# Such a weird processor

## messing with opcodes

(...and a little bit of PE)

Ange Albertini

28th October 2011

@ange4771
@corkami (news only)


hash days

# (if you read this without the presentation)

HIDDEN SLIDE

- introduce Corkami.com, a RCE site
- why correct disassembly is important for analysis,
  - why undocumented opcodes are a dead end
- a few examples of undocumented opcodes and CPU weirdness
- theory-only sucks, so I created CoST for practicing and testing.
- CoST also tests PE, but it's not enough by itself
- So I documented PE separately, and give some examples.

[version: release 1]

# presented by...

- a reverse-engineering enthusiast
  - ...since dos 3.21
  - Corkami.com
  - Mame (the arcade emulator)
- a malware analyst

# Corka-what ?

- RCE project, only technical stuff
- free to:
  - browse, download
  - test, modify, compile
- updated
- useful daily

- but.... only a hobby !

# what is in Corkami ?

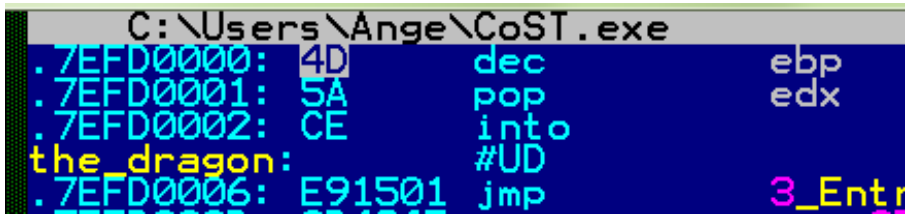
- wiki pages, cheat sheets
- many PoCs
  - hand-written (not generated), minimalists
  - binaries available
- on PDF, x86, PE...
- 100% open
  - BSD, CC BY
    - sources, images, docs

# Story

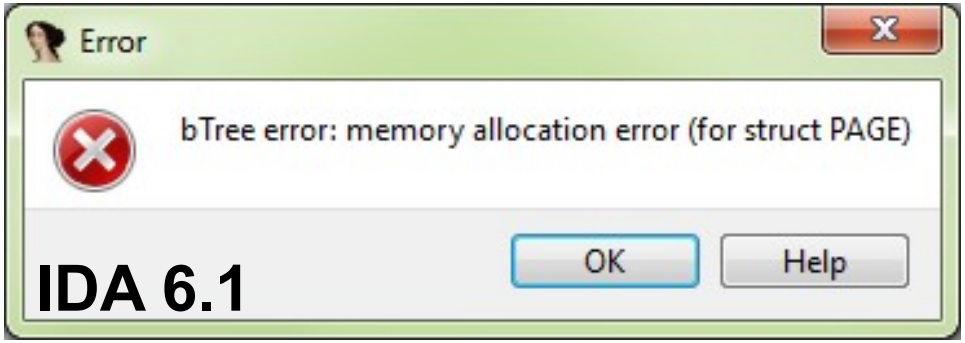
0. CPU are electronic, thus perfect
1. tricked by a malware
2. back to the basics
3. documented on Corkami
4. this presentation

# “Achievement unlocked”

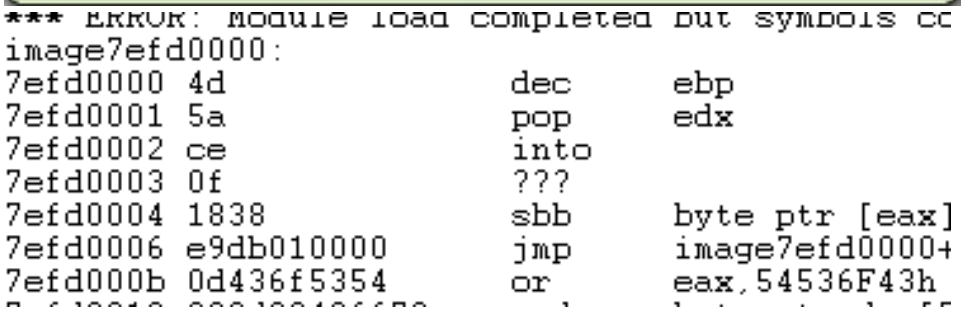

**Odbg 2.1a4**




**Hiew 8.15**




**IDA 6.1**




**WinDbg 6.12.0002.633**

(Authors notified, and most bugs already fixed)

# Agenda

I. why does it matter ?

(an easy introduction, for everybody)

II. a bunch of tricks

(technical stuff starts now, for technical people)

III. CoST

IV. a bit more of PE

AWESOME ! LEET !

2011 EDITION

# X86 ASSEMBLY FOR NEWBIES

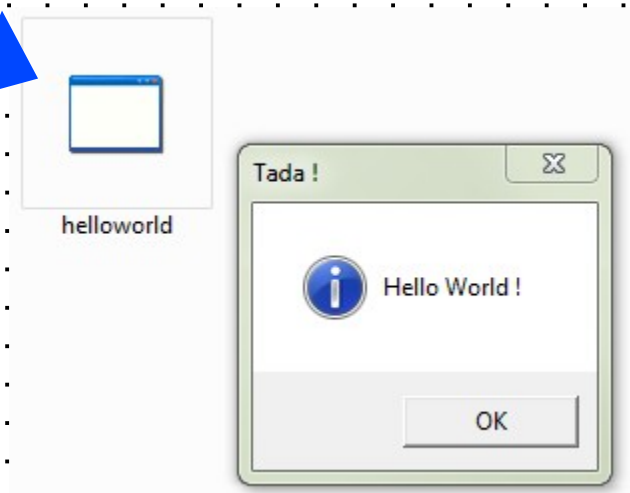
CHANGED MY LIFE FOR EVER !

IMPRESS GIRLS WITH LOW-LEVEL STUFF THAT NO ONE UNDERSTANDS...

www.txt2pic.com

# from C to binary

```
#include "stdafx.h"
#include "helloworld.h"

int APIENTRY _tWinMain(HINSTANCE hInstance,
                       HINSTANCE hPrevInstance,
                       LPTSTR    lpCmdLine,
                       int       nCmdShow)
{
    MessageBoxA(0, "Hello World !", "Tada !", MB_ICONINFORMATION);
    ExitProcess(0);
}
```



# inside the binary

```
#include "stdafx.h"
#include "helloworld.h"

int APIENTRY _tWinMain(HINSTANCE hInstance,
                       HINSTANCE hPrevInstance,
                       LPTSTR    lpCmdLine,
                       int       nCmdShow)
{
    MessageBoxA(0, "Hello World !", "Tada !", MB_ICONINFORMATION);
00121000 6A 40                push        40h
00121002 68 F4 20 12 00       push        offset string "Tada !" (1220F4h)
00121007 68 FC 20 12 00       push        offset string "Hello World !" (1220FCh)
0012100C 6A 00                push        0
0012100E FF 15 AC 20 12 00    call        dword ptr [__imp__MessageBoxA@16 (1220ACh)]
    ExitProcess(0);
00121014 6A 00                push        0
00121016 FF 15 00 20 12 00    call        dword ptr [__imp__ExitProcess@4 (122000h)]
```

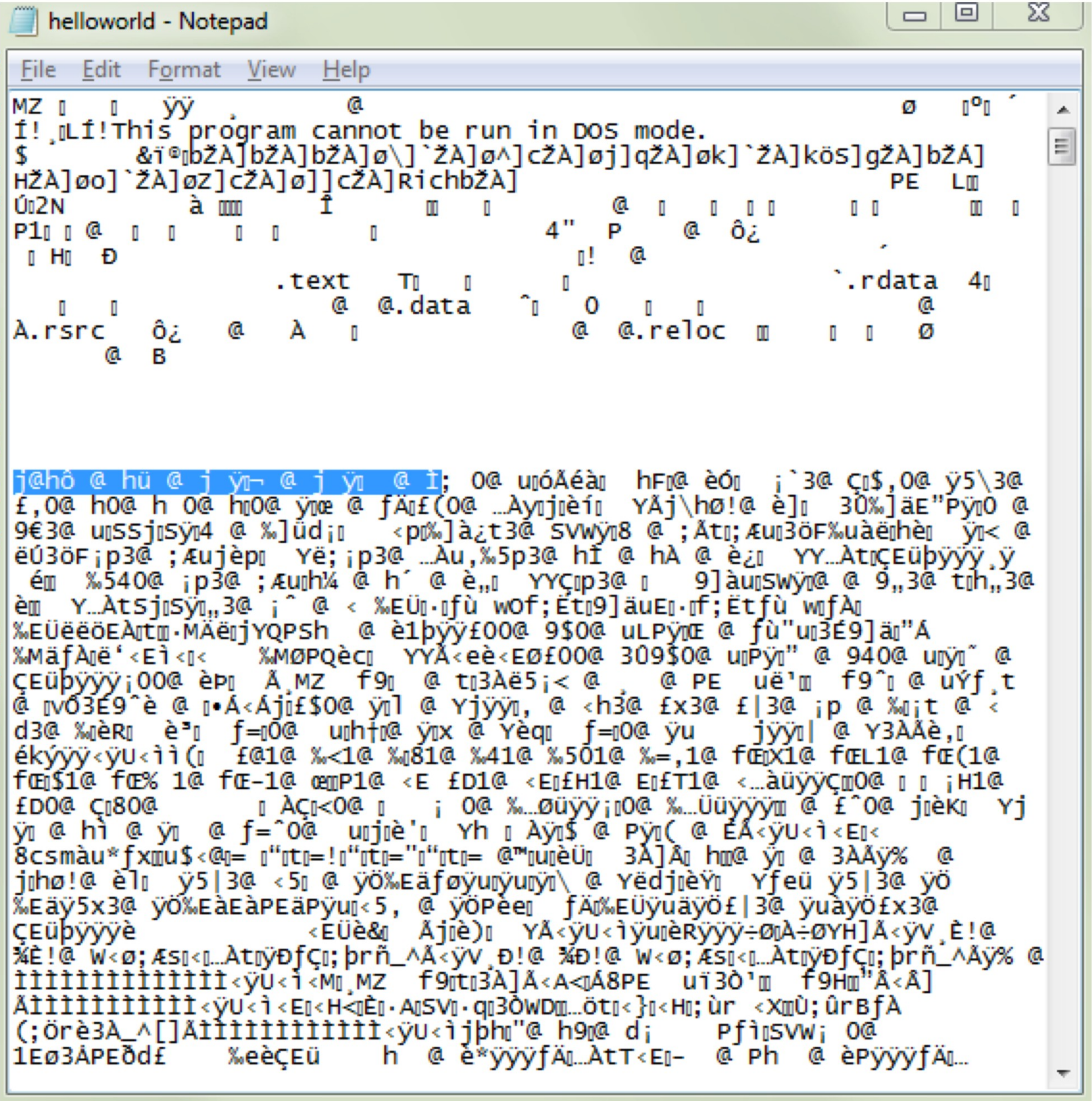
# our code, 'translated'

```
#include "stdafx.h"
#include "helloworld.h"

int APIENTRY _tWinMain(HINSTANCE hInstance,
                       HINSTANCE hPrevInstance,
                       LPTSTR    lpCmdLine,
                       int       nCmdShow)
{
    MessageBoxA(0, "Hello World !", "Tada !", MB_ICONINFORMATION);
00121000 6A 40                  push        40h
00121002 68 F4 20 12 00         push        offset string "Tada !" (1220F4h)
00121007 68 FC 20 12 00         push        offset string "Hello World !" (1220FCh)
0012100C 6A 00                  push        0
0012100E FF 15 AC 20 12 00      call        dword ptr [__imp__MessageBoxA@16 (1220ACh)]
    ExitProcess(0);
00121014 6A 00                  push        0
00121016 FF 15 00 20 12 00      call        dword ptr [__imp__ExitProcess@4 (122000h)]
```

