00000a7e	41eeffd8		LEA -40(A6),A0
00000a82	12312000	L71	MOVE.B 0(A1,D2.W),D1
00000a86	1001		MOVE.B D1,D0
00000a88	0600ff9f		ADDI.B #-97!-\$61,D0
00000a8c	0c000019		CMPI.B #25!\$19,D0
00000a90	6208		BHI L72
00000a92	0601ffe0		ADDI.B #-32!-\$20,D1
00000a96	6000000e		BRA L73
00000a9a	1001	L72	MOVE.B D1,D0
00000a9c	0600ffbf		ADDI.B #-65!-\$41,D0
00000aa0	0c000019		CMPI.B #25!\$19,D0
00000aa4	6206		BHI L74
00000aa6	11813000	L73	MOVE.B D1,0(A0,D3.W)
00000aaa	5243		ADDQ.W #1,D3
00000aac	5242	L74	ADDQ.W #1,D2
00000aae	4a312000		TST.B 0(A1,D2.W)
00000ab2	6706		BEQ L75
00000ab4	0c42000b		CMPI.W #11!\$b,D2
00000ab8	6fc8		BLE L71

Figure 66 - The location

All the code in Figure 66 is of major importance. Here are the checks for our serial. In line 00000ab4 checks if our name is smaller than 12 characters. A few lines below we have the important check although. Also you should observe that we have many jumps in this part of code. Many checks? We will see..

00000ab4	0c42000b		CMPI.W #11!\$b,D2
00000ab8	6fc8		BLE L71
00000aba	0c430001	L75	CMPI.W #1,D3
00000abe	6e2a		BGT L76
00000ac0	1d7c0041ffd8		MOVE.B #65!\$41,-40(A6)
00000ac6	1d7c0042ffd9		MOVE.B #66!\$42,-39(A6)
00000acc	1d7c0043ffda		MOVE.B #67!\$43,-38(A6)
00000ad2	1d7c0044ffdb		MOVE.B #68!\$44,-37(A6)
00000ad8	1d7c0045ffdc		MOVE.B #69!\$45,-36(A6)
00000ade	1d7c0046ffdd		MOVE.B #70!\$46,-35(A6)
00000ae4	422effde		CLR.B -34(A6)
00000ae8	7606		MOVEQ #6,D3
00000aea	3403	L76	MOVE.W D3,D2
00000aec	0c42000b		CMPI.W #11!\$b,D2
00000af0	6e12		BGT L78
00000af2	41eeffd8		LEA -40(A6),A0
00000af6	11bc00422000	L77	MOVE.B #66!\$42,0(A0,D2.W)
00000afc	5242		ADDQ.W #1,D2
00000afe	0c42000b		CMPI.W #11!\$b,D2
00000b02	6ff2		BLE L77
00000b04	177c0001000c	L78	MOVE.B #1,12(A3)

Figure 67 - A check

Now we have to analyze the code down from L78. In this line we see the instruction MOVE.B #1,12 (A3). Although except for this “mov” there is one also...

00000b04	177c0001000c	L78	MOVE.B #1,12 (A3)
00000b0a	4a14		TST.B (A4)
00000b0c	6766		BEQ L85
00000b0e	0c530033		CMPI.W #51!\$33, (A3)
00000b12	6704		BEQ L79
00000b14	422b000c		CLR.B 12 (A3)
00000b18	0c6b00390002	L79	CMPI.W #57!\$39,2 (A3)
00000b1e	6704		BEQ L80
00000b20	422b000c		CLR.B 12 (A3)
00000b24	0c6b00340004	L80	CMPI.W #52!\$34,4 (A3)
00000b2a	6704		BEQ L81
00000b2c	422b000c		CLR.B 12 (A3)
00000b30	0c6b00360006	L81	CMPI.W #54!\$36,6 (A3)
00000b36	6704		BEQ L82
00000b38	422b000c		CLR.B 12 (A3)
00000b3c	0c6b00300008	L82	CMPI.W #48!\$30,8 (A3)
00000b42	6704		BEQ L83
00000b44	422b000c		CLR.B 12 (A3)
00000b48	0c6b0038000a	L83	CMPI.W #56!\$38,10 (A3)
00000b4e	6704		BEQ L84
00000b50	422b000c		CLR.B 12 (A3)
00000b54	4a2b000c	L84	TST.B 12 (A3)
00000b58	671a		BEQ L85
00000b5a	1b7c0001c90e		MOVE.B #1,-14066 (A5)
00000b60	206de9a4		MOVEA.L -5724 (A5), A0
00000b64	d1fc00000056		ADDA.L #86!\$56, A0
00000b6a	4e90		JSR (A0)
00000b6c	6100fe70		BSR L67
00000b70	6000025c		BRA L95
00000b74	177c0001000c	L85	MOVE.B #1,12 (A3)

Figure 68 - Second mov instruction

These 2 instructions are the same. We can check it with SouthDebugger. Execute till the point I am with F7 (remember that F7 does not get into call and does not follow jumps). You must be here

00080E84	B.	42A7	CLR.L	-(A7)
00080E86	B.	42A7	CLR.L	-(A7)
00080E88	NO..	4E4FA2A9	TRAP	#15!\$F, #41641!\$A2A9 [sysTrapDlkGetSyncInfo]
00080E8C	BC	4243	CLR.W	D3
00080E8E	BB	4242	CLR.W	D2
00080E90	O.	4FEF0018	LEA	#24!\$18(A7), A7
00080E94	I...	49EDC90D	LEA	#-14067!\$-36F3(A5), A4

Figure 69 - Our location

From that location you have to trace again with F7 till here:

00080E8E	BB	4242	CLR.W	D2
00080E90	O.	4FEF0018	LEA	#24!\$18(A7), A7
00080E94	I...	49EDC90D	LEA	#-14067!\$-36F3(A5), A4
00080E98	J.	4A12	TST.B	(A2)
00080E9A	g>	673E	BEQ	#62!\$3E ;00080EDA
00080E9C	"J	224A	MOVEA.L	A2, A1
00080E9E	A...	41EEFFD8	LEA	#-40!\$-28(A6), A0
00080EA2	.l.	12312000	MOVE.B	#\$0(A1,D2), D1
00080EA6	..	1001	MOVE.B	D1, D0
00080EA8	.	0600FF9F	ADDI.B	#-97!\$-61, D0
00080EAC	.	0C000019	CMPI.B	#25!\$19, D0
00080EB0	b.	6208	BHI	#\$8 ;00080EBA
00080EB2	0601FFE0	ADDI.B	#-32!\$-20, D1
00080EB6	`	6000000E	BRA	#14!\$E ;00080EC6
00080EBA	..	1001	MOVE.B	D1, D0
00080EBC	.	0600FFBF	ADDI.B	#-65!\$-41, D0
00080EC0	.	0C000019	CMPI.B	#25!\$19, D0
00080EC4	b.	6206	BHI	#\$6 ;00080ECC
00080EC6	..0	11813000	MOVE.B	D1, #\$0(A0,D3)
00080ECA	RC	5243	ADDQ.W	#\$1, D3
00080ECC	RB	5242	ADDQ.W	#\$1, D2
00080ECE	Jl.	4A312000	TST.B	#\$0(A1,D2)
00080ED2	g.	6706	BEQ	#\$6 ;00080EDA
00080ED4	.B	0C42000B	CMPI.W	#11!\$B, D2
00080ED8	o.	6FC8	BLE	#-56!\$-38 ;00080EA2
00080EDA	.C	0C430001	CMPI.W	#\$1, D3
00080EDE	n*	6E2A	BGT	#42!\$2A ;00080FOA

Figure 70 - Executed code

From here you hit once the F8 because we want to follow that branch (=jump) and you land here:

00080EDE	n*	6E2A	BGT	#42!\$2A ;00080FOA
00080EB0	.	1D7C0041FFD8	MOVE.B	#65!\$41, #-40!\$-28(A6)
00080EB6	.	1D7C0042FFD9	MOVE.B	#66!\$42, #-39!\$-27(A6)
00080EBC	.	1D7C0043FFDA	MOVE.B	#67!\$43, #-38!\$-26(A6)
00080EF2	.	1D7C0044FFDB	MOVE.B	#68!\$44, #-37!\$-25(A6)
00080EF8	.	1D7C0045FFDC	MOVE.B	#69!\$45, #-36!\$-24(A6)
00080EFE	.	1D7C0046FFDD	MOVE.B	#70!\$46, #-35!\$-23(A6)
00080F04	B...	422EFFFDE	CLR.B	#-34!\$-22(A6)
00080F08	v.	7606	MOVEQ	#\$6, D3
00080FOA	4.	3403	MOVE.W	D3, D2

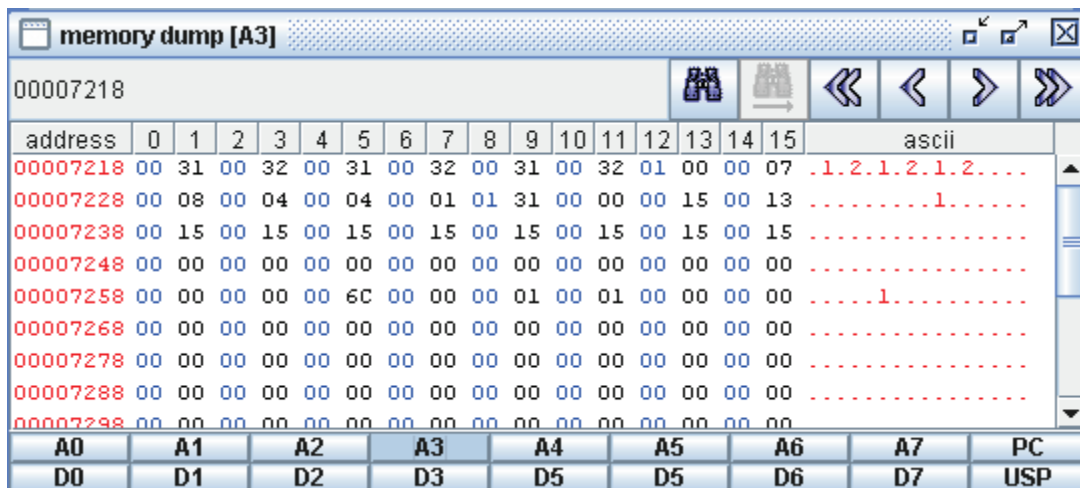
Figure 71 - The new location

Now we only need to trace with F7 until the second mov command to see if the byte in register A3 is 01. Remember it. And finally we are here

00080F56	g.	6704	BEQ	#\$4 ;00080F5C
00080F58	B+	422B000C	CLR.B	#12!\$C(A3)
00080F5C	.k	0C6B00300008	CMPI.W	#48!\$30, #\$8(A3)
00080F62	g.	6704	BEQ	#\$4 ;00080F68
00080F64	B+	422B000C	CLR.B	#12!\$C(A3)
00080F68	.k	0C6B0038000A	CMPI.W	#56!\$38, #10!\$A(A3)
00080F6E	g.	6704	BEQ	#\$4 ;00080F74
00080F70	B+	422B000C	CLR.B	#12!\$C(A3)
00080F74	J+	4A2B000C	TST.B	#12!\$C(A3)
00080F78	g.	671A	BEQ	#26!\$1A ;00080F94
00080F7A	.l	1B7C0001C90E	MOVE.B	#\$1, #-14066!\$-36F2(A5)
00080F80	.m..	206DE9A4	MOVEA.L	#-5724!\$-165C(A5), A0
00080F84	..	D1FC00000056	ADDA.L	#86!\$56, A7
00080F8A	N.	4E90	JSR	(A0)
00080F8C	a	6100FE70	BSR	#-400!\$-190 ;00080DFE
00080F90	`	6000025C	BRA	#604!\$25C ;000811EE
00080F94	.l	177C0001000C	MOVE.B	#\$1, #12!\$C(A3)

Figure 72 - The instruction we wanted

Now do a right click, select new memory dump window and go to register A3.



address	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	ascii
00007218	00	31	00	32	00	31	00	32	00	31	00	32	01	00	00	07	.1.2.1.2.1.2....
00007228	00	08	00	04	00	04	00	01	01	31	00	00	00	15	00	131.....
00007238	00	15	00	15	00	15	00	15	00	15	00	15	00	15	00	15
00007248	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00007258	00	00	00	00	00	6C	00	00	00	01	00	01	00	00	00	001.....
00007268	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00007278	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00007288	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00007298	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Figure 73 - 12th byte

As you see the 12th byte is 01 so till here our registration is correct. So what we think now? After the first and the second mov instruction there are some CLR.B 12(A3) which will zero the Test. B and the L55 will be executed and we take invalid message. Now I'll list the CLR.B 12(A3).

00000c40	422b000c	CLR.B 12(A3)
00000c8e	422b000c	CLR.B 12(A3)
00000cdc	422b000c	CLR.B 12(A3)
00000d28	422b000c	CLR.B 12(A3)
00000d7a	422b000c	CLR.B 12(A3)

00000dc6	422b000c	CLR.B 12(A3)
00000b14	422b000c	CLR.B 12(A3)
00000b20	422b000c	CLR.B 12(A3)
00000b2c	422b000c	CLR.B 12(A3)
00000b38	422b000c	CLR.B 12(A3)
00000b44	422b000c	CLR.B 12(A3)
00000b50	422b000c	CLR.B 12(A3)

4.6.1.1 PATCHING THE TARGET

Now we have to patch all CLR.B 12(A3) by NOP. Firstly we locate the patches

```

0x2C440: 7C00 46C9 383B 7C00 1AC9 3A0C 6A00 0301
0x2C450: 6066 0C3B 7CFF ECC9 383B 7C00 1AC9 3A26
0x2C460: 2EFF F424 6EFF F84E 5E4E 754E 5600 001F
0x2C470: 3C00 013F 3C00 1A48 6DF1 683F 3C00 0142
0x2C480: 672F 3C42 7752 4D4E 4FA2 D44E 5E4E 754E
0x2C490: 56FF FC3D 7C00 1AFF FE1F 3C00 0148 6EFF
0x2C4A0: FE48 6DF1 6842 672F 3C42 7752 4D4E 4FA2
0x2C4B0: D34E 5E4E 754E 56FF AC48 E71C 3847 EDF1
0x2C4C0: 684A 2DC9 0C67 1E4A 2B00 0C67 184A 2DC9
0x2C4D0: 0D67 0003 8820 6DE9 A4D1 FC00 0000 564E
0x2C4E0: 9060 0003 7842 A742 A745 EEFF B02F 0A42
0x2C4F0: A742 A742 A74E 4FA2 A942 4342 424F EF00
0x2C500: 1849 EDC9 0D4A 1267 3E22 4A41 EEFF D812
0x2C510: 3120 0010 0106 00FF 9F0C 0000 1962 0806
0x2C520: 01FF E060 0000 0E10 0106 00FF BF0C 0000
0x2C530: 1962 0611 8130 0052 4352 424A 3120 0067
0x2C540: 060C 4200 0B6F C80C 4300 016E 2A1D 7C00
0x2C550: 41FF D81D 7C00 42FF D91D 7C00 43FF DA1D
0x2C560: 7C00 44FF DB1D 7C00 45FF DC1D 7C00 46FF
0x2C570: DD42 2EFF DE76 0634 030C 4200 0B6E 1241
0x2C580: EEFF D811 BC00 4220 0052 420C 4200 0B6F
0x2C590: F217 7C00 0100 0C4A 1467 660C 5300 3367
0x2C5A0: 0442 2B00 0C0C 6B00 3900 0267 0442 2B00
0x2C5B0: 0C0C 6B00 3400 0467 0442 2B00 0C0C 6B00
0x2C5C0: 3600 0667 0442 2B00 0C0C 6B00 3000 0867
0x2C5D0: 0442 2B00 0C0C 6B00 3800 0A67 0442 2B00
0x2C5E0: 0C4A 2B00 0C67 1A1B 7C00 01C9 0E20 6DE9
0x2C5F0: A4D1 FC00 0000 564E 9061 00FE 7060 0002
0x2C600: 5C17 7C00 0100 0C42 4342 4442 4595 CA42
0x2C610: 4249 EDE9 9C43 EEFF D810 3120 0048 801D
0x2C620: 40FF AF30 40D6 4845 F280 0056 420C 4200
0x2C630: 0B6F E674 0132 0306 4100 9443 EEFF D810
0x2C640: 3120 0048 801D 40FF AF30 40D8 4845 F280
0x2C650: 0056 420C 4200 0B6F E674 0206 4400 E743
0x2C660: EEFF D810 3120 0048 801D 40FF AF30 40DA
  
```

Figure 74 - Original code

Now I'll show you the patched code..(Figure 75)

```

0x2C5A0: 044E 714E 710C 6B00 3900 0267 044E 714E
0x2C5B0: 710C 6B00 3400 0467 044E 714E 710C 6B00
0x2C5C0: 3600 0667 044E 714E 710C 6B00 3000 0867
0x2C5D0: 044E 714E 710C 6B00 3800 0A67 044E 714E
0x2C5E0: 714A 2B00 0C67 1A1B 7C00 01C9 0E20 6DE9

0x2C6C0: C910 2EFF AF48 8050 8FB0 5367 044E 714E
0x2C6D0: 7130 04C1 FC14 7B42 4048 4032 00E6 4130
0x2C6E0: 0474 0FE4 6092 40C3 FC00 6430 4490 C132
0x2C6F0: 0830 01C1 FC66 6742 4048 40E4 40E4 6190
0x2C700: 4130 402F 0848 6EFF AF4E 4FA0 C910 2EFF
0x2C710: AF48 8050 8FB0 6B00 0267 044E 714E 7130
0x2C720: 05C1 FC14 7B42 4048 4032 00E6 4130 0574
0x2C730: 0FE4 6092 40C3 FC00 6430 4590 C132 0830
0x2C740: 01C1 FC66 6742 4048 40E4 40E4 6190 4130
0x2C750: 402F 0848 6EFF AF4E 4FA0 C910 2EFF AF48
0x2C760: 8050 8FB0 6B00 0467 044E 714E 7120 54D1
0x2C770: FC00 0010 6C48 7800 642F 0A4E 9050 8F32
0x2C780: 00C3 FC00 649A C132 0A30 01C1 FC66 6742
0x2C790: 4048 40E4 4074 0FE4 6190 4130 402F 0848
0x2C7A0: 6EFF AF4E 4FA0 C910 2EFF AF48 8050 8FB0
0x2C7B0: 6B00 0667 044E 714E 7151 4406 45FF D630
0x2C7C0: 04C1 FC14 7B42 4048 4032 00E6 4130 0474
0x2C7D0: 0FE4 6092 40C3 FC00 6498 4132 0430 01C1
0x2C7E0: FC66 6742 4048 40E4 40E4 6190 4130 402F
0x2C7F0: 0848 6EFF AF4E 4FA0 C910 2EFF AF48 8050
0x2C800: 8FB0 6B00 0867 044E 714E 7130 05C1 FC14
0x2C810: 7B42 4048 4032 00E6 4130 0574 0FE4 6092
0x2C820: 40C3 FC00 649A 4132 0530 01C1 FC66 6742
0x2C830: 4048 40E4 40E4 6190 4130 402F 0848 6EFF
0x2C840: AF4E 4FA0 C910 2EFF AF48 8050 8FB0 6B00
0x2C850: 0A67 044E 714E 7161 00FC 124C EE1C 38FF
  
```

Figure 75 - Modified code

Save the new prc file...

