FunCap

RAPID REVERSING WITH IDA PRO DEBUGGER

ANDRZEJ DERESZOWSKI

Who am I ?

Security consultant with focus on incident handling, forensics and malware analysis

Not a dedicated reverser – RE is just part of my job

=> I avoid RE as much as possible as it is just too time consuming

Tools we use

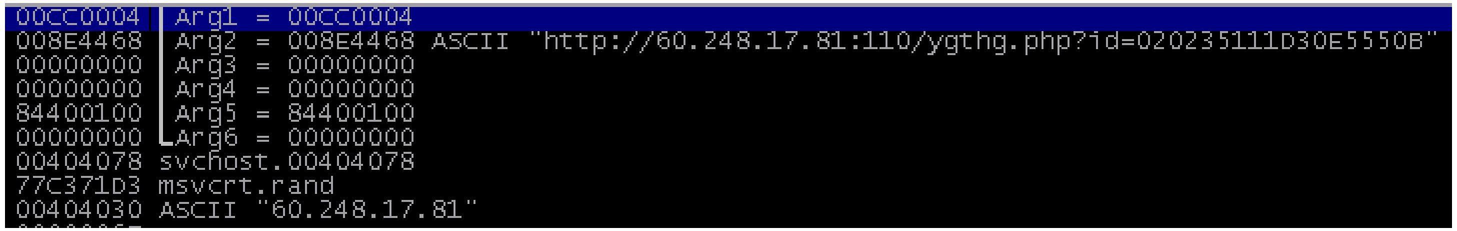
IDA Pro for static analysis

OllyDbg for debugging

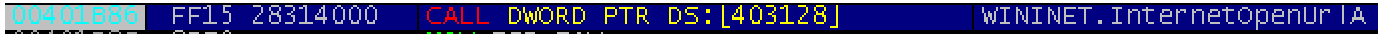
(other tools used by real reversing gurus like PIN, metasm etc. are out of scope here)

Problem

=> Olly gives a lot of good info during debugging



```
00CC0004  ┌Arg1 = 00CC0004
008E4468  │Arg2 = 008E4468 ASCII "http://60.248.17.81:110/ygthg.php?id=020235111D30E5550B"
00000000  │Arg3 = 00000000
00000000  │Arg4 = 00000000
84400100  │Arg5 = 84400100
00000000  └Arg6 = 00000000
00404078 svchost.00404078
77C371D3 msvcrt.rand
00404030 ASCII "60.248.17.81"
```



```
00401B86  FF15 28314000   CALL DWORD PTR DS:[403128]        WININET.InternetOpenUrlA
```

… but this won’t be visible in IDA

=> Unpacked code – needs rebuilding to load in IDA, not always easy

IN SHORT: No automatic connection between the two tools

Idea

Why not connect both worlds and provide automated solution ?

First I wanted to use IDA Pro tracer but realized it is too slow and generating not easily-readable data with too much noise

The inspiration:

⇒PaiMei Stalker by Pedram Amini - old and not developed any more, with only win32 userland support (uses PyDbg)

⇒Places breakpoints at each function start based on imported IDB from IDA

⇒Exports a script to load comments from the debugger to IDA’s listing

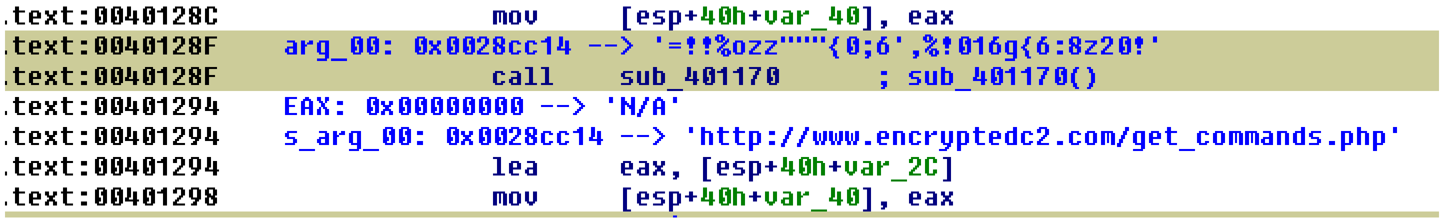
Let’s implement a solution by using IDA debugger !