# opcodes <=> assembly

```
#include "stdafx.h"
#include "helloworld.h"

int APIENTRY _tWinMain(HINSTANCE hInstance,
                       HINSTANCE hPrevInstance,
                       LPTSTR    lpCmdLine,
                       int       nCmdShow)
{
    MessageBoxA(0, "Hello World !", "Tada !", MB_ICONINFORMATION);
00121000 6A 40                push        40h
00121002 68 F4 20 12 00       push        offset string "Tada !" (1220F4h)
00121007 68 FC 20 12 00       push        offset string "Hello World !" (1220FCh)
0012100C 6A 00                push        0
0012100E FF 15 AC 20 12 00    call        dword ptr [__imp__MessageBoxA@16 (1220ACh)]
    ExitProcess(0);
00121014 6A 00                push        0
00121016 FF 15 00 20 12 00    call        dword ptr [__imp__ExitProcess@4 (122000h)]
```

```
#include "helloworld.h"

int APIENTRY _tWinMain(HINSTANCE
                       HINSTANCE h
                       LPTSTR    l
                       int       n
{
    MessageBoxA(0, "Hello World
00121000 6A 40                pu
00121002 68 F4 20 12 00       pu
00121007 68 FC 20 12 00       pu
0012100C 6A 00                pu
0012100E FF 15 AC 20 12 00    ca
    ExitProcess(0);
00121014 6A 00                pu
00121016 FF 15 00 20 12 00    ca
```



```
MZ ▯    ▯    ÿÿ    .          @                              ø    ▯º▯
í!¸▯Lí!This program cannot be run in DOS mode.
$        &ï®▯bŽÀ]bŽÀ]bŽÀ]ø\]`ŽÀ]ø^]cŽÀ]øj]qŽÀ]øk]`ŽÀ]kös]gŽÀ]bŽÀ]
HŽÀ]øo]`ŽÀ]øZ]cŽÀ]ø]]cŽÀ]RichbŽÀ]                               PE   L▯
Ú▯2N         à ▯▯       Î        ▯     ▯           @   ▯    ▯  ▯ ▯      ▯ ▯       ▯   ▯
P1▯ ▯ @   ▯   ▯     ▯   ▯         ▯              4"   P      @   ô¿
 ▯ H▯   Ð                                          ▯!   @
                     .text    T▯     ▯          ▯                        `.rdata  4▯
    ▯    ▯                @   @.data    ˆ▯    0    ▯    ▯                        @
À.rsrc    ô¿     @     À     ▯                   @   @.reloc  ▯     ▯   ▯    Ø
       @    B
```

# Assembly

- generated by the compiler
- executed directly by the CPU
- the only code information in a standard binary
  - what 'we' (analysts, hackers...) read

- disassembly is only for humans
  - no text code in the final binary

let's mess a bit now...

# let's insert 'something'

```
{
    __asm {__emit 0xd6}
    MessageBoxA(0, "Hello World !", "Tada !", MB_ICONINFORMATION);
    ExitProcess(0);
}
```

```
    __asm {__emit 0xd6}
00051000 ??                    db          d6h
    MessageBoxA(0, "Hello World !", "Tada !", MB_ICONINFORMATION);
00051001 6A 40                 push        40h
00051003 68 F4 20 05 00        push        offset string "Tada !" (
00051008 68 FC 20 05 00        push        offset string "Hello Wor
0005100D 6A 00                 push        0
0005100F FF 15 AC 20 05 00     call        dword ptr [__imp__Messag
```

## Table A-2. One-byte Opcode Map: (00H — F7H) *

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | ADD | | | | | | PUSH ES$^{i64}$ | POP ES$^{i64}$ |
| | Eb, Gb | Ev, Gv | Gb, Eb | Gv, Ev | AL, Ib | rAX, Iz | | |
| 1 | ADC | | | | | | PUSH SS$^{i64}$ | POP SS$^{i64}$ |
| | Eb, Gb | Ev, Gv | Gb, Eb | Gv, Ev | AL, Ib | rAX, Iz | | |
| 2 | AND | | | | | | SEG=ES (Prefix) | DAA$^{i64}$ |
| | Eb, Gb | Ev, Gv | Gb, Eb | Gv, Ev | AL, Ib | rAX, Iz | | |
| 3 | XOR | | | | | | SEG=SS (Prefix) | AAA$^{i64}$ |
| | Eb, Gb | Ev, Gv | Gb, Eb | Gv, Ev | AL, Ib | rAX, Iz | | |
| 4 | INC$^{i64}$ general register / REX$^{o64}$ Prefixes | | | | | | | |
| | eAX REX | eCX REX.B | eDX REX.X | eBX REX.XB | eSP REX.R | eBP REX.RB | eSI REX.RX | eDI REX.RXB |
| 5 | PUSH$^{d64}$ general register | | | | | | | |
| | rAX/r8 | rCX/r9 | rDX/r10 | rBX/r11 | rSP/r12 | rBP/r13 | rSI/r14 | rDI/r15 |
| 6 | PUSHA$^{i64}$/ PUSHAD$^{i64}$ | POPA$^{i64}$/ POPAD$^{i64}$ | BOUND$^{i64}$ Gv, Ma | ARPL$^{i64}$ Ew, Gw MOVSXD$^{o64}$ Gv, Ev | SEG=FS (Prefix) | SEG=GS (Prefix) | Operand Size (Prefix) | Address Size (Prefix) |
| 7 | Jcc$^{f64}$, Jb - Short-displacement jump on condition | | | | | | | |
| | O | NO | B/NAE/C | NB/AE/NC | Z/E | NZ/NE | BE/NA | NBE/A |
| 8 | Immediate Grp 1$^{1A}$ | | | | TEST | | XCHG | |
| | Eb, Ib | Ev, Iz | Eb, Ib$^{i64}$ | Ev, Ib | Eb, Gb | Ev, Gv | Eb, Gb | Ev, Gv |
| 9 | NOP PAUSE(F3) XCHG r8, rAX | XCHG word, double-word or quad-word register with rAX | | | | | | |
| | | rCX/r9 | rDX/r10 | rBX/r11 | rSP/r12 | rBP/r13 | rSI/r14 | rDI/r15 |
| A | MOV | | | | MOVS/B Xb, Yb | MOVS/W/D/Q Xv, Yv | CMPS/B Xb, Yb | CMPS/W/D Xv, Yv |
| | AL, Ob | rAX, Ov | Ob, AL | Ov, rAX | | | | |
| B | MOV immediate byte into byte register | | | | | | | |
| | AL/R8L, Ib | CL/R9L, Ib | DL/R10L, Ib | BL/R11L, Ib | AH/R12L, Ib | CH/R13L, Ib | DH/R14L, Ib | BH/R15L, Ib |
| C | Shift Grp 2$^{1A}$ | | RETN$^{f64}$ Iw | RETN$^{f64}$ | LES$^{i64}$ Gz, Mp | LDS$^{i64}$ Gz, Mp | Grp 11$^{1A}$ - MOV | |
| | Eb, Ib | Ev, Ib | | | | | Eb, Ib | Ev, Iz |
| D | Shift Grp 2$^{1A}$ | | | | AAM$^{i64}$ Ib | AAD$^{i64}$ Ib | | XLAT/ XLATB |
| | Eb, 1 | Ev, 1 | Eb, CL | Ev, CL | | | | |
| E | LOOPNE$^{f64}$/ LOOPNZ$^{f64}$ Jb | LOOPE$^{f64}$/ LOOPZ$^{f64}$ Jb | LOOP$^{f64}$ Jb | JrCXZ$^{f64}$/ Jb | IN | | OUT | |
| | | | | | AL, Ib | eAX, Ib | Ib, AL | Ib, eAX |
| F | LOCK (Prefix) | | REPNE (Prefix) | REP/REPE (Prefix) | HLT | CMC | Unary Grp 3$^{1A}$ | |
| | | | | | | | Eb | Ev |

# What did we do?

- Inserting an unrecognized byte
  - directly in the binary
  - not even documented nor identified !!

it could only crash...

# the CPU doesn't care

```
__asm {__emit 0xd6}
MessageBoxA(0, "Hello World !", "Tada !", MB_ICONINFORMATION);
ExitProcess(0);
```

# what happened ?

- D6 = S[ET]ALC
  - Set AL on Carry
  - AL = CF ? -1 : 0
- trivial, but not documented
  - unreliable or shameful ?

# Intel: 'do what I do...'

| | Intel's XED | MS' WinDbg |
|---|---|---|
| F1 | int1 | ?? |
| D6 | salc | ?? |
| F7C890909090 | test eax, 0x90909090 | ?? |
| 0F1E84C090909090 | nop dword ptr [eax+eax*8-0x6f6f6f70], eax | ?? |
| 0F2090 | mov eax, cr2 | ?? |
| 660FC8 | bswap ax | bswap eax |

# the problem

- the CPU does its stuff
- if we/our tools don't know what's next, we're blind.

- no exhaustive or clean test set
  - deep into malwares or packers
  - scattered

let's start the real stuff...

