Codi segur

José María Alonso Cebrián
Jordi Gay Sensat
Antonio Guzmán Sacristán
Pedro Laguna Durán
Alejandro Martín Bailón
Jordi Serra Ruiz

PID_00208409

UOC
Universitat Oberta de Catalunya

Índex

1. *Integer overflow*

A partir d'aquest mòdul estudiarem una sèrie de mòduls centrats en l'explotació d'errors de seguretat en el codi mitjançant l'absència de control de les dades d'entrada. En aquest mòdul, els desbordaments es produiran en els tipus de dades numèriques, i la tècnica es coneix com a *integer overflow*.

Aquest error no ens permetrà executar codi com en els exemples que veurem posteriorment o com en els que hem esmentat en punts anteriors, sinó que ens permetrà provocar errors en l'aplicació per a comprovar-ne l'estabilitat o per a saltar filtres de comprovació.

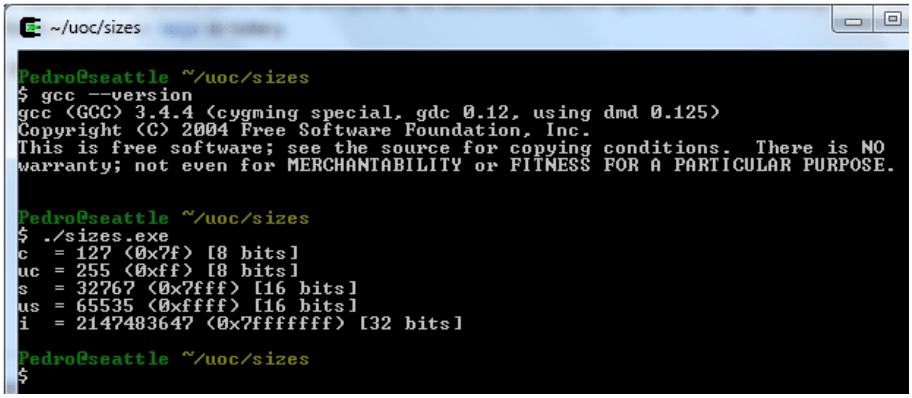
1.1. Tipus de dades numèriques

Els ordinadors (la majoria) desen les dades numèriques en registres que són de la mida dels punters que utilitzen. Això, en l'ús diari, és vàlid, ja que correspon a 32 bits (des de –2.147.483.648 fins a 2.147.483.647). A més, els ordinadors poden treballar amb nombres més grans i també més petits, pensats per a adaptar-se a les necessitats del programador i del programa. En ANSI C, les variables i les seves mides màximes són:

Nom	Mida
Char	De –128 a 127 (8 bits)
unsigned char	De 0 a 255 (8 bits)
Short	De –32.768 a 32.767 (16 bits)
unsigned short	De 0 a 65.535 (16 bits)
Int	De –2.147.483.648 a 2.147.483.647 (32 bits)

Tipus de dades numèriques en ANSI C

Per a provar això farem un petit programa en C que comprovi les mides de cada tipus de dada. Cal notar que el resultat pot variar d'un ordinador a un altre depenent de la versió del compilador que s'usi i l'arquitectura de l'equip. En aquest exemple s'usa el compilador GCC en un entorn amb Cygwin sobre un sistema operatiu Windows 7 Beta 1 de 32 bits:



Versió de GCC i execució del programa `sizes.c`

Com es pot observar en la figura anterior, el programa mostra la mida màxima de cadascuna de les variables que podem definir. Al mateix temps el programa mostra la seva representació en hexadecimal i la seva mida en bits.

El codi utilitzat per a aquesta prova es pot veure a continuació perquè el pugueu provar en la vostra màquina, en el vostre entorn programari i amb el compilador que utilitzeu.

```
#include <stdio.h>
int main(void)
{
   char c = 127;
   unsigned char uc = 255;
   short s = 32767;
   unsigned short us = 65535;
   int i = 2147483647;

   printf("c = %d (0x%x) [%d bits]\n", c, c, sizeof(c) * 8);
   printf("uc = %d (0x%x) [%d bits]\n", uc, uc, sizeof(uc) * 8);
   printf("s = %d (0x%x) [%d bits]\n", s, s, sizeof(s) * 8);
   printf("us = %d (0x%x) [%d bits]\n", us, us, sizeof(us) * 8);
   printf("i = %d (0x%x) [%d bits]\n", i, i, sizeof(i) * 8);

   return 0;
}
```

`sizes.c`

Com es pot observar, en la sortida del programa la diferència entre *signed* i *unsigned* és que el bit més significatiu, conegut com a MSB, en les variables *signed* és 0 en els valors positius (7 en hexadecimal és 0111).

1.2. Representació de nombres negatius

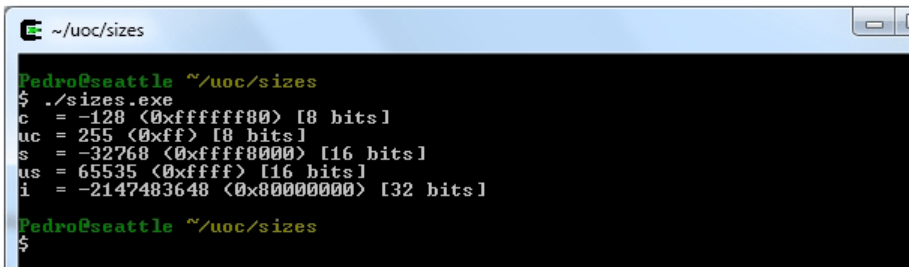
Com es pot veure en la imatge 3.2, en el tipus `char`, amb només 8 bits, es pot representar el 127 com els bits 0111 1111 [0x7f]. És a dir, l'MSB a 0 que indica un valor positiu i després el valor 127. Per contra, en els valors negatius això funcionarà de manera diferent. En aquest cas el primer bit serà l'1, que indica el signe negatiu del nombre, i després hi anirà el valor per restar del

valor més gran. Així, el valor 1000 0000 [0x80] serà el –128. El –127 serà 1000 0001 [0x81] i així successivament fins a arribar al valor 1111 1111 [0xff], que serà el –1.

Si establim els valors negatius dels tipus de dades que són compatibles amb el signe podrem veure com s'emmagatzemen les dades. Ara canviarem la inicialització de les variables al programa anterior pels valors següents i veurem com es representen els tipus de dades *signed*:

```
char c = -128;
unsigned char uc = 255;
short s = -32768;
unsigned short us = 65535;
int i = -2147483648;
```

En la figura següent es pot veure com els valors amb signe són representats tal com s'acaba d'exposar.



Sortida del programa `sizes.c` amb valors negatius

Això s'implementa així per a poder representar nombres negatius i, depenent de la manera com es processin els nombres entre tipus de dades de diferents mides, pot donar lloc a un error d'*integer overflow*, com veurem més endavant.

1.3. Desbordament de tipus de dada

Ateses les limitacions i les maneres de representar les dades en els tipus vistos, la pregunta que cal fer-se és: què succeirà si intentem instanciar una variable de més mida, un `int` o un `short`, en una de més petita, `short` o `char`?

La resposta és que el computador truncarà els valors de dades introduint en la variable destinació els bits que entrin, començant pels bits menys significatius i perdent, per tant, els bits més significatius del valor.