4.6.1.2 TESTING THE TARGET

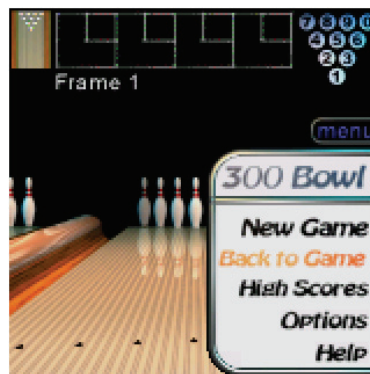


Figure 76 - Registered version

As you see we cracked that difficult target..

4.7 CONCLUSIONS AND FURTHER READINGS

This tutorial just covers an introduction to reversing Palm OS applications and it's tools. I had given practical examples which will help you start learning and advancing.

There are not much *public* tutorials in that field but from my searching I got that tutorials which the basic idea:

- <http://www.quequero.org/store/palmos/introduzione.htm>
- <http://www.quequero.org/store/palmos/opcodes.html>
- <http://www.quequero.org/store/palmos/tetris.htm>
- <http://www.quequero.org/store/palmos/fnox.htm>
- <http://www.quequero.org/store/palmos/willypms.htm>
- http://www.quequero.org/store/palmos/linee_guida.zip
- <http://www.quequero.org/store/palmos/xgrimator.html>
- http://www.quequero.org/store/palmos/epokh_serialfishing.html

Also Quequero page includes some of the tools for the others you have to search google , yahoo etc..

- <http://www.quequero.org/store/palmos/tools/emulatore.zip>
- <http://www.quequero.org/store/palmos/tools/pilotdis.zip>
- <http://www.quequero.org/store/palmos/tools/PRC2BIN.zip>
- http://www.quequero.org/store/palmos/tools/rom_palmllxe.zip
- <http://www.quequero.org/store/palmos/tools/MsgSrc.zip>
- <http://www.quequero.org/store/palmos/tools/debuffer.zip>



In Russian we have a tutorial from TSRH

<http://www.sendspace.com/file/jvxopi>

In English

<http://www.reteam.org/papers/e38.pdf>

4.8 GREETINGS

My greetings are sent to ARTeam, SnD, ICU, TSRH, exetools, unpack.cn, crackmes.de, Virus2qp1 and Shub-Nigurath. Also I want to thank a friend who always reads beta versions. Off course everyone who has read till here, after a long and difficult tutorial.

[In the Supplements folder “0.4 Wast3d_Bytes” you can find also:

- A video tutorial from Suntzu about “Nag screen & Limit Removal and Preregistered Patching (How to Patch Word Monaco 1)”. Thanks him for this contribution.
- The method to directly patch applications on the palm, see the file

Patching_on_palm_in_a_snap.txt]



5 REVERSING THE PROTECTION'S SCHEME OF ALEXEY PAJITNOV'S GAME DWICE BY GYVER75

5.1 INTRODUCTION

Who has never played with Tetris? This's my favourite game, i should stay hours and hours in front of my 17" monitor to rotate and drop its Tetraminis! So, to satisfy my curiosity, i have searched others Alex Pajitnov's game and i have found this game produced by WildSnake Software House: Dwice. Well, in a one word: it's a drug! The Tetris's father has mixed very well the ideas that stay behind of its most popular game and ... Mahjong!!! There's only one problem: it's a demo that requires a serial number to activate its full functionality. So, why not to reverse it? Furthermore, the only patch distributed on the Net doesn't work (forgot somethings Team DiGERATI?), then I convinced myself to write this tutorial for the ARTeam, i hope to like it and, as always, sorry for my bad english.

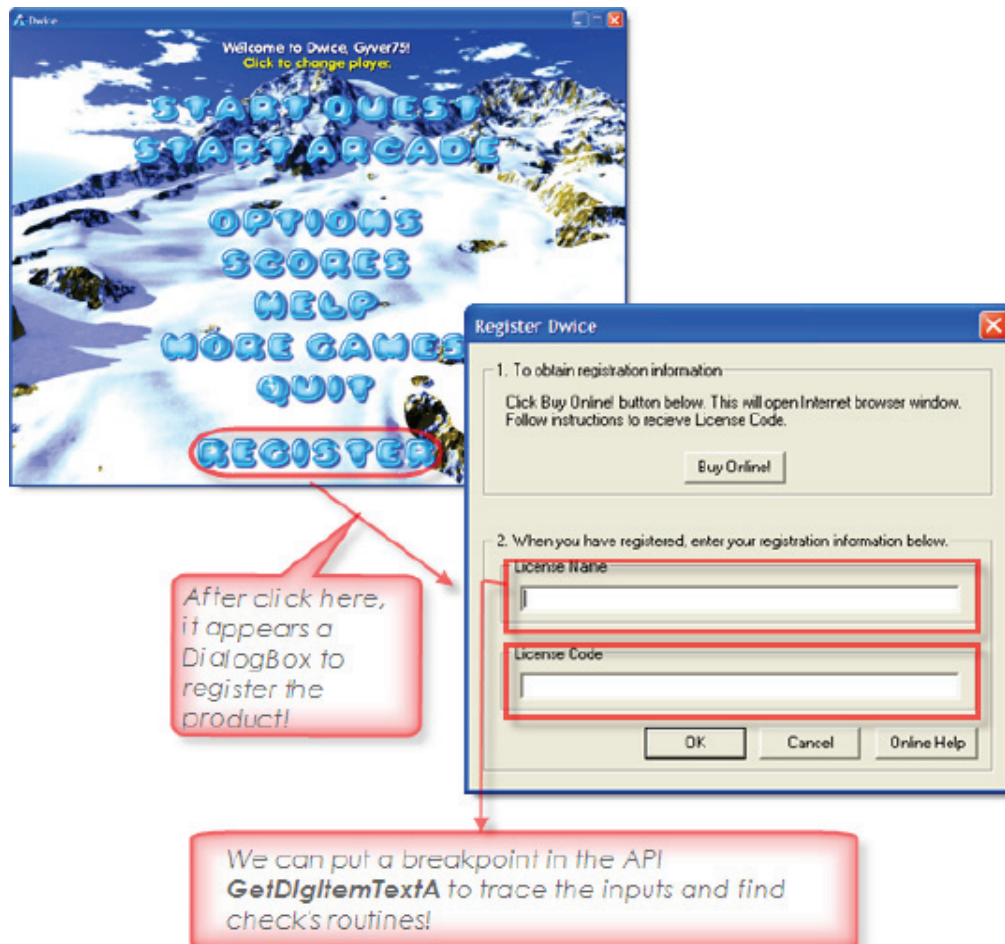
5.2 TARGET AND TOOLS USED TO REVERSE IT

These the tools used:

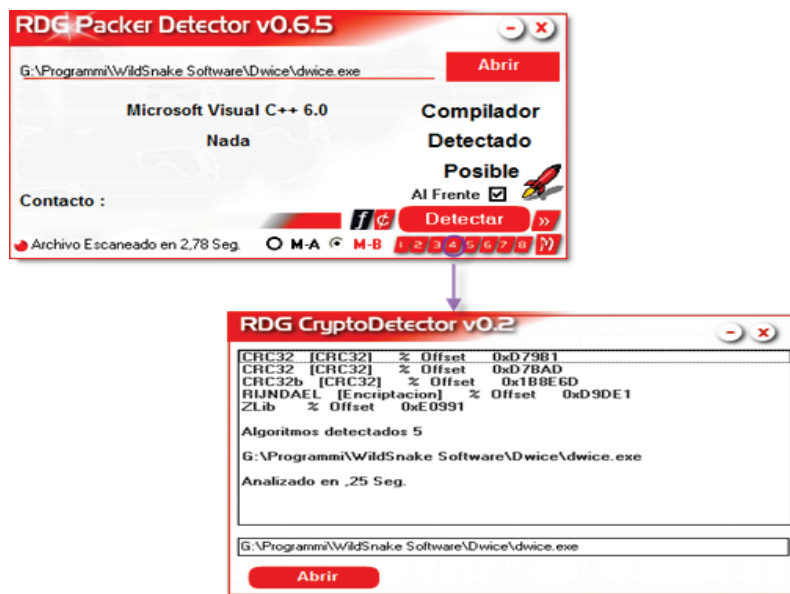
- **Target:** Alexey's Dwice; you can found it at: <http://www.wildsnake.com/puzzle/dw/>
- **Tools:** OllyDbg v1.10, the most powerful ring 3 debugger; RDG Packer Detector and CFF Explorer. I spend a few words to a useful plugin written by Scherzo for Ollydbg: LCB Plugin. As you know, Olly stores the information of a debugged program (i.e breakpoints, comments but also path of debuggee file) in a proprietary format: .Udd file; so, when we change any its 'parameter', (i.e we uninstall the prog or simply patch it), we will lost anything! Instead, LCB plugin allows to import and export comments and breakpoints of a debug's session in a external file; in other words, any change we do on the program under debug, is independent by the machine where the prog itself is installed! So, thanks to this feature, I was able to reverse this game in differents PC without to lose data or comments.

5.3 ANALYSIS

First of all, before to start with Ollydbg, we should examine the victim to find any “attack’s point”:



As you can see above, the registration’s scheme appears very simply; clearly, if the target were packed, first of all we should unpacked it. So, to check this, I use RDG Packer detector, but obviously, you can use any other PE analyser:

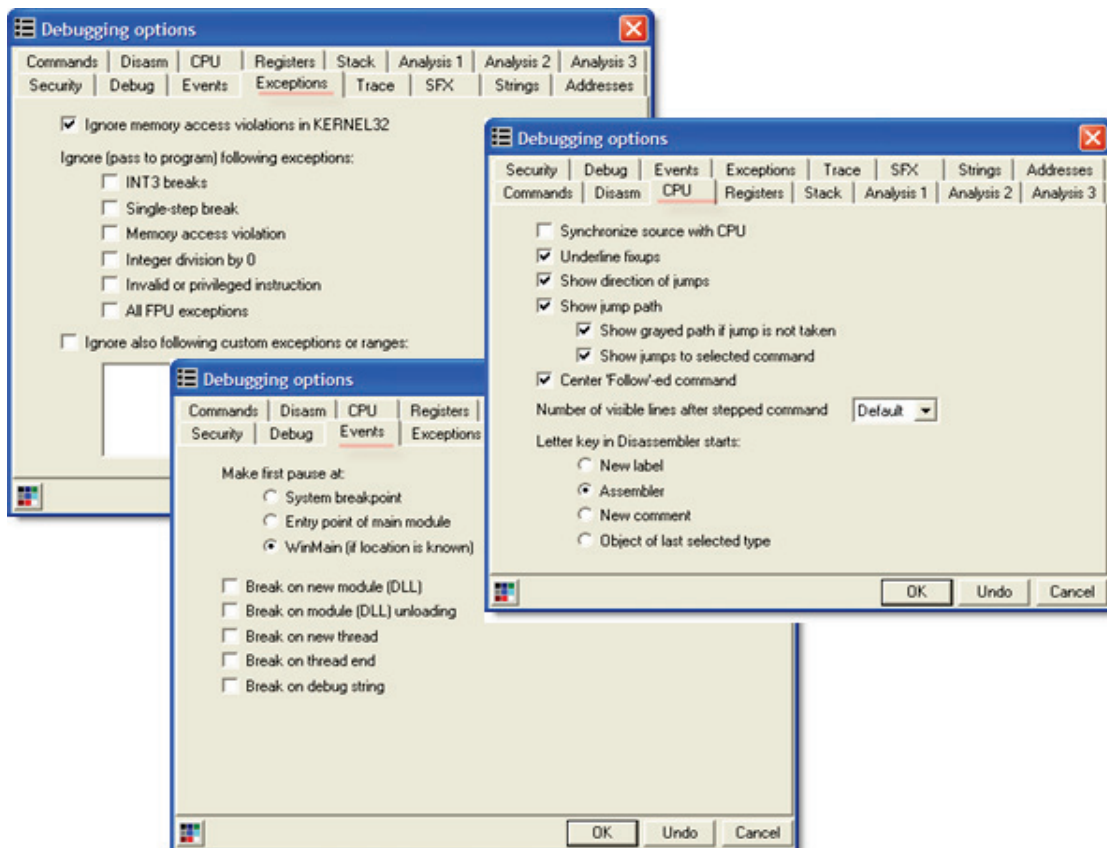


Fortunately, this program isn't packed or compressed; furthermore, it was written in Visual C++ and uses some crypto algorithms... have you seen RDG CryptoDetector recognizes RIJNDAEL signature? Not worry, Crypto scanners are useful tools but not always say the true!! Infact, I used also the Crypto scanner module of SND_Reverse Tool v 1.4 and, in this case, the only signatures identified were the CRC32 hashes.

Well, we can make us (in reality, we must!! ;)) many other questions, i.e. how many sections have the program, where's the entry point, which libraries are imported or what 's the resource's identifier of dialogbox's template seen above, but for all these questions there's only answer: CFF Explorer (Thanks Daniel Pistelli, aka NtosKrn!).

After this static analysis of the prog, we can finally fire up our favourite debugger: OllyDbg; i configured it as Lena151 has suggested in her video tutorials, without any hiding plugins; only in this way we can learn somethings!

These are my settings:



5.4 IDENTIFICATION OF CHECK'S ROUTINES

Having loaded Dvice under debug, I searched for all intermodular call and I put a INT 3 breakpoints on 2 API: **DialogBoxParamA** and **GetDlgItemTextA**. So, after clicked in the 'REGISTER button', I have landed immediately in these pieces of code:

00419F71	PUSH 0	lParam = NULL DlgProc = dvice.00419CA0 hOwner pTemplate = "REGISTERDIALOG" hInst => NULL DialogBoxParamA
00419F73	PUSH dvice.00419CA0	
00419F78	MOV EDX, DWORD PTR DS:[ECX+2E0]	
00419F7E	PUSH EDX	
00419F7F	PUSH dvice.004FD81C	
00419F84	PUSH EAX	
00419F8E	CALL NEAR DWORD PTR DS:[<&USER32	

↓

00419DBE	MOV ESI, DWORD PTR SS:[ESP+C]	Case 1 of switch 00419D92 USER32.GetDlgItemTextA Count = 52 (82.) Buffer = dvice.004F1020 ControlID = 3E8 (1000.) hWnd GetDlgItemTextA Count = 52 (82.) Buffer = dvice.004F1220 ControlID = 3E9 (1001.) hWnd GetDlgItemTextA This Sub analyses the 'Name': Remove from this empty This Sub analyses the 'Name': Remove from this DIGIT This Sub analyses the 'Serial'; take of this only dig Result = 1 hWnd EndDialog
00419DC2	MOV EDI, DWORD PTR DS:[<&USER32.Ge	
00419DC8	PUSH 52	
00419DCA	PUSH dvice.004F1020	
00419DCF	PUSH 3E8	
00419DD4	PUSH ESI	
00419DDE	CALL NEAR EDI	
00419DD7	PUSH 52	
00419DD9	PUSH dvice.004F1220	
00419DDE	PUSH 3E9	
00419DE3	PUSH ESI	
00419DE4	CALL NEAR EDI	
00419DE6	CALL dvice.00425640	
00419DEB	CALL dvice.00425830	
00419DF0	CALL dvice.00425A20	
00419DF5	PUSH 1	
00419DF7	PUSH ESI	
00419DF8	CALL NEAR DWORD PTR DS:[<&USER32	
00419DFE	POP EDI	
00419DFF	MOV EAX, 1	
00419E04	POP ESI	
00419E05	RETN 10	

Figure 77.