# a multi-generation CPU: standard...

| English | Assembly |
| --- | --- |
| let's go! | *push* |
| you win | *mov* |
| sandwich | *call* |
| hello | *retn* |
| f*ck | *jmp* |

# ...old-style...

| | |
|---|---|
| thou | *aaa* |
| porpentine | *xlat* |
| enmity | *verr* |
| hither | *smsw* |
| unkennel | *lsl* |

```
CE          INTO
6202        BOUND EAX,QWORD PTR DS:[EDX]
0F00E1      VERR CX
0F02C1      LAR EAX,ECX
0F00CA      STR DX
37          AAA
0F03C1      LSL EAX,ECX
0FAEF8      SFENCE
63C1        ARPL CX,AX
D40A        AAM
0FC9        BSWAP ECX
F0:0FC70E   LOCK CMPXCHG8B QWORD PTR DS:[ESI]
C51E        LDS EBX,FWORD PTR DS:[ESI]
D7          XLAT BYTE PTR DS:[EBX+AL]
27          DAA
0FC1C1      XADD ECX,EAX
0F0D00      PREFETCH QWORD PTR DS:[EAX]
```

# ...newest generation

| | |
|---|---|
| tweet | *crc32* |
| poke | *aesenc* |
| google | *pcmpistrm* |
| pwn | *vfmsubadd132ps* |
| apps | *rcpss* |

and *MOVBE*, the rejected offspring

# registers

- Initial values (Windows)
  - eax = <your OS generation>

    version = (eax != 0) ? Vista_or_later : XP
  - gs = <number of bits>

    bits = (gs == 0) ? 32 : 64

- Complex relations
  - FPU changes FST, STx, Mmx (ST0 overlaps MM7)
    - changes CR0, under XP

# *smsw*

- CR0 access, from user-mode
  - 286 opcode
- higher word of reg32 'undefined'
- under XP
  - influenced by FPU
  - eventually reverts

```
 smsw        eax
 cmp         ax,03B ;';'
 jnz         bad --↓1
 fnop
 smsw        eax
 cmp         ax,031 ;'1'
 jnz         bad --↓1
2smsw        eax
 cmp         ax,031 ;'1'
 jz          wait_loop --↑2
```

# GS

- reset on thread switch (Windows 32b)
- eventually reset
  - debugger stepping
  - wait
  - timings

```
 mov     ax,3
 mov     gs,eax
1mov     ax,gs
 cmp     ax,3
 jz      gs loop --↑1
```

# *nop*

- *nop* is *xchg *ax, *ax*
  - but *xchg *ax, *ax* can **do** something, in 64b !

    87 c0: xchg eax, eax

    `.. .. .. .. 01 23 45 67 => 00 00 00 00 01 23 45 67`
- *hint nop* `0F1E84C090909090 nop dword ptr [eax+eax*8-0x6f6f6f70], eax`
  - partially undocumented, actually 0f 18-1f
  - can trigger exception

# *mov*

- documented, but sometimes tricky
  - *mov [cr0], eax* *mov cr0, eax*
    - mod/RM is ignored
  - *movsxd eax, ecx* *mov eax, ecx*
    - no REX prefix
  - *mov eax, cs* *movzx eax,cs*
    - 'undefined' upper word

# *bswap*

rax

```
12 34 56 78 90 ab cd ef => ef cd ab 90 78 56 34 12
```

eax

```
.. .. .. .. 01 23 45 67 => 00 00 00 00 67 45 23 01
```

ax

```
.. .. .. .. .. .. 01 23 => .. .. .. .. .. .. 00 00
```

# *push+ret*

```
.00401000:   push          next ;'j@h4' --↓1
.00401005:   retn ; _^_^_^_^_^_^_^_^_^_^_^_^_
.00401007:   int           3
next:       1push          040 ;'@'
.0040100A:   push          0000401034 ;'Tada!' -
.0040100F:   push          000040103A ;'Hello Wo
.00401014:   push          0
.00401016:   call          MessageBoxA --↓4
```

```
<ModuleEn  $   68 08  PUSH <ret.next>
00401005   .   66:C3  RETN                    RET used as a jump to next
00401007       CC     INT3
<next>     >   6A 40  PUSH 40                 Style = MB_OK|MB_ICONASTERISK|MB_APPLMOD
0040100A   .   68 34  PUSH ret.00401034       Title = "Tada!"
0040100F   .   68 3A  PUSH ret.0040103A       Text = "Hello
00401014   .   6A 00  PUSH 0                  hOwner = NULL
00401016   .   E8 09  CALL ret.00401024       MessageBoxA
0040101B   .   6A 00  PUSH 0                  ExitCode = 0
0040101D   .   E8 08  CALL ret.0040102A       ExitProcess
00401022       CC     INT3
00401023       CC     INT3
00401024   $-  FF25   JMP DWORD PTR DS:[4     user32.Message
0040102A   .-  FF25   JMP DWORD PTR DS:[4     kernel32.ExitP
00401030       CC     INT3
00401031       CC     INT3
00401032       CC     INT3
00401033       CC     INT3
00401034   .   54 61  ASCII "Tada!",0
0040103A   .   48 65  ASCII "Hello World!"
```

# ...and so on...

- much more @ http://x86.corkami.com
  - also graphs, cheat sheet...

- too much theory for now...

Corkami Standard Test

# CoST

- http://cost.corkami.com
- testing opcodes
- in a hardened PE
  - available in easy mode

```
C>CoST.exe
CoST - Corkami Standard Test BETA 2011/09/XX
Ange Albertini, BSD Licence, 2009-2011 - http://corkami.com

Info: Windows 7 found
Starting: jumps opcodes...
Starting: classic opcodes...
Starting: rare opcodes...
Starting: undocumented opcodes...
Starting: cpu-specific opcodes...
Info: CPUID GenuineIntel
Info[cpu]: MOVBE (Atom only) not supported
Starting: undocumented encodings...
Starting: os-dependant opcodes...
Starting: 'nop' opcodes...
Starting: opcode-based anti-debuggers...
Starting: opcode-based GetIPs...
Starting: opcode-based exception triggers...
Starting: 64 bits opcodes...
Starting: registers tests

...completed!
```

# more than 150 tests

- classic, rare
- jumps (JMP to IP, IRET, …)
- undocumented (IceBP, SetALc...)
- cpu-specific (MOVBE, POPCNT,...)
- os-dependant, anti-VM/debugs
- exceptions triggers, interrupts, OS bugs,...
- ...

```
mov         eax,3
cmp         eax,3
jz          .07EFD0593
```

# a documented binary

exports + VEH = self commented assembly

```
    CoST.exe                ↓FRO --------              a32 PE .7EFD0220|Hiew 8.15
4_Main:       mov          d,[0CAFEBABE],07EFD2CF7 ;'Starting: jumps opco
.7EFD022A:    call         jumps --↓2
.7EFD022F:    nop
.7EFD0230:    mov          d,[0CAFEBABE],07EFD2D14 ;'Starting: classic op
.7EFD023A:    call         classics --↓4
```

a lot of DbgOutput

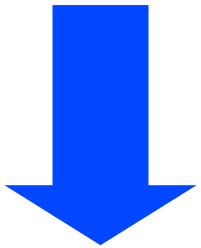
```
1   [trick] Adding TLS 2 in TLS callbacks list
2   [trick] the next call's operand is zeroed by the loader
3   CoST - Corkami Standard Test BETA 2011/09/XX
4   Ange Albertini, BSD Licence, 2009-2011 - http://corkami.com
5
6
7   [trick] TLS terminating by unhandled exception (EP is executed)
8   [trick] allocating buffer [0000-ffff]
9   testing: NULL buffer
10  checking OS version
11  Info: Windows 7 found
12  [trick] calling Main via my own export
13  Starting: jumps opcodes...
14  Testing: RETN word
```

# 32+64 = ...

```
.7EFD2540:   mov                eax,0F570D67C
.7EFD2545:   mov                ebx,3
.7EFD254A:   push               cs
.7EFD254B:   push               end --↓1
.7EFD2550:   push               033 ;'3'
.7EFD2552:   call               push_eip --↓2
push_eip:   2arpl               ax,bx
.7EFD2559:   dec                eax
.7EFD255A:   add                eax,eax
.7EFD255C:   retf ;  _^_^_^_^_^_^_^_^_^_
end:        1cmp                ebx,0EAE1ACFC
.7EFD2563:   jz                 next --↓3
.7EFD2565:   call               bad --↓4
next:       3cmp                eax,0D5C359F8
```

# same opcodes, different code

```
push_eip:    63D8     arpl        ax,bx
.7EFD2559:   48       dec         eax
.7EFD255A:   01C0     add         eax,eax
.7EFD255C:   CB       retf ; _^_^_^_^_^_^.
```



```
push_eip:    63D8     movsxd      rbx,eax
.7EFD2559:   4801C0   add         rax,rax
.7EFD255C:   CB       retf ; _^_^_^_^_^_^.
```

# CoST vs WinDbg & Hiew

WinDbg 6.12.0002.633

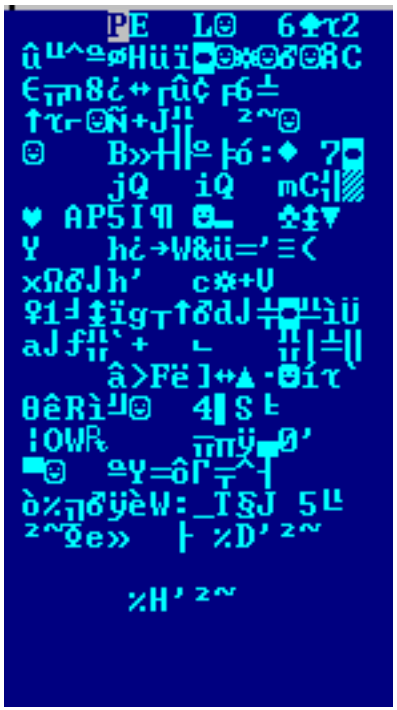
```
*** ERROR: Module load completed but symbols co
image7efd0000:
7efd0000 4d              dec     ebp
7efd0001 5a              pop     edx
7efd0002 ce              into
7efd0003 0f              ???
7efd0004 1838            sbb     byte ptr [eax]
7efd0006 e9db010000      jmp     image7efd0000+
7efd000b 0d436f5354      or      eax,54536F43h
```

Hiew 8.15

```
C:\Users\Ange\CoST.exe
.7EFD0000: 4D      dec        ebp
.7EFD0001: 5A      pop        edx
.7EFD0002: CE      into
the_dragon:        #UD
.7EFD0006: E91501  jmp        3_Entr
```

