Cours Système n°9 : l'assembleur

Éric Gaudefroy

École Hexagone
Cursus Cyberdéfense - M1

1. Avertissement

Ce cours n'est pas un cours de programmation en assembleur. Il vise à fournir les éléments permettant de comprendre comment s'effectuent à bas niveau (instructions CPU) un certain nombre d'opérations usuelles. A ce titre, il fournit les prérequis à l'étude de certaines attaques (buffer overflow) et des moyens de s'en prémunir. Il ouvre également des pistes de réflexion concernant l'optimisation des programmes ou encore l'analyse de code binaire non maîtrisé.

Le cours est axé principalement sur l'architecture Intel 32 bits (IA32, i386), tout comme le cours sur la mémoire. On retiendra cependant que l'essentiel des concepts abordés ici restent valables, à quelques détails près, sur d'autres architectures. Les divergences essentielles entre architectures seront par ailleurs évoquées en remarques.

2. Assembleur Intel 32 bits

2.1. Principales spécificités

La caractéristique la plus notable des processeurs Intel 32 bits est qu'il s'agit de processeurs de type CISC (Complex Instruction Set Computer), par opposition à l'architecture RISC (Reduced Instruction Set Computer) employée par essentiellement toutes les autres familles modernes de processeurs (PowerPC, ARM, MIPS, SPARC ...). La principale différence entre ces deux principes de conception des processeurs tient à la complexité des instructions : une unique instruction CISC peut réaliser des opérations très complexes, et travaille souvent directement sur des données en mémoire, le cas échéant en combinant des accès à la mémoire en lecture et en écriture dans une même instruction. A contrario, les instructions RISC se veulent simples (les opérations complexes nécessitent en général plusieurs instructions), et travaillent essentiellement sur les registres. Dans une architecture « purement » RISC, les seules instructions réalisant un accès à la mémoire sont les instructions dédiées de lecture (*load*) et d'écriture (*store*), et pas les instructions arithmétiques par exemple.

Ce dernier principe n'est cependant pas systématiquement appliqué. De cette différence fondamentale découlent d'autres divergences notables entre CISC et RISC :

- la taille des instructions : les instructions d'un processeur RISC (code de l'instruction et opérandes) occupent en général toutes une taille constante, propre à l'architecture (typiquement 32 ou 16 bits), tandis que les instructions CISC ont des tailles variables (de 1 à 7 octets

pour les instructions courantes i386, mais des instructions plus longues sont théoriquement possibles) ;

- le nombre de registres génériques : les architectures RISC disposent typiquement d'un grand nombre de registres génériques, où sont stockées temporairement les opérandes des instructions, alors que les architectures CISC utilisent typiquement beaucoup moins de registres, la plupart des opérations travaillant au moins pour partie directement en mémoire.

L'approche RISC a progressivement remplacé CISC pour la conception des processeurs, car elle permet des architectures électroniques plus simples et plus homogènes, qui se prêtent beaucoup mieux à la mise en œuvre de fréquences d'horloge élevées. Le maintien d'une approche CISC dans les processeurs i386 est une conséquence de l'ancienneté de la famille i386, et de la volonté affirmée de maintenir la compatibilité du jeu d'instruction à travers les générations successives de processeurs de la famille. On notera cependant que les processeurs i386 récents ne sont plus CISC que « vus de l'extérieur ». Les instructions exécutées par le processeur sont en fait décodées par une unité de traitement spécifique, qui les traduit en instructions plus simples, lesquelles sont ensuite exécutées par un cœur RISC qui seul tourne à la fréquence d'horloge maximale.

Par ailleurs, comme déjà abordé dans le cours sur la mémoire, les processeurs i386 incorporent un mécanisme de segmentation de la mémoire particulièrement riche, qui n'a généralement pas d'équivalent sur les autres architectures processeur. Ce mécanisme peut être utilisé conjointement à celui de pagination, qui est quant à lui plus généralisé parmi les autres architectures de processeurs. Il est en enfin rappelé que l'architecture i386 utilise la convention « Little Endian » pour la représentation des entiers en mémoire et dans les registres, à la différence de nombreuses autres architectures qui adoptent la convention « Big Endian » (*cf.* cours sur les sockets).

2.2. Registres CPU

Les principaux registres des processeurs Intel i386 (32 bits) et compatibles sont :

- Les huit registres dits génériques, utilisables de manière générale pour le calcul arithmétique ou le pointage mémoire, qui sont par ailleurs chacun associé à un usage particulier. Ces registres sont tous sur 32 bits (« E » = extended). Il existe également des versions 16 bits de ces registres (AX, BX, etc.), qui utilisent les 16 bits de poids faible des registres 32 bits correspondants. Les quatre registres AX, BX, CX et DX peuvent encore être subdivisés en sous-registres sur 8 bits (AX = AL (poids faible), AH (poids fort)). Les registres génériques 32 bits et leurs usages spécifiques sont les suivants :
  - EAX (accumulator) : accumulateur pour certains calculs, passage de paramètres et récupération de résultats ;
  - EBX (base) : pointeur pour des données classiques ;
  - ECX (counter) : compteur de boucle (dans les opérations itératives sur les chaînes) ;
  - EDX (data) : pointeur d'entrée/sortie ;
  - ESI (source index) : pointeur pour des données classiques (source d'opérations sur les chaînes) ;
  - EDI (destination index) : pointeur pour des données dans un segment alternatif (destination d'opérations sur les chaînes) ;
  - EBP (base pointer) : pointeur sur la base relative de la pile (adresse du premier octet suivant les variables locales de la fonction courante) ;

  - ESP (stack pointer) : pointeur sur le sommet de la pile (adresse de l'octet de la pile ayant l'adresse la plus basse).

  On notera bien que les usages particuliers décrits ici ne s'appliquent qu'à certaines instructions spécifiques, et n'interdisent pas en général (exception faite de ESP, *cf.* nota plus bas) l'utilisation de ces mêmes registres comme opérandes d'instructions génériques.