As you can notice, Ollydbg gives us many information: name of Dialog template (REGISTERDIALOG) and, especially the DialogBox Procedure localized in the memory's offset at *00419CA0h*. Here, the target stores the License Name and License Code in two buffer long 82 bytes (respectively, at the offset *004F1020h* and *004F1220h*). After the last indirect call used to invoke the **GetDlgItemTextA**, we can see 3 strange calls:

00419DE6	CALL dvice.00425640
00419DEB	CALL dvice.00425830
00419DF0	CALL dvice.00425A20

First call simply removes space's chars at the beginning of License Name buffer; also it substitutes the spaces in the middle with only one space char, for example see Figure 78.

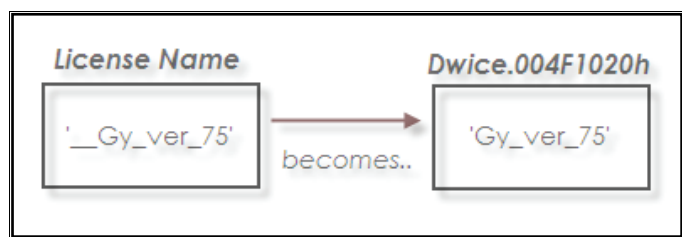


Figure 78. You consider the underline chars as simply spaces.

The offset `004F1020h` is the end of a String buffer that begins at the address `004F0FE0h`: `'9V4BKIGUVYQACMYEXMABTBZAUQGBWMBYELHLSO7Z50PO23LWFT3ZREDSMHN08LJI '`; so, as side effect, the License Name is appended at the end of this `"Hash String"`.

Second call copies the chars stored at the offset `004F1020h` in new buffer at the address `004F1120h`; also, it removes the digit chars of the previous buffer, see Figure 79.

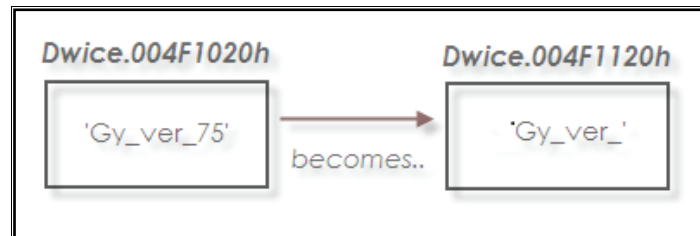


Figure 79.

Third call simply converts the chars of the License Code stored at the address `004F1220h` in uppercase chars (`'a' -> 'A', 'y' -> 'Y' ...`).

Clearly, I exposed only the results of these calls; if you are interested in tracing there, load my comments exported through Scherzo's plug in, you can find it in the `src` folder with the plug in itself (I anyway suggest you to read this tutorial with these comments, because each procedure mentioned here, was described in depth!). As final result, I did a snapshot of the memory's block from the address `004F0FE0h` to the address `004F1230h`:



Address	Hex dump	ASCII
004F0FE0	39 56 34 42 4B 49 36 55 56 59 51 41 43 4D 59 45	9V4BKI6UVYQACMYE
004F0FF0	58 4D 41 42 54 42 5A 41 55 51 47 42 57 4D 42 59	XMABTBZAUQGBWMBY
004F1000	45 4C 48 4C 53 4F 37 5A 35 30 50 4F 32 33 4C 57	ELHLSO7Z50PO23LW
004F1010	46 54 33 5A 52 45 44 53 4D 48 4E 30 38 4C 4A 49	FT3ZREDSMHN08LJI
004F1020	47 79 76 65 72 37 35 00 00 00 00 00 00 00 00	Gyver75.
004F1030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
004F1040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
004F1050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
004F1060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
004F1070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
004F1080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
004F1090	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
004F10A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
004F10B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
004F10C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
004F10D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
004F10E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
004F10F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
004F1100	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
004F1110	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
004F1120	47 79 76 65 72 00 35 00 00 00 00 00 00 00 00	Gyver5.
004F1130	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
004F1140	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
004F1150	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
004F1160	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
004F1170	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
004F1180	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
004F1190	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
004F11A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
004F11B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
004F11C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
004F11D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
004F11E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
004F11F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
004F1200	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
004F1210	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
004F1220	44 33 43 41 4E 34 58 36 50 35 59 43 41 55 53 33	D3CAN4X6P5YCAUS3
004F1230	47 52 42 59 45 52 36 4C 42 4B 4C 5A 45 58 00 00	GRBYER6LBKLEZEX..
004F1240	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	

Figure 80.

As you can see in figure 1, after these calls, the API **EndDialog** is invoked with return value = 1; so, in this piece of code, there isn't any check's routine! To find them, as first approach, I put a memory breakpoint at the offset of License Code, but, after pressing F9, only one time it's stopped and later, the Bad Boy Message appears:

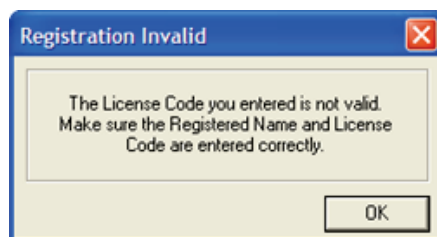


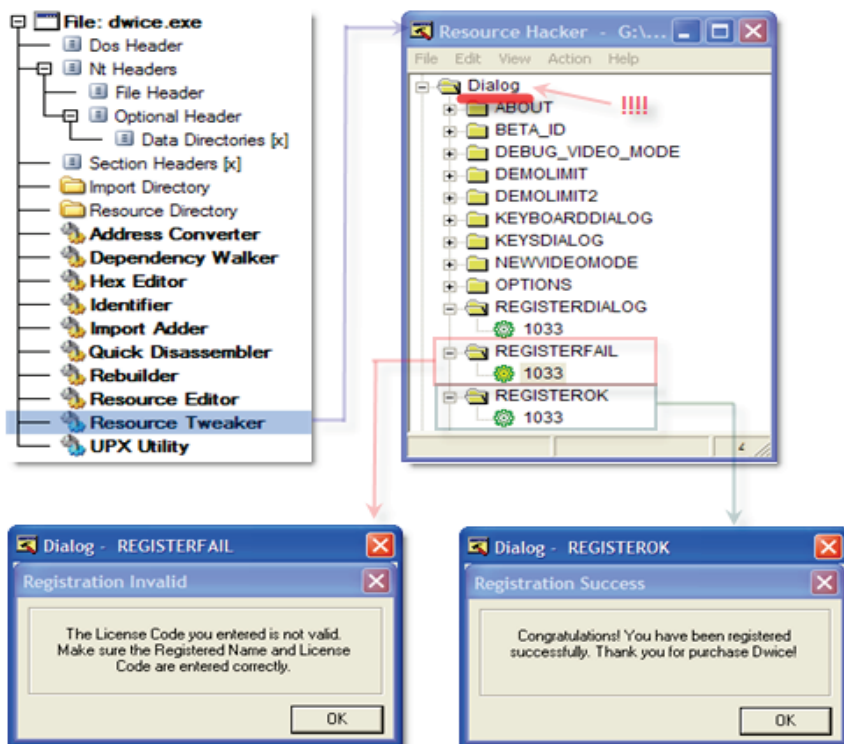
Figure 81.



Hmm, very strange! Indeed, when the debugger stopped its execution at the address `004F1220h`, where I put a M. Breakpoint on access, the code simply copied these bytes in another place and went ahead. Well, in a traditional way, I set another M. Breakpoint on access in the new locations but I have landed in the same code: another copy of Memory Block (shown in Figure 4) in a new place. Despite having tried to trace this routine (it's repeated many times) setting breakpoints everywhere, I however lost "the main wire" and I reached in the same disappointing result: "The License Code you entered is invalid...".

5.4.1 OBFUSCATION

If the standard mode to trace doesn't work, we can attach the victim in other way: indeed, we can recognize the resource identifier displayed in Figure 5 and then, start from here to reverse. So, with CFF Explorer and its Extension Plug in Resource Tweaker, I discovered that the "Bad or Good Boy Message" are in reality a Dialog Boxes:



Then in Ollydbg, I set breakpoints on every call to `DialogBoxParamA` and, after inserted a bogus License Code, I have landed here:

```

0041A020 8B0D 18E7AC0 MOV ECK,DWORD PTR DS:[ACE718]
0041A026 C705 00EF9E0 MOV DWORD PTR DS:[9EEF00],1
0041A030 8B01 MOV EAX,DWORD PTR DS:[ECK]
0041A032 FF50 64 CALL NEAR DWORD PTR DS:[EAX+64]
0041A035 8B0D 18E7AC0 MOV ECK,DWORD PTR DS:[ACE718]
0041A038 8B4424 04 MOV EAX,DWORD PTR SS:[ESP+4]
0041A03F 6A 00 PUSH 0
0041A041 68 B09F4100 PUSH dwice.00419FB0
0041A046 8B91 E002000 MOV EDX,DWORD PTR DS:[ECK+2E0]
0041A04C 85C0 TEST EAX,EAX
0041A04E 52 PUSH EDX
0041A04F 74 07 JE SHORT dwice.0041A058
0041A051 68 38D84F00 PUSH dwice.004FD838 ASCII "REGISTERFAIL"
0041A056 EB 05 JMP SHORT dwice.0041A05D
0041A058 68 2CD84F00 PUSH dwice.004FD82C ASCII "REGISTEROK"
0041A05D A1 4CF2AC00 MOV EAX,DWORD PTR DS:[ACF24C]
0041A062 50 PUSH EAX
0041A063 FF16 D4624D0 CALL NEAR DWORD PTR DS:[<4USER32.Dialog
0041A069 8B0D 18E7AC0 MOV ECK,DWORD PTR DS:[ACE718]
0041A06F C705 08EF9E0 MOV DWORD PTR DS:[9EEF08],0
0041A079 8B11 MOV EDX,DWORD PTR DS:[ECK]
0041A07B FF52 68 CALL NEAR DWORD PTR DS:[EDX+68]
0041A07E C705 00EF9E0 MOV DWORD PTR DS:[9EEF00],0
0041A088 C3 RETN
00400000 hInst = 00400000
004F0838 pTemplate = "REGISTERFAIL"
00260616 hOwner = 00260616 ('Dwice',class='MainWindow')
00419FB0 DlgProc = dwice.00419FB0
00000000 lParam = NULL
00419FB8 RETURN to dwice.0040F398 from dwice.0041A020
00000001
00000111
000008B9
> 6A 01 PUSH 1 Case BB9 of switch 0040E7A4
E8 88 CALL dwice.0041A020 the final BAD judgment!!!
89C4 ADD ESP,4
39C0 XOR EAX,EAX

```

Figure 82.

As you can see above, the Good or Bad Boy Message depends by the parameter passed to routine at the offset `0041A020h`: if equal to 1 (like my case), it's passed a 'REGISTERFAIL' template as parameter in `DialogBoxParamA`, else it's set 'REGISTEROK' message! So, for this reason, I searched any reference to the command at the beginning of this procedure and I found two calls:

```

0040F391 PUSH 1 ; Case BB9 of switch 0040E7A4
0040F393 CALL dwice.0041A020 ; the final BAD judgment!!!

```

```

0040F2B8 MOV DWORD PTR DS:[4F1448],EBX ; Case BB8 of switch
0040E7A4
0040F2BE MOV DWORD PTR DS:[9EEC84],EBX

```

Well, my comments are quite clear, they comes from few tests and from the relative jumps at the address `0040F391h` (see below):

<pre> > CALL dwice.00419F50 CMP EAX,1 JNZ dwice.0040F56D MOV AL,BYTE PTR DS:[4F1020] TEST AL,AL JE SHORT dwice.0040F391 MOV AL,BYTE PTR DS:[4F1220] TEST AL,AL JE SHORT dwice.0040F391 MOV DWORD PTR DS:[A9447C],dwice.004F0FE0 CALL dwice.00425EA0 CALL dwice.00425FE0 CALL dwice.00422540 CALL dwice.0041A270 MOV DWORD PTR DS:[9EEC8C],3 XOR EAX,EAX POP EDI POP ESI POP EBX MOV ESP,EBP POP EBP RETN PUSH 1 CALL dwice.0041A020 </pre>	<pre> Cases 3F1,C1D of switch 0040E7A4 We approde here later the REGISTERDLG! AL == first char of the name!; if the Name buffer is empty... jump, else go away; AL == first char of the Serial buffer; if the Serial Buffer is empty... jump, else go away; ASCII "9V4BKI6UVYQACMYEXMABTBZAUQGBWMBYEL [9EEC8C] == Time flag for Reg Dialog! Case BB9 of switch 0040E7A4 the final BAD judgment!!! </pre>
---	---

Figure 83 – obfuscation calls

These relative jumps are made if the buffers of License Name (004F1020h) or License Code (004F1220h) are empty (so, it's clear that we did an error and the prog should let us know!); else, if it's all OK, the base offset of memory block, displayed in figure 4 (004F0FE0h), is copied at new address and then, fourth calls are invoked. We're finally arrived to the Obfuscations' routine!

Stepping in to the first call, it appears to us a (strange...) sequence of calls:

\$	CALL	dwice.00425C70	[A9447Ch] -> [A94474h];
.	CALL	dwice.00410560	[A94474h] -> [9EED0Ch];
.	CALL	dwice.00418B90	[9EED0Ch] -> [9EEEB4h];
.	CALL	dwice.00418E30	[9EEEB4h] -> [9EEEF0h];
.	CALL	dwice.00421C80	[9EEEF0h] -> [A943F8h];
.	CALL	dwice.00422A40	[A943F8h] -> [A94410h];
.	CALL	dwice.00423220	[A94410h] -> [A94428h];
.	CALL	dwice.004238B0	[A94428h] -> [A94440h];
.	CALL	dwice.004245D0	[A94440h] -> [A94458h];
.	CALL	dwice.00421F20	[A94458h] -> [A943F4h];
.	CALL	dwice.00425B20	[A943F4h] -> [A9446Ch];
.	CALL	dwice.00410A20	[A9446Ch] -> [9EED04h];
.	CALL	dwice.004188F0	[9EED04h] -> [9EEEACh];
.	CALL	dwice.00418F80	[9EEEACh] -> [9EEEE8h];
.	CALL	dwice.00421890	[9EEEE8h] -> [A943F0h];
.	CALL	dwice.00422B90	[A943F0h] -> [A9440Ch];
.	CALL	dwice.00423370	[A9440Ch] -> [A94424h];
.	CALL	dwice.00423B50	[A94424h] -> [A9443Ch];
.	CALL	dwice.004241E0	[A9443Ch] -> [A94450h];
.	CALL	dwice.004221C0	[A94450h] -> [A943E8h];
.	CALL	dwice.00425940	[A943E8h] -> [A94470h];
.	CALL	dwice.004108D0	[A94470h] -> [9EED08h];
.	CALL	dwice.00418A40	[9EED08h] -> [9EEEB0h];
.	CALL	dwice.00419810	[9EEEB0h] -> [9EEEECh];
.	CALL	dwice.004219E0	[9EEEECh] -> [A943E0h];
.	CALL	dwice.00422CE0	[A943E0h] -> [A94404h];
.	CALL	dwice.004230D0	[A94404h] -> [A9441Ch];
.	CALL	dwice.00423DF0	[A9441Ch] -> [A94434h];
.	CALL	dwice.00424090	[A94434h] -> [A94454h];
.	CALL	dwice.00421DD0	[A94454h] -> [A943ECh];
.	CALL	dwice.00425750	[A943ECh] -> [A94464h];

Figure 84

Every call allocates a Memory Block in to the Heap (through the API **HeapAlloc** ...), then copies the Memory's zone displayed in figure 4 in it. The pointer returned by the API is stored in a particular location in the DATA section of the target (see my comments in the Figure 8) and it will be used, in the next call, as source to new copy's operation. Indeed, the only differences between a subroutine and the other, are the offsets of pointers of Src. and Dest. Heap Memory's blocks; the structure of each call is the same, see Figure 85.

