HEIG-VD | REDS Institut Reconfigurable and Embedded Digital Systems

Programmation assembleur (ASM)

Prof. Daniel Rossier
Assistants : Bruno Da Rocha Carvalho, Thomas Rieder

Appels et contextes de fonctions

lab07 (24.05.2023)

Objectifs de laboratoire

L'objectif de ce laboratoire est d'apprendre à créer et utiliser les fonctions en assembleur ARM et x86.

Validation du laboratoire

Un script de rendu vous est fourni afin de packager les fichiers sources modifiés sans inclure tous les fichiers générés (code objet, exécutables, etc.). Ce script s'appelle ***rendu.sh*** et sera fourni pour tous les laboratoires.

```
reds@reds2021:~/asm/asm_lab0X$ ./rendu.sh
```

Cette commande va générer un fichier *asm22_lab0X_rendu.tar.gz* (où X est le numéro du labo) et c'est cette archive qui sera à remettre sur Cyberlearn. Il est demandé de toujours préparer l'archive du laboratoire grâce au script de rendu fourni.

Récupération des sources & mise en place de l'environnement

```
reds@reds2022:~$ retrieve_lab asm lab07
```

Étape 1 – ARM – ABI – Fonctions Variadiques

Une fonction « *variadique* » est une fonction dont l' « *arité* » est indéfinie. L' « *arité* » étant le nombre d'arguments que prend une fonction[1]. Un exemple bien connu est la fonction « printf() », elle peut prendre un nombre d'arguments variables, par exemple :

```
printf("Salut !\n"); // 1 argument
printf("La valeur vaut %d\n", valeur); // 2 arguments
printf("La valeur vaut %d et en hexadécimal 0x%0x\n", valeur, valeur); // 3
arguments
```

En C une telle fonction s'implémente grâce aux macros définies dans le header « stdarg.h »[2]. Ici nous allons en implémenter une à la main en assembleur ARM.

Nous allons réaliser la fonction avec la signature suivante :

```
uint32_t sum_asm(uint32_t n, ...); // Somme n arguments
```

[1] https://en.wikipedia.org/wiki/Variadic_function
[2] https://en.wikipedia.org/wiki/Stdarg.h

Afin que la fonction puisse être appelée depuis du code C il va falloir respecter la convention d'appel ARM AAPCS (cf. Document sur Cyberlearn et le cours sur les fonctions).

a) Implémenter la fonction « sum_asm() » dans le fichier « ***arm/sum/sum_asm.S*** »
b) Testez le bon fonctionnement de votre solution avec le programme « ***sum*** »

Étape 2 – x86 – ABI – *Recursion / Tail Recursion*

Dans cette étape nous allons écrire une fonction récursive de trois manières, une version naïve, une un peu meilleure et finalement une version « tail recursive ».

Lorsqu'on appelle une fonction il faut créer un nouveau contexte de fonction, faire de la place pour les variables locales, et ceci fait grandir la pile. Au-delà d'une certaine limite cela ne pourra plus continuer.

a) Implémenter la fonction « silly_fib_asm() » qui permet de calculer le nième nombre de Fibonacci[3] de manière naïve, selon le code C ci-dessous. respecter l'ABI x86 CDECL. (fichier « ***x86/fib/fib_asm.S*** »)

```
uint32_t fib(uin32_t n) {
        if (n < 2)
                return n;
        return fib(n-1) + fib(n-2); // Pas terrible...
}
```

b) Tester votre fonction avec l'application « ***fib_silly*** »

La version naïve n'est pas terrible car elle va appeler récursivement deux fois la fonction avant de pouvoir sommer et retourner le résultat, ceci résulte en un nombre de stack frames exponentiel.

c) Implémenter la fonction « fib_asm() » suivante, qui est un peu meilleure (ABI CDECL).

```
uint32_t fib(uin32_t n, uint32_t a = 0, uint32_t b = 1) {
```

[3] https://en.wikipedia.org/wiki/Fibonacci_number

```
        if (n == 0)
                return a;
        if (n == 1)
                return b;
        return fib(n-1, b, a + b);
}
```

d) Tester votre code à l'aide de l'application « ***fib*** ». Cela devrait aller bien plus vite déjà.

⇨ Attention, les variables ici sont passées par copie, alors il faut réserver de la place sur la pile (variables locales) pour les stocker. (Par exemple en C++ le passage par référence avec & permet de modifier directement la variable en place, alors que le passage par copie va travailler avec une copie locale).

Une fonction « tail recursive » est une fonction récursive dont la dernière action est de s'appeler elle-même. Par exemple la fonction ci-dessus est « tail recursive » et la fonction naïve ne l'est pas car elle doit faire un second appel et une addition après le premier appel à elle-même.