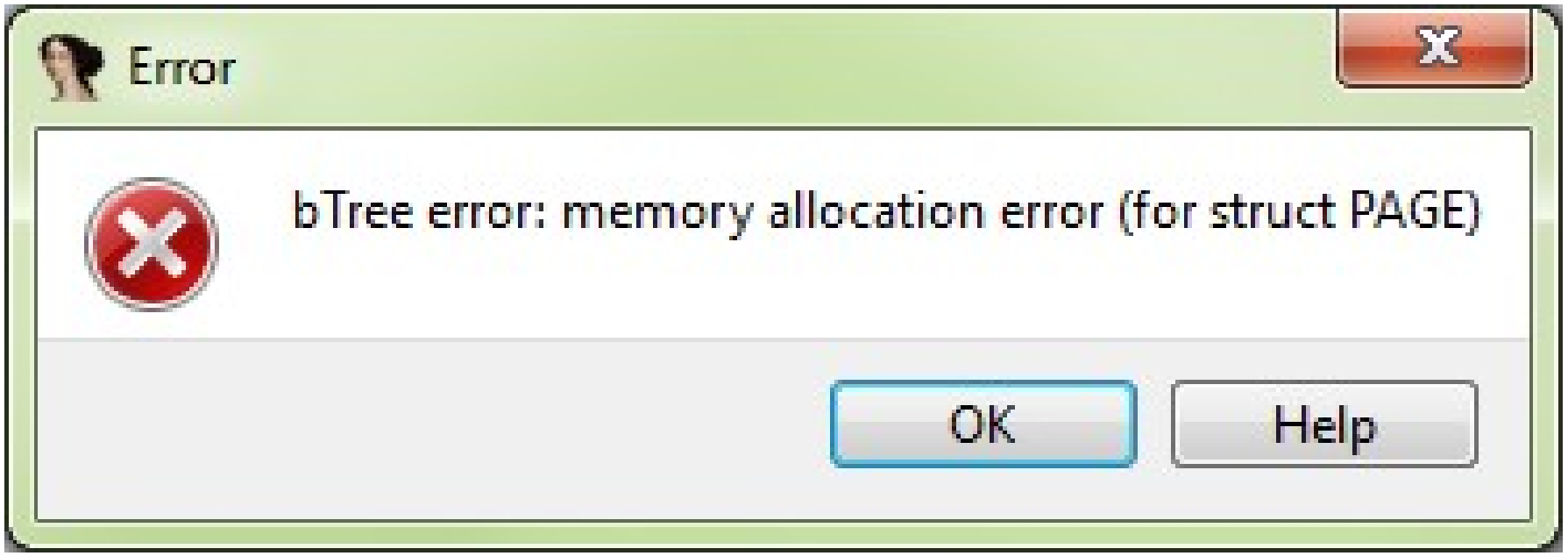
# a hardened PE


Top



PE 'footer'

# CoST vs IDA


Error
bTree error: memory allocation error (for struct PAGE)
OK
Help

# CoST vs Dumpbin

HIDDEN SLIDE

Microsoft (R) COFF/PE Dumper Version 10.00.30319.01



Dump of file CoST.exe

File Type: EXECUTABLE IMAGE

LINK : fatal error LNK1248: image size (9B097F81) exceeds maximum allowable size (80000000)

a bit more of PE...

# PE on corkami

- some graphs
- a wiki page
  - http://pe.corkami.com
  - not “finished”
  - more than 100 PoCs
  - good enough to break <you name it>

# virtual section table vs Hiew

```
VIRTSE~1.EXE          ↓FR0 --------                    00000000|Hiew 8.15 (c)SEN
00000000:  4D 5A 00 00-00 00 00 00-00 00 00 00-00 00 00 00   MZ
00000010:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
00000020:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
00000030:  00 00 00 00-00 00 00 00-00 00 00 00-40 00 00 00                @
00000040:  50 45 00 00-4C 01 52 00-00 00 00 00-00 00 00 00   PE  L☺R
00000050:  00 00 00 00-58 02 02 01-0B 01 00 00-00 00 00 00       X☻☻☺♂☺
00000060:  00 00 00 00-00 00 00 00-38 01 00 00-00 00 00 00           8☺
00000070:  00 00                                              @ ♦    ♦
00000080:  00 00                                                 ♦
00000090:  00 00                                                 ♥
000000A0:  00 00
000000B0:  00 00
000000C0:  90 01
000000D0:  00 00
000000E0:  00 00
000000F0:  00 00
00000100:  00 00
00000110:  00 00                                                 ▸☻  ▸
00000120:  00 00
00000130:  00 00                                                hP☺@  §↑
00000140:  02 40                                              —♦j  §▸☻@ ╠╠
00000150:  20 2A                                              ow alignment
00000160:  20 50                                              with a virtu
00000170:  61 6C                                              ection table
00000180:  20 28                                              )o
00000190:  D0 01                                                     0☻
000001A0:  10 02 00 00-D8 01 00 00-00 00 00 00-00 00 00 00   ▸☻  ╪☺
000001B0:  3D 02 00 00-18 02 00 00-00 00 00 00-00 00 00 00   =☻  ↑☻
000001C0:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
000001D0:  F0 01 00 00-00 00 00 00-FE 01 00 00-00 00 00 00   ≡☺      ▪☺
000001E0:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
000001F0:  00 00 45 78-69 74 50 72-6F 63 65 73-73 00 00 00     ExitProcess
1      2      3      4      5      6      7      8      9      10
```

at 00000000

| | |
|---|---|
| Signature | 5A4D |
| Bytes on last page | 0000 |
| Pages in file | 0000 |
| Relocations count | 0000 |
| Paragraphs in header | 0000 |
| Minimum memory | 0000 |
| Maximum memory | 0000 |
| SS:SP setting | 0000:0000 |
| Checksum | 0000 |
| CS:IP setting | 0000:0000 |
| Relocations table address | 0000 |
| Overlay number | 0000 |
| Overlay length | 00000248 |
| NewExe offset | 00000040 |
| Entry point | 00000000 |

# Folded header

| Name | RVA | Size |
|---|---|---|
| Export | 88660001 | 10009988 |
| Import | 86600010 | 01000998 |
| Resource | 66000100 | 00100099 |
| Exception | 6000100F | F0010009 |
| Security | 000100FF | FF001000 |
| Fixups | 00100FF0 | 0FF00100 |
| Debug | 0100FF05 | 20FF0010 |
| Description | 100FF055 | 220FF001 |
| MIPS GP | 100FF055 | 220FF001 |
| TLS | 0100FF05 | 20FF0010 |
| Load config | 00100FF0 | 0FF00100 |
| Bound Import | 000100FF | FF001000 |
| Import Table | 6000100F | F0010009 |
| Delay Import | 66000100 | 00100099 |
| COM Runtime | 86600010 | 01000998 |
| (reserved) | 88660001 | 10009988 |

# Weird export names

- exports = <anything non null>, 0

```
00401000: 6A01                                  push
00401002: 58                                    pop
00401000: 8BFF                                → retn
00401000: 8BFF                                → int
00401000: 8BFF                                → push
**************************************        → call
* Insert subliminal message here *            → add
**************************************        → retn ;
00401000: 8BFF                                → int
00401018: 202A                                 1and
```

# 65535 sections vs OllyDbg

# one last...

- TLS AddressOfIndex is overwritten on loading
- Import are parsed until Name is 0

- under XP, overwritten after imports
  - imports are fully parsed
- under W7, before
  - truncated

same PE, loaded differently

under different Windows

# conclusion

- x86 and PE are far from perfectly documented
- still some gray areas of PE or x86
  - but a bit less, every day

## official documentations lead to FAILURE

1. visit Corkami.com
2. download the PoCs
3. fix the bugs ;)

# Thanks

- Peter Ferrie
- Candid Wüest

Adam Błaszczyk, BeatriX, Bruce Dang, Cathal Mullaney, Czerno, Daniel Reynaud, Elias Bachaalany, Ero Carrera, Eugeny Suslikov, Georg Wicherski, Gil Dabah, Guillaume Delugré, Gunther, Igor Skochinsky, Ilfak Guilfanov, Ivanlef0u, Jean-Baptiste Bédrune, Jim Leonard, Jon Larimer, Joshua J. Drake, Markus Hinderhofer, Mateusz Jurczyk, Matthieu Bonetti, Moritz Kroll, Oleh Yuschuk, Renaud Tabary, Rewolf, Sebastian Biallas, StalkR, Yoann Guillot,...

# Questions ?

# Such a weird processor

## messing with opcodes

(...and a little bit of PE)

Ange Albertini

28th October 2011

@ange4771
@corkami (news only)


hash days

# Such a weird processor

## messing with opcodes

(...and a little bit of PE)

Ange Albertini

28th October 2011

@ange4771
@corkami (news only)


hash days

Welcome !

I will talk about opcodes weirdness,
and also a little bit of PE files...

HIDDEN SLIDE

## (if you read this without the presentation)

- introduce Corkami.com, a RCE site
- why correct disassembly is important for analysis,
  - why undocumented opcodes are a dead end
- a few examples of undocumented opcodes and CPU weirdness
- theory-only sucks, so I created CoST for practicing and testing.
- CoST also tests PE, but it's not enough by itself
- So I documented PE separately, and give some examples.

[version: release 1]

this extra slide to let you decide if you really want to read further ;)

Short summary =

1. I studied ASM and PE, from scratch
2. I failed all tools I tried: IDA, OllyDbg, Hiew, pefile, WinDbg, HT, CFF Explorer...
3. here is how...

## presented by...

- a reverse-engineering enthusiast
  - ...since dos 3.21
  - Corkami.com
  - Mame (the arcade emulator)
- a malware analyst

I've been a reverse-engineering enthusiast for some time.
I created a website called Corkami
in the past, I worked on decrypted arcade games, in Mame, the arcade emulator.

and professionally, I'm a malware analyst.

## Corka-what ?

- RCE project, only technical stuff
- free to:
  - browse, download
  - test, modify, compile
- updated
- useful daily

- but.... only a hobby !

I have a hobby research project, called Corkami.

only technical stuff
No ads, no login required, no annoying licenses, binaries available to test and learn immediately

it's now a wiki, so it's constantly updated...

Some cheat sheets and graphs to use daily at work...

I'm trying to make it good, but, it's only a hobby, so it's not as good as I'd like it to be, and that's why it doesn't look very professional!

## what is in Corkami ?

- wiki pages, cheat sheets
- many PoCs
  - hand-written (not generated), minimalists
  - binaries available
- on PDF, x86, PE...
- 100% open
  - BSD, CC BY
    - sources, images, docs

So what's there ?

some wiki pages, and some cheat sheets on particular topics, such as PDF, x86, PE.

I create by hand (with no external tool) minimalists proof of concepts to focus on specific points.

these PoCs are open-source, and compilable with free tools, under a permissive license

## Story

0.CPU are electronic, thus perfect

1.tricked by a malware

2.back to the basics

3.documented on Corkami

4.this presentation

the motivation behind this presentation is:

long ago, I was young and innocent, and believed that since CPU are made of electronic component, they should be perfectly logic.

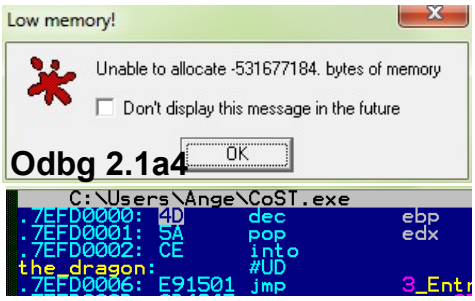
several years ago, I was tricked by a malware, using stuff undocumented at the time.