- les registres de segmentation, sur 16 bits, qui stockent les références (segment selectors, relatifs à la GDT ou à la LDT) des segments associés au contexte d'exécution courant (cf. cours sur la mémoire) :
  - CS (code segment) : référence au segment de code courant (et définition du CPL en mode protégé) ;
  - DS (data segment) : référence au segment par défaut pour les données (y.c. tas) ;
  - ES (extended segment) : référence au segment par défaut pour certaines opérations de copies (copie de chaîne, instructions movs et stos) ;
  - FS, GS (extended segments) : références explicites à d'autres segments de données ;
  - SS (stack segment) : référence au segment de la pile courante (c'est-à-dire le segment au regard duquel le contenu de ESP est interprété par certaines instructions comme push ou pop, et lors des appels de fonctions).
- le registre d'état EFLAGS (32 bits), qui rassemble plusieurs bits décrivant différents éléments de l'état du CPU (manipulables individuellement et au cas par cas par certaines instructions assembleur, par ex. saut conditionnel), notamment :
  - des éléments système : le niveau de permissions d'entrées/sorties de la tâche courante (IOPL), l'éventuelle inhibition des interruptions, etc... ;
  - des éléments utilisateur : les retenues, les résultats de comparaisons, etc...
- compteur ordinal EIP (instruction pointer, sur 32 bits), qui contient l'adresse de la prochaine instruction à exécuter (relativement au segment référencé par CS).

Il existe également de nombreux autres registres d'utilisation plus spécialisée, notamment :

- les registres de calcul sur les flottants ST(<n>) ;
- les registres de contrôle CR<n> pour paramétrer la pagination et le cache (exemple : CR3 contient l'adresse physique du répertoire de tables de pages) ;
- les registres GDTR et IDTR définissant la localisation des tables GDT et IDT ;
- les registres LDTR et TR référençant les "segments" définissant la LDT et le descripteur de tâche (le Task State Segment, qui détermine notamment le bitmap de permissions d'entrées/-sorties et la localisation des piles associées aux rings privilégiés pour la tâche courante) ;
- les registres associées aux unités de calcul vectoriel : MM<n> (extension MMX) ou XMM<n> (extensions SSE) ;
- etc...

Naturellement, outre ces différents registres adressables par des instructions assembleur, le CPU utilise un nombre important de registres internes, qui ne sont pas directement utilisables à travers une instruction. Il n'est d'ailleurs pas rare, sur les processeurs modernes, qu'un registre logique comme EAX corresponde en fait à plusieurs registres physiques, utilisés de manière transparente par le matériel pour paralléliser et réordonnancer les opérations (techniques de register renaming et de réordonnancement - out of order execution) à des fins d'optimisation.

Nota : Bien qu'il soit classiquement listé parmi les registres génériques, le registre ESP ne doit en aucun cas être utilisé pour autre chose que la gestion de la pile. Toute autre utilisation risque d'interférer avec le fonctionnement de la pile, surtout dans le cas d'une interruption du flot d'exécution (voir remarque ci-dessous) : il n'est pas possible de définir a priori un passage du code où ESP puisse contenir sans risque autre chose que l'adresse du sommet de la pile. De même, le registre EBP doit en général être réservé à la gestion de la pile, sauf le cas échéant dans un programme entièrement écrit en assembleur, où compilé avec l'option gcc -fomit-frame-pointer (qui entraîne la génération par le compilateur de code n'utilisant pas EBP pour la gestion de la pile - ce qui réduit souvent le nombre d'instructions mais complique en général le debug). Les autres registres génériques peuvent être utilisés pour stocker ou calculer sur des données arbitraires de 32 bits ou moins : les rôles mentionnés plus haut ne sont effectifs que lors de l'emploi de certaines instructions CPU bien particulières.

Le mode 64 bits supporté par les processeurs x86 récents introduit en particulier les modifications suivantes :

- les registres génériques sont étendus à 64 bits et renommés RAX, RBX, ..., RSP ; leurs composantes 32, 16 et 8 bits restent par ailleurs accessibles (EAX, AX, AH et AL) ;
- le compteur ordinal passe également sur 64 bits (RIP), de même que les autres registres stockant des adresses mémoire (GDT, IDT, CR3, etc.) ;
- huit registres génériques supplémentaires deviennent accessibles : R8 à R15 ;
- comme déjà évoqué dans le cours sur la mémoire, si les descripteurs de segments sont encore présents, les adresses de base et limites des segments référencés par CS, DS, ES et SS sont en revanche ignorées ; les adresses de base (mais pas les limites) des segments référencés via FS et GS sont toujours utilisées, ce qui permet de s'appuyer sur ces sélecteurs de segments pour implémenter certains mécanismes (thread-local storage en couche utilisateur, per-cpu data en mode noyau).

2.3. Quelques instructions

La syntaxe adoptée dans toute la suite du document correspond aux conventions du désassembleur *objdump* et de l'assembleur GNU as, utilisé pour les inclusions de code assembleur dans les projets gcc. Ces conventions ne sont pas celles adoptées par les documentations Intel ou l'assembleur nasm (ordre des opérandes, nommage des registres). Les principales caractéristiques de cette syntaxe sont :

- les noms des registres sont précédés du caractère
- les opérandes multiples sont dans l'ordre source, destination ;
- les noms des instructions travaillant sur la mémoire sont adaptables pour préciser, en cas d'ambiguïté, si elles agissent sur des octets (b), des mots de 16 bits (w) ou des longs de 32 bits (l) ; ainsi, mov peut se décliner en mov, movb, movw, movl.