Introducing FunCap

IDApython script/plugin

Aims to combine runtime info and feeds it into the static listing

IN SHORT: you can run some code in the sandbox VM and it will add useful comments to your IDA listing based on the recorded code execution



```
.text:0040128C                 mov     [esp+40h+var_40], eax
.text:0040128F    arg_00: 0x0028cc14 --> '=!!%ozz""""{0;6',%!016g{6:8z20!'
.text:0040128F                 call    sub_401170      ; sub_401170()
.text:00401294    EAX: 0x00000000 --> 'N/A'
.text:00401294    s_arg_00: 0x0028cc14 --> 'http://www.encryptedc2.com/get_commands.php'
.text:00401294                 lea     eax, [esp+40h+var_2C]
.text:00401298                 mov     [esp+40h+var_40], eax
```

RESULT: you understand some functions without even looking at them ➔ SAVES TIME!

Funcap – how it works

Places breakpoints on function call instructions (alternatively breakpoints can be places on function start and end)

Runs IDA debugger

When a breakpoint is hit it captures the arguments and function address and tries to dereference them and guess their type (currently only string, int and pointers)

Places a breakpoint directly after the call instruction

When the call returns they are dereferenced again to see how the memory was changed

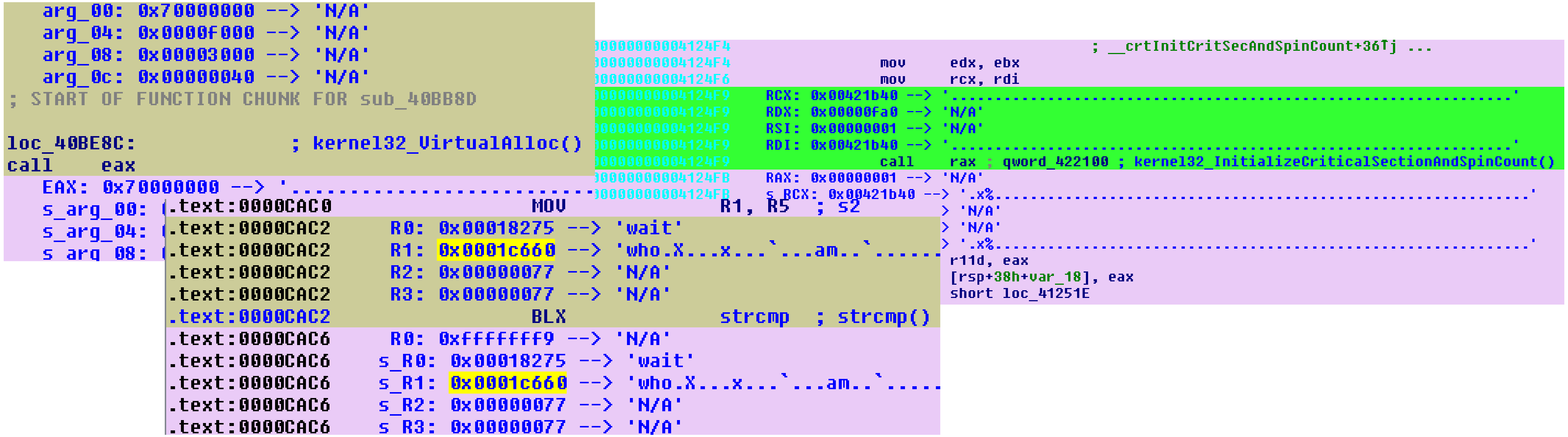
This information is dumped to a text file and inserted into the IDA listing

Funcap – features (1)

Supports ia-32, ia-64 and ARM – more can easily be added

Supports Win32, Win64, Linux32, Linux64, Android. No MacOS or iOS yet.

