Crackme Xorcist

Résolution concrète du crackme

Le mot de passe est la chaine de 16 octets : `"l0IHaYT5Cfuf1ERo"`
Pour valider le crackme, il suffit d'exécuter :

```
    xorcist "l0IHaYT5Cfuf1ERo"
```

Les headers elf

Le crackme est assemblé en mode bin. Les headers elf sont construits manuellement.

Ils sont faits de sorte à permettre l'exécution de la stack, et à rendre le programme incompatible avec `gdb`

```
; ELF Header minimal
ehdr:
    db 0x7F, "ELF"          ; Magic
    db 2, 1, 1, 0           ; 64-bit, little-endian, version, padding
    times 8 db 0
    dw 2                     ; e_type: ET_EXEC
    dw 0x3E                  ; e_machine: x86-64
    dd 1                     ; e_version
    dq _start                ; e_entry
    dq phdr - ehdr           ; e_phoff
    dq 1                     ; e_shoff (no sections) <------------- modifié pour corrompre le header
    dd 2                     ; e_flags <------------- modifié pour corrompre le header
    dw ehdr_size             ; e_ehsize
    dw phdr_size             ; e_phentsize
    dw 2                     ; e_phnum
    dw 0                     ; e_shentsize
    dw 2                     ; e_shnum <------------- modifié pour corrompre le header
    dw 1                     ; e_shstrndx <------------- modifié pour corrompre le header
ehdr_size equ $ - ehdr

; Program Header
phdr:
    dd 1                     ; p_type: PT_LOAD
    dd 5                     ; p_flags: PF_R | PF_X ;; exécution de la stack
    dq 0                     ; p_offset
    dq $$                    ; p_vaddr
    dq $$                    ; p_paddr
    dq filesize              ; p_filesz
    dq filesize              ; p_memsz
    dq 0x1000                ; p_align
phdr_size equ $ - phdr

; Program Header 2 pour exécution de la stack
phdr2:
    dd 0x6474e551            ; p_type = PT_GNU_STACK
    dd 7                     ; p_flags = PF_R | PF_W | PF_X
    dq 0                     ; p_offset = 0
    dq 0                     ; p_vaddr = 0
    dq 0                     ; p_paddr = 0
    dq 0                     ; p_filesz = 0
    dq 0                     ; p_memsz = 0
```

```
    dq 8                    ; p_align = 8 (pas important)
phdr2_size equ $ - phdr2
```

évidemment, il n'y a pas de sections, ni de nom de fonctions.

De l'instruction overlapping à gogo

Le code machine du crackme est parsemé de données arbitraires, qui essayent de limiter la capacité des crackers,

à avoir une vision globale du programme. Cela commence dès le point d'entrée du programme

```
_start:
    pop rdi ;; le nombre d'arguments
    pop rdi ;; le premier argument : l'exécutable
    pop rdi ;; le second argument : la chaine en entrée
    call get_length ;; fausse condition de taille pour embrouiller les camarades
    db 0x62
    dq 0x5403565545125344 ;; instruction overlapping
    dq 0x3104840387452923 ;; instruction overlapping
    dq 0x1320444702638491 ;; instruction overlapping
    dq 0x4380109662308712 ;; instruction overlapping
    dq 0x1684063102329871 ;; instruction overlapping
```

Un crackme à deux niveaux

Le crackme possède deux niveaux, il commence par un faux crackme,
C'est à dire un crackme qui ne débouche pas sur un succès.

Accéder au vrai crackme, implique d'avoir échoué au faux crackme.

Le faux crackme

1. Une disjonction de cas inutile

Il y a une disjonction de cas inutile, puisqu'elle à lieu au sein du faux crackme.

Elle n'est destinée qu'a semer la confusion chez nos camarades.

```
.fake_condition:
    cmp dl,9
    mov r11, $ ;; sauvegarde de la position du mdp, qui est donc à r11+19
    jge .fake_check_cesar1
    jl .fake_check_cesar2
    db 0x33
    dq 0xD4649694C6B6A466 ;; mot de passe partie 1
    dq 0x8AC8E6C86686A8DA ;; mot de passe partie 2
    dq 0x6235411016105610 ;; instruction overlapping
    dq 0x8461338410134865 ;; instruction overlapping
```

2. Déchiffrement César

Affectation à `r8`, de chiffré de César, avec un décalage de 2, de `"cosmogol"`.

```
    mov r8,0x6E7169716F757165
```