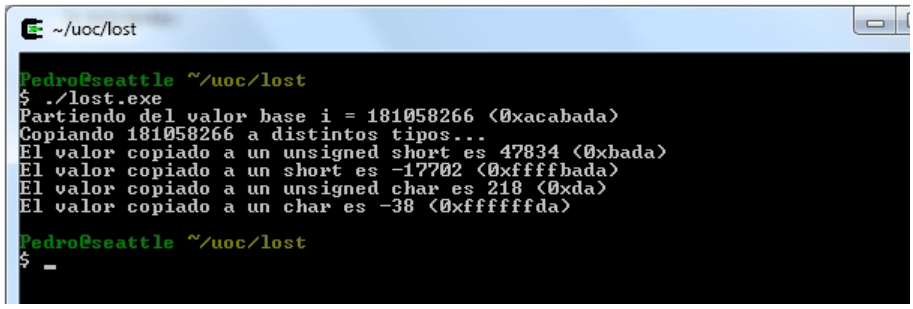
Ara modificarem el codi `sizes.c` per assignar valors de més mida a variables de menys mida. Mostrarem per pantalla els resultats obtinguts i podrem veure com s'ha comportat el sistema.

```
#include <stdio.h>

int main(void)
{
   char c = 0;
   unsigned char uc= 0;
   short s = 0;
   unsigned short us = 0;
   int i = 181058266;

   printf("Partint del valor base i = %d (0x%x)\n", i, i);
   printf("Copiant %d a diferents tipus...\n", i);

   us = i;
   printf("El valor copiat a un unsigned short és %d (0x%x)\n",us,us);

   s = i;
   printf("El valor copiat a un short és %d (0x%x)\n",s,s);

   uc = i;
   printf("El valor copiat a un unsigned char és %d (0x%x)\n",uc,uc);

   c = i;
   printf("El valor copiat a un char és %d (0x%x)\n",c,c);

   return 0;
}
```

Còpia de valors més grans que el tipus de dades

En aquest codi s'ha inicialitzat la variable de tipus `integer i` amb un valor gran, en aquest cas concret `181058266`. Aquest nombre és prou gran per a desbordar la resta dels tipus de dades usades en el programa. Com es pot veure, aquest nombre va essent copiat dins de variables de menys mida i mostrat per pantalla.

L'execució d'aquest programa compilat donarà una sortida com la que es pot apreciar en la imatge següent:



Desbordament de tipus

Com es pot observar, els valors es trunquen a l'alçada del màxim nombre de bits que pugui contenir el tipus de dades sobre el qual s'ha copiat el valor de la variable original. Això ocorre de manera dràstica, sense apropar-se si més no al valor màxim permès de cadascun dels tipus.

Això pot implicar, com es pot veure en el cas de l'exemple, que per a algunes mides de variables els valors continguts siguin positius i altres siguin negatius perquè hagi quedat un bit a 1 en el bit de signe.

1.4. Atacs *integer overflow*

Després de veure com funcionen les variables en memòria en el mòdul d'introducció i com es tracten i interpreten, podem començar a analitzar els processos que ocorren internament perquè es produeixi un *integer overflow*.

Els atacs d'*integer overflow* no ens permetran sobreescriure zones de memòria, variables o codi, però sí canviar la lògica de l'aplicació i fins i tot desbordar estructures de memòria creades per mitjà de variables insegures. Això serà a causa que s'introdueixen en variables valors mai esperats mitjançant el truncatge indiscriminat de valors.

Ara farem un exemple que mostri els dos errors en un únic codi.

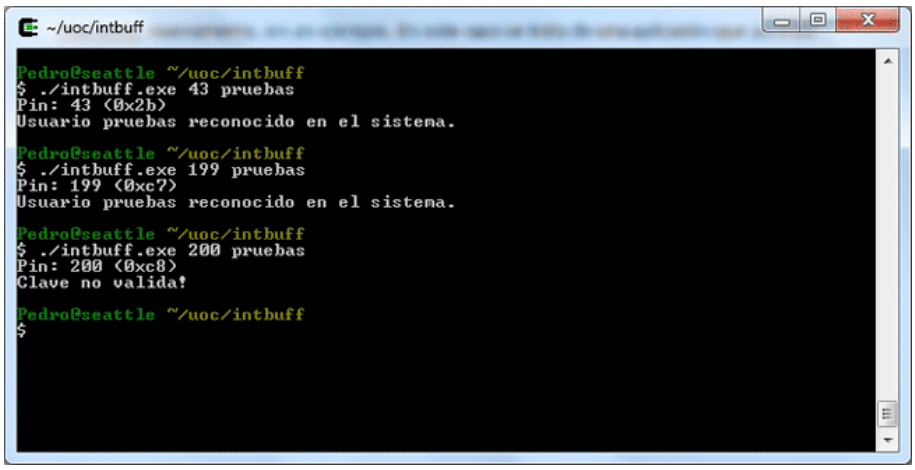
1.4.1. De major positiu a menor negatiu

Suposem un programa que rebi un número de PIN i un usuari per a accedir a una aplicació. Suposem que només els usuaris amb valor de PIN inferior a 200 poden entrar en el sistema. Una mala comprovació podria fer que s'utilitzi una variable *short* que permeti valors negatius, a la qual assignaríem un valor de tipus `unsigned short`.

El tipus `unsigned short` permet valors positius que, si s'assignen a valors *short*, es convertiran en valors negatius.

Així, si assumim que tots els usuaris tenen un PIN positiu però assignem d'alguna manera un valor d'entrada `integer` o `unsigned short` a una variable `signed short`, aconseguirem que un valor més gran es converteixi en un valor més petit.

Si implementem aquesta comprovació, és a dir, que només puguin entrar usuaris amb valors de PIN, amb una simple comparació del tipus PIN<200 tindrem un problema a l'hora d'assignar valors `integer` o `unsigned` de gran mida a la variable `short pin`.



Execució de control per PIN

Com es pot observar en la figura anterior, el programa ha rebut tres valors i només l'últim ha resultat incorrecte, ja que arribava al límit imposat per l'aplicació. Si fem un cop d'ull al codi, només obtindrem aquests dos valors, és a dir, usuari no reconegut o clau no vàlida, ja que és només això, una aplicació sintètica per a provar el desbordament.