Supports almost any IDA debugger connector, even PIN tracer connector ☺



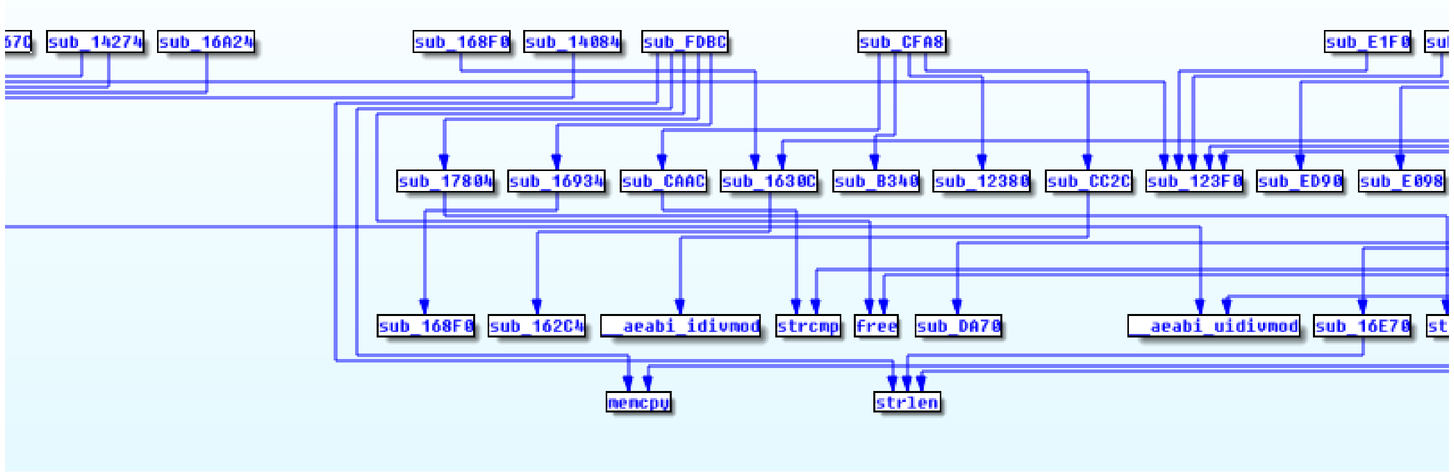
```
   arg_00: 0x70000000 --> 'N/A'
   arg_04: 0x0000f000 --> 'N/A'
   arg_08: 0x00003000 --> 'N/A'
   arg_0c: 0x00000040 --> 'N/A'
; START OF FUNCTION CHUNK FOR sub_40BB8D

loc_40BE8C:                 ; kernel32_VirtualAlloc()
call    eax
   EAX: 0x70000000 --> '.........................
   s_arg_00: (
   s_arg_04: (
   s_arg_08: (
```

```
00000000004124F4                                         ; __crtInitCritSecAndSpinCount+36↑j ...
00000000004124F4                   mov     edx, ebx
00000000004124F6                   mov     rcx, rdi
00000000004124F9         RCX: 0x00421b40 --> '...........................................................'
00000000004124F9         RDX: 0x00000fa0 --> 'N/A'
00000000004124F9         RSI: 0x00000001 --> 'N/A'
00000000004124F9         RDI: 0x00421b40 --> '...........................................................'
00000000004124F9                   call    rax ; qword_422100 ; kernel32_InitializeCriticalSectionAndSpinCount()
00000000004124FB         RAX: 0x00000001 --> 'N/A'
00000000004124FB         s_RCX: 0x00421b40 --> '.x%...........................................................'
                                           > 'N/A'
                                           > 'N/A'
                                           > '.x%...........................................................'
                                             r11d, eax
                                             [rsp+38h+var_18], eax
                                             short loc_41251E
```

```
.text:0000CAC0                    MOV                R1, R5   ; s2
.text:0000CAC2     R0: 0x00018275 --> 'wait'
.text:0000CAC2     R1: 0x0001c660 --> 'who.X...x...`...am..`......
.text:0000CAC2     R2: 0x00000077 --> 'N/A'
.text:0000CAC2     R3: 0x00000077 --> 'N/A'
.text:0000CAC2                    BLX                strcmp   ; strcmp()
.text:0000CAC6     R0: 0xfffffff9 --> 'N/A'
.text:0000CAC6   s_R0: 0x00018275 --> 'wait'
.text:0000CAC6   s_R1: 0x0001c660 --> 'who.X...x...`...am..`.....
.text:0000CAC6   s_R2: 0x00000077 --> 'N/A'
.text:0000CAC6   s_R3: 0x00000077 --> 'N/A'
```

