Année Universitaire 2021-2022 session 1 de printemps

Parcours : TCY Master CSI **UE :** 4TCY801U
Épreuve : Examen Sécurité logicielle
Date : 10 mai **Heure :** 9h **Durée :** 3h
Modalités : Épreuve sans document, calculatrice autorisée
Épreuve de : Mr Samuel Thibault 3 **pages**

université de BORDEAUX

Collège Sciences et technologie

Exercice 1. Les buffers contre-attaquent

Q1.1 `ret@` désigne l'endroit dans la pile où se trouve stockée une adresse de retour dans la fonction appelante. Ici, on avait placé le breakpoint sur `printf`, on vient donc seulement de rentrer dedans. Sur le sommet de la pile (en `ffffc23c`) on a donc l'adresse de retour vers la fonction `f` (l'instruction juste après l'appel à `printf`). L'adresse indiquée par `ret@` est donc probablement plutôt l'adresse de retour vers la fonction `main`.

Q1.2 On remarque quinze apparitions de `41` entre les adresses `0xffffc250` et `0xffffc25c`, c'est donc très probablement les `A`.

Q1.3 En vrai, en assembleur il n'y a rien qui permet de savoir combien de paramètres la fonction prend, c'est seulement une convention entre l'appelant et l'appelé. `pframe` montre où sont les paramètres, mais ne sait pas vraiment combien il y en a.

Q1.4 Pourquoi `pframe` ne montre par contre pas 3 paramètres ?

On remarque qu'à l'adresse `0xffffc268`, vers laquelle pointe `bp`, et qui contient donc la valeur précédente de `bp`, contient `0xffffc278`. Donc `0xffffc278` ne peut pas être l'adresse d'un paramètre puisqu'elle est censée contenir la valeur encore-précédente de `bp`.

Q1.5 `gets` n'a pas de paramètre pour spécifier la taille du tampon utilisé, il remplit donc sans se poser de question et peut déborder du tampon, écrasant ainsi selon les ces addresse de retour (le `ret@` dans l'affichage de `pframe`, justement), structures d'allocation de `malloc`, ...

Q1.6 Lorsqu'on ressort d'une fonction, les variables ne sont pas nettoyées, elles sont seulement désallouées (`esp` est juste remonté). Quand on rentre dans une autre fonction, les variables locales nouvellement allouées se retrouve donc contenir les valeurs qui "traînent" un peu partout sur la pile, venant des variables locales de la fonction précédemment appelée. Dans `main` il n'y a pas d'autre appel avant `f`, mais avant l'appel à `main` la libc a besoin de s'initialiser, ce qui appelle diverses fonctions.

Q1.7 C'est l'Address-Space Layout Randomization : le programme est placé à une adresse aléatoire tirée au lancement du programme. Toutes les adresses des fonctions du programme se retrouvent ainsi décalées de la même façons. Ici `0xffffc26c` et `0xffffc23c` contiennent des adresses de retour, donc décalées de la même façon. On fait cela pour rendre les attaques plus difficile : l'attaquant ne sait pas à l'avance où sont les variables à inspecter, les fonctions à faire appeler, etc.

Q1.8 Il s'agit a priori de l'adresse de retour de `main` lui-même (on avait vu que `0xffffc78` contenait la sauvegarde de `bp`), vers la fonction de la libc qui s'occupe d'appeler `main`. Son changement aléatoire correspond donc au déplacement aléatoire de la libc elle-même.

Q1.9 `printf` interprète son premier paramètre. Si l'utilisateur a tapé des `%`, il va donc les interpréter. Typiquement, s'il tape `%p`, `%2$p`, `%3$p`, il va pouvoir faire afficher par `printf` le contenu de la pile ! Et potentiellement dévoiler ce qui est stocké, comme des mots de passe.

Q1.10 On a vu que `0xffffc23c` est l'adresse de retour de `printf` dans `f`. Juste au-dessus on a donc le premier paramètre de `printf`, l'adresse du format. Si l'utilisateur tape `%p` c'est la valeur juste au-dessus (`0x0000001`) qui va être imprimée. Avec `%2$p` c'est la valeur encore au-dessus, etc. jusqu'à `%11$p` qui va imprimer la valeur `ret@`, l'adresse de retour dans `main`. En regardant le code assembleur de `main`, on peut en déduire le nombre d'octets à y soustraire pour remonter au début de `main`.

Q1.11 Une fois qu'on a l'adresse de `main`, il est très facile d'ajouter/soustraire le bon décalage pour obtenir l'adresse de n'importe quelle variable ou fonction du programme. On peut alors imprimer le contenu de variables en mettant son adresse dans `buf` suivi d'un `%4$s`, voire les modifier avec `%4$n` ! Ou alors on peut faire appeler une fonction en bourrant `buf` avec $4 * 7 = 28$ octets avant de mettre l'adresse de la fonction, qui se retrouve alors écrite à `ret@`, et donc appelée lors du retour de `f` au lieu de retourner dans `main` !

Exercice 2. Lecture d'assembleur

Q2.1 La fonction met en place un registre `ebp`, et donc ce sont les accès à `0xn(%ebp)` ($n \geq 8$) qui sont des accès aux paramètres. Donc les instructions aux adresses `6`, `9`, `c`.