No obstant això, veiem que estem assignant el valor que vingui per línia d'entrada `argc[1]` convertit a `integer` amb la funció `atoi` a una variable de tipus `short`. A més, la comprovació del PIN es fa com una comparació, com ja hem explicat.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
   short pin;
   char user[200];

   if(argc < 3)
   {
      printf("Ús: intbuff <pin> <usuari>\n");
      return -1;
   }

   pin = atoi(argv[1]);

   printf("Pin: %d (0x%x)\n", pin, pin);

   if(pin >= 200)
   {
      printf("Clau no vàlida!\n");
      return -1;
   }

   memcpy(user, argv[2], pin);

   printf("Usuari %s reconegut en el sistema.\n", user);

   return 0;
}
```

Codi de comprovació de PIN

A primera vista el codi sembla bo, però a l'hora d'executar-lo pot arribar a fallar.

Si en l'aplicació introduïm un valor més gran que el valor màxim d'un `short` (32767) ens trobarem que l'aplicació falla. Per què? Doncs per la falta de comprovació de les variables.

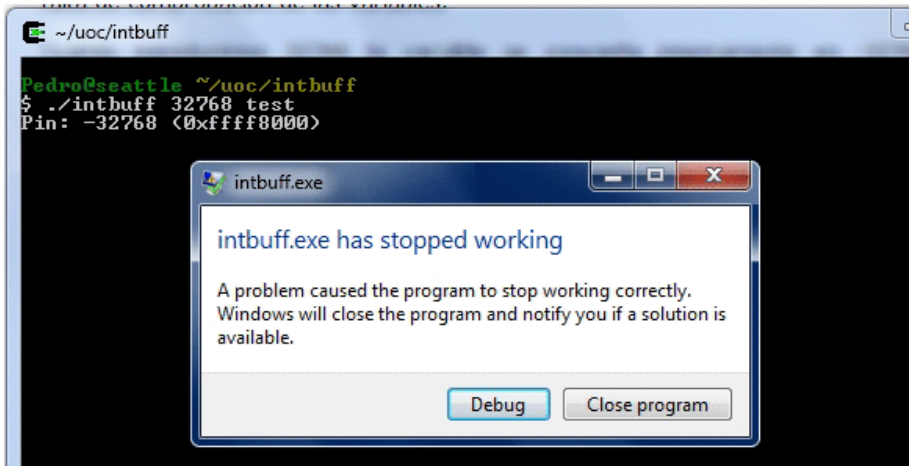
Quan introduïm 32768, un valor més gran que el suportat pel tipus *short*, la variable es converteix internament en –32768 a causa de l'ús que fan els ordinadors del bit més significatiu (MSB). Per a l'ordinador, el nombre `0xffff8000`, en valer 1 l'MSB, és negatiu. Per tant, la comprovació de si el nombre és més gran o igual que 200 serà, sens dubte, falsa.

Com es pot veure, gràcies a una mala comprovació en el tipus de dades s'ha produït un comportament erroni en la lògica de l'aplicació.

1.4.2. Desbordament de l'aplicació

En la segona part del codi d'aquesta aplicació es fa ús de la variable per a reservar memòria amb la funció `memcpy`. L'habitual en aquesta funció és fer una reserva de memòria de la mida de la variable que es vol copiar, és a dir, una cosa com `sizeof(user)`, però en aquest cas s'està fent una assignació de memòria de la variable PIN, que és de tipus `short` després d'haver estat truncada d'una variable `integer`.

El resultat que obtenim si posem una mida molt gran és que desbordarem la memòria de l'aplicació i obtindrem un error com el que es pot veure en la figura següent.

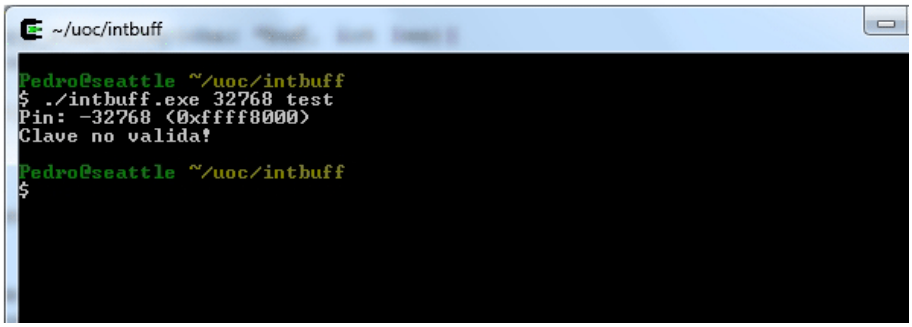


Desbordament de la memòria

No és l'exemple més comú del món, però volíem deixar clara la importància de controlar l'ús correcte dels tipus de dades numèriques en totes les assignacions de variables. De tota manera, `memcpy`, com ja veurem més endavant, és una de les funcions més utilitzades en la creació d'*exploits* de desbordament de memòria.

Com es pot observar, la comprovació dels valors numèrics ha de ser exhaustiva perquè no es produeixin errors inesperats. Per exemple, en el codi anterior, la solució hauria implicat incloure una comprovació per a detectar si el PIN introduït, a més de ser més petit que 200, era més gran que 0 i, per descomptat, comprovar la mida del tipus de dades abans de començar a treballar-hi.

```
if(pin < 0 || pin >= 200)
{
   ...
}
```



Execució amb el codi amb el pegat

Com hem vist, les conversions de tipus poden ser molt perilloses; per això, en treballar amb aquest tipus de variables hem de tenir en compte tots els possibles valors que puguin prendre segons el tipus de dades, com per exemple:

- Més petit que el valor mínim.
- Més petit que zero.
- Zero.
- Més gran que zero.
- Més gran que el valor màxim.

Aquestes són les proves més utilitzades per les eines de *fuzzing* a l'hora de detectar valors insegurs i conversions errònies de tipus de dades numèriques en aplicacions.

2. Desbordament de pila

Un dels errors més importants en el món de la seguretat informàtica ha estat el desbordament de variables en memòria. Aquest error és un dels que més vegades han possibilitat la creació d'*exploits* per a l'execució de codi en la màquina amb el programari vulnerable. Amb aquest tipus d'atacs en moltes aplicacions vulnerables es podrà prendre el control sobre el flux d'execució de l'aplicació.

Abans de començar és convenient recordar els conceptes bàsics de la memòria vistos anteriorment. La memòria té, segons hem vist, una zona dedicada a les variables del programa que es divideix en dos: la pila o *stack*, *heap* o zona de memòria dinàmica. La zona de pila té un creixement de dalt a baix pel que fa a posicions de memòria. S'hi va reservant memòria a mesura que el programa va definint variables i emmagatzemant aquestes últimes en format *little endian*, és a dir, amb el bit menys significatiu a l'esquerra. És important tenir present aquestes peculiaritats, ja que seran de gran importància en tots els *exploits* que anirem veient l'hora de modificar i sobreescriure la memòria.

2.1. Un exemple de desbordament de memòria

Comencem amb un petit programa que ens permetrà entendre i comprovar com funciona el nostre ordinador pel que fa a registres i memòria. Els exemples que segueixen a continuació s'han fet sobre una màquina de 32 bits, usant el compilador GCC en la versió 3.4.4 per mitjà de l'aplicació Cygwin. Pot ocórrer que les dades usades aquí variïn una mica respecte a les que s'hagin d'introduir en altres entorns, però es proposarà un mètode per a aprendre a ajustar els valors a qualsevol plataforma.

Per a poder desenvolupar els exercicis, treballarem amb el programa que es pot veure en el codi d'exemple. S'hi pot apreciar la funció `function`, que rep tres valors numèrics, que són recollits en les variables `a`, `b` i `c`.

La funció inicialitza dues matrius de 5 i 10 posicions, `buffer1` i `buffer2`, que inicialitza amb dues *strings* de lletres `A` i lletres `B`.

Després, la funció crea un punter a un valor numèric. Aquest punter es fa apuntar a l'adreça de memòria situada 28 bytes més enllà del començament del `buffer1`. Recordem que `buffer1` en realitat és un punter a la primera posició de la matriu.

Finalment, la funció incrementa en 7 el valor existent en aquesta posició de memòria.

El codi principal del programa crea una variable `integer x` que és inicialitzada a 0. Després es crida la funció creada amb els valors `1,2,3`, s'assigna el valor `1` a la variable `x` i s'imprimeix el valor de `x`.