Funcap – features (2)

Builds a runtime call graph



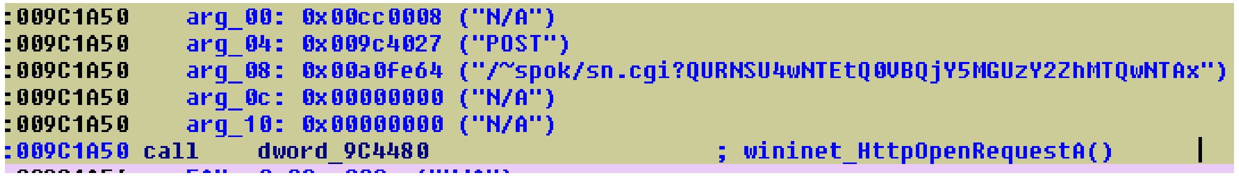
code_discovery mode to automatically deal with packers

```
Python> code_discovery = True

…
0x9c299a: new code section detected: [0x9c1000, 0x9c3000]
hooking function: sub_9C299A()
Function call: sub_1000156E+147 to sub_9C299A (0x9c299a)
```

Funcap – features (3)

Resolves indirect calls

API calls can be captured as well

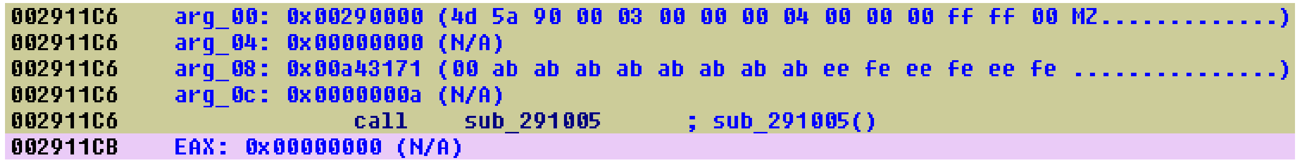


```
:009C1A50     arg_00: 0x00cc0008 ("N/A")
:009C1A50     arg_04: 0x009c4027 ("POST")
:009C1A50     arg_08: 0x00a0fe64 ("/~spok/sn.cgi?QURNSU4wNTEtQ0VBQjY5MGUzY2ZhMTQwNTAx")
:009C1A50     arg_0c: 0x00000000 ("N/A")
:009C1A50     arg_10: 0x00000000 ("N/A")
:009C1A50 call     dword_9C4480                     ; wininet_HttpOpenRequestA()
```

Full context is dumped to the file, subset of the context is pasted into IDAs' listing annotations