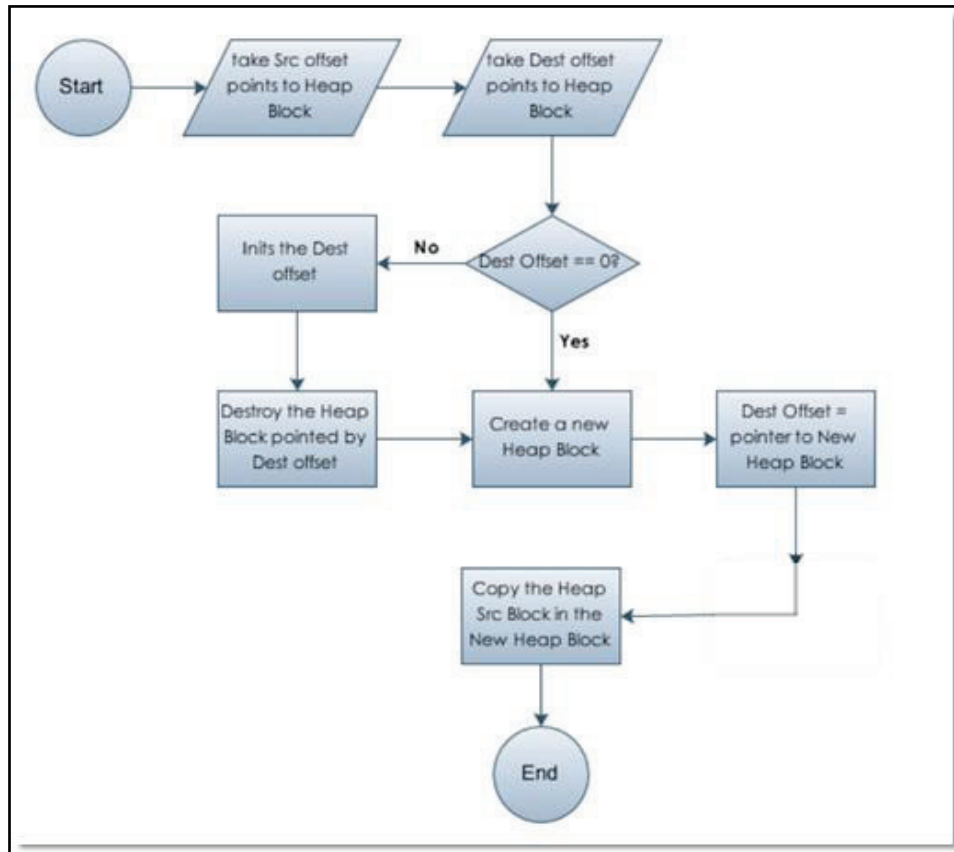


Figure 85 – structure of each call of the obfuscation list

The others 3 Obfuscation's calls use this routine to destroy and create Memory blocks allocating in to the heap: the offsets that point to it, are stored in the same locations you can see in Figure 87. It's important to notice the presence of garbage call inside this scheme: before to copy the Heap source block in to the New one, a subroutine is invoked:

```

PUSH 100
CALL dwice.00427220
ADD ESP,4
... some calculation
CMP CL,1
JNZ SHORT Alternative copy operation
...
1° copy's mode
PUSH 444
PUSH offset of Source Heap Block
PUSH offset of New Heap Block
CALL dwice.004B5060
...
MOV ECX,6
MOV EDI,DWORD PTR DS:[offset of New Heap Block]
IDIV ECX
MOV ECX,111
MOV EAX,1
REP MOVSD DWORD PTR ES:[offset of New Heap Block],
           DWORD PTR DS:[offset of Source Heap Block]

```

The call underlined in blue, generates a pseudo random number; this selects, after a few calculation, 2 subroutines that make the same thing: copy's operation of Heap Blocks!!!

Well, at the end of this deep analysis, we can statically trace locations that point to Heap objects created; in example, by the address constant `00A9447Ch` (I selected it because it's the first address used to point the Original Memory's Block of Figure 80 !), we will find many subroutines having the same "structure":

<pre> 00410410 PUSH ESI 00410411 MOV ESI, DWORD PTR DS:[9EEFC] 00410417 TEST ESI, ESI 00410419 PUSH EDI 0041041A JNB SHORT dvice.0041042B 0041041C MOV ESI, DWORD PTR DS:[A9447C] 00410422 TEST ESI, ESI 00410424 JNB SHORT dvice.0041042B 00410426 POP EDI 00410427 XOR EAX, EAX 00410429 POP ESI 0041042A RETN 0041042B > PUSH 100 00410430 CALL dvice.00427220 00410435 AND EAX, 3 00410438 ADD ESP, 4 0041043B AND EAX, 80000001 00410440 JNB SHORT dvice.00410447 00410442 DEC EAX 00410443 OR EAX, FFFFFFFF 00410446 INC EAX 00410447 JNB SHORT dvice.00410462 00410449 MOV ESI, DWORD PTR DS:[A94464] 0041044F TEST ESI, ESI 00410451 JNB SHORT dvice.00410462 00410453 MOV ESI, DWORD PTR DS:[A9447C] 00410459 TEST ESI, ESI 0041045B JNB SHORT dvice.00410462 0041045D POP EDI 0041045E XOR EAX, EAX 00410460 POP ESI 00410461 RETN 00410462 > MOV EDI, DWORD PTR SS:[ESP+C] 00410466 MOV ECX, 111 0041046B REP MOVSD WORD PTR ES:[EDI], DWORD PTR DS:[ESI] 0041046D POP EDI 0041046E MOV EAX, 1 00410473 POP ESI 00410474 RETN </pre>	<pre> saves ESI on the stack; ESI points to a Heap Object; saves EDI on the stack; it jumps if really Heap Object exists; ESI points to the original Heap Block; it jumps surely; parameter passed by value; this proc generates a pseudo random number < to the parameter passed by value; ESI points to another Heap Object; it jumps if really Heap Object exists; ESI points to the original Heap Block; it jumps surely; EDI == parameter passed (top of the stack...); ECX = number of bytes to copy; copy operation; restore EDI; restore ESI; </pre>
---	--

Ctrl + R (points to 00410410)

References to the address constant A9447Ch (points to 0041041C and 00410453)

Copy's operation on the stack (points to 00410462)

Figure 86.

As you can see above, at the end of subroutine `00410410h`, there's a simply copy operation of `111h` bytes from Heap Area to Stack one; indeed, if we find references to select command at the beginning of this procedure (Ctrl+R), we will land here:

<pre> \$ SUB ESP,444 . PUSH 11 . CALL dwice.00427220 . ADD ESP,4 . CMP EAX,0F . JZ dwice.004187B2 . JMP NEAR DWORD PTR DS:[EAX*4+418834] . LEA EAX,DWORD PTR SS:[ESP] . PUSH EAX . CALL dwice.00418F10 . JMP dwice.004187BC . LEA ECX,DWORD PTR SS:[ESP] . PUSH ECX . CALL dwice.00423AE0 . JMP dwice.004187BC . LEA EDX,DWORD PTR SS:[ESP] . PUSH EDX . CALL dwice.00422DC0 . JMP dwice.004187BC . LEA EAX,DWORD PTR SS:[ESP] . PUSH EAX . CALL dwice.00410860 . JMP dwice.004187BC . LEA ECX,DWORD PTR SS:[ESP] . PUSH ECX . CALL dwice.00417FF0 . JMP dwice.004187BC . LEA EDX,DWORD PTR SS:[ESP] . PUSH EDX . CALL dwice.00423300 . JMP dwice.004187BC . LEA EAX,DWORD PTR SS:[ESP] . PUSH EAX . CALL dwice.00423840 . JMP SHORT dwice.004187BC . LEA ECX,DWORD PTR SS:[ESP] . PUSH ECX . CALL dwice.00418880 . JMP SHORT dwice.004187BC . LEA EDX,DWORD PTR SS:[ESP] . PUSH EDX . CALL dwice.00425C00 </pre>	<p>Parameter passed by value;</p> <p>This sub generates a pseudo random number by the use of the API GetTickCount EAX <= 10h; it 's used as a random index; Switch (cases 0..F)</p> <p>Case 0 of switch 004186D0 EAX == offset of stack's top; This proc copies a Heap object in to the stack;</p> <p>Case 1 of switch 004186D0 ECX == offset of stack's top; This proc copies a Heap object in to the stack;</p> <p>Case 2 of switch 004186D0 EDX == offset of stack's top; This proc copies a Heap object in to the stack;</p> <p>Case 3 of switch 004186D0 EAX == offset of stack's top; This proc copies a Heap object in to the stack;</p> <p>Case 4 of switch 004186D0 ECX == offset of stack's top; This proc copies a Heap object in to the stack;</p> <p>Case 5 of switch 004186D0 EDX == offset of stack's top; This proc copies a Heap object in to the stack;</p> <p>Case 6 of switch 004186D0 EAX == offset of stack's top; This proc copies a Heap object in to the stack;</p> <p>Case 7 of switch 004186D0 ECX == offset of stack's top; This proc copies a Heap object in to the stack;</p> <p>Case 8 of switch 004186D0 EDX == offset of stack's top; This proc copies a Heap object in to the stack;</p>
---	---

Figure 87.

The first call (*dwice.00427220h*) generates a pseudo random index used to select a subroutine from 16 ones (here every procedure has the same structure displayed in Figure 10!); so, it's clear why we don't trace the serial in the standard way: the stack area where the target copies License Name, License Code and validates its, derives from a heap object randomly chosen! Graphically see Figure 88:

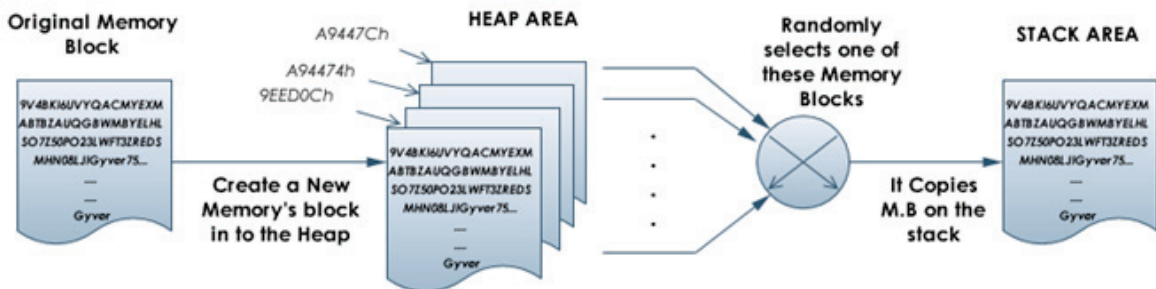


Figure 88.

Funny, isn't it?

Well, at the end we came in the right place (procedure represented in Figure 87): we can recognize a test on the License Code's length and many checks about the chars or digits belonging to the Serial! Before to describe the General Check's scheme, I want to present to you a global overview of this protection; it's time to talk about Guard's Checks.

5.4.2 GUARD'S CHECKS AND TIME'S FLAGS

If you have noticed in Figure 83 I stressed in green a address constant `9EEC8Ch`. This location is a sort of Time's Flag: its value is decremented by a unit until to a certain constant; then a Check's routine is executed. To discover this, I found any reference to this address constant:

```
0040F37E MOV DWORD PTR DS:[9EEC8C],3
0040F76B MOV EAX,DWORD PTR DS:[9EEC8C]
0040F778 MOV DWORD PTR DS:[9EEC8C],EAX
```

So, I traced the 2 instructions and I landed here:

```

0040F720 MOV EAX,DWORD PTR DS:[9EEC94]
0040F725 . PUSH EDI
0040F726 . XOR EDI,EDI
0040F728 . CMP EAX,EDI
0040F72A . JNZ SHORT dwice.0040F73C
0040F72C . CMP DWORD PTR DS:[9EEC98],EDI
0040F732 . JNZ SHORT dwice.0040F73C
0040F734 . CMP DWORD PTR DS:[9EEC9C],EDI
0040F73A . JE SHORT dwice.0040F76B
0040F73C . MOV EAX,DWORD PTR DS:[ACE718]
0040F741 . MOV DWORD PTR DS:[9EEC94],EDI
0040F747 . MOV DWORD PTR DS:[9EEC98],EDI
0040F74D . MOV DWORD PTR DS:[9EEC9C],EDI
0040F753 . MOV ECX,DWORD PTR DS:[EAX+2E0]
0040F759 . PUSH EDI
0040F75A . PUSH 0BB9
0040F75F . PUSH 111
0040F764 . PUSH ECX
0040F765 . CALL NEAR DWORD PTR DS:[<&USER32.PostMessageA
0040F76B . MOV EAX,DWORD PTR DS:[9EEC8C]
0040F770 . CMP EAX,EDI
0040F772 . JLE SHORT dwice.0040F79E
0040F774 . DEC EAX
0040F775 . CMP EAX,1
0040F778 . MOV DWORD PTR DS:[9EEC8C],EAX
0040F77D . JNZ SHORT dwice.0040F79E
0040F77F . CALL dwice.004250A0
0040F784 . TEST EAX,EAX
0040F786 . JNZ SHORT dwice.0040F794
0040F788 . MOV DWORD PTR DS:[9EEC94],1
0040F792 . JMP SHORT dwice.0040F79E
0040F794 . MOV DWORD PTR DS:[9EEED0],5
0040F79E . CMP DWORD PTR DS:[9EEC90],EDI
0040F7A4 . JLE SHORT dwice.0040F7BB
0040F7A6 . MOV DWORD PTR DS:[9EEC90],EDI
0040F7AC . CALL dwice.004250A0
0040F7B1 . MOV DWORD PTR DS:[9EEEC4],1
0040F7BB . POP EDI
0040F7BC . RETN

```

Annotations and comments:

- `[9EEC94]` == Flag control, if it sets to 1 we have a MISTAKE;
- `[9EEED0]` == Time flag for Reg Dialog!
- `[9EEC90]` == 0Ah; it's the first Time Flag for Registry Check!
- `[9EEC90]` = 0, Reset the Time Counter;
- `[9EEEC4]` == 1; it's a next Time's flag for Registry Check;
- These are Flag Controls!!; ;)
- PostMessageA
- wParam = 0BB9
- Message = WM_COMMAND
- hWnd
- EAX == Time Flag, at the begin its value is 3;
- When EAX == 1, it begins the control of the serial;

Callout box: **Guard's Checks!!!**

Figure 89.

My comments are quite clear but however I will explain to you this subroutine in a few words. First of all, have you noticed the API `PostMessageA`? It receives a `WM_COMMAND` uMsg with `wParam = 0BB9h`... where we met this constant? Do you remember good or bad boy messages (in reality, dialog boxes...)? Ok, now this API is invoked if one of these locations (`[9EEC94h]`, `[9EEC98h]`, `[9EEC9Ch]`) is not equal to 0; else we jump in the instructions underlined in grey. Here, the Time's flag addressed by `9EEC8Ch` is decremented until to 1; then a

Check's routine (CALL *dwice.004250A0h*) is invoked and returns 0 if is not passed. If this happens, a guard's check [*9EEC94h*] is set to 1 (so, in the next cycle, the prog. calls the API *PostMessageA* in Figure 89), else it sets a new Time's Flag: [*9EEED0h*]== 5. It's clear that a next Check's routine is linked to this parameter; so I found any reference to this new address constant and I have landed here:

```

0040F7C0 |> MOV EAX,DWORD PTR DS:[9EEED0] EAX = {9EEED0}, Time Flag; When EAX == 2, starts another Dialog check control;
0040F7C5 |> TEST EAX,EAX
0040F7C7 |> JLE SHORT dwice.0040F7F3
0040F7C9 |> DEC EAX
0040F7CA |> CMP EAX,2
0040F7CD |> MOV DWORD PTR DS:[9EEED0],EAX
0040F7D2 |> JNZ SHORT dwice.0040F7F3
0040F7D3 |> CALL dwice.00410640
0040F7D9 |> TEST EAX,EAX
0040F7DB |> JNZ SHORT dwice.0040F7E9
0040F7DD |> MOV DWORD PTR DS:[9EEC94],1 [9EEC94] == Flag control, if it sets to 1 we have a MISTAKE;
0040F7E7 |> JMP SHORT dwice.0040F7F3
0040F7E9 |> MOV DWORD PTR DS:[9EEEC8],6 [9EEEC8] == 6; it's a next Time's Flag for Reg Dialog;
0040F7F3 |> MOV EAX,DWORD PTR DS:[9EEEC4] EAX = {9EEEC4}, Time's Flag; when EAX == 1 starts another Registry check control;
0040F7F8 |> TEST EAX,EAX
0040F7FA |> JLE SHORT dwice.0040F815
0040F7FC |> MOV DWORD PTR DS:[9EEEC4],0 [9EEEC4] = 0, Reset the Time Counter;
0040F80B |> CALL dwice.00410640
0040F80B |> MOV DWORD PTR DS:[9EEED4],5 [9EEED4] == 5; it's a next Time's flag for Registry Check;
0040F815 |> RETN

```

Figure 90.

