

TP5 - Florian POMPIDOU

Fichier reçu

Le client a reçu un fichier `crackme_esdi` ce matin.

En première analyse :

- `file` : ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, [...] for GNU/Linux 3.2.0, stripped
- `objdump -f` : file format elf64-x86-64, architecture: i386:x86-64, flags 0x00000112: EXEC_P, HAS_SYMS, D_PAGED
- `strings` : Le contenu mentionne un CTF{mot_de_passe}, et surtout un `Usage: %s <password>`

Un petit binaire sensé renvoyer un string si on entre le bon mot de passe. En apparence, rien ne permet de voir si un sous-programme s'exécute.

GDB

Avec GDB, on peut voir un point d'entrée à l'adresse `0x401060`. Un `disas` ne marche pas, le programme est stripped.

J'ai fait un `x/30i 0x401060` (examine 30 éléments comme instructions) et j'ai reçu des données assembleur :

```
0x401060:  xor    %ebp,%ebp
0x401062:  mov    %rdx,%r9
0x401065:  pop    %rsi
0x401066:  mov    %rsp,%rdx
0x401069:  and    $0xfffffffffffffff0,%rsp
0x40106d:  push   %rax
0x40106e:  push   %rsp
0x40106f:  xor    %r8d,%r8d
0x401072:  xor    %ecx,%ecx
0x401074:  mov    $0x4011fa,%rdi
0x40107b:  call   *0x2f57(%rip)          # 0x403fd8
0x401081:  hlt
0x401082:  cs nopw 0x0(%rax,%rax,1)
0x40108c:  nopl   0x0(%rax)
0x401090:  ret
0x401091:  cs nopw 0x0(%rax,%rax,1)
0x40109b:  nopl   0x0(%rax,%rax,1)
0x4010a0:  mov    $0x404028,%eax
0x4010a5:  cmp    $0x404028,%rax
0x4010ab:  je     0x4010c0
0x4010ad:  mov    $0x0,%eax
0x4010b2:  test   %rax,%rax
0x4010b5:  je     0x4010c0
0x4010b7:  mov    $0x404028,%edi
0x4010bc:  jmp    *%rax
0x4010be:  xchg   %ax,%ax
```

```

0x4010c0:    ret
0x4010c1:    data16 cs nopw 0x0(%rax,%rax,1)
0x4010cc:    nopl    0x0(%rax)
0x4010d0:    mov     $0x404028,%esi

```

Je vois l'adresse `0x4011fa` qui inscrit dans le registre RDI. Je fouille 40 éléments dessus.

Décortiquage :

1. `0x401209: cml $0x2, -0x4(%rbp)` => le programme attend deux `argc`, sous le format `./binaire <argument>`, comme le texte "Usage:" le laissait entendre, avec un `je` (Jump if Equal) qui enquille
2. Si le `je` n'est pas activé, il continue et renvoie un premier `call` de la fonction `<printf@plt>`, certainement le texte "Usage:" en question, avec un `jmp`
3. `0x40123f: mov (%rax), %rax` => L'argument du binaire est mis dans le registre RDI et appelle la fonction adresse `0x401146`, avant un `test` type `%eax`
4. Si le résultat n'est pas égal à 0 (donc pas de renvoi `False`), un `jne` (Jump if Not Equal) envoie le résultat sur une deuxième vérification adresse `0x401277`
5. Si le deuxième test renvoie 0, il appelle un premier `<puts@plt>`, certainement un message d'erreur
6. Si non, il renvoie une dernière fois à un `call` qui renvoie à un `jne` qui renvoie le résultat à l'adresse `0x401291`. C'est très certainement là qu'on a la fonction de fin réussie

Les messages des fonctions :

Les adresses en questions sont en commentaires des lignes `lea` :

- `x/s 0x402008: 0x402008: "Usage: %s <password>\n"`
- `x/s 0x40201e: 0x40201e: "[-] Incorrect length."`
- `x/s 0x402034: 0x402034: "[-] Checksum failed."`

Je retrouve les informations du `strings ./crackme_esdi` du début.