Les opérandes (registres, mémoire) sont désignés de la façon suivante :

- $0x00000004 ou $0x4 : entier explicite égal à 4 ;
- %eax : contenu du registre EAX ;
- 0xbffffffc : contenu de la mémoire à l'adresse (logique) fixe 0xbffffffc (on parle de déplacement) ;
- (%eax) : contenu de la mémoire à l'adresse pointée par le registre EAX (on parle de base) ;

- 0x8(%ebp) : contenu de la mémoire huit octets après l'adresse pointée par EBP ;
- 0xfffffffc(%ebp) ou -4(%ebp) : contenu de la mémoire quatre octets avant l'adresse pointée par EBP ;
- (%eax, %ecx, 2) : contenu de la mémoire à l'adresse pointée par EAX avec un offset précisé dans ECX (on parle d'index), cet offset étant multiplié par le facteur d'échelle 2, c'est-à-dire à l'adresse donnée par %eax + 2 * %ecx ; les facteurs d'échelle possibles sont 1, 2, 4 et 8 ; une telle représentation est utile pour le parcours de tableaux d'entiers : $tab[i] = (tab, i, sizeof(tab[0]))$;
- 0xbfffffffc(, %ecx, 1) : contenu de la mémoire à l'adresse fixe 0xbfffffffc décalée d'un offset précisé dans ECX (à l'échelle 1) ;
- 0x00000003(%eax, %ecx, 4) : contenu de la mémoire trois octets après l'adresse pointée par EAX « offsettée » par ECX (à l'échelle 4) – utile pour le parcours de tableaux de structures de taille convenable : $tab[i].x = off\_x(tab, i, sizeof(tab[0]))$.

Les principales instructions d'accès à la mémoire et aux registres sont les suivantes :

- mov <val>,<cible> (move) : copie de contenus, par exemple :
  - mov $0x10,%eax : affecter la valeur 16 à EAX ;
  - mov 0xfffffffc(%ebp),%ebx : copier les 4 cases mémoire précédant l'adresse pointée par EBP dans EBX ; typiquement, il s'agit de charger la première variable locale si elle est de type int - cf. organisation de la pile décrite plus loin dans le document ;
  - movl $0x10,(%esp) : écrire 16 en mémoire, à l'adresse pointée par ESP (copie sur 32 bits) ;
- lea <addr>,<cible> (load effective address) : calcul et copie d'une adresse, par exemple :
  - lea 0xfffffffc(%ebp),%eax : copier l'adresse pointée par EBP dans EAX après avoir retiré 4 (= adresse de la première variable locale si elle est de type int ; ne pas confondre avec le contenu pointé, qui aurait été copié si on avait appelé mov au lieu de lea) ;
- push <val> : ajouter une valeur de type entier au sommet de la pile (effet de bord : décrémente ESP), par exemple :
  - push %eax : empiler la valeur contenue dans EAX (équivalent à réaliser atomiquement les deux instructions : sub $0x4, %esp puis mov %eax, (%esp)) ;
  - pushl $0x10 : empiler la valeur 16 (sur 32 bits) ;
  - pushl 0xfffffffc(%ebp) : empiler la valeur de la première variable locale (si elle est de type int).
- pop <cible> : retirer une valeur du sommet de la pile (effet de bord : incrémente ESP), par exemple :
  - pop %eax : dépiler un entier et le placer dans EAX (équivaut à réaliser atomiquement mov (%esp), %eax puis add $0x4, %esp).

Nota : Il est important de voir que la pile fonctionne à l'envers par rapport au tas : l'extension de la pile se fait vers les adresses basses.

Les principales instructions de calcul entier sont les suivantes :

- cmp <val1>,<val2> : comparaison entière entre deux valeurs (le résultat - égal, supérieur ou inférieur - est stocké dans les bits de EFLAGS prévus à cet effet, et est typiquement utilisé par une instruction de saut conditionnel, *cf.* ci-dessous) ;

- test <val1>,<val2> : test du ET bit à bit entre deux valeurs (essentiellement équivalent à un cmp <val1>&<val2>,$0), par exemple :
  - test $0x01,%eax : renvoie une « égalité » si le bit de poids faible est à 0 dans EAX et une « inégalité » s'il est à 1 ;
- inc <cible> : incrémentation de la cible (cible++) ;
- dec <cible> : décrémentation de la cible (cible–) ;
- add <val>,<cible> : somme entière (cible += val) ;
- sub <val>,<cible> : soustraction entière (cible -= val) ;
- xor <val>,<cible>, and <val>,<cible>, etc. : opérations bit à bit sur la cible.

Les principales instructions de saut (utilisées dans la compilation des sauts conditionnels, des boucles, etc...) sont, outre les appels de fonction (voir plus bas) :

- jmp <addr> : saut inconditionnel ;
- je <addr> : saut conditionnel si la dernière comparaison (par cmp ou test) a trouvé une égalité (==) ;
- jne <addr> : saut si la dernière comparaison a trouvé une inégalité (!=) ;
- jge <addr> (>=), jg <addr> (>), jle <addr> (<=), jl <addr> (<) : sauts sur comparaison d'entiers signés (lesser/greater) - par exemple :

  ```
  cmp %eax, %ebx
  jge 0x2a
  ```

  réalise un saut de 42 octets si et seulement si le contenu de EAX est supérieur ou égal à celui de EBX ;
- jae <addr> (>=), ja <addr> (>), jbe <addr> (<=), jb <addr> (<) : sauts sur comparaison d'entiers non signés (above/below).

Les sauts sont généralement relatifs : l'adresse passée dans l'instruction de saut est un offset à interpréter relativement à l'adresse de l'instruction suivant celle du saut (et non à l'adresse de l'instruction de saut elle-même : penser que EIP a déjà été incrémenté lorsque le saut est exécuté). Exception : jmp *%eax (saut à l'adresse absolue stockée dans EAX).

Les principales instructions pour les appels de fonction sont (voir aussi les sections sur les appels de fonction et les appels système) :

- call <addr> : appel d'une fonction, essentiellement équivalent à un saut précédé de la sauvegarde de l'adresse de l'instruction suivante sur la pile, de manière à pouvoire y retourner à l'issue de l'appel), par exemple :
  - call <addr> : appel d'une fonction située à une adresse donnée (typiquement, cette adresse n'est pas connue avant l'édition de liens) – comme pour un saut, l'adresse addr est un offset par rapport à l'adresse de l'instruction suivant le call). Cette instruction équivaudrait à réaliser de manière atomique les instructions push %eip puis jmp <addr> (la première instruction n'est pas supportée en réalité : on ne peut pas manipuler EIP directement).
  - call *%eax : appel de la fonction dont l'adresse absolue est stockée dans EAX (appel via un pointeur de fonction).