Well, we can notice the same structure seen in Figure 89; so, to find all Check's routines, we can simply trace Time's Flags (now, we recognize it! I.e I traced the [*9EEEC8h*]):

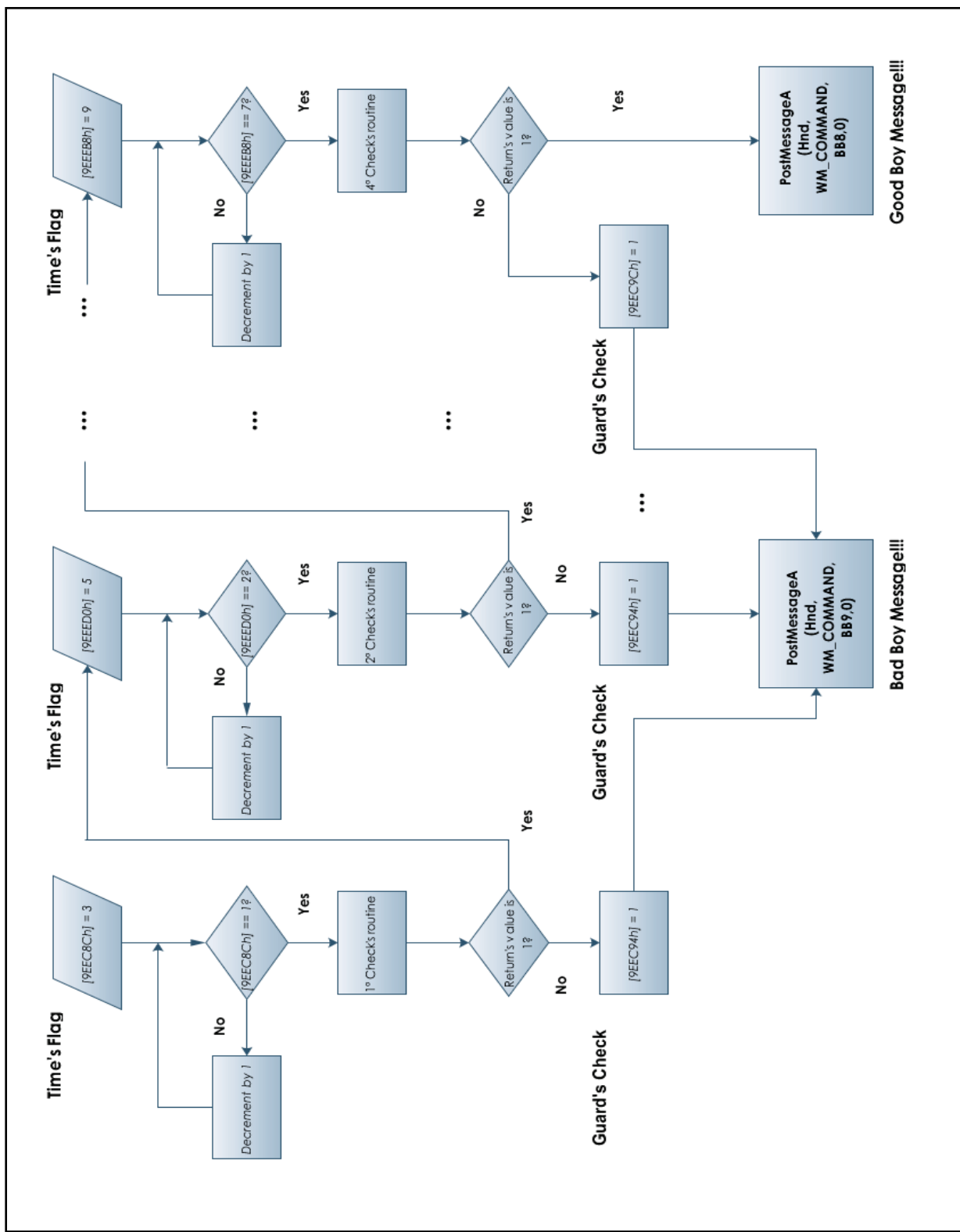
```

|> MOV EAX,DWORD PTR DS:[9EEEC8] EAX = {9EEEC8}, Time Flag; When EAX == 6, starts another check control;
|> PUSH ESI
|> XOR ESI,ESI
|> CMP EAX,ESI
|> JLE SHORT dwice.0040F892
|> DEC EAX
|> CMP EAX,4
|> MOV DWORD PTR DS:[9EEEC8],EAX
|> JNZ SHORT dwice.0040F84C
|> CALL dwice.004186C0
|> TEST EAX,EAX
|> JNZ SHORT dwice.0040F892
|> MOV DWORD PTR DS:[9EEC94],1 [9EEC94] == Flag control, if it sets to 1 we have a MISTAKE;
|> JMP SHORT dwice.0040F892
|> CMP EAX,3
|> JNZ SHORT dwice.0040F86C
|> CALL dwice.0041A0C0
|> TEST EAX,EAX
|> JNZ SHORT dwice.0040F892
|> MOV DWORD PTR DS:[9EEEC8],ESI [9EEEC8] = 0, reset the Time Counter;
|> MOV DWORD PTR DS:[9EEC98],1 [9EEC98] == Flag control, if it sets to 1 we have a MISTAKE;
|> JMP SHORT dwice.0040F892
|> CMP EAX,2
|> JNZ SHORT dwice.0040F878
|> CALL dwice.0041A100
|> JMP SHORT dwice.0040F856
|> CMP EAX,1
|> JNZ SHORT dwice.0040F884
|> CALL dwice.0041A140
|> JMP SHORT dwice.0040F856
|> CMP EAX,ESI
|> JNZ SHORT dwice.0040F892
|> MOV DWORD PTR DS:[9EEEB8],9 [9EEEB8] == 9; it's a next Time's Flag;
|> MOV EAX,DWORD PTR DS:[9EEED4] EAX = {9EEED4}, Time's Flag; when EAX == 4 starts another Registry check control;

```

At the end, a *PostMessageA* with *uMsg* = *WM_COMMAND* and *wParam* = *0BB8h* is invoked. You know what it means? EhEh ...

This is a possible flowchart for the general Protection's scheme:



Now, we are ready to analyze the Checks' routines of License Code.

5.4.3 GENERAL CHECK'S SCHEME

Figure 87 shows the first part of a more complex subroutine that identifies General Check's scheme used in this target:

```

00410640 SUB ESP,444
00410646 PUSH 11
00410648 CALL dwice.00427220
0041064D ADD ESP,4
00410650 CMP EAX,0F
00410653 JMP NEAR DWORD PTR DS:[EAX*4+address of indexes table]
...
A Heap object is randomly selected to be copied in the stack area;
...
0041073F OR ECX,-1
00410742 XOR EAX,EAX
00410744 PUSH EDI
00410745 LEA EDI, stack's offset of License Code
0041074C REPNE SCAS BYTE PTR ES:[EDI]
0041074E NOT ECX
00410750 DEC ECX
00410751 CMP ECX,1E ; ECX = Length of Serial;
00410754 JE Next
...
Next
00410765 PUSH Parameter passed by address
00410766 CALL Hashing Block procedure
0041076B MOV ECX, index of License Code Buffer
...
00410786 MOV CL,BYTE PTR SS:[Base offset of License Code + ECX]
0041078D MOV EDX,EAX ; EAX is a return's value of the Hashing Block Procedure;
0041078F AND EDX,1F ; EDX is the index of Hash String Buffer;
00410792 CMP CL,BYTE PTR SS:[Base offset of Hash String + EDX*2]
00410796 JE Next Check
00410798 POP EDI
00410799 XOR EAX,EAX ; if EAX == 0, we did a mistake!;
0041079B ADD ESP,444
004107A1 RETN
...
Next Check ←
004107A2 MOV ECX, another index of License Code
...
004107BA MOV CL,BYTE PTR SS:[ Base offset of License Code + ECX]
004107C1 MOV EDX,EAX
004107C3 SHR EDX,5
004107C6 AND EDX,1F ; EDX is another index of Hash String Buffer;
004107C9 CMP CL,BYTE PTR SS:[ Base offset of Hash String + EDX*2]
004107CD JE Next next Check
004107CF POP EDI
004107D0 XOR EAX,EAX ; if EAX == 0, we did a mistake!;
004107D2 ADD ESP,444
004107D8 RETN
...
Next next Check
...

```




	Hashing Block	Input	Output:32 bit value
1° Time's Flag Control	Direct Code 0042521Bh...00425278h	License Code [15:10]	3 indexes compressed
	CALL dwice.00424DE0h	License Name	2 indexes compressed
	CALL dwice.00427550h	License Name, CDATA Buffer	5 indexes compressed
2° T. Flag Control	CALL dwice.004100C0h	License Name	3 indexes compressed
3° T. Flag Control	CALL dwice.00418220h	License Name	4 indexes compressed
4° T. Flag Control	CALL dwice.004190D0h	License Name	6 indexes compressed+1 index of 2 bits

Table 1. Clearly, the offsets present in this table are relative, they will change from PC to PC.

Before to describe every Hashing Block, I will spend a few word to the only condition used to verify the correctness of License Code's elements: **License Code[i] = Hash String [Base offset + 2*i]**. This test can be execute in the form I have presented before (green code's blocks), or it can be "obfuscated" in this way:

<pre> MOV EAX,DWORD PTR DS:[9EEEC8] PUSH ESI XOR ESI,ESI CMP EAX,ESI JLE SHORT dwice.0040F892 DEC EAX CMP EAX,4 MOV DWORD PTR DS:[9EEEC8],EAX JNZ SHORT dwice.0040F84C CALL dwice.004186C0 TEST EAX,EAX JNZ SHORT dwice.0040F892 MOV DWORD PTR DS:[9EEEC94],1 JMP SHORT dwice.0040F892 CMP EAX, JNZ SHORT dwice.0040F86C CALL dwice.0041A0C0 TEST EAX,EAX JNZ SHORT dwice.0040F892 MOV DWORD PTR DS:[9EEEC8],ESI MOV DWORD PTR DS:[9EEEC98],1 JMP SHORT dwice.0040F892 CMP EAX, JNZ SHORT dwice.0040F878 CALL dwice.0041A100 JMP SHORT dwice.0040F856 CMP EAX, JNZ SHORT dwice.0040F884 CALL dwice.0041A140 JMP SHORT dwice.0040F856 CMP EAX,ESI JNZ SHORT dwice.0040F892 MOV DWORD PTR DS:[9EEEB8],9 </pre>	<pre> EAX = [9EEEC8], Time Flag; When EAX == LEA EDX,DWORD PTR SS:[ESP+140] PUSH EDX CALL dwice.00418220 MOV ECX,DWORD PTR DS:[4F1498] ADD ESP,4 CMP ECX,100 MOV DWORD PTR DS:[9EEEC0],EAX </pre>
<pre> Case 5 of switch 0040F82C [9EEEC94] == Flag control, if it sets to </pre>	<pre> Case 4 of switch 0040F82C [9EEEC8] = 0, reset the Time Counter; [9EEEC98] == Flag control, if it sets to </pre>
<pre> Case 3 of sw </pre>	<pre> Case 2 of sw </pre>
<pre> Default case [9EEEB8] == </pre>	<pre> MOV EAX,DWORD PTR DS:[9EEEC0] MOV ECX,DWORD PTR DS:[4F14A0] SHR EAX,0A AND EAX,1F CMP ECX,100 JNB SHORT dwice.0041A11C XOR EAX,EAX RETN MOV CL, BYTE PTR DS:[ECX+4F1220] AND EAX,1F PUSH EBX XOR EDX,EDX MOV BL, BYTE PTR DS:[EAX*2+4F0FE0] POP EBX SETE DL MOV EAX,EDX RETN </pre>

1. A Hashing procedure is calculated

2. The result of Hashing Block is stored in a COSTANT ADDRESS

3. At every value of Time's Flag, it's executed the same test by these calls. For simplicity, i reported only one of these subroutines.

This address constant changes from call to call

10.
- ECX = Index of license Code;
- EAX = Index of Hash String;
- [4F1220h] = Base offset of License Code;
- [4F0FE0h] = Base offset of Hash String;

Figure 91.

As you can see in Figure 91, stepping in to the call `dwice.004186C0h`, (1. Red arrow) we recognize the first part of General Check's scheme; the only difference is the absence of green block code. At its place, there's a simple store operation of 32 bits value in to an address constant (2. Blue comment). Now, for every step of Time's Flag (Figure 91). It's stored at the address constant `[9EEEC8h]`, a different subroutine is invoked (3. Green comment): here, thanks to hash value previously stored, the same checks of green code block in the general Check's scheme are executed (10. Black Comment).

You can read my Dvice's comments, exported thank to LCB plug in, for a deeper analysis... ;-).

Now I will explain the Hashing Blocks routines presented in the previous table.

5.4.4 HASHING BLOCKS

- 1° Time's Flag Control - 1° Hashing Block

```

XOR EDI,EDI
...
XOR ESI,ESI
First Loop:
MOV CL,BYTE PTR SS:[Offset of License Code[10h]+EDI]
LEA EDX, Offset of Hash String
PUSH ECX
PUSH EDX
CALL dwice.00424950
ADD ESP,8
CMP EAX,0
JL Error Code
MOV ECX,ESI
ADD ESI,4
SHL EAX,CL
MOV ECX,Temp Value
OR ECX,EAX
INC EDI
CMP ESI,18
MOV Temp Value,ECX
JB First Loop
MOV ESI,987AC16Bh
MOV EAX,ECX
MOV EDI,EAX
MOV ECX,6
Second Loop:
MOV EBX,ESI
MOV EDX,EAX
SHL EBX,5
AND EDX,0F
SUB EBX,ESI
XOR EDX,EBX
SHR EAX,4
DEC ECX
MOV ESI,EDX
JNZ Second Loop

```

The registers EDI and ESI are used respectively as index of License Code Buffer and counter of First Loop. Here, every element of License Code belonging to interval 10h...15h, must to be also an element of Hash String at the even position: License Code[i] == Hash String [2*j] (do you recognize this condition?). The subroutine underlined in red returns in EAX proper j index. Now, for every cycle, the counter i (stored in ESI), is incremented by 4 and it's used as 2 power; then, j is multiplied to 2^(4*i) and stored in a temp value. This result is OR-ed with the value obtained in the next cycle. So, at the end: ECX == OR (j[i] * 16ⁱ) where i belongs to interval [0...6[and j depends to i. In the green code block, every nibble of ECX is then XOR-ed with a value stored in EBX initialized to 987AC16Bh and multiplied 2⁵-1. Then, the result is copied in ESI and the entire cycle is repeated 6 times.

This Hashing Block produces a 32 bit value containing 3 indexes compressed used to verify License Code[16h:18h].

- 1° Time's Flag Control - 2° Hashing Block, 2° and 3° Time's Flag Controls

I decided to group these tests because they present, more or less, the same structure and they have the same algorithm to compute the 32 bits hash value: Carry Less Multiplication.

Normally, when we must to multiply 2 numbers, we apply the algorithm of the partial sums; for example, if we have 15 and 12 and we must to do its product, we will make:

$$15 \times 12 = 15 \times (1 \times 10 + 2) = (15 \times 1) \times 10 + (15 \times 2) = 150 + 30 = 180$$



Clearly, also 15 must to be decomposed in to positional notation! Now, in a digit system, the presentation's base is obviously 2, so:

$$(15)_{10} = (0F)_{16} = (1111)_2 ;$$

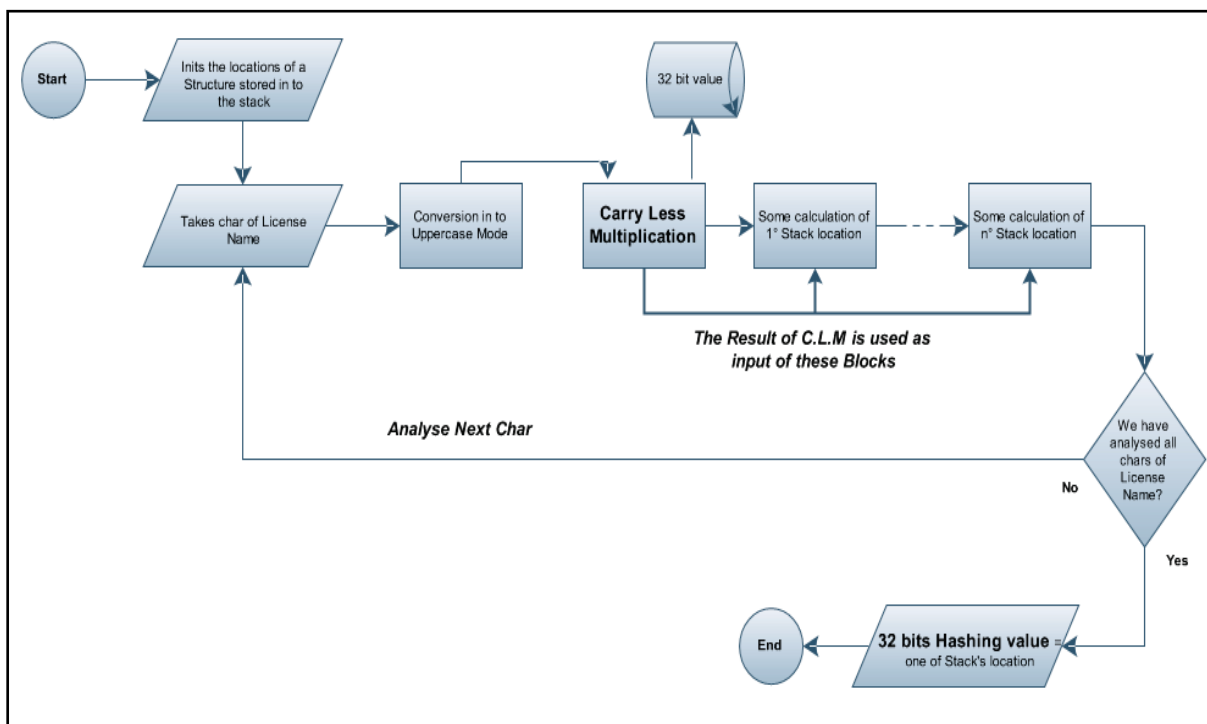
$$(12)_{10} = (0C)_{16} = (1100)_2 ;$$

$$\begin{aligned} (15)_{10} \times (12)_{10} &= (1111)_2 \times (1100)_2 = (1111)_2 \times (1 \times 2^3 + 1 \times 2^2) = \\ &= (1111)_2 \times 2^3 + (1111)_2 \times 2^2 = (1111000)_2 + (111100)_2 = (10110100)_2 = \\ &= (180)_{10} \end{aligned}$$

