Retour d'expérience sur l'enseignement des attaques par débordement de *buffer* à CentraleSupélec.

Frédéric Tronel et Guillaume Hiet
CentraleSupélec, Inria, CNRS, IRISA
F-35000 Rennes

I. Introduction

Dans ce papier nous présentons un retour d'expérience d'une dizaine d'années sur l'enseignement des techniques d'attaques par débordement de tampon aussi communément appelée *buffer overflow*. Il nous a semblé utile de partager notre expérience avec la communauté francophone de l'enseignement supérieur en cybersécurité.

II. Bref historique

Ce cours a été créé lors de la réforme des spécialités de troisième année du cursus d'ingénieur Supélec initié en 2011 qui a vu le découpage de cette dernière année du cursus en différentes majeures (spécialités) dotées chacune d'un tronc commun complété par des cours de mineures laissés aux choix des étudiants. À cette occasion, pour la majeure intitulée Systèmes d'Information Sécurisés il nous a semblé utile de proposer d'une part un cours de tronc commun concernant la connaissance de la menace et plus particulièrement les attaques visant des codes binaires développés dans des langages dont la sécurité des manipulations de mémoire n'est pas garantie (comme par exemple les langages C et C++) précédé d'un cours sur l'assembleur x86 et d'autre part un certain nombre de cours connexes proposés sous forme de mineure, et notamment un cours sur le fonctionnement des systèmes d'exploitation.

III. Les notions abordées

A. Cours d'assembleur

Durant le cours de rappel sur l'assembleur x86, nous abordons l'organisation des registres du processeur en insistant sur la compatibilité arrière assurée par l'inclusion des registres apparus dans les gammes de processeurs qui ont précédé le 80386. Nous nous limitons (pour le moment) à une explication détaillée de l'architecture 32 bits, même si nous mentionnons rapidement les nouveaux registres introduits par le passage à une architecture 64 bits. Nous détaillons ensuite les différents modes d'adressage offerts par le processeur (registre, immédiat, indirect (avec ou sans décalage)). Puis nous passons en revue une petite partie du jeu d'instruction, en nous limitant aux instructions les plus couramment rencontrées lorsque l'on désassemble du code généré par un compilateur (ici GCC) :

- Instructions arithmétiques et logiques ;
- Instruction de transfert depuis/vers la mémoire : `mov`.
- Instruction de calcul d'adresse : `lea`.
- Instruction de comparaison (`cmp`) en expliquant l'effet sur le registre d'état.
- Instructions de branchement conditionnel en lien avec le registre d'état (`jXX`).
- Instruction de saut inconditionnel (`jmp`).
- Instruction de manipulation de la pile (`push` et `pop`)
- Instruction d'appel et de retour de fonction (`call` et `ret`), en lien avec la pile.
- Instruction permettant de déclencher un appel système (`int` et/ou `sysenter`).

B. Cours de fonctionnement des systèmes d'exploitation

Dans ce cours proposé sous forme de mineure (donc optionnel, mais fortement recommandé), nous détaillons le fonctionnement d'un système d'exploitation moderne (Linux, Windows NT ou plus récent, Mac OS X) fonctionnant avec un processeur de type x86. Ce cours permet entre autres aux étudiants d'acquérir des connaissances sur le fonctionnement de la mémoire virtuelle, de la pagination et de la segmentation, des interruptions, des exceptions, de l'implémentation des appels systèmes. Tous ces points techniques permettent d'aborder le cours sur les vulnérabilités des programmes binaires dans de bonnes conditions.

C. Cours sur les attaques par débordement de buffer

Ce cours commence par un certain nombre d'expérimentations permettant aux étudiants de constater que lors d'un débordement de buffer, le programme va être généralement arrêté brutalement par le système d'exploitation via l'envoi d'un signal (par exemple `SIGSEGV`). Ceci permet de faire le lien avec les notions vues dans le cours de système d'exploitation sur la protection mémoire apportée par la MMU et programmée via la pagination et la segmentation. Puis on passe à une expérience où le flot de contrôle est modifié par une fonction *magic* (via des manipulations de pointeurs) afin de sauter l'exécution d'une instruction située juste après l'appel à cette même fonction. Ceci permet d'illustrer le fait que la modification d'une adresse de retour peut produire des effets intéressants sans causer d'arrêt brutal du processus. On expose ensuite le fonctionnement d'une attaque par débordement de *buffer* simple avec retour direct dans la pile (que l'on suppose donc exécutable). On conçoit alors une charge d'attaque du type *shellcode* semblable à celle proposée par Aleph One [1] dans son fameux article [4]. Ceci permet de réviser notamment la gestion des appels systèmes (via une interruption