Comparaison de r8 déchiffré octet par octet avec la sous chaine de l'entrée utilisateur composée des 8 premiers bits

```
.loop_fake_check_cesar1:
    cmp cl,8
    je .pre_fake_check_cesar2
    sub r8b,2
    cmp BYTE [rdi+rcx],r8b

    mov rbx,rsp ;; future adresse du code + 262 dans la stack, subtilement stockée

    jne .pre_continue ;; sortie dans le cas du mdp
    shr r8,8
```

Parallèlement, un hash est construit dans r9, au fur et à mesure qu'on compare les caractères. Ce dernier est déterministe.

```
    mov r10b,BYTE [rdi+rcx]
    imul r10,r10,2
    add r9,r10
    imul r9,r9,42
    inc cl

    jmp .loop_fake_check_cesar1
```

- Si la chaîne n'est pas égale, on jump à `.continue` et on accède au vrai crackme
- Si la première sous chaine de 8 caractère de l'entrée utilisateur valide `cosmogol`, on accède au déchiffrement de la seconde sous chaine.

3. Exploitation du hash généré, pour déchiffrer la deuxième sous chaîne

On stocke dans `r8` une valeur très particulière, choisie spécifiquement pour vérifier la propriété qui suis

```
mov r8,0x6E716c10a85cdbfb ;; r9 = 0x6ff5fbc058d78
```

On compare l'octet de poids faible du résultat de la somme des octets de poids faible de r8 et r9,

au caractrère courant.

```
.loop_fake_check_cesar2:
    cmp BYTE [rdi+rcx],0
    je .pre_exit_with_error

    mov r10b,r8b
    mov r11b,r9b

    add r10,r11
    mov dl,r10b
    cmp dl,byte[rdi+rcx]

    jne .continue
    shr r8,8
    shr r9,8
    inc cl
    jmp .loop_fake_check_cesar2
```

Choix spécifique de r8 pour que lors de la somme, la solution imposée soit shadok.

Si on échoue la comparaison, on jump aussi à `.continue` -> on est tenté de penser qu'on est en échec. (il peut arriver d'avoir un segfault avant le jump)

Si la comparaison est réussie, ie `cosmogolshadok` on jump à exit_with_error. Il y a cependant une subtilité supplémentaire.

En effet, on ne saute pas directement à `.exit_with_error`, on saute à `.pre_exit_with_error` qui va faire jouer l'état actuel des registres dans la construction du syscall.

De cette manière on fait croire à un fausse piste dans le faux crackme.

```
.pre_exit_with_error:
    mov rax,r9
    add rax,0x36
    jmp .exit_with_error
```

```
.exit_with_error:
    ;; exit 1
    push 1
    pop rdi
    syscall
```

Mais d'où ça vient cosmogolshadok ?

Le vrai crackme

Génération dynamique de code et exécution de la stack

Le vrai crackme est exécuté dans la stack, et construit dynamiquement en fonction du premier caractère de l'entrée utilisateur.

Le code exécuté sera donc corrompu, et provoquera une erreur, si le premier caractère n'est pas le bon.

Cela permet de donner le sentiment à nos camarades qu'échouer au faux crackme implique d'échouer au crackme.

L'emploi du bruteforce pour cette étape, bien qu'il ne soit pas nécessaire car en suivant le flux d'exécution on peut comprendre ce qui se passe,

est efficace et doit se résoudre rapidement (128 caractères imprimables).

Dissimulation d'instructions parmis les instructions

Le vrai crackme commence après l'appel de la fonction get_length

```
.fake_condition:
    cmp dl,9
    mov r11, $ ;; sauvegarde de la position du mdp, qui est donc à r11+19
    jge .fake_check_cesar1
    jl .fake_check_cesar2
    db 0x33
    dq 0xD4649694C6B6A466 ;; mot de passe partie 1
    dq 0x8AC8E6C86686A8DA ;; mot de passe partie 2
    dq 0x6235411016105610 ;; instruction overlapping
    dq 0x8461338410134865 ;; instruction overlapping
```

Le mot de passe chiffré est stocké dans les instructions.