- ret : retour à la fonction appelante (à l'instruction suivant le call ayant réalisé l'appel), en dépilant l'adresse sauvegardée lors du call.
- lcall <seg>,<addr> (far call) : appel d'une fonction avec changement du segment de code – typiquement pour passer dans un ring plus privilégié (méthode obsolète pour réaliser un appel système), <seg> doit référencer une call gate plutôt qu'un descripteur de segment standard (l'adresse de la fonction appelée sera alors déterminée par la call gate et non par <addr>).
- lret : retour à la fonction appelante avec restauration du segment de code (et de la pile s'il y a changement de ring) ;
- int <val> : déclenchement d'une interruption logicielle, par exemple :
  - int $0x80 exécution d'un appel système Linux ou BSD (*cf.* plus bas).
- iret : retour au flot d'instructions principal après traitement d'une interruption (utilisé par le noyau) ;
- sysenter : appel d'une routine noyau prédéfinie en ring 0 (avec une pile prédéfinie) ;
- sysexit : passage en ring 3 (EIP chargé depuis EDX et ESP depuis ECX).

Enfin, l'instruction nop consiste à ne rien faire (à part augmenter EIP pour passer à l'instruction suivante, comme toutes les instructions hors appels et sauts).

3. Opérations de base en assembleur

Les descriptions fournies correspondent aux binaires produits sous Linux pour une architecture de type i386. Les principes de base s'étendent aux autres OS sur CPU Intel et, dans une large mesure, à d'autres architectures matérielles.

3.1. Appel de fonction (convention *cdecl*)

Un appel de fonction est, de manière conventionnelle, réalisé de la façon suivante :

- la fonction appelante place les arguments dans la pile, en commençant par le dernier :
  - soit par push ou pushl (empilement de valeurs),
  - soit par sub sur ESP puis movl (extension de la pile puis renseignement des valeurs) ; à l'issue de cette étape, les arguments sont placés ainsi, dans l'ordre des adresses mémoire croissantes : arg1, arg2, arg3, etc... (donc ESP pointe sur arg1) ;
- la fonction appelante fait un call à l'adresse de la fonction appelée, ce qui revient à faire un push de l'adresse de l'instruction suivante et à charger l'adresse de la fonction appelée dans EIP ;
- la fonction appelée prépare la pile :
  - soit explicitement en :
    - faisant un push de EBP pour sauver la base relative de la pile de la fonction appelante,
    - copiant ESP sur EBP pour initialiser son étage de la pile (qui est encore vide, puisque la base relative et le sommet sont confondus) – à l'issue, EBP pointe sur la sauvegarde de l'ancien EBP et les arguments de l'appel de fonction sont accessibles à partir de 0x8(%ebp),
    - allouant de l'espace pour ses propres variables locales non statiques (sub <size>,%esp) ;

  - soit implicitement, grâce à l'instruction enter <size>,$0 (qui fait exactement la même chose) ;
- la fonction appelée initialise si nécessaire ses variables locales (mov) – à l'issue de cette étape, les variables sont placées ainsi, dans l'ordre des adresses mémoire croissantes : ..., var3, var2, var1 (donc ESP pointe sur la dernière variable locale) ;
- la fonction appelée fait son travail (calculs, appels de sous-fonctions, etc.) ; si l'on travaille avec des variables locales et des arguments stockés sur 4 octets, les arguments sont accessibles en 0x8(%ebp), 0xc(%ebp), etc., et les variables locales en -0x4(%ebp), -0x8(%ebp), etc... ;
- la fonction appelée place son code de retour dans EAX ;
- la fonction appelée remet en état la pile pour la fonction appelante :
  - soit explicitement, en copiant EBP sur ESP (ce qui supprime son étage) puis en faisant pop sur EBP (ce qui restaure la valeur qu'elle avait sauvegardé dans la pile),
  - soit implicitement, grâce à l'instruction leave (qui fait exactement la même chose) ;
- la fonction appelée sort par ret, ce qui revient à faire un pop dans EIP (retour à l'instruction suivante dans la fonction appelante) ;
- la fonction appelante raccourcit sa pile (add sur ESP) pour libérer l'espace utilisé par les arguments de l'appel de fonction, et peut récupérer le code de retour de la fonction appelée dans EAX.

L'état de la pile pendant l'exécution d'une fonction est résumé dans la figure 1.