1. Elias Levy.

logicielle) et de rappeler la notion de convention d'appels et d'*Application Binary Interface* (ABI). Puis des mécanismes de protection classiques sont rappelés :

1) protections via la chaîne de compilation : canari, code indépendant de la position ;
2) protections apportées (principalement) par le système d'exploitation : randomisation de l'espace d'adressage et support du bit NX ;
3) protections apportées au support d'exécution (chargeur dynamique notamment) : politique W^X.

Puis les principales techniques de coutournement utilisées par les attaquants sont à leur tour détaillées. Nous commençons par expliquer les attaques du type retour vers la librairie C (*return to libc*) inventées par Solar Designer[2] [1]. Puis nous expliquons les attaques en *Return Oriented Programming* (ROP ci-après) dérivées des précédentes et introduites pour la première par Sébastian Kramer dans [3].

IV. Travaux pratiques

A. Le scénario

Le travail de laboratoire (travaux pratiques dans le jargon de SUPÉLEC) associé au cours sur la connaissance de la menace est scénarisé autour d'une prestation commanditée par la société fictive (PRESSOARE) à une société d'audit en sécurité après que les services informatique de la société PRESSOARE ont détecté un trafic anormal sur l'un de leur serveur et soupçonne une tentative d'intrusion sur celui-ci. Une trace réseau suspicieuse est fournie aux étudiants. Les élèves par binôme (ou éventuellement trinôme) jouent le rôle du prestataire. Notons ici que le travail demandé relève plus de la réponse à incident que de l'audit de sécurité.

B. Aspects fonctionnels du serveur attaqué

Le serveur ayant subi une attaque est déployé sur une machine virtuelle fournie aux étudiants. Il s'agit d'un simple serveur utilisant la distribution Debian dans sa version stable. Le code du serveur est censé avoir été développé par un prestataire de services (la société de services fictive SOFT.BZH). Il est développé en langage C. Il s'agit d'un code client/serveur dont le rôle est de stocker des transactions envoyées par des clients de la société PRESSOARE à un reverse proxy qui sert à déchiffrer le trafic supposé être envoyé en TLS au serveur. Le reverse proxy est supposé être localisé dans une DMZ publique, tandis que le serveur attaqué est localisé dans une DMZ interne. Le système d'exploitation socle sur lequel tourne le serveur attaqué est raisonnablement durci en suivant le guide d'hygiène de l'ANSSI (surface d'attaque minimale, pas d'interface graphique, application des mises à jour de sécurité automatique, administration par SSH sur une interface réseau dédiée, authentification par clé SSH de taille suffisante, parefeu limitant le trafic réseau au minimum nécessaire). Le serveur répond aux commandes textuelles suivantes :

2. Alexander Peslyak.

— `QUIT` : permet de fermer proprement la connexion et de quitter.
— `EXISTS ressource` : détermine si la ressource passée en paramètre existe ou pas.
— `GET ressource` : permet de récupérer la valeur associée à la ressource en question (si elle existe).
— `PUT ressource valeur` : permet d'associer la valeur passée en paramètre à la ressource.
— `ECHO texte` : commande de *debug* renvoyant le texte passé en paramètre en réponse.

Les ressources sont créées sous la forme de fichiers résidant dans le répertoire de travail du serveur. Le nom des ressources est contraint à l'ensemble de caractères [A-Za-z0-9] par le code du serveur. Seules les valeurs entières positives sont quant à elles acceptées. Le protocole est donc minimaliste, et il pourrait être intéressant afin de crédibiliser un peu plus le scénario de l'enrichir avec une commande d'authentification afin de n'autoriser que les commandes `ECHO` et `QUIT` dans le mode non authentifié.

Le binaire ainsi que le code source du serveur sont fournis aux étudiants. Le code source est recompilabe via une chaîne de compilation basée sur les *autotools* [5]. Le code peut être compilé en activant ou en désactivant un ensemble de protections (proches de l'état de l'art) :

1) Protection de la pile par canari (option `-fstack-protector` de GCC) ;
2) Code indépendant de sa position dans l'espace d'adressage (option `-pie` de GCC).