So I decided to go back to the basics, and study ASM and PE from scratch, and share my discoveries on the way, on Corkami.

and in the process, I failed all tools I tried in one way or another.

now, I'm presenting the result.

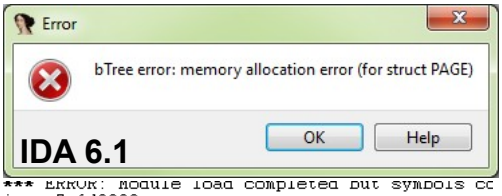
## “Achievement unlocked”



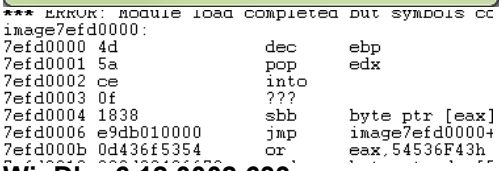

**Odbg 2.1a4**




**Hiew 8.15**




**IDA 6.1**




**WinDbg 6.12.0002.633**

(Authors notified, and most bugs already fixed)

but, if I was just a guy learning ASM, I probably wouldn't be presenting here at Hashdays.

So, here is why I'm here :)

## Agenda

I. why does it matter ?

(an easy introduction, for everybody)

II. a bunch of tricks

(technical stuff starts now, for technical people)

III. CoST

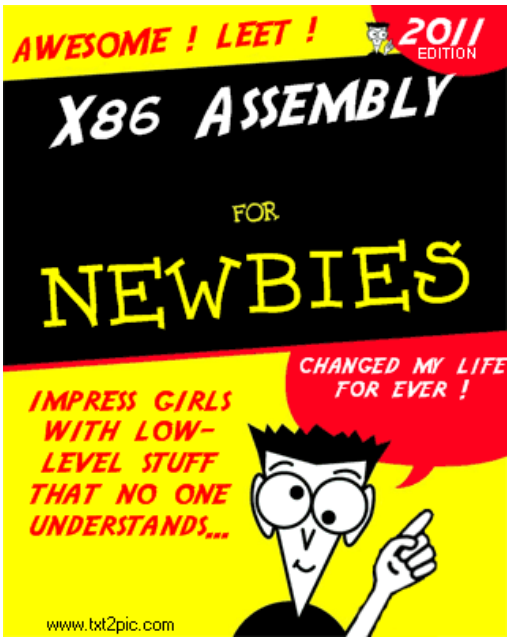
IV. a bit more of PE

so, first, I'll start slowly, trying to introduce assembly to beginners, and make them understand the problem of undocumented opcodes.

then it will get more technical:
I'll cover a few assembly tricks, including some found in malware.

then I'll introduce my opcode tester, CoST.

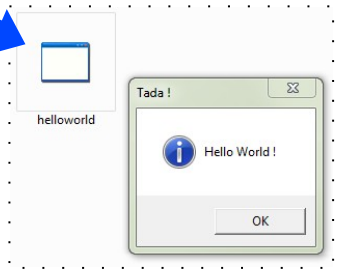
and I'll also present my last project which deals with the PE format.




So, let's start and try to make everybody understand the problem of undocumented opcodes.

# from C to binary

```
#include "stdafx.h"
#include "helloworld.h"

int APIENTRY _tWinMain(HINSTANCE hInstance,
                       HINSTANCE hPrevInstance,
                       LPTSTR    lpCmdLine,
                       int       nCmdShow)
{
    MessageBoxA(0, "Hello World !", "Tada !", MB_ICONINFORMATION);
    ExitProcess(0);
}
```




so, we create a simple program in a language, such as C.

Here, in Visual Studio, Microsoft standard development environment.

this program shows a simple message box on screen, then terminates.

an executable is generated, and indeed does what we expected.

## inside the binary

```
#include "stdafx.h"
#include "helloworld.h"

int APIENTRY _tWinMain(HINSTANCE hInstance,
                       HINSTANCE hPrevInstance,
                       LPTSTR    lpCmdLine,
                       int       nCmdShow)
{
    MessageBoxA(0, "Hello World !", "Tada !", MB_ICONINFORMATION);
00121000 6A 40                push        40h
00121002 68 F4 20 12 00       push        offset string "Tada !" (1220F4h)
00121007 68 FC 20 12 00       push        offset string "Hello World !" (1220FCh)
0012100C 6A 00                push        0
0012100E FF 15 AC 20 12 00    call        dword ptr [__imp__MessageBoxA@16 (1220ACh)]
    ExitProcess(0);
00121014 6A 00                push        0
00121016 FF 15 00 20 12 00    call        dword ptr [__imp__ExitProcess@4 (122000h)]
```

what the Visual Studio compiler did from our C code is actually generate sequences of assembly code instruction that will generate the wanted actions.

## our code, 'translated'

```
#include "stdafx.h"
#include "helloworld.h"

int APIENTRY _tWinMain(HINSTANCE hInstance,
                       HINSTANCE hPrevInstance,
                       LPTSTR    lpCmdLine,
                       int       nCmdShow)
{
    MessageBoxA(0, "Hello World !", "Tada !", MB_ICONINFORMATION);
00121000 6A 40                  push        40h
00121002 68 F4 20 12 00         push        offset string "Tada !" (1220F4h)
00121007 68 FC 20 12 00         push        offset string "Hello World !" (1220FCh)
0012100C 6A 00                  push        0
0012100E FF 15 AC 20 12 00      call        dword ptr [__imp_MessageBoxA@16 (1220ACh)]
    ExitProcess(0);
00121014 6A 00                  push        0
00121016 FF 15 00 20 12 00      call        dword ptr [__imp_ExitProcess@4 (122000h)]
```

Here, you can see calls to MessageBox, then ExitProcess (the names are self-explaining), with the parameters above.

these assembly operations are stored in opcodes directly in the binary, as visible on the left.

## opcodes <=> assembly

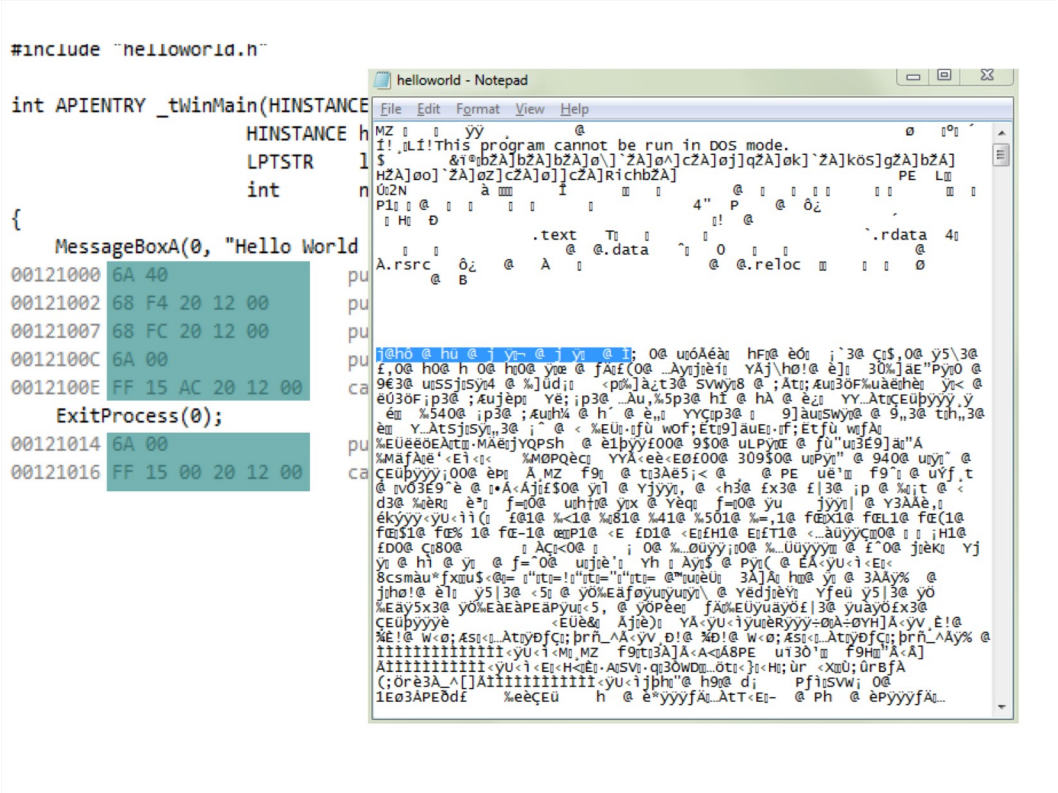
```
#include "stdafx.h"
#include "helloworld.h"

int APIENTRY _tWinMain(HINSTANCE hInstance,
                       HINSTANCE hPrevInstance,
                       LPTSTR    lpCmdLine,
                       int       nCmdShow)
{
    MessageBoxA(0, "Hello World !", "Tada !", MB_ICONINFORMATION);
00121000 6A 40                 push        40h
00121002 68 F4 20 12 00        push        offset string "Tada !" (1220F4h)
00121007 68 FC 20 12 00        push        offset string "Hello World !" (1220FCh)
0012100C 6A 00                 push        0
0012100E FF 15 AC 20 12 00     call        dword ptr [__imp__MessageBoxA@16 (1220ACh)]
    ExitProcess(0);
00121014 6A 00                 push        0
00121016 FF 15 00 20 12 00     call        dword ptr [__imp__ExitProcess@4 (122000h)]
```

now you know that this is what is in the file itself.
this is how it's read by 'us' (reverse engineers, malware analysts, exploit developers...).

the CPU itself only reads the hex.

as you can see, there is a relation:
68 - in hex - is used to push offsets
calls starts with FF 15...
and you can see the used addresses here (read them backward).

so, you see the first byte determine the actual opcode.
and depending on each opcode, the length is variable.




This is what is actually in the file on the hard disk (the 'hex').

If you'd accidentally open the file in, say notepad - it doesn't really make sense, but at least you have that on your machine - you could find it here (remember, it's hex).

Note that it's actually a very tiny part of the whole file (<30bytes out of 56000).