Si une fonction est « tail recursive » on peut réutiliser l'état courant de la pile (les variables locales) car elles ne seront plus utilisées après l'appel à soi-même et on n'a pas besoin de faire grandir la pile (ceci revient à faire une boucle). Ceci permet d'éviter d'exploser la pile !

e) Copier le code de votre fonction « fib_asm() » dans « fib_asm_tail() » et modifier le code afin de rendre la fonction « tail recursive ». Donc la transformer afin qu'elle utilise une boucle au lieu que la fonction s'appelle elle-même. Elle doit toujours être compatible avec l'ABI x86 CDECL.

f) Tester votre code à l'aide de l'application « ***fib_tail*** ». Cette version est la plus optimale en nombre d'instructions exécutées et en utilisation de la mémoire, car la taille de la pile est constante lors de la récursion (on a transformé une récursion en boucle).

Le compilateur est parfois capable d'appliquer ce genre d'optimisations lorsqu'on utilise « -Olevel » vous pouvez explorer ceci avec le « Compiler Explorer » par exemple. Lorsqu'on écrit des fonctions récursives en C il est toujours bien de contrôler le code assembleur résultant afin de voir si la pile grandit ou non.

Étape 3 – x86 – ABI – *Stack Overflow / Stack Smashing*

Dans cette étape il s'agira d'exploiter une application vulnérable afin d'exécuter du code arbitraire. (Cette étape se fait sur la machine hôte et **non** dans l'émulateur **QEMU**).

L'application « ***hello*** » définie par « ***x86_host/main.c*** » va simplement saluer l'utilisateur après lui avoir demandé son nom. Pour ce faire le développeur a alloué un tableau de 16 caractères sur la pile, étant donné qu'il n'y a pas de restrictions par rapport au nombre de caractères que l'utilisateur peut entrer on va pouvoir dépasser les limites du tableau et manipuler la pile !

L'application contient aussi une fonction secrète qui permet d'afficher un trésor ! Le but sera de manipuler la pile afin d'appeler cette fonction et d'obtenir le trésor.

a) Utilisez « ***objdump*** » afin de voir comment la fonction vulnérable « print_user_name() » a été compilée.

b) Faites un schéma de la pile avec EIP, EBP ainsi que le tableau (variable locale).

⇨ Ici, il est possible que le compilateur aie émis du code qui utilise la pile avant de réserver l'espace pour les variables locales (pour le tableau) prenez-le en compte.

On voit que si l'on continue à écrire au-delà du buffer on va écraser ce qui se trouvait sur la pile, par exemple le précédant « base pointer » ainsi que EIP (et plus encore si on continue…). Le but ici sera d'écraser EIP avec une valeur de notre choix.

a) Utilisez « ***objdump*** » afin de trouver l'adresse de la fonction « treasure() ».

b) Préparez le code dans le programme C « ***main_attacc.c*** » afin de générer la chaine de caractères qui permet d'obtenir le trésor (avec un « printf() » par exemple, ceci nous permettra d'émettre des valeurs qu'on ne peut pas saisir au clavier).

c) Utilisez le programme « ***attacc*** » pour attaquer le programme « ***hello*** » et obtenir le trésor !

```
$ ./attacc | ./hello # redirection de la sortie du programme attacc dans hello
```

Afin d'éviter ce genre d'attaques il est possible de compiler avec l'option « -fstack-protector ». Elle est activée par défaut sur beaucoup de systèmes. Avec cette option, le compilateur va ajouter un « canari » sur la pile avant de réserver de l'espace pour les variables locales. Ce « canari » est en général un nombre aléatoire et au retour de la fonction avant de restaurer l'ancienne pile et récupérer EIP, une fonction va vérifier que le canari est « vivant », donc qu'il avait la même valeur qu'au début. Si le canari est « mort », « smashé », donc que la valeur a changé, c'est qu'il y a eu un « stack overflow ». A ce moment-là une exception est levée et on arrête l'exécution du programme car il est compromis.

d) Essayez votre attaque sur le programme « ***hello_protecc*** » et observez le résultat.

⇨ Vous pouvez regarder la différence entre « ***hello*** » et « ***hello_protecc*** » avec « ***objdump*** ».

Une autre mitigation des attaques qui permettent de compromettre EIP est de charger le programme a une adresse aléatoire, ceci rend plus difficile la génération d'une valeur utile pour écraser EIP, par

exemple dans notre cas, si la fonction « treasure() » était chaque fois à un endroit différent l'attaque serait bien plus difficile.