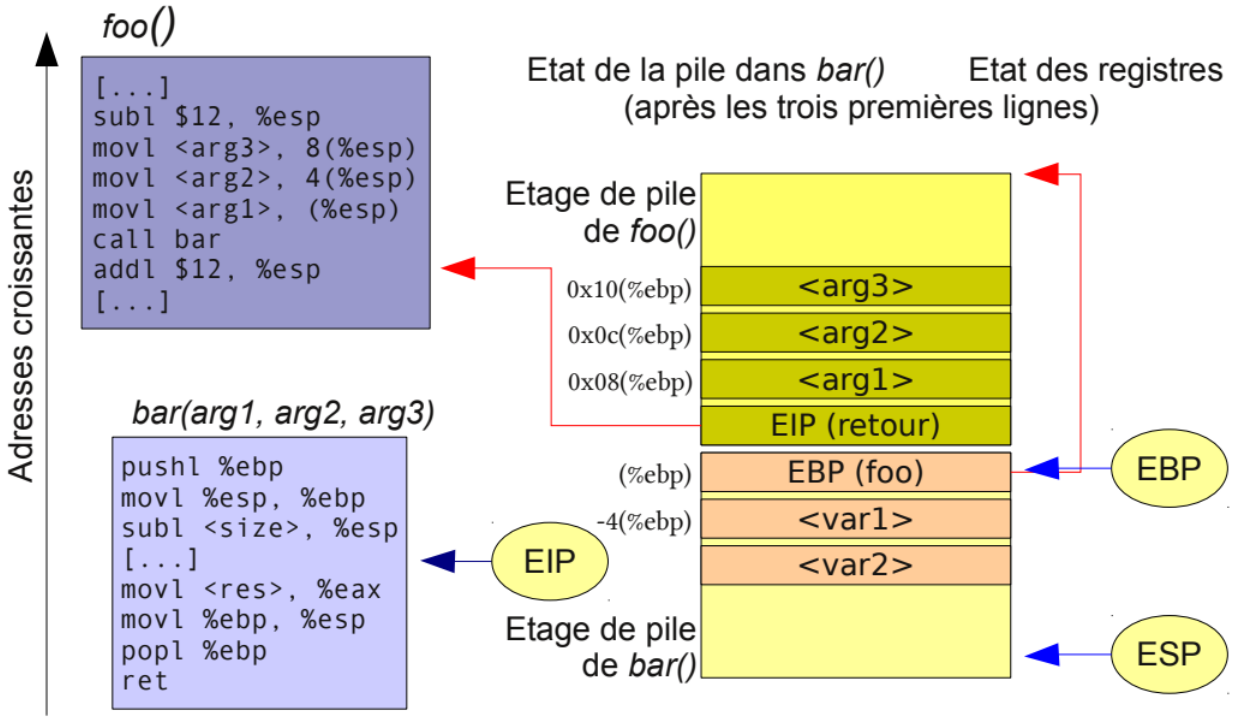

FIGURE 1 – Etat de la pile et de quelques registres après un appel de fonction.

Il faut voir que l'emboîtement des appels de fonctions fait qu'au cours de l'exécution d'un programme, la pile est en fait un empilement d'étages (stack frames) de ce type. L'état initial de la pile au lancement du programme sera illustré dans le cours sur les exécutables.

L'enchaînement d'opérations présenté ici correspond à un appel de fonction « classique ». Il peut en pratique être remplacé par diverses variantes, en fonction des options d'optimisation du compilateur, notamment :

- La gestion du pointeur de base EBP peut être omise (option gcc -fomit-frame-pointer), auquel cas il n'y aura pas de sauvegarde de EBP sur la pile, et toutes les variables locales seront référencées par rapport à ESP. Le compilateur est dans ce cas chargé de rétablir explicitement l'état de la pile par des add <size>, %esp avant de sortir de la fonction. Le principal intérêt de cette option est de rendre EBP disponible comme registre générique ; en contrepartie, sa mise en œuvre complique les opérations de débug (interprétation des différents étages de pile).
- Les premiers arguments de la fonction appelée peuvent être passés par des registres plutôt que par la pile (option gcc -mregparm=n, où n - inférieur ou égal à 3 - désigne le nombre maximum d'arguments à passer par la pile). Dans ce cas, les n premiers arguments d'une fonction appelée seront passés dans les registres EAX, EDX et ECX (dans cet ordre). Si la fonction appelée nécessite plus de n arguments, les derniers (à partir du (n+1)-ième) seront tout de même passés par la pile. L'optimisation attendue tient au fait qu'un accès aux registres est nettement moins long qu'un accès à la pile (mémoire). On notera incidemment que le passage d'arguments par registre est en général la convention d'appel par défaut sur la plupart des architectures RISC, qui bénéficient d'un plus grand nombre de registres que les processeurs x86. De même, la convention d'appel associé au mode 64 bits des processeurs Intel/AMD précise que les six premiers arguments entiers peuvent être passés par registres.

En particulier, le noyau Linux 2.6 est par défaut compilé avec les options -fomit-frame-pointer -mregparm=3 sur les architectures x86 32 bits. Des options du menu Kernel Hacking permettent, lors de la configuration du noyau, de rétablir la méthode d'appel de fonction classique.

3.2 Appel système

L'appel système traditionnel (interruption logicielle) sous Linux est réalisé de la façon suivante :

- on place les arguments de l'appel dans certains registres génériques ; le nombre exact d'arguments dépend de l'appel, et on utilise dans l'ordre les registres suivants : EBX, ECX, EDX, ESI et EDI (pour les appels à 6 arguments, on s'arrange pour utiliser ponctuellement EBP) ;
- on place le numéro de l'appel dans le registre EAX (ce numéro détermine s'il s'agit d'un read, d'un write, etc. – la liste des numéros est fournie dans /usr/include/asm/unistd.h) ;
- on lance l'interruption logicielle (trap) $0x80 (syscall).

On voit ainsi que les appels système ne peuvent travailler que sur des paramètres de type int ou assimilés (types stockés sur 32 bits ou moins et pointeurs).

La routine enregistrée par le noyau pour traiter l'interruption 0x80 est alors exécutée par le CPU. Elle récupère les arguments dans les registres, les copie dans sa propre pile, et appelle la fonction noyau correspondant au traitement de l'appel invoqué (choix de l'entrée correspondant à EAX dans le tableau des pointeurs de fonctions de traitement d'appels système, sys_call_table).
En fin d'exécution, la routine noyau place son code de retour – 0 ou une valeur positive en cas de succès, ou une valeur négative (-E<err>) en cas d'échec – dans le registre EAX, restaure les autres registres et repasse la main à la couche utilisateur.

Les autres approches possibles pour réaliser un appel système sur architecture i386, en particulier les instructions spécifiques sysenter et syscall (cette dernière étant plutôt réservée au long mode

64 bit), sont utilisées selon des principes similaires : inscription du numéro d'appel système dans le registre EAX, et des arguments dans les autres registres. Contrairement à un appel de fonction classique, la pile n'est jamais utilisée pour passer des arguments lors d'un appel système, ce qui est logique dans la mesure où le traitement de l'appel par le noyau est réalisé avec une pile différente de celle utilisée par l'appelant. Voir aussi le cours sur la mémoire pour la manière dont le noyau Linux offre une abstraction commune de ces différentes méthodes d'appel système via un *Virtual Dynamic Shared Object* (VDSO).

Gestion de errno : Les appels système sont en temps normal invoqués au travers de fonctions de la librairie standard plutôt que directement par le programme utilisateur. Au retour, la librairie standard récupère le code de retour (valeur de EAX) et détermine s'il s'agit d'un succès ou d'un échec :

- s'il s'agit d'un succès, il est renvoyé comme code de retour de la fonction de bas niveau encapsulant l'appel (read, write, etc...) ;
- s'il s'agit d'une erreur, sa valeur absolue est copiée dans errno (située à une adresse donnée dans l'espace mémoire du processus) et la fonction de bas niveau renvoie -1. Dans une application multithreads, la librairie standard peut gérer une variable errno par thread.