```
#include <stdio.h>
void function(int a, int b, int c)
{
   char buffer1[5] = "AAAAA";
   char buffer2[10] = "BBBBBBBBBB";
   int *ret;

   ret = buffer1 + 28;
   (*ret) += 7;
}

int main(void)
{
   int x;

   x = 0;
   function(1,2,3);
   x = 1;
   printf("%d\n",x);

   return 0;
}
```

Codi d'exemple

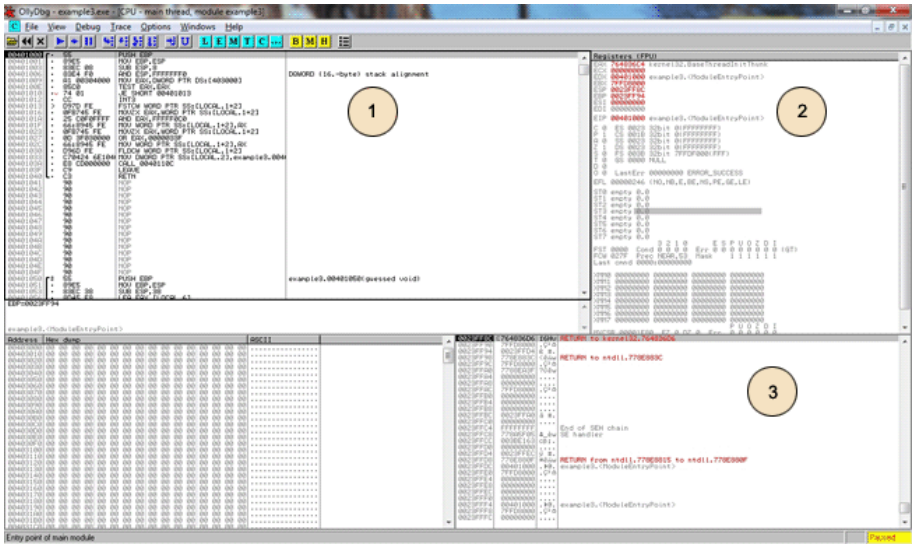
Aquest programa de propòsit simple hauria de crear una variable que s'inicia a `0`, cridar una funció que aparentment no fa res, posar la variable a `1` i acabar imprimint aquest valor. No obstant això, i gràcies al coneixement que adquirirem sobre la memòria, aconseguirem que el nostre programa "salti" l'assignació d'un nou valor a la variable `x` i imprimeixi per consola el valor de `0`.

2.1.1. Anàlisi inicial

Per a començar l'anàlisi hem de compilar el programa i usar algun tipus de depurador per a observar el codi assemblador generat. D'aquesta manera es podrà depurar a poc a poc.

En aquesta ocasió usarem el programa gratuït anomenat Ollydbg en la versió 2 (beta 1). Aquest depurador, com ja hem vist, ens permetrà abastar d'un cop d'ull diferents zones importants de l'execució del nostre codi, tal com hem vist en els exercicis del primer mòdul:

- Codi assemblador
- Registres de memòria
- Memòria del programa



Zones de pantalla en l'Ollydbg

Després de compilar el programa anterior i obrir-lo amb l'Ollydbg, podrem observar que s'ha obtingut el codi assemblador mostrat a continuació. Com es pot veure, el codi generat no és gaire gran, la qual cosa ens permetrà analitzar-lo més fàcilment.

```
00401050 /$ 55              PUSH EBP
00401051 |. 89E5            MOV EBP,ESP
00401053 |. 83EC 38          SUB ESP,38
00401056 |. 8D45 E8          LEA EAX,[LOCAL.6]
00401059 |. 83C0 1C          ADD EAX,1C
0040105C |. 8945 D4          MOV DWORD PTR SS:[LOCAL.11],EAX
0040105F |. 8B55 D4          MOV EDX,DWORD PTR SS:[LOCAL.11]
00401062 |. 8B45 D4          MOV EAX,DWORD PTR SS:[LOCAL.11]
00401065 |. 8B00             MOV EAX,DWORD PTR DS:[EAX]
00401067 |. 83C0 07          ADD EAX,7
0040106A |. 8902             MOV DWORD PTR DS:[EDX],EAX
0040106C |. C9               LEAVE
0040106D \. C3               RETN