Dans le scénario le plus complexe, ces deux protections sont défaites par un ensemble de techniques avancées.

C. La principale vulnérabilité exploitée par l'attaquant

Nous proposons deux scénarios d'attaque possibles en fonction de la difficulté souhaitée et du niveau des étudiants en début de travaux pratiques :

1) Scénario simple : attaque par débordement de buffer classique avec retour sur la pile. Ce scénario ne devrait a priori plus fonctionner sur un socle système suffisamment durci.
2) Scénario complexe : attaque en plusieurs phases permettant de faire fuir de l'information sur l'espace d'adressage via une seconde vulnérabilité afin d'outrepasser l'ASLR, puis attaque en ROP afin de passer outre la protection apportée par le bit NX du processeur.

Plus précisément, chacun de ces deux scénarios repose sur l'exploitation d'une faille de type débordement de *buffer*. Le code fautif peut se résumer à :

```
int vulnerable(char *unsafe){
  int res;
  char *copie;
  int len;
  int i;
  char buffer[MAXLENGTH];

  len = strlen(unsafe);
  if(len > MAXLENGTH)
    return ERROR;
```

```
  else{
    copie = buffer;
    for(i=0; i<=len; i++){
      *copie=*unsafe;
      copie++;
      unsafe++;
    }
  }

  res = analyse(buffer);
  return res;
}
```

Ce code est sensé analyser la chaîne de caractères qui lui est passée en paramètre (supposée contenir des caractères interdits par exemple). Le corps de la fonction recopie cette chaîne de caractères dans un buffer alloué sur la pile et de taille fixe (ici une constante `MAXLENGTH`). Cette recopie se fait par arithmétique de pointeur, sous le prétexte fallacieux de vitesse d'exécution (commentaire laissé dans le code source par le développeur). On notera que la taille de la chaîne de caractères passée en paramètre est testée au début de la fonction. Cependant le développeur commet ici une faute en oubliant que la fonction `strlen` ne renvoie pas la longueur de la chaîne de caractères en incluant l'octet utilisé pour marquer la fin de chaîne. Donc si la chaîne de caractères mesure exactement `MAXLENGTH` caractères, le test va autoriser l'exécution de la boucle `for`. Or celle-ci recopie bien le caractère de terminaison de la chaîne (via l'utilisation d'un test de fin de boucle `i<=len`). La faute de conception commise par le programmeur se traduit donc par l'écriture d'un unique octet en dehors du *buffer* alloué sur la pile. Si on suppose (ce qui n'est pas forcément le cas) que les variables sont disposées sur la pile dans le même ordre que celui de leurs déclarations dans l'entête de la fonction, alors ce dépassement d'un seul octet va avoir des conséquences importantes. Le lecteur pourra se reporter à la figure 1 pour se représenter l'organisation de la pile de la fonction vulnérable. L'octet nul écrit en dehors du *buffer* va venir écraser l'octet de poids faible[3] de la variable `i`. Ceci a pour effet de réinitialiser la valeur de i à zéro, puis à un par l'incrément de boucle. L'attaquant peut alors injecter des zéros pour les 3 octets de poids forts de la variable `i`, avant de venir écraser à son tour la variable `len`, ce qui va lui permettre de piloter finement le nombre d'octets qu'il souhaite écraser sur la pile. Enfin, il va venir écraser la variable `copie`. Or celle-ci pilote l'adresse qui est en train d'être copiée. L'attaquant va donc venir écraser l'octet de poids faible de cette variable ce qui aura pour conséquence de provoquer un saut dans la recopie. L'attaquant va profiter de ce saut pour sauter directement sur l'adresse de retour de la fonction vulnérable, passant ainsi au-dessus du canari qui aura été positionné par la chaîne de compilation. La recopie peut s'arrêter à cet endroit dans le scénario simple ou continuer dans le cas d'une charge plus complexe. Nous discutons un peu plus loin de la possibilité d'obtenir une attaque qui fonctionne malgré les mises à jour des chaînes de compilation et qui soit stable dans le temps.

3. On notera ici que le fait que la pile soit décroissante, et qu'inversement l'architecture x86 soit petit-boutiste joue en notre faveur.