3.3. Application : principe des attaques par buffer overflow

Les attaques par buffer overflow contre un programme exploitent le plus souvent des vulnérabilités issues de passages de paramètres par adresse sans contrôle de la taille de la zone mémoire pointée ou avec un contrôle défectueux. L'exemple typique est l'utilisation de l'instruction *strcpy* pour copier une chaîne de caractères non maîtrisée sur un buffer de taille fixe : si un attaquant parvient à faire passer en argument d'une telle fonction une chaîne plus longue que le buffer réservé pour la copie, *strcpy* débordera sur la mémoire située au delà de la zone allouée au buffer, donc modifiera la valeur des données situées en mémoire derrière ce buffer.

Les possibilités d'exploitation malveillante d'un buffer overflow dépendent de la zone mémoire dans laquelle il se produit :

- dans les données ou le tas : l'attaquant peut modifier la valeur des variables situées après le tableau (ou équivalent) dans lequel se produit le débordement ; les effets d'une telle attaque dépendent de la nature des variables ainsi accessibles, par exemple (liste évidemment non exhaustive) :
  - remplacement du numéro d'un descripteur de fichier : simule une redirection, par exemple pour faire écrire un contenu sensible dans un fichier sous contrôle de l'attaquant, ou au contraire faire écrire un contenu contrôlé par l'attaquant dans un fichier sensible (/etc/shadow...) ;
  - modification d'un profil client dans une application multi-utilisateurs, en vue usurper des privilèges applicatifs ou de contourner une étape d'authentification ;
  - écrasement d'un pointeur de fonction : prépare un détournement du flot d'instructions (le pointeur de fonction peut appartenir à une structure allouée dynamiquement dans le tas, ou faire partie d'une structure globale des données, par exemple la PLT qui sera abordée dans le cours sur les exécutables) ;
  - écrasement des métadonnées associées à la gestion du tas, si cela est rendu possible par le positionnement de ces métadonnées : permet potentiellement une attaque en deux temps,

conduisant à une écriture arbitraire en mémoire, pas nécessairement limitée au tas (cf. remarque plus bas) ;

- dans la pile : l'attaquant peut en général 9 modifier la valeur des variables situées plus haut (c'est-à-dire « avant » dans la pile) que celle dans laquelle se produit le débordement ; ceci permet notamment :
  - de réaliser des opérations similaires à celles réalisables dans les données (modification de certaines valeurs), mais cette fois sur des variables locales ;
  - de modifier à la volée les arguments d'appel de la fonction courante (qui sont situés plus haut en mémoire que les variables locales) ;
  - de modifier la valeur de l'adresse de retour de la fonction courante pour provoquer l'exécution d'un code alternatif (code existant à l'intérieur du programme ou de ses bibliothèques ; code injecté par l'attaquant dans la pile, les données ou le tas) lors du retour de la fonction.

Nota : Il n'y a pas de buffer overflow dans la zone de code puisqu'on n'y accède jamais en écriture. En revanche, certains types de programmes sont susceptibles de créer des projections mémoire (via *mmap*) accessibles à la fois en écriture et exécution, pour y exécuter du code généré dynamiquement. Les interpréteurs de langages de haut niveau - java, flash, C#, etc. - qui mettent en œuvre des techniques de compilation Just In Time sont un exemple typique d'un tel comportement. A la différence de la zone de code, ces projections sont susceptibles d'être corrompues par un buffer overflow (interne à la projection ou éventuellement depuis une autre projection contigue en mémoire). Un buffer overflow permettant une telle corruption constitue une vulnérabilité particulièrement critique, dans la mesure où il permet à un attaquant de faire quasi-trivialement exécuter du code arbitraire, et ce quelques soient les protections mémoires mises en œuvre par ailleurs.

Pour cette raison, les vulnérabilités de type buffer overflow conduisant à une exécution de code arbitraire s'exploitent traditionnellement comme suit :

- on réalise un buffer overflow dans la pile ;
- la chaîne copiée dans la pile est constituée de :
  - une tête quelconque permettant de s'aligner sur la sauvegarde de l'adresse de retour (écrasement de variables locales et de la sauvegarde de EBP) ;
  - l'adresse (absolue) du mot suivant (écrasement de la sauvegarde de EIP avec l'adresse du début du code arbitraire) ;
  - du code arbitraire en langage machine (ce code pourra par exemple réaliser un exec(« /bin/sh »), ce qui fournit à l'attaquant un shell interactif avec les privilèges de l'application compromise – on parle souvent de shellcode pour désigner le code d'exploitation, même lorsque ce dernier ne lance pas un shell).

Une fois que le buffer overflow a eu lieu, la fonction courante, dont le code est intact, continue son exécution normale (ou anormale : certaines variables locales et les arguments de l'appel sont corrompus) jusqu'au ret, censé repasser la main à la fonction appelante. L'adresse de retour ayant été modifiée, c'est en fait le shellcode qui est alors exécuté.

Le principe d'une telle exploitation est résumé dans la figure 2, où l'on suppose qu'une fonction bar, elle-même appelée par foo, copie un argument contrôlable par l'attaquant, sans vérification de taille, dans un tableau de taille fixe buf alloué sur la pile, par exemple à l'aide de *strcpy*. On

notera cependant que l'exploitation d'un buffer overflow est désormais rarement aussi simple, vu la nécessité de contourner différents mécanismes de protection qui peuvent être en place : voir aussi la discussion de quelques variantes ci-dessous.