Then, the partial sums can to be implemented in the assembly language as a sequence of *SHL* and *ADD* instructions; instead, in Carry Less Multiplication, the *ADD* instructions are substituted with *XOR* ones. Considering the previous example and identifying the Carry Less Multiplication with ' * ' symbol:

$$\begin{aligned} (15)_{10} * (12)_{10} &= (1111)_2 * (1100)_2 = (1111)_2 * (1 \times 2^3 + 1 \times 2^2) = \\ &= (1111)_2 \times 2^3 \wedge (1111)_2 \times 2^2 = (1111000)_2 \wedge (111100)_2 = (1000100)_2 = \\ &= (68)_{10} \quad (\wedge = XOR \text{ symbol}) \end{aligned}$$

As you can notice, a great difference between normal and Carry Less multiplication, is the bit's number of final product: in the first, this is equal to sum of 2 bit's numbers; in the second, the result has the same bit's number of greater multiplier. After this brief Mathematical introduction, I can present you the Flowchart of Hashing Blocks belonging to this group:



The 32 bit value used as input in a Carry Less Multiplication block, is stored in a particular way: the position of every bit set to 1 is saved in `.data` section or, from a mathematical point of view, there's every exponent of 2 power associated to bits set to 1. For example, if we consider the number $(10)_{10} = (1010)_2$:

1	0	1	0	<i>Value</i>
3	2	1	0	<i>Position</i>

Inside the program, this number should be stored as a structure of 2 dwords: (3,1). Thanks to these infos, the following figure should be clear:

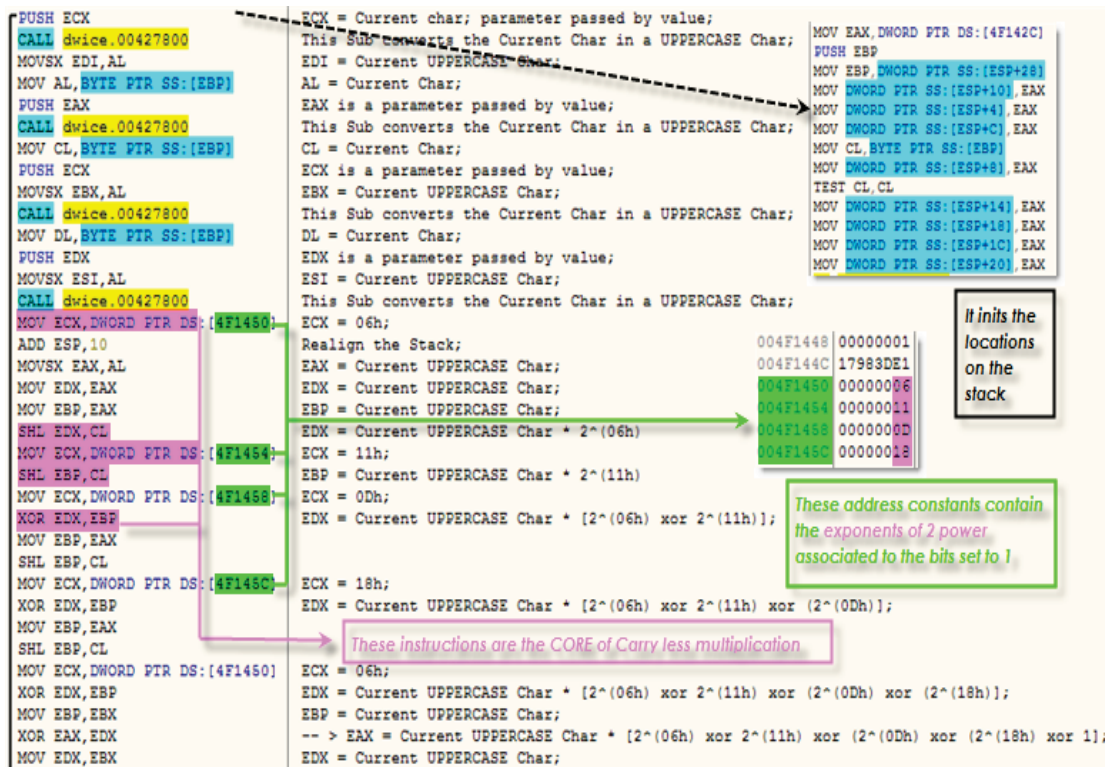


Figure 92.

I have found to mark the init phase (pointed to black arrow) and base instructions used to Carry Less Multiplication (violet color); instead, the address constants underlined in green, identify the representation of a 32 bit value that changes from Hashing Block to Hashing Block. For example, in the figure above, this number is:

$$(18h, 11h, 0Dh, 06h) = (0000\ 0001\ 0000\ 0010\ 0010\ 0000\ 0100\ 0000)_2 = (01022040)_{16}$$

It's clear, no? For the following analysis, we can identify the Carry Less Multiplication Block with this function: Unsigned int C.L.M(int, int). So, in this context, we can write:

$$\text{HashingValue} = \text{C.L.M}(\text{UpLicenseName}[i], 32 \text{ bit value});$$

The first parameter is the current char of License Name converted in Uppercase mode.

▪ 1° Time's Flag Control - 3° Hashing Block

In reality, the subroutine invoked in this group, is used many times and not only to return a 32 bit value used to verify License Code. In general terms, we can define its prototype as: Unsigned int CDATAHash(short int, short int, *char[]); the first parameter, passed by value, is a binary digit (0 or 1) and identifies the type of calculation inside this subroutine; the second parameter, passed by value, represents the number of bytes to elaborate in the third parameter passed by address. In this context, this function becomes:

$HashingValue = CDATHash(1,19h,*LicenseCode)$

Naturally, as you can see in Table 1, there's also another parameter inside this procedure: a **CDATA** buffer long 400 bytes and stored in *.data* section. It's used in this way:

```

MOV EDI, Third parameter;
MOV EBP, Second parameter;
Here:
MOV EAX, First parameter;
MOV EBX,ESI ; Initially, ESI = -1
SHR EBX,8
TEST EAX,EAX
JE 1° calculation
;2° calculation
MOV AL,BYTE PTR DS:[EDI]
INC EDI
PUSH EAX
CALL dwice.00427800 ; It's converts the literal char in to
upper case!;
AND EAX,0FF
AND ESI,0FF
ADD ESP,4
XOR EAX,ESI
MOV ESI,DWORD PTR DS:[EAX*4+Base offset of CDATA buffer]
JMP Next
1° calculation:
XOR EAX,EAX
AND ESI,0FF
MOV AL,BYTE PTR DS:[EDI]
XOR ESI,EAX
INC EDI
MOV ESI,DWORD PTR DS:[ESI*4+ Base offset of CDATA buffer]
Next:
XOR ESI,EBX
DEC EBP
JNZ Here
...
MOV EAX,ESI
POP ESI
NOT EAX ; EAX is a return value of
this Sub;
RETN

```

As you can notice, the only difference between first type of calculation and second one (yellow code), is the presence of a call (red code) invoked to convert literal char in to uppercase mode. The other instructions are the same: after a XOR operation, EAX is the index (multiply of 4) of CDATA Buffer; ESI stores the Dword pointed by EAX and its value is XOR-ed with EBX that contains the result of precedent loop; EBP is its counter. At the end, EAX stores return value of this subroutine (grey block).

Now it's time to talk about last test...

▪ 4° Time's Flag Control

The structure of this check is not very different by the flowchart presented before; there 's always a init phase of a Structure saved in to the stack and there's a compute phase where the Structure's locations are elaborated in a certain way; the final phase selects one of these as 32 bit Hash Value. The main difference stays in compute phase where the **Carry Less Multiplication Block** is substituted with a chain of **XOR – ing and Permutation Blocks**:

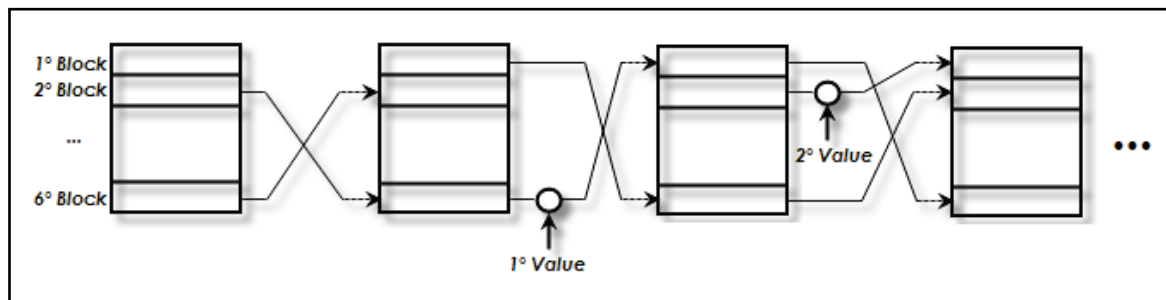


Figure 93. Cryptographic scheme inside 4° Time's Flag Control

In the `CALL dwice.004190D0h`, that implements the Hashing Block of this Time's Flag Control, we can recognize 3 types of permutation:

	Subroutines	N° Blocks to permute (or parameters passed by address)
1° Permutation's Type	<code>CALL dwice.00419060h</code>	2
2° Permutation's Type	<code>CALL dwice.00419080h</code>	3
3° Permutation's Type	<code>CALL dwice.004190A0h</code>	4

Table 2. As always, the offsets present in this table are relative, they will change from PC to PC.

The three Subroutines above are invoked in this way:

<pre>LEA EAX, DWORD PTR SS:[ESP+48] LEA ECX, DWORD PTR SS:[ESP+60] PUSH EAX PUSH ECX CALL dwice.00419060</pre>	<p>EAX = address of 1 block; ECX = address of 6 block; First parameter passed by address: offset of 1 block; Second parameter passed by address: offset of 6 block; This sub scrambles the positions of two blocks;</p>
Ctrl + R = Step in to...	
<pre>MOV EDX, DWORD PTR SS:[ESP+8] MOV EAX, DWORD PTR SS:[ESP+4] PUSH ESI MOV ESI, DWORD PTR DS:[EDX] MOV ECX, DWORD PTR DS:[EAX] MOV DWORD PTR DS:[EAX], ESI MOV DWORD PTR DS:[EDX], ECX POP ESI RETN</pre>	<p>EDX = offset of first block; (first parameter); EAX = offset of second block; (Second parameter); save ESI in to the Stack; ESI = first block; ECX = second block; First block goes in the position of Second block; Second block goes in the position of First block; restore original value of ESI;</p>

Figure 94. Description of `CALL dwice.00419060h`; the addresses of blocks change from invoke to invoke.

LEA EAX, DWORD PTR SS:[ESP+40]	EAX = address of 2 block;
LEA ECX, DWORD PTR SS:[ESP+48]	ECX = address of 4 block;
PUSH EAX	First parameter passed by address: offset of 2 block;
LEA EDX, DWORD PTR SS:[ESP+48]	EDX = address of 3 block;
PUSH ECX	Second parameter passed by address: offset of 4 block;
PUSH EDX	Third parameter passed by address: offset of 3 block;
CALL dwice.00419080	This Sub permutes the blocks: [first parameter] -> [second parameter]

Step in to...

MOV EDX, DWORD PTR SS:[ESP+8]	EDX = offset of second block; (second parameter);
MOV EAX, DWORD PTR SS:[ESP+4]	EDX = offset of third block; (third parameter);
PUSH ESI	save ESI in to the stack;
MOV ESI, DWORD PTR DS:[EDX]	ESI = second block;
MOV ECX, DWORD PTR DS:[EAX]	ECX = third block;
MOV DWORD PTR DS:[EAX] , ESI	Second block goes in the position of Third block;
MOV EAX, DWORD PTR SS:[ESP+10]	EAX = offset of first block; (first parameter);
MOV ESI, DWORD PTR DS:[EAX]	ESI = first block;
MOV DWORD PTR DS:[EDX] , ESI	First block goes in the position of Second block;
MOV DWORD PTR DS:[EAX] , ECX	Third block goes in the position of First block;
POP ESI	restore the value of ESI;
RETN	

Figure 95. Description of CALL *dwice.00419080h*; the addresses of blocks change from invoke to invoke.

LEA EAX, DWORD PTR SS:[ESP+44]	EAX = address of 6 block;
LEA ECX, DWORD PTR SS:[ESP+2C]	ECX = address of 1 block;
PUSH EAX	First parameter passed by address: offset of 6 block;
LEA EDX, DWORD PTR SS:[ESP+3C]	EDX = address of 4 block;
PUSH ECX	Second parameter passed by address: offset of 1 block;
LEA EAX, DWORD PTR SS:[ESP+3C]	EAX = address of 3 block;
PUSH EDX	Third parameter passed by address: offset of 4 block;
PUSH EAX	Fourth parameter passed by address: offset of 3 block;
CALL dwice.004190A0	This Sub permutes the blocks: [first parameter] -> [second

Following in to...

MOV EDX, DWORD PTR SS:[ESP+8]	EDX = offset of third block; (third parameter);
MOV EAX, DWORD PTR SS:[ESP+4]	EAX = offset of fourth block; (fourth parameter);
PUSH ESI	save ESI in to the stack;
MOV ESI, DWORD PTR DS:[EDX]	ESI = third block;
MOV ECX, DWORD PTR DS:[EAX]	ECX = fourth block;
MOV DWORD PTR DS:[EAX] , ESI	Third block goes in the position of Fourth block;
MOV EAX, DWORD PTR SS:[ESP+10]	EAX = offset of second block; (second parameter);
MOV ESI, DWORD PTR DS:[EAX]	ESI = second block;
MOV DWORD PTR DS:[EDX] , ESI	Second block goes in the position of Third block;
MOV EDX, DWORD PTR SS:[ESP+14]	EDX = offset of first block; (first parameter);
MOV ESI, DWORD PTR DS:[EDX]	ESI = first block;
MOV DWORD PTR DS:[EAX] , ESI	First block goes in the position of Second block;
MOV DWORD PTR DS:[EDX] , ECX	Fourth block goes in the position of First block;
POP ESI	restore the value of ESI;
RETN	

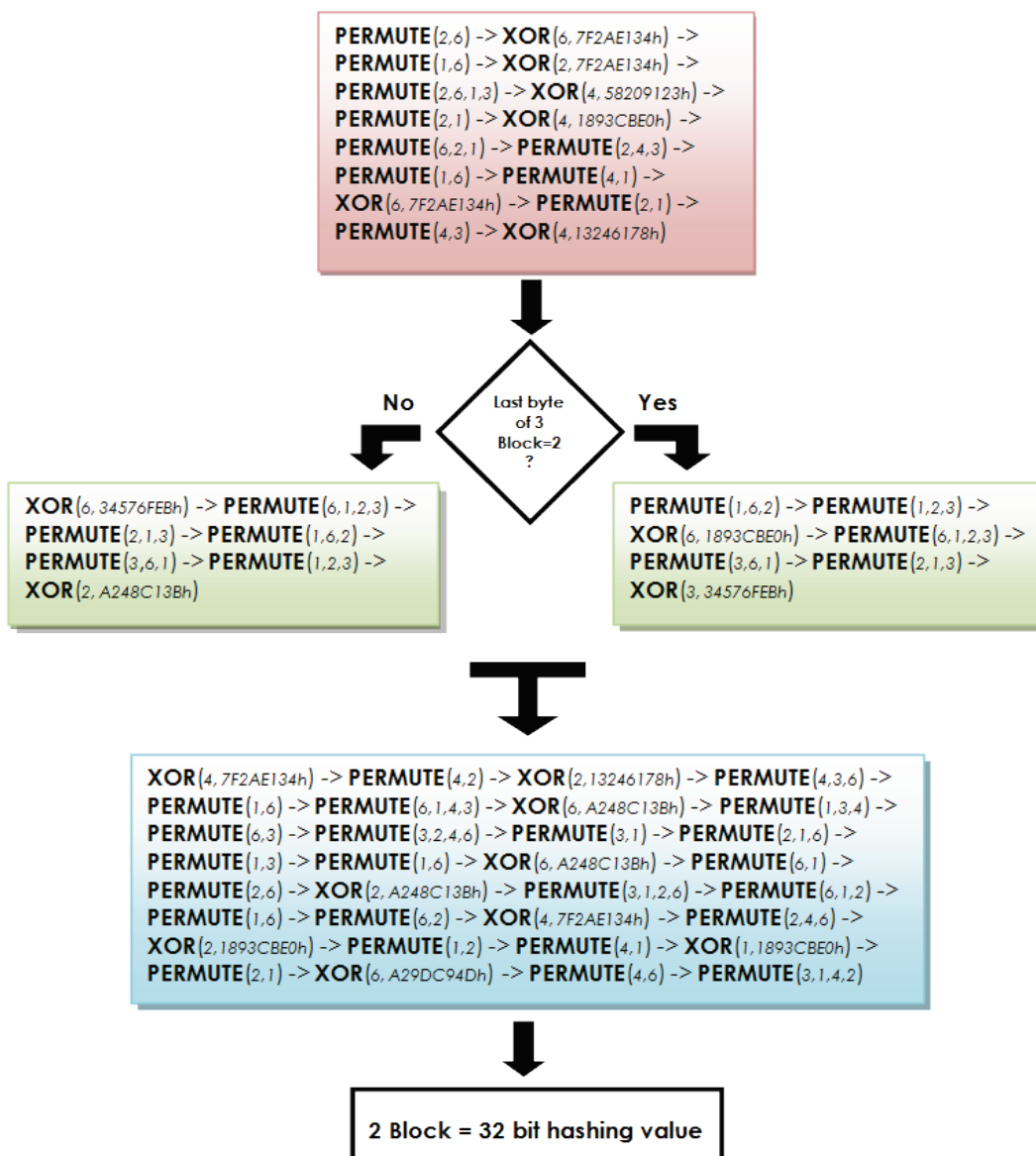
Figure 96. Description of CALL *dwice.004190A0h*; the addresses of blocks change from invoke to invoke.