0040106E /. 55              PUSH EBP
0040106F |. 89E5            MOV EBP,ESP
00401071 |. 83EC 18          SUB ESP,18
00401074 |. 83E4 F0          AND ESP,FFFFFFF0
00401077 |. B8 00000000      MOV EAX,0
0040107C |. 83C0 0F          ADD EAX,0F
0040107F |. 83C0 0F          ADD EAX,0F
00401082 |. C1E8 04          SHR EAX,4
00401085 |. C1E0 04          SHL EAX,4
00401088 |. 8945 F8          MOV DWORD PTR SS:[LOCAL.2],EAX
0040108B |. 8B45 F8          MOV EAX,DWORD PTR SS:[LOCAL.2]
0040108E |. E8 49000000      CALL 004010DC
00401093 |. E8 D4000000      CALL <JMP.&cygwin1.__main>
00401098 |. C745 FC 00000    MOV DWORD PTR SS:[LOCAL.1],0
0040109F |. C74424 08 030    MOV DWORD PTR SS:[ESP+8],3
004010A7 |. C74424 04 020    MOV DWORD PTR SS:[ESP+4],2
004010AF |. C70424 010000    MOV DWORD PTR SS:[ESP],1
004010B6 |. E8 95FFFFFF      CALL 00401050
004010BB |. C745 FC 01000    MOV DWORD PTR SS:[LOCAL.1],1
004010C2 |. 8B45 FC          MOV EAX,DWORD PTR SS:[LOCAL.1]
004010C5 |. 894424 04        MOV DWORD PTR SS:[ESP+4],EAX
004010C9 |. C70424 002040    MOV DWORD PTR SS:[ESP],00402000
004010D0 |. E8 A7000000      CALL <JMP.&cygwin1.printf>
004010D5 |. B8 00000000      MOV EAX,0
004010DA |. C9               LEAVE
004010DB \. C3               RETN
```

Codi d'assemblador resultant

Analitzem el codi assemblador per entendre com funciona l'ordinador internament en aquesta arquitectura concreta, però per a entendre'l, fem un breu repàs a l'arquitectura interna de la màquina que necessitarem ara.

2.1.2. Els registres

Dins de l'arquitectura x86 hi ha diverses maneres de programar en assemblador a l'hora de gestionar els registres del sistema, però hi ha certs registres i instruccions bastant comunes en la majoria dels compiladors que s'utilitzen. De totes maneres, com ja hem anticipat en el segon mòdul, hem de tenir present que un mateix codi en llenguatge ANSI C pot generar diferents codis en assemblador depenent del compilador utilitzat. Entre tots els registres, els més importants que utilitzarem aquí són:

- `EIP`. És l'*instruction pointer* i desa l'adreça actualment en execució, és a dir, el registre on s'emmagatzema el control de programa.

- `ESP`. És l'*stack pointer* i desa el cap de la pila. Cal tenir en compte que la pila creix de manera invertida, és a dir, cada vegada que la pila creix l'adreça de memòria decreix.

- `EBP`. És el *base pointer* i desa l'adreça de memòria on comença la pila. Com que creix de manera invertida, és l'adreça més gran dins de la pila.

A part d'aquests tres registres especials del codi, es pot veure que es fa ús del registre EAX com a variable auxiliar per a moure valors entre variables.

Quan una funció està en execució, el registre EIP apunta a la instrucció en execució, el registre EBP a l'adreça base de la pila i el registre ESP al capdavant de la pila.

2.1.3. Gestió de la pila

La zona de memòria situada entre l'EBP i l'ESP es coneix com l'*stack frame*, i marca la zona de memòria de la funció assignada a la pila.

Quan es fa una crida a la funció `POP` amb un registre com a paràmetre, el valor situat en la posició de memòria apuntat per ESP serà assignat al registre i el valor d'ESP es desplaçarà per a treure aquest valor de la pila. En aquest cas, la pila es desplaça a la mida del registre extret, de manera que ESP quedarà assignat a ESP +4 [en arquitectures de 32 bits].

Quan es fa una crida a la funció PUSH amb un registre o valor com a paràmetre, aquest serà posat al cim de la pila i el registre ESP es desplaçarà per a indicar que la pila és en la nova posició. En aquest cas ESP serà assignat a ESP –4.

2.1.4. Crida i retorn de funcions

Quan es produeix la invocació d'una funció, hi ha tres funcions importants més que també afecten els registres i la pila. És obligatori conèixer-les per a comprendre com funciona el codi de l'exemple:

- CALL. Quan es produeix una crida a una altra funció, és necessari fer una sèrie d'accions. CALL automatitza aquestes accions. En primer lloc fa un PUSH del valor següent d'EIP, és a dir, de l'adreça on es troba la instrucció següent per executar després que el control de programa torni de la crida a la funció. Aquest valor serà el valor de retorn de la funció cridada. En segon lloc actualitzarà el valor d'EIP a l'adreça de la funció cridada. És a dir, CALL Address genera un PUSH EIP següent i una crida a la funció MOV (moure) EIP, Address.

- LEAVE. Quan s'abandona l'execució d'una rutina, es pot usar la crida a LEAVE per a preparar la sortida. Per a això, aquesta funció situa el cap de la pila en l'adreça de la base, és a dir, MOV ESP, EBP. Després fa un POP al registre EBP, és a dir, restaura el valor original d'EBP. Aquest valor d'EBP és desat en el començament de la funció, com veurem en l'exemple d'aquest tema. Aquesta situació deixa el registre ESP apuntant a l'adreça de retorn CALL, és a dir, la instrucció següent per executar després de la finalització de la funció.

- RTN. La crida a RTN genera la fi de l'execució d'una funció i el que es fa és actualitzar el valor d'EIP al valor d'ESP, és a dir, és un POP EIP.

2.2. Anàlisi del codi de l'exemple

Com es pot observar a simple vista, tenim dues zones de codi, que corresponen a les dues funcions del nostre programa: la funció function i el programa principal main. La primera acaba en 0040106D i la segona comença en la línia següent i acaba en 004010DB, és a dir, en l'última línia del codi assemblador mostrat. En treballar amb un entorn Cygwin es pot veure que al principi el programa fa una sèrie de funcions d'inicialització i preprocés.

Després podem veure com s'assigna el valor 0 a la variable X en la posició de memòria 00401098:

```
00401098 |.     C745 FC 00000      MOV DWORD PTR SS:[LOCAL.1],0
```

Seguidament s'introdueixen en la pila els valors 1, 2 i 3, que es passen a `function` en les línies `0040109F`, `004010A7` i `004010AF`. Es podrien haver utilitzat crides a `PUSH`, però és decisió del compilador copiar els valors que es passen per paràmetre amb la funció `MOV` i, en el començament de la funció cridada, actualitzar l'*stack frame*.

```
0040109F |.      C74424 08 030      MOV DWORD PTR SS:[ESP+8],3
004010A7 |.      C74424 04 020      MOV DWORD PTR SS:[ESP+4],2
004010AF |.      C70424 010000      MOV DWORD PTR SS:[ESP],1
```

Com es pot veure, la memòria funciona com una pila *LIFO*, per la qual cosa s'entén d'aquesta manera que les variables siguin passades en ordre invers.

En la línia següent, la `004010B6`, es fa una crida a `00401050` amb la instrucció `CALL`, o el que és el mateix, una crida a la funció `function`.

```
004010B6 |      E8 95FFFFFF      CALL 00401050
```

Aquesta instrucció apila el valor de l'EIP següent per executar, és a dir, l'adreça de retorn de la funció cridada, i actualitza el valor d'EIP a l'adreça passada per paràmetre en la crida a `CALL`.

Si anem al començament de la funció, ens hem de parar per a analitzar les tres primeres instruccions que acompanyen tota funció, i que són de gran importància.

```
00401050 /$      55        PUSH EBP
00401051 |.      89E5      MOV EBP,ESP
00401053 |.      83EC      38 SUB ESP,38
```

La primera instrucció desa en la pila el valor actual del registre EBP, és a dir, l'adreça base de la pila de la funció que el crida, i actualitza el valor d'EBP amb el valor actual del cim de la pila. Després situa el registre ESP en el lloc on es troba el cim de la pila actualment. La pila reserva per a les variables que usa la funció, en aquest cas 38 en hexadecimal, per la qual cosa reserva 56 bytes, és a dir, 14 posicions de 4 bytes:

- 16 bytes de `char buffer1[5]`
- 16 bytes de `char buffer2[10]`
- 4 bytes de l'`int *ret`
- 20 bytes falten

2.3. La funció `function`

Una vegada enteses aquestes idees, ens centrarem en la funció `function` i analitzarem com modifica l'adreça de retorn de la funció `function` perquè salti la instrucció d'assignació del valor `1` a la variable `x`. Aquesta instrucció es troba en l'adreça de memòria següent:

```
004010BB |. C745 FC 01000 MOV DWORD PTR SS:[LOCAL.1],1
```

Si es vol saltar aquesta instrucció, haurem de modificar el valor que obtindrà EIP quan acabi l'execució de la rutina cridada. EIP serà actualitzat amb el valor emmagatzemat per la instrucció `CALL` que, en aquest cas, portaria a l'execució de la instrucció anterior. Si volem saltar l'execució de la instrucció `004010BB` haurem de desplaçar el valor del registre d'adreça de retorn tantes posicions com distància hi hagi entre `004010C2` (la instrucció següent) i `004010BB` (la instrucció que volem saltar). Una senzilla resta en hexadecimal ens revela que la distància és 7 bytes. Necessitem, llavors, que l'adreça de retorn de la funció es desplaci 7 bytes.

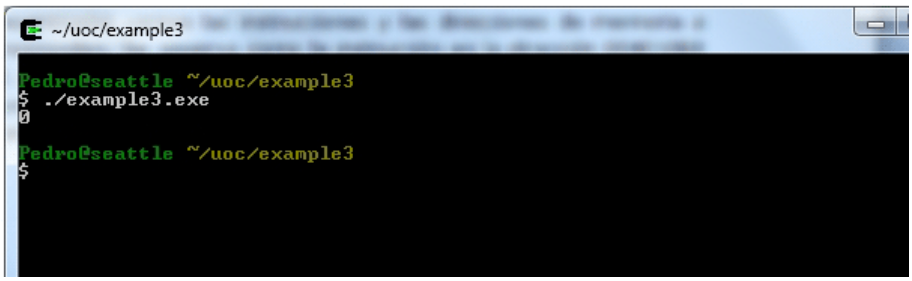
La part del codi en C que farà el salt d'aquesta adreça de memòria és la següent:

```
ret = buffer1 + 28;
(*ret) += 7;
```

La primera instrucció en el codi C localitza la posició en memòria de l'adreça de retorn. Per a descobrir el valor que cal desplaçar-se des de l'inici de la variable `buffer1` cal fer una anàlisi de l'estat de la pila amb l'Ollydbg o el GDB.

Després de fer el buidatge es pot veure que la variable `Buffer1` és a 28 bytes de l'adreça de retorn. N'hi ha prou amb desplaçar la variable `ret` a aquesta posició i incrementar el valor emmagatzemat en aquesta adreça en 7 bytes.

Amb això s'aconsegueix que quan la rutina acabi la seva execució EIP sigui actualitzat a la instrucció següent a l'assignació del valor `1` a `x`, i s'obté com a resultat la sortida per pantalla següent.



Sortida del codi sense execució de la instrucció *x=1*

2.4. Atac de desbordament de pila

L'exemple anterior ens ha servit per a analitzar com funcionen els registres i com és possible alterar el flux del programa alterant els valors dels registres en temps real. Lògicament, a l'hora de crear un *exploit* no es podrà compilar un codi en C, però a la pràctica sí que podrem modificar valors en els registres mitjançant vulnerabilitats de desbordament de pila.

En què consisteix el desbordament de pila? Com hem vist anteriorment, en el programa hem modificat el flux de la nostra aplicació fent que salti una instrucció. Això es pot aconseguir fent ús d'una variable no controlada que ens permeti col·locar com a adreça de retorn l'adreça de memòria que nosaltres vulguem.

De nou, per a aconseguir un aprenentatge basat en l'experiència, compilarem un petit programa vulnerable i l'explotarem per aconseguir l'efecte que vulguem.

```
#include <stdio.h>

int main(void)
{
   char text[16];
   int i = 0;

   printf("text:");
   scanf("%s", text);

   i++;
   printf("El valor de i és %d\n", i);

   return 0;
}
```

Codi en ANSI C vulnerable a desbordament de pila

Aquest codi està assignat al valor de l'entrada per pantalla recollida per la funció *scanf* a una matriu de caràcters de 16 bytes. No obstant això, no s'està fent cap comprovació de mida, per la qual cosa el codi és vulnerable al desbordament. En executar aquest codi podem començar a provar a sobre seu diferents tipus d'entrada en la variable text. Els valors que provarem són:

- X < 16 "A". Amb aquesta entrada el nostre programa funcionarà normalment.

Execució correcta amb "A" < 16

- X = 16 "A". El programa continuarà funcionant perfectament, ja que no sobreescrivim cap zona de memòria que sigui crítica per a l'execució.

Execució correcta amb "A" = 16

- X > 16 "A". Aquí hem d'observar dos possibles casos:

- X < 28 "A"

Aquest cas s'assembla a l'anterior, el programa s'executa normalment i no sembla que quedi afectat per la sobreescriptura de la memòria. Per a entendre això és necessari veure com s'està assignant la memòria a les diferents variables. Si fem una anàlisi amb l'Ollydbg, obtindrem les següents:

Col·locació de les variables en memòria

Com es pot observar en la imatge, la variable `i` ocupa la posició `0023CC5C` i la variable `text` hauria d'ocupar des de `0023CC60` fins a `0023CC6C`, però com que s'ha sobreescrit la memòria intermèdia, gairebé arribem fins a l'adreça `0023CC7C`, on es desa l'adreça de retorn.

- `X` >= 28 "`A`". Recordem que en C, en prémer la tecla d'entrada, quan s'introdueix una cadena s'afegeix també el caràcter `\0` de final de línia, de manera que seran 29 els caràcters emmagatzemats en memòria. Això vol dir que si introduïm 28 o més `A` s'estarà començant a escriure en la posició `0023CC7C` i, tret que es trobi una adreça i memòria vàlida, s'obtindrà un error en l'execució.

Sobreescriptura de l'adreça de retorn amb `A`

El programa arriba a un punt en què se li diu que la instrucció següent per executar es troba en la posició `41414141 (AAAA)` i, com que no troba instruccions vàlides, falla i l'execució es talla.

2.5. Execució de codi per desbordament de pila

En aquest exemple queda clar que si podem sobreescriure l'adreça de memòria per mitjà d'un desbordament de les variables de la pila, seria possible fer que el control de programa fos a qualsevol part de la memòria, és a dir, es podria executar qualsevol programa carregat en el sistema o fins i tot injectat en les variables.

Com que encara no hem introduït codi executable en memòria i no sabem encara com fer-ho, ens aprofitarem del codi mateix de l'aplicació per a crear un bucle en el qual s'incrementi per segona vegada la variable `i`.

Per a això, el que farem és determinar quina és la instrucció que augmenta en 1 la variable `i`. Si mirem el codi assemblat que s'ha generat amb aquest programa obtenim el següent:

```
00401050 /. 55                PUSH EBP
00401051 |. 89E5              MOV EBP,ESP
00401053 |. 83EC 48           SUB ESP,48
00401056 |. 83E4 F0           AND ESP,FFFFFFF0
00401059 |. B8 00000000       MOV EAX,0
0040105E |. 83C0 0F           ADD EAX,0F
00401061 |. 83C0 0F           ADD EAX,0F
00401064 |. C1E8 04           SHR EAX,4
00401067 |. C1E0 04           SHL EAX,4
0040106A |. 8945 D4           MOV DWORD PTR SS:[LOCAL.11],EAX
0040106D |. 8B45 D4           MOV EAX,DWORD PTR SS:[LOCAL.11]
00401070 |. E8 4B000000       CALL 004010C0
00401075 |. E8 D6000000       CALL <JMP.&ygwin1.__main>
0040107A |. C745 E4 00000     MOV DWORD PTR SS:[LOCAL.7],0
00401081 |. C70424 002040     MOV DWORD PTR SS:[ESP],stack.00402000
00401088 |. E8 E3000000       CALL &lt;JMP.&cygwin1.printf&gt;
0040108D |. 8D45 E8           LEA EAX,[LOCAL.6]
00401090 |. 894424 04         MOV DWORD PTR SS:[ESP+4],EAX
00401094 |. C70424 072040     MOV DWORD PTR SS:[ESP],stack.00402007
0040109B |. E8 C0000000       CALL <JMP.&cygwin1.scanf>
004010A0 |. 8D45 E4           LEA EAX,[LOCAL.7]
004010A3 |. FF00              INC DWORD PTR DS:[EAX]
004010A5 |. 8B45 E4           MOV EAX,DWORD PTR SS:[LOCAL.7]
004010A8 |. 894424 04         MOV DWORD PTR SS:[ESP+4],EAX
004010AC |. C70424 0A2040     MOV DWORD PTR SS:[ESP],stack.0040200A
004010B3 |. E8 B8000000       CALL <JMP.&cygwin1.printf>
004010B8 |. B8 00000000       MOV EAX,0
004010BD |. C9                LEAVE
004010BE \. C3                RETN
```

Codi assemblador vulnerable

Com es pot veure en el codi anterior, la instrucció que augmenta en 1 la variable `i` es troba en `004010A3`, per la qual cosa hauríem d'establir l'adreça de retorn a `004010A0`.

2.6. Introducció d'adreces de memòria per teclat

Al final, a l'hora d'escriure l'adreça de memòria en la funció de retorn, necessitarem escriure mitjançant el teclat els valors que volem introduir. Per desgràcia, en un codi en ANSI C no es pot introduir pel teclat cap caràcter que no sigui en l'estàndard ASCII mostrat en la taula següent.

Com es pot veure en la taula, el màxim valor representat en hexadecimal amb l'estàndard ASCII és el `7F`, per la qual cosa ens haurem de limitar, ara com ara, a zones de memòria que continguin valors inferiors a `7F` (128).

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
00	00	NUL	43	2B	+	86	56	V
01	01	SOH	44	2C	,	87	57	W
02	02	STX	45	2D	-	88	58	X

Taula estàndard ASCII

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
03	03	ETX	46	2E	.	89	59	Y
04	04	EOT	47	2F	/	90	5a.	Z
05	05	ENQ	48	30	0	91	5B	[
06	06	ACK	49	31	1	92	5C	\
07	07	BEL	50	32	2	93	5D	]
08	08	BS	51	33	3	94	5E	^
09	09	HT	52	34	4	95	5F	_
10	0A	LF	53	35	5	96	60	`
11	0B	VT	54	36	6	97	61	a
12	0C	FF	55	37	7	98	62	b
13	0D	CR	56	38	8	99	63	c
14	0E	SO	57	39	9	100	64	d
15	0F	SI	58	3A	:	101	65	e
16	10	DEL	59	3B	;	102	66	f
17	11	DC1	60	3C	<	103	67	g
18	12	DC2	61	3D	=	104	68	h
19	13	DC3	62	3E	>	105	69	i
20	14	DC4	63	3F	?	106	6a.	j
21	15	NAK	64	40	@	107	6B	k
22	16	SYN	65	41	A	108	6C	l
23	17	ETB	66	42	B	109	6D	m
24	18	CAN	67	43	C	110	6E	n
25	19	EM	68	44	D	111	6F	o
26	1A	SUB	69	45	E	112	70	p
27	1B	ESC	70	46	F	113	71	q
28	1C	FS	71	47	G	114	72	r
29	1D	GS	72	48	H	115	73	s
30	1E	RS	73	49	I	116	74	t
31	1F	US	74	4A	J	117	75	u
32	20	(es-pai)	75	4B	K	118	76	v
33	21	¡	76	4C	L	119	77	w

Taula estàndard ASCII

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
34	22	"	77	4D	M	120	78	x
35	23	#	78	4E	N	121	79	y
36	24	$	79	4F	O	122	7a.	z
37	25	%	80	50	P	123	7B	{
38	26	&	81	51	Q	124	7C	|
39	27	'	82	52	R	125	7D	}
40	28	(	83	53	S	126	7E	~
41	29	)	84	54	T	127	7F	DEL
42	2A	*	85	55	U			

Taula estàndard ASCII

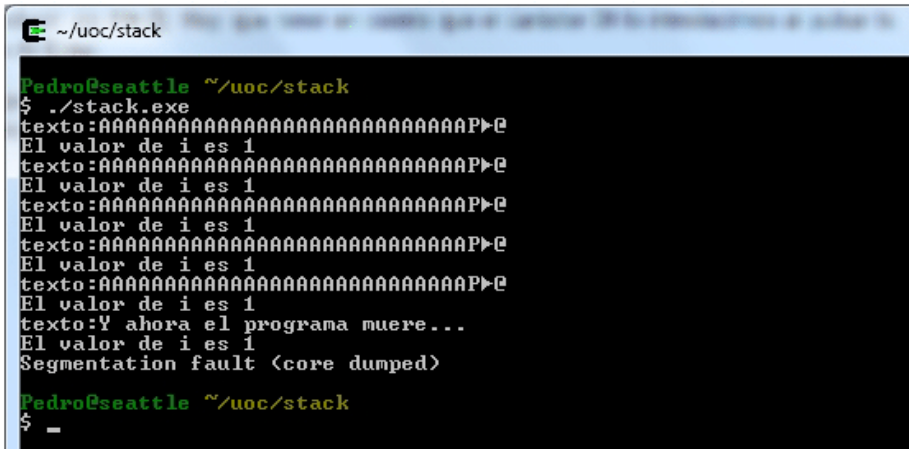
Amb aquesta limitació en l'adreçament podrem fer, en aquest exemple i sense veure cap tècnica més avançada, que el programa s'executi dues vegades si fem que l'adreça de retorn apunti a l'adreça de memòria `00401050`, que és una adreça totalment escrivible amb ASCII. Això provocarà que el programa s'executi mentre introduïm adreces vàlides, fins que no el sobreescriguem o fins que posem una adreça invàlida i puguem comprovar com es canvia el flux del programa amb un desbordament de pila.

Per a fer això introduirem 28 caràcters, tants com ens calen en aquest exemple per a arribar a col·locar-nos just davant de la zona de memòria on es desa l'adreça de retorn. Posteriorment introduirem l'adreça de memòria `00401050`.

Cal recordar que en la memòria les dades es desen a l'inrevés de la manera com les hem introduïdes nosaltres, per la qual cosa s'haurà d'introduir la cadena `50104000` perquè al final quedi situada correctament l'adreça de memòria.

Consultant la taula de valors en la figura 67, l'adreça de memòria `00401050` es converteix en la cadena P►@. Cal tenir en compte que el caràcter `00` l'introduïm en prémer la tecla d'entrada, mentre que els altres caràcters s'introduiran prement Alt + el nombre corresponent en decimal, usant per a introduir el nombre en qüestió el teclat numèric.

Si executem el programa des de la línia d'ordres i introduïm 28 "`A`" i la cadena especificada anteriorment, obtindrem el resultat següent:



Sobreescriptura recurrent d'adreça de retorn

En la captura es pot veure com s'ha fet executar el mateix programa sis vegades fins que li hem introduït una adreça no vàlida en l'adreça de retorn, la qual cosa ha fet que el programa produeixi un error i es detingui.

Amb això queda demostrat que modificar l'execució d'un programa és possible mitjançant la introducció d'una variable prou gran i correctament mal formada si el codi és vulnerable. Això es torna útil més endavant quan veurem com podem llançar un *shellcode* gràcies a aquest tipus de vulnerabilitats.

3. Desbordament de *heap*

Les tècniques de desbordament de *heap* són potser les més complicades d'explotar amb èxit (encara que una denegació de serveis serà sempre més fàcil) a causa que depèn més que cap de les tècniques anteriors de les versions concretes del programari.

No ens referim ara només al programari vulnerable que, evidentment, continua essent l'element més important d'aquesta equació, sinó que aquí entren en joc, i molt, el compilador usat i les biblioteques utilitzades en el procés de compilació.

I a què és degut això? Quan un programa emmagatzema dades en la pila, com ja vam veure, és el compilador el que s'encarrega de reservar la memòria i organitzar-la. Quan parlem de desbordament de *heap*, és el compilador mateix el que organitza les variables del programa que podrem arribar a sobreescriure de la manera que li sembli més convenient.

En aquest mòdul en veurem un exemple senzill del funcionament.

3.1. El *heap*

Abans de continuar discutint aspectes de l'explotació d'un error d'aquest tipus, recordarem com funciona la memòria, i en concret el *heap*. El *heap* és una zona de memòria que usa el programa per a emmagatzemar variables inicialitzades de tipus estàtic i per a reservar blocs de memòria en temps d'execució.

> Les cadenes de text d'un programa per als menús i diàlegs localitzats en diferents idiomes són un bon exemple de variable que s'emmagatzema en el *heap* d'un programa.

Una altra zona de memòria o segment de dades molt similar és el *BSS*, que és una zona destinada a emmagatzemar variables estàtiques no inicialitzades. En temps d'execució aquestes variables són emplenades amb zeros fins que se'ls assigna un nou valor. En la resta del mòdul explicarem els errors relacionats amb el desbordament de *heap*, que són idèntics als ocorreguts en el *BSS*.

El *heap* es reserva, com hem dit, en compilar un programa, i és per això que depenem, més que mai, del compilador usat i de les biblioteques instal·lades. En generarem i compilarem un exemple per a comprovar com és el funcionament.