Les deux inconnues à déterminer au cas par cas (dépendent de l'application, du compilateur qui a généré l'exécutable, des options d'optimisation à la compilation et de l'environnement d'exécution) sont l'adresse absolue où est stockée l'adresse de retour (dont on déduit l'adresse du code injecté) et sa position relative par rapport au buffer écrasé (dont on déduit la taille que doit faire la tête).
En cas d'incertitude sur l'adresse absolue, il est possible d'inclure une série de nop devant le code utile. De cette façon, il n'est plus indispensable de faire pointer la nouvelle adresse de retour précisément au début du code injecté : il suffit de sauter quelque part dans la série de nop pour finalement exécuter le code d'exploitation.
Une telle approche est généralement dénommée NOP sled, NOP slide ou NOP ramp. Elle est d'autant plus efficace que l'attaquant est capable d'écrire une quantité de données arbitraire - une « rampe » de plusieurs milliers de nop permet ainsi de lever l'essentiel de l'incertitude sur la position du code d'attaque en mémoire, même en présence de randomisation.
De même, en cas d'incertitude sur la position relative, on peut répéter plusieurs fois la nouvelle adresse de retour après la tête pour s'assurer qu'une de ces copies recouvre bien l'ancienne adresse de retour.

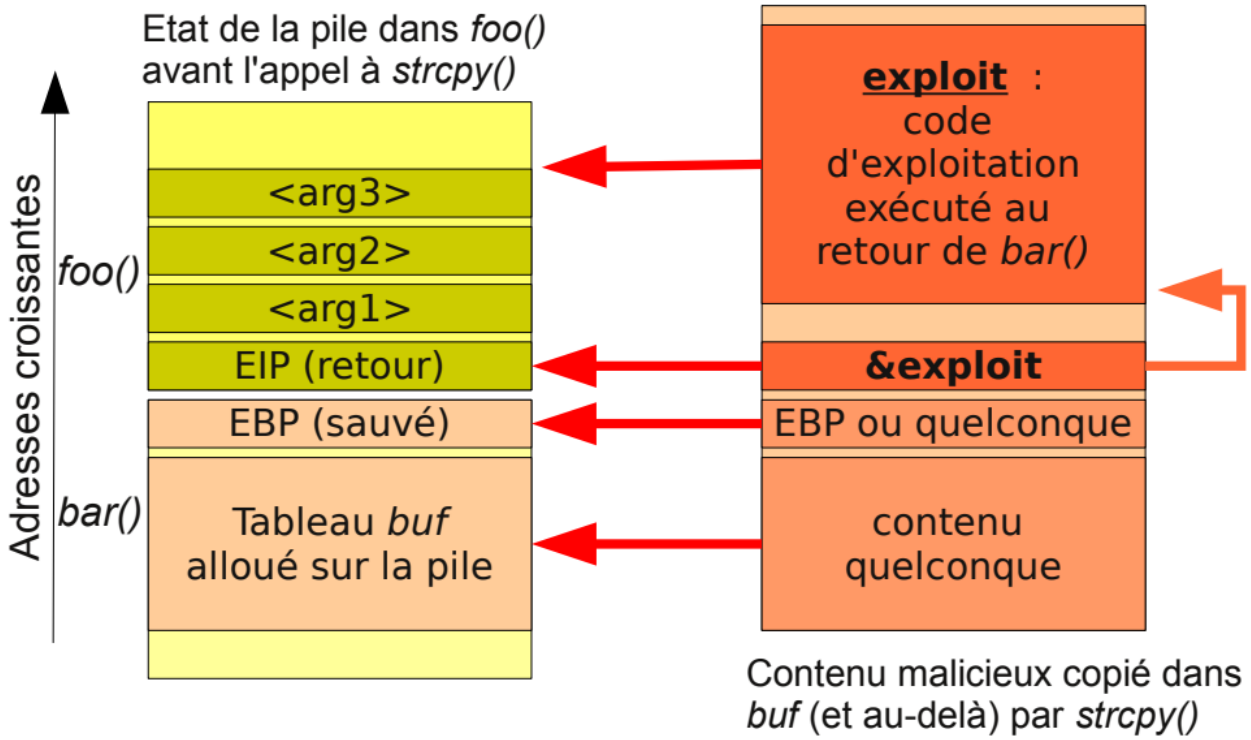

FIGURE 2 – Principe d'exploitation basique d'un buffer overflow dans la pile.

Généralement, il faut aussi s'arranger pour que la chaîne ainsi construite ne contienne pas de caractère nul (pour s'assurer qu'une fonction comme *strcpy* la copie intégralement), ce qui nécessite quelques astuces lors de l'écriture du code à injecter (remplacer le chargement d'un zéro par le calcul du xor d'un entier avec lui-même, etc.). Enfin, il est évident qu'une telle attaque simple ne

fonctionnera pas si la pile correspond à une projection mémoire non exécutable (et que le matériel contrôle effectivement cette propriété avant de permettre l'exécution).

Variantes : Selon les mesures de protection mises en œuvre (pile non exécutable, etc.), différentes variantes de ce type d'attaque peuvent être envisagées :

- **Injecter d'abord le code dans les données ou le tas** (ne nécessite pas d'exploiter une vulnérabilité) puis écraser la pile pour que l'adresse de retour pointe vers le code injecté (utile par exemple lorsque le buffer overflow exploité est limité en taille, et ne permet pas de copier l'intégralité du code d'exploitation, ou lorsque la pile est non exécutable, mais pas le tas).
- **Ecraser la pile** pour que l'adresse de retour pointe vers un code existant mais qui n'est pas supposé être exécuté dans ce contexte (par exemple, pour invoquer la fonction system de la libc – on parle de return-to-libc – ou pour court-circuiter une fonction interne d'authentification). Cette approche peut permettre à un attaquant de contourner des permissions mémoire très restrictives, interdisant tout exécution des zones de mémoire accessibles en écriture. La randomisation des positions des différentes projections mémoire (tas, pile, bibliothèques, mais aussi adresse de base de l'exécutable) complexifie sensiblement ce type d'attaque, et constitue donc un complément indispensable à la mise en œuvre de permissions mémoire restrictives.
- **Ecraser une variable de type pointeur de fonction** (adresse d'une routine en mode callback, référence à une « méthode » en programmation orientée objet, etc.) pour renvoyer vers du code injecté ou vers une routine existante et attendre que le programme fasse un appel de fonction à travers ce pointeur – le pointeur de fonction peut indifféremment être stocké dans les données, le tas ou la pile. Cette méthode est également utilisée pour l'exploitation de buffer overflow dans le tas, *cf.* remarque ci-dessous.