It's not necessary to comment these figures because I explained every instruction in OllyDbg; instead, to describe the whole Cryptographic scheme, I'll introduce 2 'Mathematical operators':

PERMUTE (*1° Block, 2° Block, ..., 6° Block*); first block goes in to second block, second block goes in to third etc etc...

XOR (*Block, dword*); naturally, the 32 bit value change from XOR to XOR.

Then, after a first phase where a structure of 6 dword is stored in to the stack and initialized, the Hashing Block does the following operations:





Funny, isn't it? Indeed, every colorized block can be drastically simplified with a barrier of XOR operations and **ONE** permutation; for example, we consider the first 4 operations in red block above:

PERMUTE(2,6) = the 2° block goes in the 6° position and the 6° block goes in the 2° position; then the block in the 6° position (2° block) is xor-ed with the value *7F2AE134h*. So:

$$\text{PERMUTE}(2,6) \rightarrow \text{XOR}(6, 7F2AE134h) \equiv \text{XOR}(2, 7F2AE134h) \rightarrow \text{PERMUTE}(2,6)$$

(Simply rule: *scramble always PERMUTE with XOR and change the block xor-ed with what precedes in PERMUTE operation. If block xor-ed is independent by PERMUTE operation, is will not be changed*)

Considering the other 2 operations, we have:

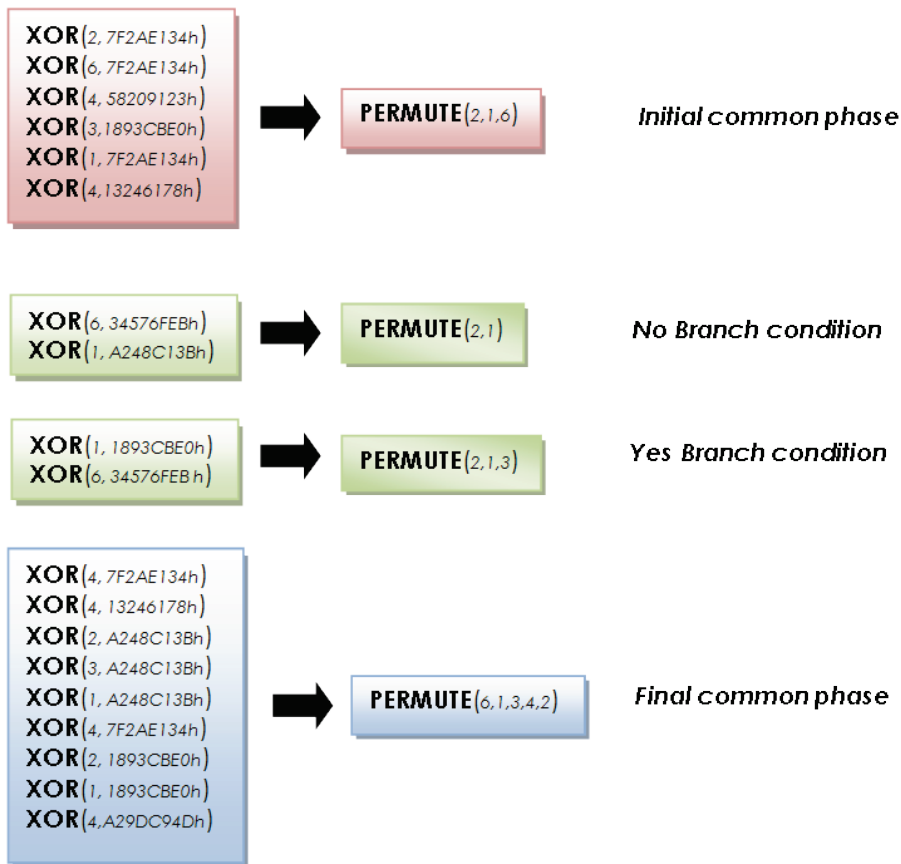
$$\text{XOR}(2, 7F2AE134h) \rightarrow \text{PERMUTE}(2,6) \rightarrow \text{PERMUTE}(1,6) \rightarrow \text{XOR}(2, 7F2AE134h) \equiv$$

$$\text{XOR}(2, 7F2AE134h) \rightarrow \text{PERMUTE}(2,6) \rightarrow \text{XOR}(2, 7F2AE134h) \rightarrow \text{PERMUTE}(1,6) \equiv$$

$$\text{XOR}(2, 7F2AE134h) \rightarrow \text{XOR}(6, 7F2AE134h) \rightarrow \text{PERMUTE}(2,6) \rightarrow \text{PERMUTE}(1,6) \equiv$$

$$\text{XOR}(2, 7F2AE134h) \rightarrow \text{XOR}(6, 7F2AE134h) \rightarrow \text{PERMUTE}(2,1,6)$$

In the last passage, I combine the 2 permutations; you can see References for more infos. So, approaching the problem in this way, I simplified the Cryptographic scheme:



We can further simplify XOR's barriers having one XOR operation for each location in colored block; for example, in blue block, we have 2 XOR operations for 2° location:

$$\mathbf{XOR}(2, A248C13Bh) \rightarrow \mathbf{XOR}(2, 1893CBE0h) = \mathbf{XOR}(2, BADB0ADBh); \text{ simply no?}$$

Before to explain some tricks I used to write a keygen, I would display the Init Phase of this Time's Flag Control:

```

MOV EAX, 578A43D1h
...
MOV ESI, address of the 'Cleaned Name'
PUSH EDI
MOV 5 block, EAX
MOV EDI,ESI
OR ECX,-1
XOR EAX,EAX
REPNE SCAS BYTE PTR ES:[EDI]
MOV EBX, A312B14Ch
MOV EBP, A29DC94Dh
NOT ECX
DEC ECX
PUSH 1
PUSH ECX ;ECX = Length of 'Cleaned Name Buffer';
PUSH ESI
CALL dwice.00427550
XOR EAX,EBX
MOV EDI,ESI ;EDI = ESI = pointer of the 'Cleaned Name';
MOV 1 block, EAX
PUSH 1
PUSH Length of 'Cleaned Name Buffer'
PUSH ESI
CALL dwice.00427550
MOV EBX,17983DE1h;
...
XOR EAX,EBX
...
MOV 2 block, EAX
PUSH 1
PUSH Length of 'Cleaned Name Buffer';
PUSH ESI
CALL dwice.00427550
XOR EAX, 58209123h
MOV EDI,ESI ;EDI = ESI = pointer of the 'Cleaned Name';
MOV 4 block, EAX
...
PUSH 1
PUSH Length of 'Cleaned Name Buffer'
PUSH ESI
CALL dwice.00427550
XOR EAX,EBP
MOV EDI,ESI ;EDI = ESI = pointer of the 'Cleaned
Name';
XOR EAX, 289F1E44h;
MOV 3 block, EAX
...
PUSH 1
PUSH Length of 'Cleaned Name Buffer';
PUSH ESI
CALL dwice.00427550
MOV ECX, 5 block;
...
XOR EAX,ECX
MOV 6 block, EAX

```

Do you recognize the call underlined in red? Yes, it's the same used in 3° Hashing Block of 1° Time's Flag control:

EAX = CDATAHash(1, ,)

where the 2° and 3° parameters are the length (yellow instructions) and offset of 'Cleaned Name'. In the green code I stressed the storing operation of locations in to the stack. We can synthesize these instructions in this way:

```

1° block = CDATAHash(1,a,b) ^ A312B14Ch
2° block = CDATAHash(1,a,b) ^ 17983DE1h
3° block = CDATAHash(1,a,b) ^ 8A02D709h
4° block = CDATAHash(1,a,b) ^ 58209123h
5° block = 578A43D1h
6° block = CDATAHash(1,a,b) ^ 5° block

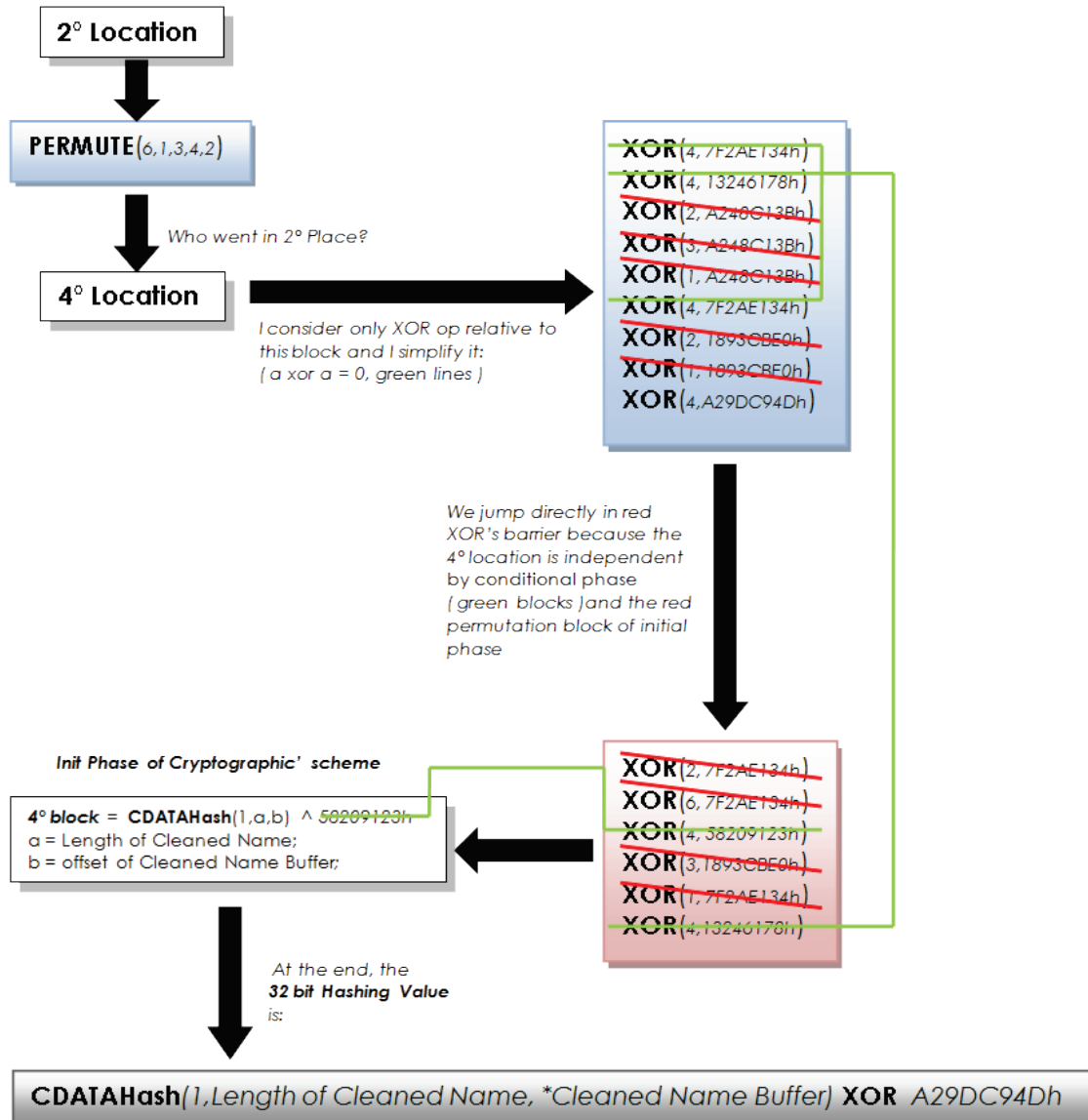
```

a = Length of Cleaned Name;
b = offset of Cleaned Name Buffer;



5.5 SUGGESTIONS TO PROGRAM A KEYGEN

There are many plug-ins for OllyDbg that allow to rip instructions, but I prefer to write a Keygen ex novo because, first of all, it's a useful exercise and also because we can optimize some part of code in Check's routines. In fact, if you have understand the previous Flowcharts of Check's routines, you have certainly noticed it's not necessary calculate everything but only changes relative to that location used to store a 32 bit Hashing value. For example, in the 4° Time's Flag control, only the 2° location of Structure saved in to the Stack is important, while the others are superfluous; so, doing little back tracing in the Cryptographic Scheme:



In the same way, we can "reverse" the Hashing's Blocks based to Carry Less Multiplication; for simplicity, I write these instructions in pseudo C language:



```

Int Temp;
HashingValue = 0x0A312B14C;
for ( i = 0, i < CleanedNameLength, i++)
{ Temp = C.L.M(UplicenseCName[i],0x01022041);
  HashingValue = 0x25 * HashingValue + Temp;
};

```

2° Time's Flag Control

```

Int Temp;
HashingValue = 0x578A43D1;
for ( i = 0, i < CleanedNameLength, i++)
{ Temp = C.L.M(UplicenseCName[i],0x21020881);
  HashingValue = 0x13 * HashingValue + Temp;
};

```

3° Time's Flag Control

```

Int Temp;
HashingValue = 0x17983DE1;
for ( i = 0, i < CleanedNameLength, i++)
{ Temp = C.L.M(UplicenseName[i],0x22208241);
  HashingValue = HashingValue ^ Temp;
  If (HashingValue < 0)
    HashingValue = 2 * HashingValue +1;
  Else HashingValue = 2 * HashingValue;
};

```

1° Time's Flag Control – 2° Hashing Block

For others Check's routines, you can see the asm source of my Keygen, it's quite commented! ;)

5.6 ADDENDUM – EXERCISE

Don't worry, it's a my little proposal; if you traced the program as I described here, you have surely notice many duplications of calls inside Time's Flag controls (see Figure 14.) and also you can see others address constants that seem to work like Time's Flags itself... why? Once registered the game, where 's saved the License Code? Exact, in Window' registry, more precisely in: **HKEY_LOCAL_MACHINE\SOFTWARE\WildSnake Software\Dwice\1.0.**

Here, both License Name and License Code are stored as REG_SZ. So, when we launch the game, the program tests the presence of these keys and, if they exist, it does the same controls; for exercise, you can trace this Checks and discover new Time's Flags (Registry Time's Flags! ;-D).

It's important to underline as License Code is saved in to registry in a cryptic form; I discovered it tracing the case *BB8* of CALL *dwice.0040E7A4h* do you remember the "final good judgment" ?). Inside it, we can find:

```

PUSH Offset of License Code
PUSH Buffer of Cryptic License Code
CALL dwice.00424AA0

```

The CALL underlined in red, simply crypts the License Code passed as 1° parameter by address and the result is stored in Cryptic License Code's Buffer (2° parameter passed by address).

Stepping inside this, we can analyze these instructions:

```

XOR ESI,ESI
Loop:
LEA EDI, address of current byte of Cryptic Serial Buffer;
MOV AL, current byte of License Code;
PUSH EAX
PUSH address of Hash String
CALL dwice.00424950
...
TEST EAX,EAX
JL Error code
CMP ESI,5
JE 1° group
CMP ESI,9
JE 1° group
CMP ESI,0B
JE 1° group
CMP ESI,0C
JE 1° group
CMP ESI,4
JE 2° group
CMP ESI,6
JE 2° group
CMP ESI,0A
JE 2° group
CMP ESI,0F
JE 2° group
TEST ESI,ESI
JE 3° group
CMP ESI,2
JE 3° group
CMP ESI,8
JE 3° group
CMP ESI,0E
JE 3° group
MOV CL,[Hash Table's base offset + 3 + 4*j];
JMP Next1
3° group
MOV DL,[Hash Table's base offset + 2 + 4*j];
MOV BYTE PTR DS:[EDI],DL
JMP Next
2° group
MOV AL,[Hash Table's base offset + 1 + 4*j];
MOV BYTE PTR DS:[EDI],AL
JMP Next
1° group
MOV CL,[Hash Table's base offset + 4*j];
Next1:
MOV BYTE PTR DS:[EDI],CL
Next:
INC ESI
CMP ESI,1E
JL Loop
POP EDI
POP ESI
POP EBP
MOV EAX,1
POP EBX
RETN

```

The subroutine underlined in red, receives in input 2 parameters: a byte of License Code and the offset of Hash String; so, it returns in EAX a index j that satisfies the same condition seen before:

License Code[i] = Hash String [Base offset + $2*j$]. Then, using ESI as pointer to an Hash Table, the target maps its bytes in to a Cryptic Serial Buffer. Indeed, as you can see to left, we can identify 4 main groups for Hash Table's index:

1° group = (5,9,0Bh,0Ch);

2° group = (4,6,0Ah,0Fh);

3° group = (0,2,8,0Eh);

4° group = others indexes;

For every group, there's a different displacement in the instruction:

MOV reg,[H.Table offset + disp + $4*j$];

where displacement can be:

0 -> for 1° group;

1 -> for 2° group;

2 -> for 3° group;

3 -> for 4° group.