Analyse de lignes supplémentaires

Avec un `x/20i 0x401280` je continue l'affichage des fonctions, et je relève les adresses en commentaires des `lea` :

- `x/s 0x402049: 0x402049: "[-] Access denied."`
- `x/s 0x402060: 0x402060: "[+] Access granted! Flag: CTF{3l1t3_R3v3rs3r}"`

J'obtiens le résultat, mais je poursuis en cherchant les deux adresses `call` vues plus haut suite au `x/40i 0x4011fa`, à savoir les adresses `0x401146` et `0x40116a`.

Je peux m'arrêter à `0x401146`, qui sort un `cmp $0xe, %rax`. Une conversion HEXA => DEC me donne 0xe comme "14". La taille du mot de passe du "CTF{".

Script Python

Pour automatiser le processus, il faut considérer qu'un script ne lira PAS le code en clair dans le retour de fonction finale. Non, il faut qu'il trouve la valeur dans le binaire.

Déchiffrer les HURDLE

Deux adresses derrière des fonctions de comparaison `cmp` (est-ce que le mot de passe en argument est égal au mot de passe dans la fonction ?) : `0x401190` et `0x4011a8`. Les résultats sont tombés :

- `0x4011c6: movzbl (%rax),%eax` : Charge un caractère de ton mot de passe : `password[i]`
- `0x4011c9: xor $0x42,%eax` : Fait un XOR avec `0x42` (66 en décimal, ou le caractère 'B')
- `0x4011cc: mov %eax,%ecx` : Stocke le résultat dans `%cl`
- `0x4011d3: lea 0xeb6(%rip),%rdx` : Calcule l'adresse de la clef cachée : `0x402090`
- `0x4011da: movzbl (%rax,%rdx,1),%eax` : Charge le vrai octet attendu : `secret[i]`
- `0x4011de: cmp %al,%cl` : Comparaison `!\'`
- `0x4011e0: je 0x4011e9` : Si c'est égal, on passe au caractère suivant

On sait par la ligne `0x4011d3`, de par son commentaire, que la clef est dans l'adresse `0x402090`.

Dont le contenu est tel :

```
0x402090: 0x71 0x2e 0x73 0x36 0x71 0x1d 0x10 0x71
0x402098: 0x34 0x71 0x30 0x31 0x71 0x30
```

Retour au script

Plutôt que "cracker" le mot de passe, on "traduit" juste ces valeurs avec le script quand on a fait le travail manuellement comme tantôt.

Script semi force-brute

Dans le cas où le script doit tout faire, il doit trouver :

1. La longueur du mot de passe attendue (14)
2. Les adresses/offset de 14 caractères
3. Les inscrire en base
4. Trouver l'endroit où le script renvoie un succès
5. Les tenter un par un jusqu'à validation en décodant l'adresse accédée au bon endroit

Le script attaché fonctionne. Voici le résultat sur une machine Linux (essentiel, pour les binaires ELF) :

```
(kali@GAWIN)-[~/TP5]
$ uv run solve_crackme.py
[*] '/home/kali/TP5/crackme_esdi'
Arch:      amd64-64-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX unknown - GNU_STACK missing
```

```
PIE:      No PIE (0x400000)
Stack:    Executable
RWX:      Has RWX segments
[*] Démarrage de la résolution automatique de ./crackme_esdi...
[+] 34 candidats potentiels identifiés.
[*] Validation dynamique des candidats via exécution...
[+] Crackme résolu : valeur = 3l1t3_R3v3rs3r
```

```
(kali@GAWIN)-[~/TP5]
$ uv run solve_crackme.py
[*] '/home/kali/TP5/crackme_esdi'
Arch:      amd64-64-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX unknown - GNU_STACK missing
PIE:       No PIE (0x400000)
Stack:     Executable
RWX:       Has RWX segments
[*] Démarrage de la résolution automatique de ./crackme_esdi...
[+] 34 candidats potentiels identifiés.
[*] Validation dynamique des candidats via exécution...
[+] Crackme résolu : valeur = 3l1t3_R3v3rs3r
```