L'emplacement mémoire est sauvegardé dans le registre `r11` avec l'instruction `mov r11, rip`

De même, au sein de la boucle principale de la première partie du faux crackme, l'instruction `lea rdx,[rbp-taille_du_code]`

Permet de stocker l'adrese de la fonction à appeler dans la stack.

```
.loop_fake_check_cesar1:
    cmp cl,8
    je .pre_fake_check_cesar2
    sub r8b,2
    cmp BYTE [rdi+rcx],r8b

    mov rbx,rsp ;; future adresse du code (rsp - 8) dans la stack, subtilement stockée <-------------
------------------------------ ici

    jne .pre_continue ;; sortie dans le cas du mdp
    shr r8,8
    mov r10b,BYTE [rdi+rcx]
    imul r10,r10,2
    add r9,r10
    imul r9,r9,42
    inc cl

    jmp .loop_fake_check_cesar1
```

Fonction de génération dynamique du code

Le script python `generator_function_from_bytecode.py` permet de générer le code assembleur qui génère à partir d'un octet d'une valeur prédéterminée,

la suite d'octet qui correspond au code assembleur souhaité, dans la stack

Comme le code obfusqué est très long, pour ne pas polluer le rapport on se contente de présenter un exemple.

Ainsi voici un code clair :

```
    ;; print ok
    push 1
    pop rax
    mov rdi,rax
    mov rdx,rdi
    add rdx,2
    push 0x000a4b4f
    push rsp
    pop rsi
    syscall

    ;; exit 0
    xor rdi,rdi
    mov rax,0x3c
    syscall
```

Et sa version obfusquée :

```
    mov cl,0x61 ;; initialisation

    sub cl,0x5c ;; construction du code dans la stack
    add rax,rcx
    shl rax,8
    add cl,0xa
    add rax,rcx
    shl rax,8
    sub cl,0xf
    add rax,rcx
    shl rax,8
    add cl,0x0
```

```
add rax,rcx
shl rax,8
add cl,0x0
add rax,rcx
shl rax,8
add cl,0x3c
add rax,rcx
shl rax,8
add cl,0x7c
add rax,rcx
shl rax,8
add cl,0x47
add rax,rcx
push rax
xor rax,rax
sub cl,0xce
add rax,rcx
shl rax,8
add cl,0x17
add rax,rcx
shl rax,8
sub cl,0x43
add rax,rcx
shl rax,8
add cl,0xa
add rax,rcx
shl rax,8
add cl,0x4f
add rax,rcx
shl rax,8
sub cl,0xa
add rax,rcx
shl rax,8
sub cl,0x54
add rax,rcx
shl rax,8
add cl,0xa
add rax,rcx
push rax
xor rax,rax
add cl,0x41
add rax,rcx
shl rax,8
add cl,0x4
add rax,rcx
shl rax,8
add cl,0x19
add rax,rcx
shl rax,8
sub cl,0x66
add rax,rcx
shl rax,8
add cl,0xc0
add rax,rcx
shl rax,8
sub cl,0x3f
add rax,rcx
shl rax,8
sub cl,0x3b
add rax,rcx
shl rax,8
add cl,0xb2
```

```
    add rax,rcx
    push rax
    xor rax,rax
    sub cl,0x71
    add rax,rcx
    shl rax,8
    sub cl,0x41
    add rax,rcx
    shl rax,8
    add cl,0x7f
    add rax,rcx
    shl rax,8
    sub cl,0x3e
    add rax,rcx
    shl rax,8
    sub cl,0x41
    add rax,rcx
    shl rax,8
    add cl,0x10
    add rax,rcx
    shl rax,8
    sub cl,0x57
    add rax,rcx
    shl rax,8
    add cl,0x69
    add rax,rcx
    push rax
    xor rax,rax
```

évidemment, la taille du programme explose.

cipher

Vue d'ensemble

`cypher` est une fonction de chiffrement simplifiée opérant sur des blocs de 16 octets. L'objectif est de créer un mécanisme de vérification de mot de passe analytiquement réversible tout en intégrant plusieurs techniques d'obfuscation pour compliquer l'analyse statique et dynamique.

Architecture du chiffrement

Principe de base

La fonction cipher prend en entrée 16 octets répartis en deux registres de 64 bits (rdi et rsi) et produit deux valeurs chiffrées (rdx et rbx). Le chiffrement s'effectue octet par octet selon la transformation suivante :

```
c = (b XOR 4) + b
```

où b est l'octet en clair et c l'octet chiffré. Cette transformation préserve la structure positionnelle : chaque octet chiffré dépend uniquement de son octet source, ce qui garantit la réversibilité analytique.