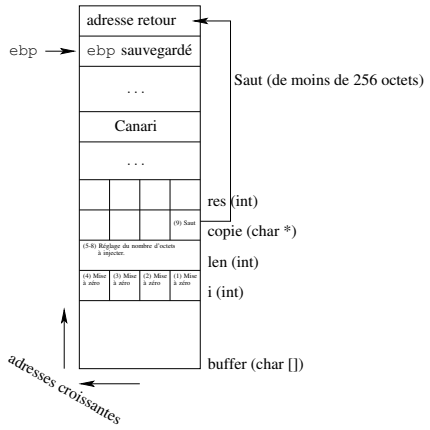

FIGURE 1. Organisation de la pile de la fonction vulnérable et résumé des premières étapes de l'attaque.

D. Scénario simple

Dans ce scénario, d'une part le serveur est mal configuré et (pour d'obscures raisons que les étudiants devront commenter) propose des options sur sa ligne de commande qui permettent de (1) désactiver la randomisation de son espace d'adressage, et (2) rendre la première page de sa pile exécutable. Ceci est obtenu respectivement via des appels aux fonctions `personality`[4] et `mprotect` au démarrage de l'application. On est clairement confronté ici à un problème de configuration du serveur, ainsi qu'à des erreurs de conception du prestataire qui a développé le code. Ces options sont inutiles au bon fonctionnement du serveur et sont contreproductives en terme de sécurité. L'attaque se trouve ainsi considérablement facilitée. L'attaquant peut alors exploiter la vulnérabilité exposée à la section IV-C pour injecter du code exécutable dans la pile et le déclencher au retour de la fonction. La charge active utilisée par l'attaquant est dérivée de celle décrite en cours. Elle est simplement adaptée ici pour réutiliser le descripteur de fichier de la connexion TCP ouverte par l'attaquant et rediriger l'entrée standard et les sorties standard et d'erreur vers celui-ci via des invocations de l'appel système `dup2`. Ceci permet donc à l'attaquant d'obtenir un shell distant s'exécutant avec les droits du serveur. Dans le scénario proposé aux étudiants, l'attaquant se contente de récupérer des informations sans grand intérêt. Mais on pourrait imaginer d'augmenter celui-ci par une élévation de privilèges et une poursuite de l'attaque par un déplacement latéral.

E. Scénario complexe

Dans le scénario plus complexe, l'attaquant commence par récolter des informations sur l'espace d'adressage du serveur via une fuite d'information découlant d'une vulnérabilité résultant de l'usage involontaire d'une chaîne de formatage contrôlée par l'attaquant (*format string*) lors du traitement de la commande `ECHO`. Via cette fuite d'information l'attaquant

4. L'auteur a découvert à cette occasion que l'on peut désactiver l'ASLR pour un processus donné via cet appel et souhaitait partager cette curiosité.

est capable de déterminer (1) la position dans la pile de l'adresse de retour de la fonction vulnérable et (2) l'adresse de retour d'une fonction appelée par la fonction vulnérable. Ceci lui permet de lever la randomisation de la position de l'exécutable dans l'espace d'adressage du serveur et donc de contourner la contremesure imposée par la chaîne de compilation (option `-pie`). Puis l'attaque continue comme dans le scénario simple. Cependant, au lieu de détourner l'exécution vers la pile, ce qui lui est impossible si le bit NX est réellement appliqué à la pile, celui-ci va avoir recours à du ROP. L'attaquant va utiliser un ensemble de cinq gadgets présents dans le binaire lui-même afin de venir réécrire la section .got.plt en mémoire pour détourner des appels de la bibliothèque C. Ne connaissant pas la position de la librairie C en mémoire, l'attaquant va utiliser un gadget du type `adc byte ptr [eax+offset], ebx`[5], qui lui permet d'additionner à l'adresse de son choix un octet quelconque. En répétant 4 fois cette opération, il peut additionner un décalage arbitraire codé sur 32 bits. Il peut répéter cette opération pour toutes les fonctions de la librairie C qu'il souhaite détourner vers des gadgets présents dans celle-ci (qui est beaucoup plus riche que le binaire vulnérable). Il peut additionner aussi bien des décalages en avant (positifs) qu'en arrière (négatif) puisque le processeur fait ses additions modulo 2^{32}. Il lui faut simplement prendre la précaution de (1) remettre à zéro le bit de retenue (*carry flag*) à chaque étape, (2) détourner des fonctions qui ont déjà été appelées par le code vulnérable, sans quoi leur entrée dans la section .got.plt pointe vers le résolveur de symboles du chargeur dynamique. La remise à zéro de bit de retenue est obtenue par l'utilisation d'un effet de bord de l'instruction `test eax, constante`[6] qui est présente comme gadget dans le binaire vulnérable.