## Assembly

- generated by the compiler
- executed directly by the CPU
- the only code information in a standard binary
  - what 'we' (analysts, hackers...) read

- disassembly is only for humans
  - no text code in the final binary

so, the compiler translates our C to a series of assembly operations, which is itself encoded in opcodes.
the resulting executable only contains the opcodes, which are directly understood and executed by the CPU. If no error happens, what is here directly affects the behavior of the program, there is no 'man in the middle' from the OS.

so our C code will just eventually lead the CPU to read and execute
6A 40 68 F4 20 40 00 68 FC 20...

if, by any chance, there is some opcodes that we are not aware of, or doesn't do what we expect, the CPU doesn't care, it just knows what to do.

let's mess a bit now...

so now, let's interfere with the compiling process

## let's insert 'something'

```
{
    __asm {__emit 0xd6}
    MessageBoxA(0, "Hello World !", "Tada !", MB_ICONINFORMATION);
    ExitProcess(0);
}
```

```
    __asm {__emit 0xd6}
00051000 ??                 db          d6h
    MessageBoxA(0, "Hello World !", "Tada !", MB_ICONINFORMATION);
00051001 6A 40              push        40h
00051003 68 F4 20 05 00     push        offset string "Tada !" (
00051008 68 FC 20 05 00     push        offset string "Hello Wor
0005100D 6A 00              push        0
0005100F FF 15 AC 20 05 00  call        dword ptr [__imp__Messag
```

let's add a command that will force a specific byte in the opcode.

this result is not known to visual studio, which only shows ??

Table A-2. One-byte Opcode Map: (00H — F7H) *

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | ADD | | | | | | PUSH ES$^{i64}$ | POP ES$^{i64}$ |
| | Eb, Gb | Ev, Gv | Gb, Eb | Gv, Ev | AL, Ib | rAX, Iz | | |
| 1 | ADC | | | | | | PUSH SS$^{i64}$ | POP SS$^{i64}$ |
| | Eb, Gb | Ev, Gv | Gb, Eb | Gv, Ev | AL, Ib | rAX, Iz | | |
| 2 | AND | | | | | | SEG=ES (Prefix) | DAA$^{i64}$ |
| | Eb, Gb | Ev, Gv | Gb, Eb | Gv, Ev | AL, Ib | rAX, Iz | | |
| 3 | XOR | | | | | | SEG=SS (Prefix) | AAA$^{i64}$ |
| | Eb, Gb | Ev, Gv | Gb, Eb | Gv, Ev | AL, Ib | rAX, Iz | | |
| 4 | INC$^{i64}$ general register / REX$^{o64}$ Prefixes | | | | | | | |
| | eAX REX | eCX REX.B | eDX REX.X | eBX REX.XB | eSP REX.R | eBP REX.RB | eSI REX.RX | eDI REX.RXB |
| 5 | PUSH$^{d64}$ general register | | | | | | | |
| | rAX/r8 | rCX/r9 | rDX/r10 | rBX/r11 | rSP/r12 | rBP/r13 | rSI/r14 | rDI/r15 |
| 6 | PUSHA$^{i64}$/ PUSHAD$^{i64}$ | POPA$^{i64}$/ POPAD$^{i64}$ | BOUND$^{i64}$ Gv, Ma | ARPL$^{i64}$ Ew, Gw MOVSXD$^{o64}$ Gv, Ev | SEG=FS (Prefix) | SEG=GS (Prefix) | Operand Size (Prefix) | Address Size (Prefix) |
| 7 | Jcc$^{f64}$, Jb - Short-displacement jump on condition | | | | | | | |
| | O | NO | B/NAE/C | NB/AE/NC | Z/E | NZ/NE | BE/NA | NBE/A |
| 8 | Immediate Grp 1$^{1A}$ | | | | TEST | | XCHG | |
| | Eb, Ib | Ev, Iz | Eb, Ib$^{i64}$ | Ev, Ib | Eb, Gb | Ev, Gv | Eb, Gb | Ev, Gv |
| 9 | NOP PAUSE(F3) XCHG r8, rAX | XCHG word, double-word or quad-word register with rAX | | | | | | |
| | | rCX/r9 | rDX/r10 | rBX/r11 | rSP/r12 | rBP/r13 | rSI/r14 | rDI/r15 |
| A | MOV | | | | MOVS/B Xb, Yb | MOVS/W/D/Q Xv, Yv | CMPS/B Xb, Yb | CMPS/W/D Xv, Yv |
| | AL, Ob | rAX, Ov | Ob, AL | Ov, rAX | | | | |
| B | MOV immediate byte into byte register | | | | | | | |
| | AL/R8L, Ib | CL/R9L, Ib | DL/R10L, Ib | BL/R11L, Ib | AH/R12L, Ib | CH/R13L, Ib | DH/R14L, Ib | BH/R15L, Ib |
| C | Shift Grp 2$^{1A}$ | | RETN$^{f64}$ Iw | RETN$^{f64}$ | LES$^{i64}$ Gz, Mp | LDS$^{i64}$ Gz, Mp | Grp 11$^{1A}$ - MOV | |
| | Eb, Ib | Ev, Ib | | | | | Eb, Ib | Ev, Iz |
| D | Shift Grp 2$^{1A}$ | | | | AAM$^{i64}$ Ib | AAD$^{i64}$ Ib | | XLAT/ XLATB |
| | Eb, 1 | Ev, 1 | Eb, CL | Ev, CL | | | | |
| E | LOOPNE$^{f64}$/ LOOPNZ$^{f64}$ Jb | LOOPE$^{f64}$/ LOOPZ$^{f64}$ Jb | LOOP$^{f64}$ Jb | JrCXZ$^{f64}$/ Jb | IN | | OUT | |
| | | | | | AL, Ib | eAX, Ib | Ib, AL | Ib, eAX |
| F | LOCK (Prefix) | | REPNE (Prefix) | REP/REPE (Prefix) | HLT | CMC | Unary Grp 3$^{1A}$ | |
| | | | | | | | Eb | Ev |

indeed, if we check Intel official documentation, there is nothing to see here...

## What did we do?

- Inserting an unrecognized byte
  - directly in the binary
  - not even documented nor identified !!

it could only crash...

so, we forced something that is not recognized by the most expensive Microsoft compiler to execute, which is not even in Intel's books.

We should only expect a crash, right ?

## the CPU doesn't care

```
__asm {__emit 0xd6}
MessageBoxA(0, "Hello World !", "Tada !", MB_ICONINFORMATION);
ExitProcess(0);
```


but the CPU doesn't care about what YOU (or VS) know, and it just executes that mysterious D6 just fine (apparently)

it doesn't look like a big problem, but if like Microsoft, you base your judgment on Intel's documentation, you just don't know what happens next. No automated analysis, proactive detection, etc... and you need to understand that undocumented opcode.

You can't even skip it:
you don't know if it will jump, do nothing, trigger an exception...
and because of variable instruction length, you can't even tell what would be the next instruction, so you can't guess easily backward from the next instruction.

## what happened ?

- D6 = S[ET]ALC
  - Set AL on Carry
  - AL = CF ? -1 : 0
- trivial, but not documented
  - unreliable or shameful ?

so what did we do in reality ?

D6 will be decoded as SETALC, which is quite simple.

It doesn't interfere with the execution of this example (it could have, of course).

surprisingly, it's not documented by Intel, but it's documented by AMD.

anyone knows why ?
I'd be curious to know.

## Intel: 'do what I do...'

| | Intel's XED | MS' WinDbg |
|---|---|---|
| F1 | int1 | ?? |
| D6 | salc | ?? |
| F7C890909090 | test eax, 0x90909090 | ?? |
| 0F1E84C090909090 | nop dword ptr [eax+eax*8-0x6f6f6f70], eax | ?? |
| 0F2090 | mov eax, cr2 | ?? |
| 660FC8 | bswap ax | bswap eax |

the funny thing is, even though Intel docs are full of holes, Intel free tools are fully aware of what to expect...

Sadly, Microsoft WinDbg decided to follow the official docs, which makes it a very bad tool against malware, which commonly use undocumented tricks.

## the problem

- the CPU does its stuff
- if we/our tools don't know what's next, we're blind.

- no exhaustive or clean test set
  - deep into malwares or packers
  - scattered

So, you now know that the CPU knows things that the Intel documentations omits.

if we or our tools are not able to tell what the CPU will do, we're just blind.

let's start the real stuff...

Now, let's start the real stuff

but, before focusing on particular opcodes,
my first questions was:
what are actually all the supported opcodes ?

## a multi-generation CPU: standard...

| English | Assembly |
|---|---|
| let's go! | *push* |
| you win | *mov* |
| sandwich | *call* |
| hello | *retn* |
| f*ck | *jmp* |

that's the problem.
like English language, assembly uses mainly always the same 'standard' opcodes.

which means, what everybody is used to hear or read:

Here, 'standard language'. What all generations understand.

most people would understand...

## ...old-style...

| | |
|---|---|
| thou | *aaa* |
| porpentine | *xlat* |
| enmity | *verr* |
| hither | *smsw* |
| unkennel | *lsl* |

but Intel CPU are from the 70's and still backward compatible...

here is an example of Shakespeare English and old x86 mnemonics

unknown to most people.
yet still fully working on a modern CPU.

```
CE          INTO
6202        BOUND EAX,QWORD PTR DS:[EDX]
0F00E1      VERR CX
0F02C1      LAR EAX,ECX
0F00CA      STR DX
37          AAA
0F03C1      LSL EAX,ECX
0FAEF8      SFENCE
63C1        ARPL CX,AX
D40A        AAM
0FC9        BSWAP ECX
F0:0FC70E   LOCK CMPXCHG8B QWORD PTR DS:[ESI]
C51E        LDS EBX,FWORD PTR DS:[ESI]
D7          XLAT BYTE PTR DS:[EBX+AL]
27          DAA
0FC1C1      XADD ECX,EAX
0F0D00      PREFETCH QWORD PTR DS:[EAX]
```

example: a PoC on corkami

works on all CPU

comfortable to read ? how old are you ?!

## ...newest generation

| | |
|---|---|
| tweet | *crc32* |
| poke | *aesenc* |
| google | *pcmpistrm* |
| pwn | *vfmsubadd132ps* |
| apps | *rcpss* |

and *MOVBE*, the rejected offspring

new generation : english and opcodes.

probably unknown to most people

single opcodes for CRC, AES, string masking...