Implémentation de la boucle

Le traitement se déroule en deux phases de 8 octets chacune. Pour chaque octet, on extrait le byte de poids faible avec `movzx rax, dil`, on applique la transformation, puis on construit progressivement le résultat en shiftant rbx de 8 bits à gauche avant d'insérer le nouvel octet chiffré.

Le registre `r8` sert de compteur décrémental pour parcourir les 8 octets, tandis que r9 indique la phase courante (0 pour la première moitié, 1 pour la seconde). À la fin de la première phase, le résultat rbx est sauvegardé dans rdx, rbx est réinitialisé, et on charge les 8 octets suivants depuis rsi dans rdi.

Vérification intégrée

Au lieu de retourner simplement les valeurs chiffrées, la fonction intègre directement une comparaison avec des valeurs cibles hardcodées. Les hashs attendus sont poussés sur la pile puis comparés octet par octet avec les résultats calculés via une implémentation de strcmp.

Si la comparaison échoue, le programme exit avec un code d'erreur. Si elle réussit, le programme affiche "OK\n" et termine proprement. Cette approche fusionne chiffrement et validation dans un même bloc de code.

Techniques d'obfuscation

Prédicats opaques

Deux prédicats opaques sont insérés stratégiquement dans le code pour créer des branches mortes qui ne seront jamais prises mais compliquent l'analyse.

Le premier prédicat exploite une propriété mathématique : le carré d'un nombre modulo 2 est toujours pair. On calcule `rdx * rdx` puis on teste le bit de parité avec `test al, 1`. Le jump conditionnel pointe vers un faux exit qui ne sera jamais exécuté.

Le second prédicat repose sur la congruence `7x mod 2 = x mod 2`. On calcule 7x via `lea r10, [rax + rax*8]` suivi de deux soustractions, puis on XOR avec la valeur originale.

Ces prédicats introduisent du bruit dans le control flow graph sans altérer le comportement réel du programme.

Dispatcher basique

Un dispatcher simple est ajouté pour briser la linéarité du flux d'exécution. Il route l'exécution selon la valeur de `r8` :

- Si `r8` = 8, on entre dans l'état 0 qui initialise `r15` puis saute vers la boucle principale
- Si `r8` = 0, on entre dans l'état 1 qui gère la transition entre les deux moitiés
- Sinon, on continue directement dans la boucle

Ce dispatcher fragmente le code en états discrets et oblige l'analyste à suivre les transitions d'état plutôt qu'un flux séquentiel simple.

Position-independent code

L'ensemble du code est position-independent pour permettre une exécution depuis la pile : les accès mémoire utilisent systématiquement un adressage relatif à RIP. De plus, tous les jumps sont naturellement PC-relatifs en x86-64.

Cette propriété permet de copier dynamiquement le code sur la pile exécutable et de l'invoquer depuis cette position, rendant l'analyse statique plus difficile puisque le code n'est plus à son emplacement nominal.

Réversibilité analytique

Le chiffrement reste facilement inversible malgré l'obfuscation. Pour chaque octet chiffré c, on peut tester les 256 valeurs possibles de b jusqu'à trouver celle qui satisfait `((b XOR 4) + b) & 0xFF = c`.

Cette recherche exhaustive nécessite au maximum 256 * 16 = 4096 tests, exécutables en quelques millisecondes. Un script Python trivial peut donc récupérer le mot de passe à partir des valeurs TARGET_HASH hardcodées dans le binaire.

L'obfuscation vise à compliquer l'identification de l'algorithme et la localisation des constantes de comparaison, pas à rendre le problème cryptographiquement dur.

Résumé de la méthode de résolution

1. Garder l'esprit ouvert et ne pas s'acharner sur le faux crackme
2. Scruter minutieusement le flux d'exécution du programme et que la stack est exécutée
3. Comprendre ce qui se joue avec la fonction de génération du code dans la stack, à savoir bonne valeur

   du premier caractère du mot de passe testé
4. Tester possibilités de première lettres selon l'heuristique de la ressemblance des premiers bytecodes

produits à du bytecode cohérents, ou bien en bruteforce
5. Arriver au vrai crackme, anaylser la fonction de chiffrement et l'inverser.
6. Localiser le mot de passe chiffré dans le programme.
7. Déchiffrer le mot de passe chiffré.

Synthèses des astuces et méthodes d'obfuscation

- corruption de headers
- instruction overlapping
- explosion superficielle des branchements conditionnels
- faux crackme, et fausses pistes
- chiffrement naifs
- construction dynamique de code
- exécution de la stack
- dissimulation d'instructions clés