Une fois cela fait, l'attaquant peut utiliser des gadgets plus classiques. Plus précisément, il utilise un ensemble de cinq gadgets permettant en premier lieu de faire un appel système à `mmap` afin de projeter dans l'espace d'adressage du processsus attaqué une zone mémoire accessible en lecture-écriture-exécution (violation de la politique W^X) puis écrit la charge principale par des appels à l'instruction `mov`. Tous les gadgets ont été découvert par l'utilitaire ROPgadget.

F. Robustesse de l'attaque dans le temps

Un premier exemple de cette attaque qui soit fonctionnel avait été obtenu par l'auteur il y a une dizaine d'année avec une version de GCC maintenant obsolète (sans doute 4.x). Il est clairement impossible de reproduire l'attaque en utilisant une version plus récente de GCC car le placement des variables dans la pile obéit à des règles assurant naturellement une plus grande sécurité. En particulier les *buffer* risquant de déborder sont systématiquement placés près de l'adresse de retour (et donc d'un éventuel canari qu'ils viendront écraser), alors que les variables qui influent potentiellement les structures de contrôle sont placées plus bas dans la pile à l'abri d'éventuels débordement. Afin de contourner cette limitation et de continuer à utiliser cet exemple de vulnérabilité qui nous paraît intéressant en ce sens qu'il met en avant le risque que présente le débordement de ne serait-ce qu'un seul octet hors du *buffer* destiné à le contenir, nous avons cherché un moyen de convaincre le compilateur d'organiser la pile de la même manière que celle illustrée par la figure 1. La spécification du langage C [2] précise que « *Within a structure object, the non-bit-field members and the units in which bit-fields reside have addresses that increase in the order in which they are declared.* ». Autrement dit, si l'on empaquète les variables qui jouent un rôle dans l'attaque au sein d'une même structure et dans un ordre de déclaration adéquat, il est possible de rendre le programme vulnérable à l'attaque décrite précédemment pour toutes les versions de la chaîne de compilation. Par exemple :

```
typedef struct msg_t{
  char safeBuffer[MAXLENGTH];
  int i;
  int len;
  char *copie;
} msg;
```

5. L'instruction `adc` procède à une addition en tenant compte du bit de retenue (*carry flag*).

6. L'instruction `test` permet de tester l'égalité de deux opérandes.

V. Évaluation des étudiants

Les étudiants sont évalués sous la forme d'une restitution à l'oral avec des supports électroniques de présentation. À l'issue de la présentation, le jury (composé de deux enseignants) qui joue le rôle du commanditaire (la société PRESSOARE) interroge le binôme sur des points techniques qui pourrait être restés obscurs durant la présentation. Le but ici est de mesurer jusqu'à quel point les étudiants ont acquis des connaissances notamment en terme de contremesures concernant les attaques par débordement de *buffer*. La note obtenue par le binôme tient compte de la qualité de la présentation, de la réalisation technique (en terme de rejeu de l'attaque), des contremesures proposées et des réponses aux différentes questions du jury.

VI. Conclusion

L'ensemble du matériel pédagogique de ce cours et du TP sera partagé publiquement sur un compte Github avant la conférence.

Références

[1] Solar DESIGNER. In : (août 1998). URL : http://insecure.org/sploits/linux.libc.return.lpr.sploit.html.

[2] JTC1/SC22/WG14. *Committe draft, ISO/IEC 9899 :1999*. Sept. 2007. URL : http : / / www . open - std.org/jtc1/sc22/wg14/www/docs/n1256.pdf.

[3] Sebastian KRAHMER. "x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique". In : (oct. 2005).

[4] Aleph ONE. "Smashing the Stack for Fun and Profit". In : *Phrack* 7.49 (nov. 1996). URL : http://www.phrack.com/issues.html?issue=49&id=14.

[5] WIKIPEDIA. *GNU Build System*. URL : https : / / en . wikipedia.org/wiki/GNU_Build_System.