Fused Multiply-Alternating Subtract/Add of Packed Single-Precision Floating-Point Values

Scalar Single-Precision Floating-Point Reciprocal

MOVBE = rejected offspring
netbook only. absent from i7
=> so much for backward compatibility

## registers

- Initial values (Windows)
  - eax = <your OS generation>

    version = (eax != 0) ? Vista_or_later : XP
  - gs = <number of bits>

    bits = (gs == 0) ? 32 : 64

- Complex relations
  - FPU changes FST, STx, Mmx (ST0 overlaps MM7)
    - changes CR0, under XP

the basics of assembly are the registers...

before any operation, registers have the value assigned to themselves by the OS.
I collected these values
under windows, specific values it's not CPU specific, but the initial values of the register on process start-up, under windows, gives a few hint that are used by malwares.

eax can immediately tell if you're on an older OS or not.
While GS can tell you if the machine is 64b or not, even in a 32b process.

registers are overlapping.
unlike many documentations, ST0 <> MM7

## *smsw*

- CR0 access, from user-mode
  - 286 opcode
- higher word of reg32 'undefined'
- under XP
  - influenced by FPU
  - eventually reverts

```
 smsw        eax
 cmp         ax,03B ;';'
 jnz         bad --↓1
 fnop
 smsw        eax
 cmp         ax,031 ;'1'
 jnz         bad --↓1
2smsw        eax
 cmp         ax,031 ;'1'
 jz          wait_loop --↑2
```

smsw is an old 286-era mnemonic (before protected mode was 'complete'): it allows usermode access to cr0.
the higher word of a reg32 target is 'undefined', yet always modified (and same as cr0)

under XP, right after an FPU operation, the returned value is modified [bits 1 and 3, called MP (Monitor Coprocessor) and TS (Task switched)], but eventually reverted after some time.

too tricky ? redirection fails. any idea why ?

# GS

- reset on thread switch (Windows 32b)
- eventually reset
  - debugger stepping
  - wait
  - timings

```
 mov        ax,3
 mov        gs,eax
1mov        ax,gs
 cmp        ax,3
 jz         gsloop --↑1
```

the GS trick is similar.
on 32b of windows, GS is reset on thread switch.
on 64b windows, it's already used by the OS (value non null at start)

ie wait long enough, it's null, whatever the value before.

if you just step manually, instantly lost.
after some time, but not a too short time, it's reset

## *nop*

- *nop* is *xchg *ax, *ax*
  - but *xchg *ax, *ax* can **do** something, in 64b !

    87 c0: xchg eax, eax

  `.. .. .. .. 01 23 45 67 => 00 00 00 00 01 23 45 67`
- *hint nop* `0F1E84C090909090 nop dword ptr [eax+eax*8-0x6f6f6f70], eax`
  - partially undocumented, actually 0f 18-1f
  - can trigger exception

xchg eax, eax is 90, which originally did nothing.
(xchg eax, ecx is 91)
thus 90 became nop
but 87 c0 is an xchg eax, eax that is not a nop and does something in 64b, as it resets the upper dword.

hint nop gives hint of what to access next. it does nothing, but it's multi-byte.
first, it's not completely documented by intel
and, being a multi-byte opcode, if it overlaps an invalid page, it can trigger an exception!

## *mov*

- documented, but sometimes tricky
  - *mov [cr0], eax* *mov cr0, eax*
    - mod/RM is ignored
  - *movsxd eax, ecx* *mov eax, ecx*
    - no REX prefix
  - *mov eax, cs* *movzx eax,cs*
    - 'undefined' upper word

Mov is documented, but has a few quirks.
* to/from control and debug registers, memory operands are not allowed. but not rejected !
* in 64b, with no REX prefix, movsxd can actually work to and from a 32b register, which is against the logic of 'sign extending'
* on the contrary, mov from a selector actually affects a complete 32b register. the upper word is theoretically undefined, but actually 0 (used by malware to see if upper part is actually reset or if wrongly emulated as 'mov ax, cs'.)

## *bswap*

rax

```
12 34 56 78 90 ab cd ef => ef cd ab 90 78 56 34 12
```

eax

```
.. .. .. .. 01 23 45 67 => 00 00 00 00 67 45 23 01
```

ax

```
.. .. .. .. .. .. 01 23 => .. .. .. .. .. .. 00 00
```

Bswap... is like an administration... rules prevent it to work correctly most of the time...

it's supposed to swap the endianness of a register.

but most of the time, it does something unexpected.

with a 64b register, it swaps the quadword around. good.

with a 32b, it resets the highest dword. 'as usual', of course...

and on 16b, it's 'undefined' but it just clears the 16b register itself (the rest stays unchanged, of course)...

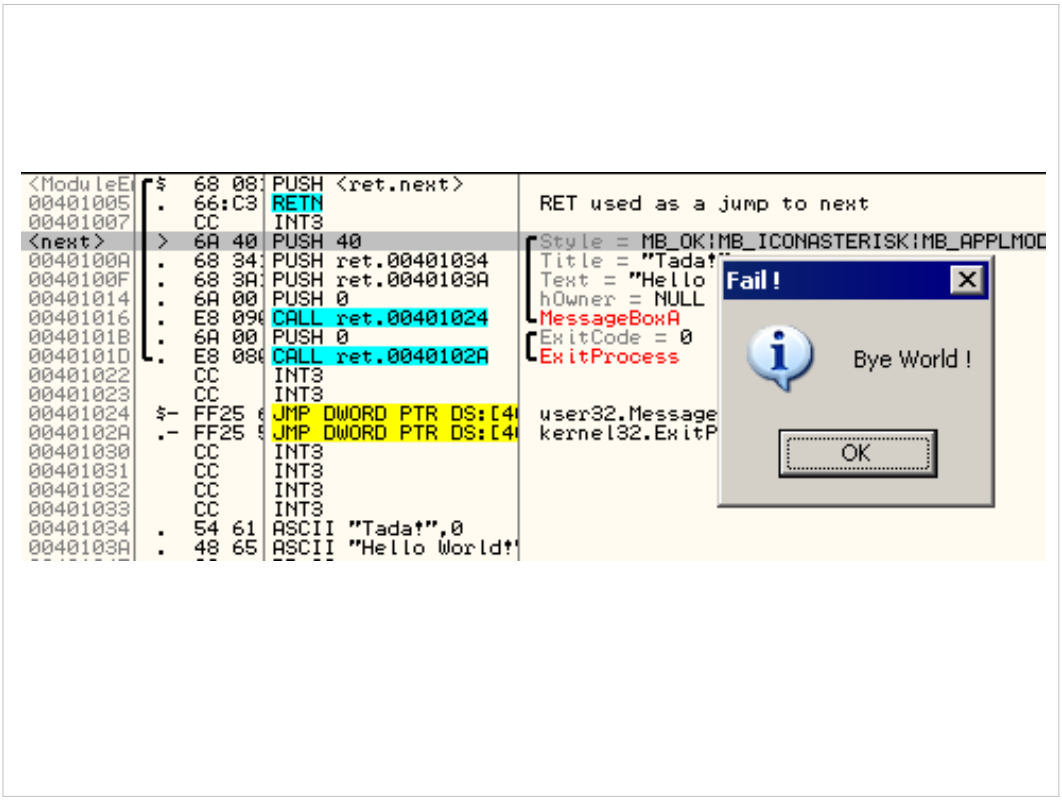
*push+ret*

```
.00401000:   push          next ;'j@h4' --↓1
.00401005:   retn ; -^-^-^-^-^-^-^-^-^-^-^-^-
.00401007:   int           3
next:       1push          040 ;'@'
.0040100A:   push          000401034 ;'Tada!' -
.0040100F:   push          00040103A ;'Hello Wo
.00401014:   push          0
.00401016:   call          MessageBoxA --↓4
```

anyone knows what will happen here ?

push, ret.
put an address on the stack, pop it and jump to it.

no possible trick, right...




so, what happened ?
olly even auto-comments the ret!

the 66: before the RETN makes return to IP, not EIP.

so here we returned to 1008, not 401008.

the other problem is that while different, there is no official name for this ret to word, 'small ret', 'ret16'....

## ...and so on...

- much more @ http://x86.corkami.com
  - also graphs, cheat sheet...
- too much theory for now...

I won't enumerate them all.
they're already on Corkami, with some other x86 stuff that might be useful to print.

too much theory with no practice never gives good results...

Corkami Standard Test

so I created CoST.

# CoST

- http://cost.corkami.com
- testing opcodes
- in a hardened PE
  - available in easy mode

```
C>CoST.exe
CoST - Corkami Standard Test BETA 2011/09/XX
Ange Albertini, BSD Licence, 2009-2011 - http://corkami.com

Info: Windows 7 found
Starting: jumps opcodes...
Starting: classic opcodes...
Starting: rare opcodes...
Starting: undocumented opcodes...
Starting: cpu-specific opcodes...
Info: CPUID GenuineIntel
Info[cpu]: MOVBE (Atom only) not supported
Starting: undocumented encodings...
Starting: os-dependant opcodes...
Starting: 'nop' opcodes...
Starting: opcode-based anti-debuggers...
Starting: opcode-based GetIPs...
Starting: opcode-based exception triggers...
Starting: 64 bits opcodes...
Starting: registers tests

...completed!
```

an opcode tester, in a tricky PE.
available in easy mode compile (less tricky), as CoST is quite difficult to debug :)

just run, and it roughly displays what happened.

## more than 150 tests

```
mov         eax,3
cmp         eax,3
jz        .07EFD0593
```

- classic, rare
- jumps (JMP to IP, IRET, …)
- undocumented (IceBP, SetALc...)
- cpu-specific (MOVBE, POPCNT,...)
- os-dependant, anti-VM/debugs
- exceptions triggers, interrupts, OS bugs,...
- ...

so, it contains a lot of various tests... (150 is the lower margin, depend how you count)

some trivial... some less trivial.

## a documented binary

exports + VEH = self commented assembly

```
CoST.exe          ↓FRO --------          a32 PE .7EFD0220|Hiew 8.15
4_Main:     mov     d,[0CAFEBABE],07EFD2CF7 ;'Starting: jumps opco
.7EFD022A:  call    jumps --↓2
.7EFD022F:  nop
.7EFD0230:  mov     d,[0CAFEBABE],07EFD2D14 ;'Starting: classic op
.7EFD023A:  call    classics --↓4
```

a lot of DbgOutput

```
1   [trick] Adding TLS 2 in TLS callbacks list
2   [trick] the next call's operand is zeroed by the loader
3   CoST - Corkami Standard Test BETA 2011/09/XX
4   Ange Albertini, BSD Licence, 2009-2011 - http://corkami.com
5
6
7   [trick] TLS terminating by unhandled exception (EP is executed)
8   [trick] allocating buffer [0000-ffff]
9   testing: NULL buffer
10  checking OS version
11  Info: Windows 7 found
12  [trick] calling Main via my own export
13  Starting: jumps opcodes...
14  Testing: RETN word
```

it's tricky, made by hand in assembly, so quite compact, yet 'commented' and self documented.

a VEH to add comments and one-line printing.

and internal exports to jump over sections of the file.

also, lots of detailed debug output.

