Sécurité Logicielle

– Examen (**1**) –

1 Assembleur x86-32 (Barême approximatif : 8 points)

Reconstituez (approximativement) le code C de la fonction qui correspond au code assembleur suivant et expliquez son usage (à votre avis) :

```
Disassembly of section .text:

00000000 <.tea>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   83 ec 0c                sub    $0xc,%esp
   6:   89 1c 24                mov    %ebx,(%esp)
   9:   89 74 24 04             mov    %esi,0x4(%esp)
   d:   89 7c 24 08             mov    %edi,0x8(%esp)
  11:   8b 45 08                mov    0x8(%ebp),%eax
  14:   8b 55 0c                mov    0xc(%ebp),%edx
  17:   8b 30                   mov    (%eax),%esi
  19:   8b 78 04                mov    0x4(%eax),%edi
  1c:   b9 b9 79 37 9e          mov    $0x9e3779b9,%ecx

00000021 <.tea_top>:
  21:   89 fb                   mov    %edi,%ebx
  23:   c1 e3 04                shl    $0x4,%ebx
  26:   03 1a                   add    (%edx),%ebx
  28:   8d 04 0f                lea    (%edi,%ecx,1),%eax
  2b:   31 c3                   xor    %eax,%ebx
  2d:   89 f8                   mov    %edi,%eax
  2f:   c1 e8 05                shr    $0x5,%eax
  32:   03 42 04                add    0x4(%edx),%eax
  35:   31 c3                   xor    %eax,%ebx
  37:   01 de                   add    %ebx,%esi
  39:   89 f3                   mov    %esi,%ebx
  3b:   c1 e3 04                shl    $0x4,%ebx
  3e:   03 5a 08                add    0x8(%edx),%ebx
  41:   8d 04 0e                lea    (%esi,%ecx,1),%eax
  44:   31 c3                   xor    %eax,%ebx
  46:   89 f0                   mov    %esi,%eax
  48:   c1 e8 05                shr    $0x5,%eax
  4b:   03 42 0c                add    0xc(%edx),%eax
  4e:   31 c3                   xor    %eax,%ebx
  50:   01 df                   add    %ebx,%edi
  52:   81 c1 b9 79 37 9e       add    $0x9e3779b9,%ecx
  58:   81 f9 d9 b0 26 65       cmp    $0x6526b0d9,%ecx
  5e:   75 c1                   jne    21 <.tea_top>
  60:   8b 45 08                mov    0x8(%ebp),%eax
  63:   89 30                   mov    %esi,(%eax)
  65:   89 78 04                mov    %edi,0x4(%eax)
  68:   8b 1c 24                mov    (%esp),%ebx
  6b:   8b 74 24 04             mov    0x4(%esp),%esi
  6f:   8b 7c 24 08             mov    0x8(%esp),%edi
  73:   83 c4 0c                add    $0xc,%esp
  76:   5d                      pop    %ebp
  77:   c3                      ret
```

2 Once upon a `free()` (Barême approximatif : 12 points)

Lisez l'article "*Once upon a `free()`*" par anonymous (Phrack #57, 2001). Puis rédigez des réponses aux questions suivantes.

2.1 System V `malloc()`

Questions

1. À quoi servent les deux derniers bits du champs qui indiquent la taille d'un bloc, et pourquoi peut-on les utiliser sans risque ?
2. Que se passe-t-il lorsque deux blocs consécutifs se retrouvent inutilisés tous les deux ?
3. Lors d'un dépassement de tampon dans le tas (*heap*), quel bloc sera touché par ce dépassement ? Et, quelle est la première condition à respecter si l'on veut pouvoir faire l'exploitation du bug et pourquoi ?
4. Donnez la deuxième condition à satisfaire pour permettre l'exploitation et donnez le morceau de code qu'elle permet d'atteindre (mettez aussi le `if` qui le conditionne).
5. Faites une synthèse qui récapitule les différentes étapes permettant d'écrire à un endroit choisi de la mémoire grâce à un bug de dépassement de tampon dans le tas (*heap*).

   Et, dites quand intervient cette écriture dans le processus de la gestion de la mémoire (À l'allocation du bloc où se situe le dépassement de tampon ? À l'allocation du bloc voisin de celui où se situe le dépassement de tampon ? À la libération de...).

2.2 Linux `malloc()` (Doug Lea)

Questions

1. Pourquoi est-ce que la taille minimale d'un `malloc()` est forcément d'au moins 8 octets ?
2. Dans la structure de l'en-tête d'un *chunk* mémoire, il est signalé qu'extraire la taille de ce *chunk* se fait en calculant `hd & ~PREV_INUSE`. Donnez la valeur de `PREV_INUSE` et expliquez pourquoi il est nécessaire de faire cette manipulation.
3. Expliquez le processus de nettoyage de la liste des *chunks* lors d'un appel à `free()`.
4. Faites une synthèse qui récapitule les différentes étapes permettant d'écrire à un endroit choisi de la mémoire grâce à un bug de dépassement de tampon dans le tas (*heap*).

   Et, dites quand intervient cette écriture dans le processus de la gestion de la mémoire (À l'allocation du bloc où se situe le dépassement de tampon ? À l'allocation du bloc voisin de celui où se situe le dépassement de tampon ? À la libération de...).

2.3 Synthèse et Mise en application

Questions

1. Quel appel système permet de réserver de l'espace mémoire sur le tas (*heap*) ?
2. En utilisant la possibilité d'écrire ce que vous désirez et où vous désirez en mémoire, expliquez comment monter une attaque qui permet de récupérer le contrôle sur l'exécution du programme.
3. Parmi les protections classiques (Canary, NX, ASLR, PIE, RELRO, ...) lesquelles empêcheraient l'exploitation de ce type de faille ? Et, expliquez pourquoi les autres techniques seraient inefficaces dans ce cas là.
4. Explicitez les conditions particulières ou les difficultés qui peuvent être rencontrées lors de l'exploitation de tels bugs.