Hexdump or ASCII capture format

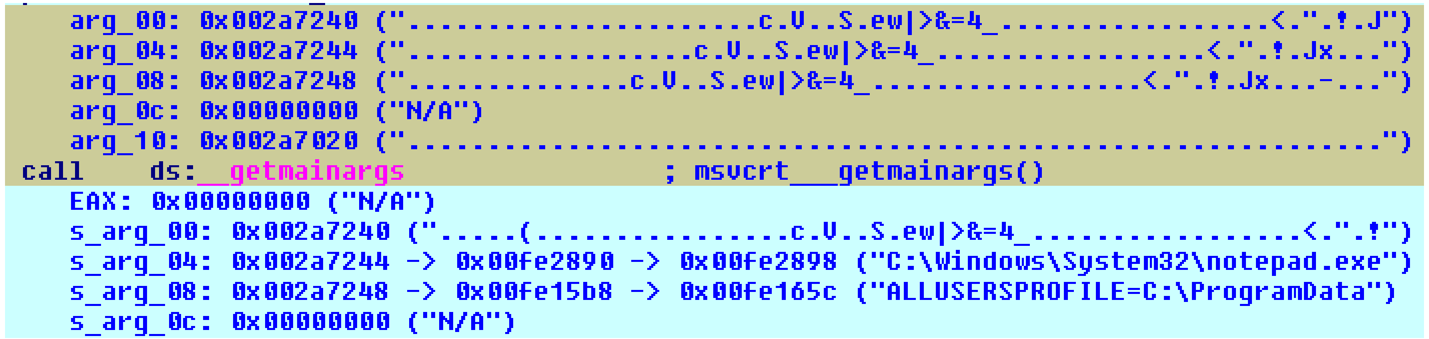
```
Python> hexdump = True
```



```
002911C6     arg_00: 0x00290000 (4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 MZ.............)
002911C6     arg_04: 0x00000000 (N/A)
002911C6     arg_08: 0x00a43171 (00 ab ab ab ab ab ab ab ab ee fe ee fe ee fe ...............)
002911C6     arg_0c: 0x0000000a (N/A)
002911C6                 call     sub_291005       ; sub_291005()
002911CB     EAX: 0x00000000 (N/A)
```

Funcap – features (4)

Recursive argument dereferencing – idea taken from PEDA for GDB



```
    arg_00: 0x002a7240 ("......................c.V..S.ew|>&=4_.................<.".!.J")
    arg_04: 0x002a7244 ("...................c.V..S.ew|>&=4_.................<.".!.Jx...")
    arg_08: 0x002a7248 ("...............c.V..S.ew|>&=4_.................<.".!.Jx...-...")
    arg_0c: 0x00000000 ("N/A")
    arg_10: 0x002a7020 ("............................................................")
call    ds:__getmainargs                ; msvcrt___getmainargs()
    EAX: 0x00000000 ("N/A")
    s_arg_00: 0x002a7240 (".....(................c.V..S.ew|>&=4_.................<.".!")
    s_arg_04: 0x002a7244 -> 0x00fe2890 -> 0x00fe2898 ("C:\Windows\System32\notepad.exe")
    s_arg_08: 0x002a7248 -> 0x00fe15b8 -> 0x00fe165c ("ALLUSERSPROFILE=C:\ProgramData")
    s_arg_0c: 0x00000000 ("N/A")
```

Capture scope easily configurable (which registers etc.)

Recursive function hooking mode for large binaries

```
Python> d.recursive = True
```

Easy command line interface in Python

Functions that were executed are marked by a different color

Funcap DEMO

1. Taidoor – basic example

2. ZEUS/Citadel – usage of the call graph

3. Unknown APT – code_discovery mode

4. Snake/Uroburos – Funcap in kernel mode (just results)

5. Android – Funcap for ARM/Thumb (just results)

Funcap – limitations

No threads following (recursive mode)

Code injected to another process is not going to be followed

Call graph a bit unfriendly to the user

Only basic types are dereferenced (no structures)

Argument count determination not very accurate on ia64 and ARM

Funcap – future directions

Threads following

- Breakpoint on thread creation ?

Remote process code injection following

- Cuckoo plugin ?
- Switching to kernel mode debugger ?

Better graph solution

- Visualize outside IDA (Gephi perhaps?)

Better argcount determination and complex types support

- Using decompiler plugin ?

Automation and database storage

Questions ?

deresz@gmail.com

http://github.com/deresz/funcap