32+64 = ...

```
.7EFD2540:   mov          eax,0F570D67C
.7EFD2545:   mov          ebx,3
.7EFD254A:   push         cs
.7EFD254B:   push         end --↓1
.7EFD2550:   push         033 ;'3'
.7EFD2552:   call         push_eip --↓2
push_eip:   2arpl         ax,bx
.7EFD2559:   dec          eax
.7EFD255A:   add          eax,eax
.7EFD255C:   retf ; -^-^-^-^-^-^-^-^-^-
end:        1cmp          ebx,0EAE1ACFC
.7EFD2563:   jz           next --↓3
.7EFD2565:   call         bad --↓4
next:       3cmp          eax,0D5C359F8
```

anyone sees what this is doing ?

executing code at push_eip...
then the same code with selector 33 (64b code)

so the same opcodes are executed twice, first in 32b mode, then in 64b.

same opcodes, different code

```
push_eip:  63D8    arpl        ax,bx
.7EFD2559: 48      dec         eax
.7EFD255A: 01C0    add         eax,eax
.7EFD255C: CB      retf ; -^-^-^-^-^-^.
```

```
push_eip:  63D8    movsxd      rbx,eax
.7EFD2559: 4801C0  add         rax,rax
.7EFD255C: CB      retf ; -^-^-^-^-^-^.
```

and these opcodes gives exclusive mnemonics to each side...

works fine under a 64b OS, but Windows itself can't even debug that.

same EIP, same opcodes, twice, and different code.

## CoST vs WinDbg & Hiew

WinDbg 6.12.0002.633

```
*** ERROR: Module load completed but symbols co
image7efd0000:
7efd0000 4d              dec     ebp
7efd0001 5a              pop     edx
7efd0002 ce              into
7efd0003 0f              ???
7efd0004 1838            sbb     byte ptr [eax]
7efd0006 e9db010000      jmp     image7efd0000+
7efd000b 0d436f5354      or      eax,54536F43h
```

Hiew 8.15

```
C:\Users\Ange\CoST.exe
.7EFD0000: 4D      dec        ebp
.7EFD0001: 5A      pop        edx
.7EFD0002: CE      into
the_dragon:        #UD
.7EFD0006: E91501  jmp        3_Entr
```

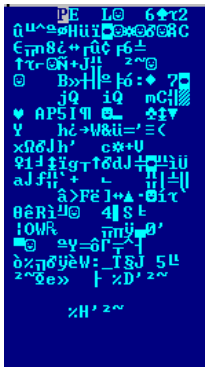
as you'd expect, WinDbg, following Intel docs too closely, will give you '??'

Hiew does that too a little.

but honestly, I found bugs in all disassemblers I looked at, no exception AFAIR. Even a crash in XED.

## a hardened PE


Top



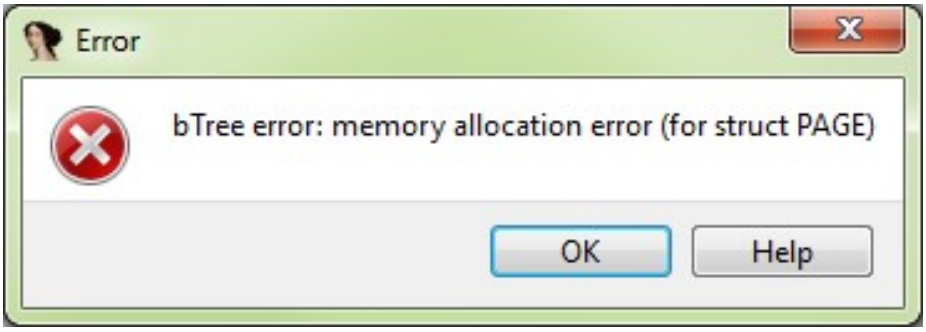

PE 'footer'

CoST was originally only an opcode tester.

then I added a few PE tricks...

have a look yourself, the top of the file, and the PE header (right at the bottom)

## CoST vs IDA




As you can see, IDA didn't really like it at first (fixed, now)

So, if CoST helps you to find a few bugs in your program, I'm not really surprised.

## CoST vs Dumpbin

HIDDEN SLIDE

Microsoft (R) COFF/PE Dumper Version 10.00.30319.01
Copyright (C) Microsoft Corporation. All rights reserved.

Dump of file CoST.exe

File Type: EXECUTABLE IMAGE

LINK : fatal error LNK1248: image size (9B097F81) exceeds maximum allowable size (80000000)

Cost vs MS's standard analysis tool...

what else would you expect ?

a bit more of PE...

but one single file, even full of tricks, is not enough to express all the possibilities of the PE file.

so I started my next project since I applied for Hashdays

## PE on corkami

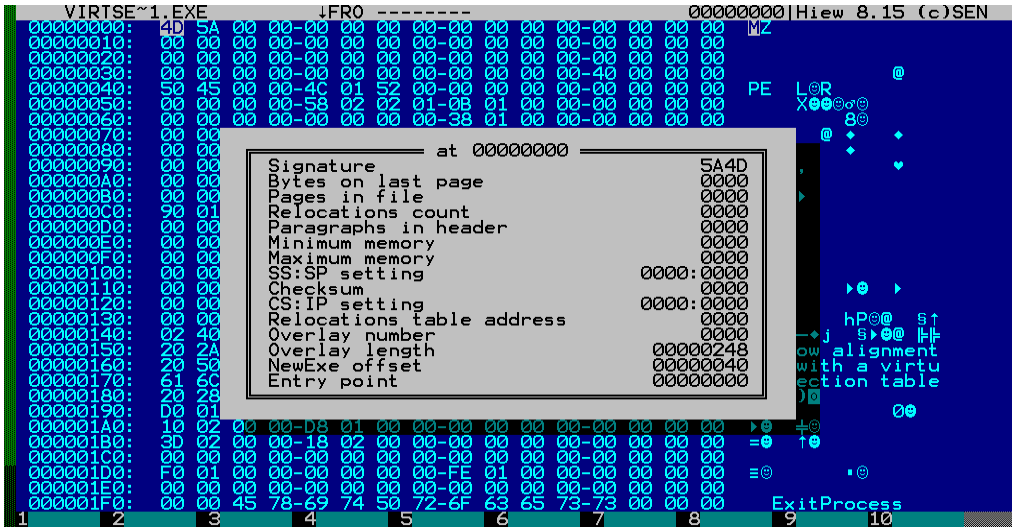
- some graphs
- a wiki page
  - http://pe.corkami.com
  - not “finished”
  - more than 100 PoCs
  - good enough to break <you name it>

I already made some useful graphs for PE files.

and I started a wiki page, with more than 100 PoCs, focusing, as usual, on precise aspects of the PE.

PE with no section, with 64k sections, with huge ImageBase, relocation encryption...

## virtual section table vs Hiew




in low alignments, the section table is checked but not used at all.
so, if it's full of zeroes, it will still work – under XP.

thus, with SizeOfOptionalHeader, you can set it in virtual space...

Hiew doesn't like that.
check the picture, it doesn't even identify it as a PE.

## Folded header

| Name | RVA | Size |
|---|---|---|
| Export | 88660001 | 10009988 |
| Import | 86600010 | 01000998 |
| Resource | 66000100 | 00100099 |
| Exception | 6000100F | F0010009 |
| Security | 000100FF | FF001000 |
| Fixups | 00100FF0 | 0FF00100 |
| Debug | 0100FF05 | 20FF0010 |
| Description | 100FF055 | 220FF001 |
| MIPS GP | 100FF055 | 220FF001 |
| TLS | 0100FF05 | 20FF0010 |
| Load config | 00100FF0 | 0FF00100 |
| Bound Import | 000100FF | FF001000 |
| Import Table | 6000100F | F0010009 |
| Delay Import | 66000100 | 00100099 |
| COM Runtime | 86600010 | 01000998 |
| (reserved) | 88660001 | 10009988 |

what do you think ?

when you can do ASCII art with the PE info, something dodgy is going on :)

this is ReversingLabs' dual PE header.
the PE header is partially overwritten (at exports directories) on loading.

the upper part is read from disk, the lower part, read in memory, is overwritten by the section that is folded over the bottom of the header.

## Weird export names

- exports = <anything non null>, 0

```
00401000: 6A01                                  push
00401002: 58                                    pop
00401000: 8BFF                                → retn
00401000: 8BFF                                → int
00401000: 8BFF                                → push
**********************************            → call
* Insert subliminal message here *            → add
**********************************            → retn ;
00401000: 8BFF                                → int
00401018: 202A                                 1and
```

export names can be anything until 0, or even null.

Hiew displays them inline, so, well, here is the PoC of weird export names

one of the other names in this PoC is LOOOONG enough to trigger a buffer overflow >:)

## 65535 sections vs OllyDbg


this is a 64k section PE against the latest Olly.

amazingly, it doesn't crash despite this funny message...

## one last...

- TLS AddressOfIndex is overwritten on loading
- Import are parsed until Name is 0

- under XP, overwritten after imports
  - imports are fully parsed
- under W7, before
  - truncated

same PE, loaded differently
under different Windows

this one is not very visual, yet quite unique.

TLS AoI points to an Import descriptor Name member...

depending on AoI or imports happening first, this is a terminator or not...

so the same PE gets loaded with more or less imports depending on the OS.

## conclusion

- x86 and PE are far from perfectly documented
- still some gray areas of PE or x86
  - but a bit less, every day

### official documentations lead to FAILURE

1. visit Corkami.com
2. download the PoCs
3. fix the bugs ;)

unlike what I used to believe, cpus and windows binaries are far from perfectly logical nor documented

I can't pretend I know it all, but I'm progressing slowly, and fully sharing with everybody.

If you only follow the official doc, you're bound to fail. especially with the malware landscape out there.

so give Corkami PoCs a try – and send me a postcard if you found some bugs

I seriously hope that MS will put WinDbg back to a more reactive release cycle, and will update it...

## Thanks

- Peter Ferrie
- Candid Wüest

Adam Błaszczyk, BeatriX, Bruce Dang, Cathal Mullaney, Czerno, Daniel Reynaud, Elias Bachaalany, Ero Carrera, Eugeny Suslikov, Georg Wicherski, Gil Dabah, Guillaume Delugré, Gunther, Igor Skochinsky, Ilfak Guilfanov, Ivanlef0u, Jean-Baptiste Bédrune, Jim Leonard, Jon Larimer, Joshua J. Drake, Markus Hinderhofer, Mateusz Jurczyk, Matthieu Bonetti, Moritz Kroll, Oleh Yuschuk, Renaud Tabary, Rewolf, Sebastian Biallas, StalkR, Yoann Guillot,...

# Questions ?

Eternal thanks to Peter Ferrie.
and I wouldn't be here if Candid didn't push me to.

a lot of people helped me in the process to make this presentation and the content on corkami, in one way or another.

Any questions ?

# Such a weird processor

## messing with opcodes

(...and a little bit of PE)

Ange Albertini

28th October 2011

@ange4771
@corkami (news only)


Thanks for your attention!

you can catch me online

I'm open to suggestions, feedback, requests...