Nota : La protection contre l'exploitation de vulnérabilités de type buffer overflow repose principalement sur deux axes d'amélioration de la sécurité : le développement de code sans vulnérabilité de ce type, et la mise en œuvre d'un environnement d'exécution qui interdise, ou du moins complexifie fortement, l'exploitation de telles vulnérabilités. Un code dépourvu de toute vulnérabilité est un idéal qui n'est en pratique qu'exceptionnellement atteint. On peut cependant chercher à s'en rapprocher, notamment par l'utilisation exclusive de fonctions dites « sures », par exemple *strncpy* plutôt que *strcpy* a, et par un audit systématique et approfondi du code. On pourra également s'appuyer sur des méthodes de développement formelles ou semi-formelles, ou sur des langages de programmation fortement typés (java, ada, etc.) qui réduisent les erreurs possibles.
De manière complémentaire à la sécurité du développement, l'environnement d'exécution peut limiter fortement les possibilités d'exploitation d'une erreur de programmation. La mise en œuvre de permissions mémoire restrictives, et la randomisation des adresses de projection, peuvent, comme cela a été évoqué plus haut, complexifier fortement la tâche d'un attaquant qui chercherait à détourner le flot d'exécution à partir d'une vulnérabilité. Par ailleurs, des méthodes peuvent être mises en œuvre pour détecter le débordement d'un buffer et forcer dans ce cas la terminaison immédiate du processus, avant toute exploitation. Les options de compilation **-fstack-protector** / **-fstack-protector-all** pour gcc (et leur équivalent /GS pour les compilateurs Microsoft) permettent de positionner des « canaris » au sommet de chaque étage de pile, dont l'écrasement sera détecté avant le retour de la fonction concernée (et donc avant l'utilisation des sauvegardes corrompues de EIP et EBP). Des gestions durcies du tas (par exemple celle d'OpenBSD), ou des bibliothèques spécifiques comme libmudflap, permettent d'obtenir des garanties similaires dans le tas, au prix d'un impact plus ou moins marqué sur les performances.

Enfin, il ne faut pas négliger l'importance de la détection d'attaques manquées. Un attaquant qui dispose d'un nombre illimité de tentatives pour exploiter une vulnérabilité finira toujours par parvenir à ses fins. En revanche, la multiplication des terminaisons anormales d'un service réseau par exemple, si elle est correctement détectée, peut permettre à l'administrateur de prendre des mesures de correction de la vulnérabilité que l'attaquant cherche à exploiter

4. Analyse d'un programme

Il existe plusieurs outils efficaces permettant d'analyser un programme à partir du fichier exécutable ou lors de son exécution.

4.1. Analyse statique

L'analyse statique s'effectue principalement à l'aide d'outils tels que *objdump* et *readelf*. La commande *objdump -d <fichier>* permet ainsi de désassembler un fichier exécutable, c'est à dire de visualiser en assembleur le code exécutable qu'il embarque. Ces outils, ainsi que la structure interne des fichiers exécutables Linux (format ELF) sont présentés plus en détail dans le cours sur les exécutables et la compilation.

4.2. Analyse dynamique

L'utilitaire strace permet sous Linux de tracer les appels système lancés par un processus ainsi que les signaux qu'il reçoit. La syntaxe est la suivante :

- strace <options> <commande> : lancer une commande en traçant ses appels système ;
- strace <options> -p <pid> : tracer les appels système d'un processus existant.

avec notamment les options :

- -e trace=<appel1>,<appel2>,... : pour tracer certains appels seulement (ou catégories d'appels : file, network, process, etc.) ;
- -e signal=... : pour choisir les signaux à tracer ;
- -f -F : pour tracer aussi les fils générés par le processus ;
- -v -s <n> : pour éviter de tronquer l'affichage (n = taille maximale pour l'affichage des chaînes) ;
- *cf.* man strace pour davantage d'options

Parmi les autres outils intéressants, on peut citer plus particulièrement :

- **gdb**, qui permet aussi de s'attacher à un processus existant en vue de le contrôler ou de l'observer (notamment à travers le gel du processus et l'accès arbitraire en lecture et écriture à sa mémoire et à ses registres) : gdb <prog> <pid> ;
- **ltrace**, qui permet de tracer les appels aux fonctions de bibliothèques dynamiques (syntaxe voisine de celle de strace) ;
- voir aussi systrace (issu de BSD), ktrace/kdump (BSD), truss (Solaris), etc.

Contrôle d'accès La plupart des outils d'analyse dynamique du comportement d'un programme, comme *strace* ou *gdb* (mais pas *ltrace*, qui repose uniquement sur une instrumentation de l'édition

de liens dynamique), s'appuient sur l'appel système ptrace. Celui-ci permet à un processus de « s'attacher » à un autre processus et d'en lire et modifier la mémoire, ce qui équivaut fondamentalement à prendre le contrôle total de cet autre processus.
Cet appel est naturellement soumis à un contrôle d'accès visant à interdire les escalades de privilèges. Plus concrètement, un processus P est autorisé à s'attacher via *ptrace* à un autre processus T si et seulement si l'une des deux conditions suivantes est vérifiée :

- l'uid réel de P est égal aux uid effectif, réel et sauvegardé de T, et son gid réel est égal aux gid effectif, réel et sauvegardé de T ;
- P dispose de la capacité CAP_SYS_PTRACE, ce qui est notamment le cas lorsque P est un processus root.

Par ailleurs, lors de l'exécution d'un fichier disposant d'un bit *setuid*, le noyau positionne automatiquement un attribut dumpable sur la tâche qui réalise l'exécution. La présence de cet attribut restreint la possibilité de s'attacher à cette tâche via ptrace aux seules tâches qui disposent de la capacité CAP_SYS_PTRACE.

Ce document est inspiré du « Cours Système n°9 : Assembleur » délivré par monsieur Vincent Strubel dans les cours dispensés à l'Agence nationale de la sécurité des systèmes d'information (Février 2013).