Q2.2 Il y a divers *jumps*, mais le seul qui remonte, c'est celui à l'adresse `1a`, vers l'adresse `c`. La boucle est donc entre ces deux adresses. On constate que `%eax` est seulement incrémenté à l'adresse `19`, il avait été initialisé à 0 à l'adresse `1`, c'est donc apparemment un simple compteur.

Q2.3 Cela signifie que l'on regarde en mémoire à l'adresse `ecx + eax * 4`. Ici c'est une lecture mémoire.

Q2.4 Il y a deux *jumps* au milieu de la boucle.

À l'adresse `f`, on sort de la boucle quand la comparaison juste au-dessus indique une égalité (résultat de la soustraction Égal à zéro). La comparaison est entre le 3e paramètre de la fonction et notre compteur.

À l'adresse `17`, on sort de la boucle quand la comparaison juste au-dessus indique une différence (résultat de la soustraction Non Égal à zéro). La comparaison est entre ce qu'on a lu à l'adresse `ecx+eax*4` ce qu'on a lu à l'adresse `edx+eax*4`.

Q2.5 Puisque `edx` et `ecx` sont simplement les 2 premiers paramètres de la fonction, et que `eax` est un simple compteur qui avance, et qu'on sort lorsque le compteur est devenu égal au 3e paramètre ou lorsque l'on constate une différence entre ce qui est lui depuis la mémoire indexée depuis `edx` et `ecx`, c'est un simple `strcmp`.

Q2.6 On a de bas en haut l'adresse de retour et les trois paramètres, les deux premiers étant par exemple des pointeurs vers le tas, et le troisième leur taille.

```
arg3      0x0000002a
arg2      0x60345456
arg1      0x60123456
@ret sp-> 0x56xxxxxx
```

Q2.7 L'encodage de `xor %eax,%eax` est plus court, cela prend moins de place, c'est intéressant pour les shellcodes d'être petits. Aussi, l'encodage n'expose pas d'octet nul, ce qui permet de l'injecter même s'il y a des fonctions utilisées s'arrêtant à nul, comme strcpy.

Exercice 3. qmail fun

Do as little as possible in setuid programs.

Q3.1 Lorsqu'on exécute un programme qui a le bit `setuid` activé, le processus lancé obtient automatiquement l'identité (`uid`) du propriétaire du programme. Cela donne ainsi des privilèges potentiellement important au programme exécuté par le processus, il faut donc qu'il fasse attention à ne pas faire plus que ce qu'il est censé faire, si l'utilisateur peut détourner ce que le programme fait, en pratique il aura obtenu les privilèges du propriétaire du programme, ce qui peut être dramatique si ce propriétaire est `root`.

Don't parse.

Q3.2 Le *parsing* est toujours un exercice difficile, il faut être sûr de ce qu'on fait. Par exemple, si l'on récupère le contenu d'un champ de formulaire rempli par un utilisateur, il vaut mieux ne pas implémenter soi-même la vérification de ce qui devrait être interdit avant de construire une requête SQL incluant ce contenu, sous risque d'oublier que certains caractères sont interprétés par SQL. Il vaut mieux déléguer cela à une bibliothèque spécialisée. Ou sinon n'accepter que ce qui est le plus simple : les lettres (et donc ne pas faire de *parsing* du tout).

Q3.3 n étant un `int` et les constantes étant simplement entières, les calculs se font en précision 32 bits. Si l'on donne par exemple $n = 2^{32} - 1$, l'appelant croit avoir alloué 4 Go de mémoire, mais une fois le calcul d'alignement fait, le résultat est... 0 (modulo 2^{32}), et donc on alloue zéro octets. Si `malloc` refuse, l'allocation échoue et retourne NULL et le programme pourrait peut-être alors crasher. Si `malloc` accepte et alloue simplement une zone avec aucune place dedans, l'appelant va essayer d'écrire 4 Go à cet endroit... et donc déborder complètement dans tout ce qui est à côté.

Q3.4 Si le calcul de la ligne 7 déborde, `x->a` va contenir une valeur très petite, et donc on réduit dramatiquement la taille du tampon `x->field`. Si par exemple n vaut $2^{32} - 1$, l'appelant de `stralloc_readyplus` va écrire jusqu'à 4 Go *après* la petite zone allouée.

Q3.5 À ce moment-ci, l'appel à `byte_copy` va écrire à la fin de l'allocation M_{i+2}, qui se trouve coïncider avec le début de la libc. Parce que l'allocation se fait par mmap, on a remplacé le début de la libc avec l'allocation M_{i+2}. On peut alors choisir ce que l'on expose à la place du début de la libc. On peut par exemple modifier la table des symboles de la libc.

Q3.6 Avec l'ASLR, la position de la libc est aléatoire, et c'est alors bien plus difficile de viser où écrire la bonne valeur pour écraser au bon endroit dans la libc.

Q3.7 On avait vu qu'elle nous donnait une adresse de retour vers la libc. Autrement dit elle nous dévoile la position de la libc. On peut alors calculer la visée pour faire fonctionner l'exploit !

`ret`