So, the Hash Table can be partitioned in this way:

36	31	73	7A	32	7A	6A	31	37	32	46	51	33	71	4B	61
35	41	64	5A	72	33	6C	73	30	51	53	78	31	45	41	41
51	65	4F	57	67	52	51	32	50	64	50	44	4C	58	72	66
48	34	65	56	39	46	54	76	4A	66	77	43	57	72	70	33
44	38	6D	67	56	59	6E	72	62	48	7A	42	7A	42	78	59
58	4E	5A	65	66	75	33	52	41	4A	4E	68	73	62	56	54
38	4D	37	4D	6D	70	30	55	45	6B	39	39	74	30	35	38
34	6F	4D	70	6B	69	34	6E	4E	39	36	69	79	4C	31	4F
AC	80	29	51	49	6E	73	74	61	6C	6C	20	50	61	74	68

As exercise, you can find the subroutine that implements the inverse operation: from Cryptic License Code to Clear License Code... (little suggestion: you can trace any reference to offset constant `00598974h`; in fact, in my pc, this is the Base offset of Hash Table).

Uff... finally we can tell **TO HAVE FINISH!!!** I hope that this tutorial you enjoyed and still sorry if I was not clear as I would have been.

5.7 REFERENCES

- [1] "Permutation", <http://en.wikipedia.org/wiki/Permutation>
- [2] "Carry-Less Multiplication and Its Usage for Computing The GCM Mode", <http://softwarecommunity.intel.com/articles/eng/3787.htm>

5.8 GREETINGS

First of all, I want to thank Shub–Nigurrath for editing and revising this long tutorial, all members of ArTeam, Quequero, Evilcry and all U.I.C; thanks also go to Oleh Yuschuk, Scherzo and NtosKrn1 for their useful tools but especially I thank you who have read my tutorial.

Dedicated to my love Angela,
My mother and my father

[In the Supplements folder "0.5 Gyver75" you can find also:

- LCB Olly plugin with sources and comments for the code
- Full ASM sources of the keygen]



6 LIVE DEBUGGING SYMBIAN APPLICATIONS USING OR NOT USING IDA BY ARGV

Here we present a short readme of what's into the Supplements folders. In folder 0.6 argv you can find 4 videos about live Symbian debugging of applications, with and without IDA and from PC or directly from the phone.

This video series dedicated to Symbian covers all the possible ways available to live debug Symbian applications, and drop away dead-listings.

You have 3 Debug Videos + Teaser One. And also, as you can see they are arranged in folders.

- **First Debug Video is about Debugging Application using a built in Emulator.**

You may think Emulator is crap and not worth exploring. Well, you are very wrong. Emulator is perfect way to test and debug applications before application goes to phone. Emulator also supports debugging which you can use to find glitches and fix them before transferring the file to the phone. It is true that Emulator isn't real phone, but do you want a buggy application on the phone? I think so also. Use debugging on Emulator ALWAYS before trying to run/debug it on the phone. Emulator has its own console you can invoke and track the progress as you are stepping through the code.

- **Second Debug Video is about Debugging Application LIVE on the phone.**

In the second video you will see how to debug an application LIVE on the phone. However you have many obstacles here. You cannot correct errors like in Emulator, you can only view how it looks on the phone. This is not a debugger of choice. Emulator is far better. You can try to debug on the phone, but execution path is very quick so you might miss some stuff going on behind the scene. Debugging on phone can be useful in cases where you need to see how application behaves on real device, but still, it is weak as a debugger.

- **Third Video is about Debugging Application using IDA Pro 5.4**

Last method is by using IDA with Remote Symbian Debugger that connects to phone's AppTRK. This is most powerful type of debugging because you are literally stepping through applications code. In IDA, you can see the stance of registers and also you can see memory layout and you can write your own memory regions.



6.1 SOME FAQ

Finally here are few questions we thought you might ask:

Q: Do these need a hacked Symbian phone?

-- *No, but hacking the phone could save time.*

Q: To which Symbian versions these solutions apply?

-- *These solutions apply to ANY 3rd device. It can be used in Second Edition but that is different story.*

Q: Which type of debugging event I can or I cannot handle?

-- *You can debug userland programs. And without extremely hard work you can't debug protected areas on the phone.*

Q: Is the TRK the only option available, is there any other similar product?

-- *Yes, Target Resident Kernel is only one that support debugging. Similar product do not exist, at least I am not aware of it.*

Q: TRK relies on undocumented Symbian kernel APIs? If yes the risk isn't that Nokia will drop support for them?

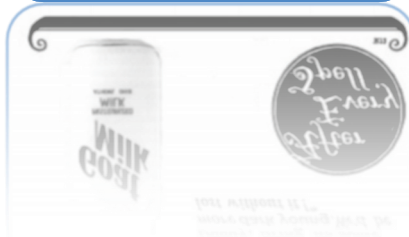
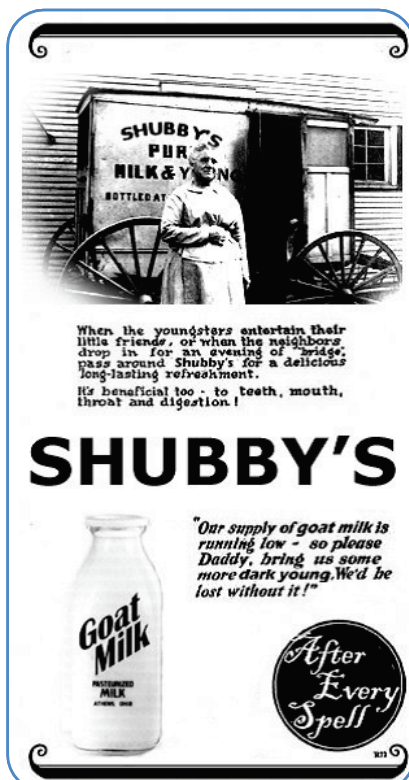
-- *Probably not because big development companies need it. In fact, Nokia updated TRK just recently.*

In the Supplements folder “0.6 argv” you can find:

- **Debugging_real_programs_with_Emulator_by_argv**
a series of 3 videos
- **Live_Debugging_Programs_using_Carbidе_by_argv**
a series of 2 videos plus tools and programs required
- **Debugging_programs_teaser_Video_by_argv.wmv**
the first video to look at
- **Debugging_programs_with_IDA_Pro_by_argv.wmv**
this is probably the second video to look
- **symbian_primer.pdf** from Hex Rays' site, a document explaining how to configure IDA 5.4 for Symbian debugging

Beware tutorials have audio comments

7 INTERVIEW WITH SHUB BY GUNTHER



Gunther: Hi Shub, thank you for your availability. Some simple questions to introduce you to our readers to start with. What is your role at ARTeam? Can you tell me a little bit about yourself and what do you do?

Shubby: Hi everybody. Which is my role in the team? Anyone coming to visit our forum should have seen that I define myself as the commander. It's a joke of course, but more or less that is what I do in the team, motivating people with new ideas, keeping the team joined, handle the forum and system (even in case of crashes) and propose to the other mates possible evolutions. But the team is truly a democratic group. I am not a real commander, I'm rather the one that most often proposes new ideas for discussion. This comes from a personal attitude, I am what is often also called a "cross-fertilizer": what happens to me is that I tend to read a lot of papers and tutorials and start thinking about possible applications with different techniques in the RCE area. I am somehow embarrassed doing interviews because I hate those types of people standing up and telling their truths as if were gold, I then wish to remark that I am just giving opinions, the word "IMHO" should be added at the beginning of any future sentence. ;-)

Gunther: How long have you been involved with Reverse Engineering? How did you become interested in it?

Shubby: This is a story that starts a lot of years ago, my first approach to reversing were with a Commodore C64, at that time the scene was incredibly quite huge and luckily there wasn't still any well defined regulation against piracy. That was, let say, a golden age! Anyway I learnt then to disassemble on the Motorola 68000 architecture (I still have a little handbook called "Pocket guide assembly language for the 68000", printed in 1984!... definitely a prehistoric age) and entered some of the 0days teams of that time (I will not mention them). That scene was mainly made of BBS and few lucky datapac connection points and of course was made through modem transmitting at 1200/2400 BAUDS (approx 240 chars/sec then approx 1920 bit/sec)! When C64 was declared dead I stopped reversing, but your passions returns and I started again few years ago with Palm (reversing Palm programs) and then found ARTeam. What I liked immediately of ARTeam was the attitude to share everything they were doing not only pushing at an increasing the cracks/day ratio. Moreover there were some very talented people! Others joined later, now ARTeam is made of very excellent reversers.



Gunther: Do you use any special tools for RE?

Shubby: the most special tool I use is my brain ;-). Beside this I use quite all the possible tools, but mostly OllyDbg, IDA and few other specialized tools like resources decompiler or specialized disassemblers like DeDe. I like to integrate all the results of these tools just to have different views of the same thing; it helps me understanding the underlying logic of the system and have an higher view, you can call this type of reversing "birdfly reversing". Ah I forgot to say that I also use Total Commander a lot, for almost everything, it's just a substitute of explorer. With the proper plugins it becomes a perfect forensic workstation. What is often forgotten is to look under the hood which are outside of usual tools (evidences or weakness are sometimes so evident that there's no need to even open OllyDbg).

Gunther: If you had to make recommendations for other engineers in your fields, what would you say to them?

Shubby: If you mean reversing that's not my field ;-), but what to say. I don't think I am anything so special to give any advice to anyone, but in general the only thing I suggest is to follow your attitude and to not get shortcuts. RCE is all but a simple art and recently the interest of industry just raise the bar. I posted few months ago on our forum¹ this post: *"Since few years the software industry created a new official competence, the security expert, the secure software development lifecycle, the professional reverse engineer. This was due to the fact that finally reversing for security needs has been regulated by laws and it's now possible to build a career on reversing. This was not possible at the times of +Fravia and industry was suffering from the lack of professional roles, able to effectively reverse programs. The effect of this change of scenario is under eyes of anyone, a great and constant raise of the bar. New protections (themida, securom,...) are very difficult. How many crackers can handle them fully? Not much and those who are, keep their secrets for their own clubs. Of course!"*. I posted this comments on the forum because I clearly see this drift happening. What this means is that if you want to stay at top you'll have to study more (if you're alone) or collaborate more (if you are in a team).

Gunther: How do you really feel about the current situation in RE and software protection?

Shubby: Indeed I already answered partially in the above question, sorry. But I see a lot of different trends happening. First of all the number of professional people doing reverse engineering for work increased. These roles are on both sides of the barricade: malware and fighting. Those people among us not working in RCE and not willing to create malware are somehow compressed between these two colliding worlds. Then the web scene (which is weakly protected) is vulnerable to government on the one hand and releasing competition with Oday (which is well protected) on the other hand. This hybrid situation could only create problems. This is why ARTeam stopped doing any release a long time ago and started to just do tutorials. Tutorials writing is that type of activity that's borderline and (till now) borne. On the other hand most of the reversers sat down on WinXP and 32 bit architectures without expanding their views to other systems (non win32, non-PC), to other architecture (embedded, hardware hacking, 64bits) and to what will come after Vista (W7, OSX). The most common comment I saw from reversers using Vista was "luckily it runs OllyDbg"..this means that they are not getting the real opportunities offered by Vista, isn't it? Whenever WinXP disappears, a lot of well known crackers will do as well. What we did (with iPhone,

¹ <http://www.accessroot.com/arteam/forums/index.php?showtopic=7602>



Symbian, WinCE,...) is to create some alternatives trying to reverse also different systems. This ezine is also a proof.

Gunther: What are you currently working on at the moment and what are you interested in the near future?

Shubby: Indeed lately I was taken out of reversing due to RL issues and due to the forum restoring, the new site setup and this ezine also...a lot of editorial and boring tasks that slowed my interest on active RCE. But was due: luckily I found several ARTeam enthusiasts and people appreciating our efforts and vision which helped us to be here. I think that once all will again be fixed I will take some resting just doing nothing, or better just doing what I was doing before: reading tutorials and reversing them without any additional distraction.

Gunther: Is RE your profession?

Shubby: I cannot answer to this question; surely it's one of my passions.

Gunther: Do you think we are at the point where software protection developers had a hard time protecting their software against you guys?

Shubby: Not indeed I think just the opposite. This is the golden and upmost point in the reversing saga, where reversers reach their more complex results against developers. Software industry started to react professionally and it's eating the disadvantage accumulated in years (mostly due to misleading regulations). What will come is the era of team working: malware is already using teamworking (look at conflicker, it's clearly a result of a team of experts), industry as well of course, Oday teams are already collaborating without telling to anyone and the web reversing "scene" will also have to do it. Why? Just because complexity is too much for a self-made man!

Gunther: you often mention the malware industry, and the professional reversing world. How these two worlds will change the way of protecting programs?

Shubby: Humm my idea of how programs' protection mechanisms will evolve is tied to malware. To start describing the idea I must start from the honeynets (nets specifically prepared to be vulnerable and catch malwares like honey, a type of trap for malware). I started to think that future evolutions of protections and thought that some programs are also like honeynets, they are probably better termed as say "honeyprogs", just prepared to be cracked and "catch" infos from who did it. Just to make it brief the concept is this: suppose you have a program that is important for a specific class of users, like specialists in something, and suppose that the program costs a lot.

There are two possible ways to protect the program: first, few simple evident protections that one generic cracker can see, at some not too easy level, just to not raise suspects. Second, a much more hidden protection that silently somehow collect data on the patching process... or instead, of shouting "I'm



debugged and can't run", just collect information about your PC and you... silently! That information can be used to modify the program behavior (like for games badly cracked for which the game's engine of a badly cracked game becomes "harder" to play, while the game doesn't complain about being a pirated copy), or to actively send information to somewhere...

Is that possible? Sure, it is. Malware is already doing these things. What I think is that developers soon will realize that malware logic could be helpful for them (indeed Sony was the first to realize it, almost a year ago when they were distributing CDs with a self installing rootkit for their DRM system). I then to start thinking that we'll have to use the same tricks that are normally used to test malwares, to avoid programs to catch information and send them to developers. This is exactly the same concept malware does, just tilted. Malware phish information from your system about accounts, start to think about a program that phish information on you, that are trying to crack it.

Moreover malware is teaching us that there are hundreds ways to catch information from the system that normal reversers doesn't even suspect that are possible. This is because nowadays malware analysis is considered a white-hat only domain, but it's not.

This is why I decided to focus the next eZine only on malware analysis, thought as a way to teach us to better reverse any program: tutorials about methodologies, primers, things that are important to know... This is not to tell that we should convert to malware analysis, our malware interest is functional to what we are already doing!

Gunther: What are your future plans? Do you plan on writing any books soon like H.D,Moore?

Shubby: humm I thought of writing something several times, or even to collect some tutorials into a book, but time is always lacking and writing is a really expensive task. It's anyway a pinpointed task and I am just waiting to find time to develop it.

Gunther: What are the directions and future for ARTeam? Any interesting projects in progress?

Shubby: just continue doing what we are already doing, hopefully with the contributions of all the team mates who shared their passions. ARTeam is first of all a group of friends, what I always missed is the opportunity to meet them all, but we are spread in so many places that it will just be impossible!



Gunther: Could you give some pieces of advice for our readers – people who might be going to look for a job in Reverse Engineering field some day?

Shubby: Advice? Humm nothing special indeed... Just that living doing RCE is really hard. Look at those guys officially doing it for work: they are travelling all around the globe to seek works. Second thing do not reveal that once you were crackers, because it exists the so called "Mitnick effect": if you're an ex-cracker anyone will ask you to do courses explaining how to do things and asking to teach them, but no one will ever give you their system for fixing. So after all this will reveal to be an own goal.

Gunther: Our interview seems to be completed. Do you have something to add to our readers? Thank you for your time and keep up the excellent work.

Shubby: Not indeed, I wrote so much and I bored so much that I cannot even think that anyone will still be alive at this point. This interview comes at the end of a giant issue which kept me busy for several months, collecting, fixing and editing all these contributions. First of all I wish to thanks all the authors (and you as well) and all the other mates sharing their experiences on our forums.

So long, and thanks for all the phish!



ARTEAM EZINE #5 CALL FOR PAPERS



ARTeam members are asking for your article submissions on subjects related to Reverse-Engineering.

We wanted to provide the community with somewhere to distribute interesting, sometimes random, reversing information. Not everyone likes to write tutorials, and not everyone feels that the information they have is enough to constitute a publication of any sort. I'm sure all of us have hit upon something interesting while coding/reversing and have wanted to share it but didn't know exactly how. Or if you have cracked some interesting protection but didn't feel like writing a whole step by step tutorial, you can share the basic steps and theory here. If you have an idea for an article, or just something

fascinating you want to share, let us know.

Examples of articles are a new way to detect a debugger, or a new way to defeat debugger detection, or how to defeat an interesting crackme..

The eZine is more about sharing knowledge, as opposed to teaching. So the articles can be more generic in nature. You don't have to walk a user through step by step. Instead you can share information from simple theory all the way to "sources included"

What we are looking for in an article submission:

1. Clear thought out article. We are asking you to take pride in what you submit.
2. It doesn't have to be very long. A few paragraphs is fine, but it needs to make sense.
3. Any format is fine, but to save our time possibly send them in WinWord Office or text format.
4. If you include pictures please center them in the article. If possible please add a number and label below each image.
5. If you use references please add them as footnotes where used.
6. If you include code snippets inside a document other than .txt please use a monospace font to allow for better formatting and possibly use a syntax colorizer
7. Anonymous articles are fine. But you must have written it. No plagiarism!
8. Any other questions you may have feel free to ask

We are accepting articles from anyone wanting to contribute. That means you.

We want to make the eZine more of a community project than a team release. If your article is not used, it's not because we don't like it. It may just need some work. We will work with you to help develop your article if it needs it.

Questions or Comments please visit <http://forum.accessroot.com>