```
#include <stdio.h>
#include <string.h>

int main()
{
   static char buffer[16] = "";
   static char nom[] = "Pedro Laguna";

   gets(buffer);

   printf("Hola %s!", nom);

   return 0;
}
```

Codi vulnerable a desbordament de *heap*

Com es pot observar, l'exemple anterior no difereix gaire dels fets anteriorment, però si que hi ha una subtil diferència. En aquest codi, les variables s'han declarat com a *static* i, per tant, s'usa la zona de *heap* per a emmagatzemar les variables perquè se'n coneix la mida exacta.

Aquesta última afirmació no és del tot certa, ja que hem generat una variable *buffer* de 16 bytes a la qual no hem assignat cap valor, una cosa que farem en temps d'execució mitjançant la funció *gets*.

3.1.1. Exemple en Windows

La compilació i execució d'aquest programa ens determinarà alguns aspectes sobre la manera com s'està reservant la memòria el nostre compilador. Començarem per executar el codi anterior en el nostre entorn habitual de proves, Cygwin sobre entorn Windows amb un compilador GCC 3.4.4. Posteriorment farem el mateix però sobre una màquina amb l'Ubuntu 8.10 i el compilador GCC 4.3.2.

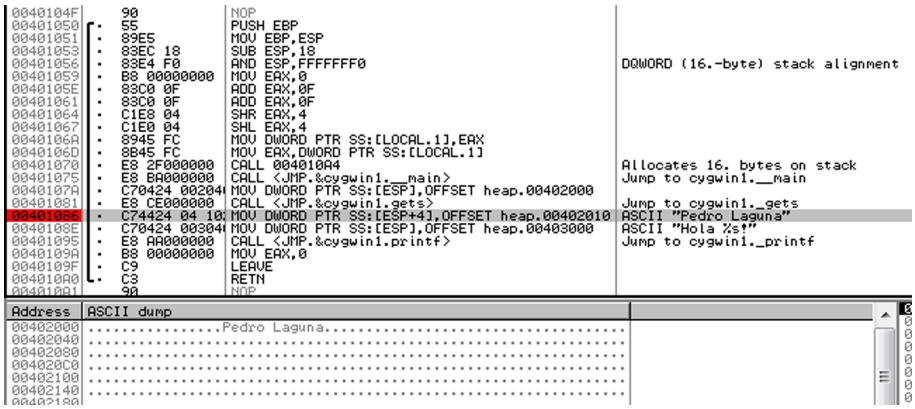
```
Pedro@seattle ~/uoc/heap
$ cat heap.c | grep -n "static"
6:      static char buffer[16] = "";
7:      static char nombre[] = "Pedro Laguna";

Pedro@seattle ~/uoc/heap
$ ./heap.exe
Texto pequeño
Hola Pedro Laguna!
Pedro@seattle ~/uoc/heap
$ ./heap.exe
Texto bastante mas grande
Hola as grande!
Pedro@seattle ~/uoc/heap
$ ./heap.exe
0123456789012345Chema Alonso
Hola Chema Alonso!
Pedro@seattle ~/uoc/heap
$ _
```

Execució del programa *heap.c* compilat pel GCC 3.4.4

Com es veu en la figura anterior, tenim un programa compilat en què la variable *buffer* està declarada abans que la variable *nom* (això es veu gràcies al paràmetre `-n` de `grep`, que ens mostra el número de línia on ha trobat la paraula `static`). En aquest cas el nostre compilador està usant una manera se-

qüencial d'emmagatzemar les variables que es declaren en el nostre codi; això vol dir que primer reservarà 16 bytes per a la variable *buffer* i posteriorment emmagatzemarà el contingut de la variable `nom`.

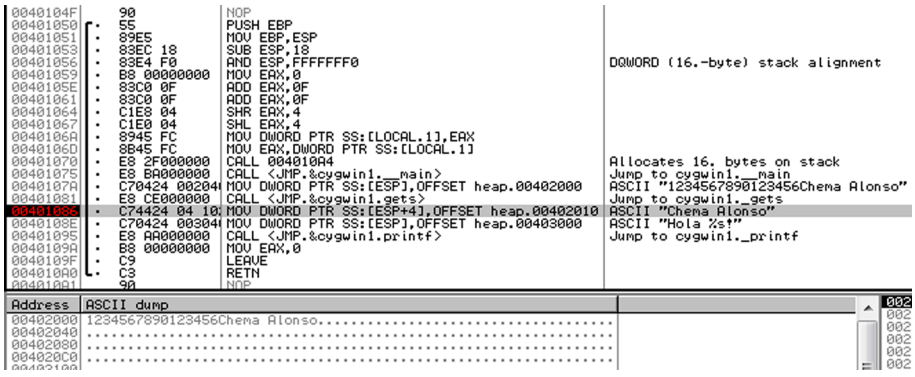
I com ocorre això vist a ulls d'un depurador com l'Ollydbg? Si arrenquem el programa des d'aquest depurador obtindrem un buidatge similar a aquest:



Programa `heap.c` sense el *heap* modificat

Com es pot observar en la línia destacada, a causa que té un punt de ruptura, el programa accedeix una posició de memòria fixa, la `00402010`, on espera trobar el valor de la variable. Si observem detingudament veurem que la memòria comença en l'adreça `00402000`. Si sumem 16 (`0x10`) al valor de la variable `buffer`, es podrà arribar fins a la posició de la variable `nom`.

Si en executar aquest programa establim com a *buffer* la cadena `1234567890123456Chema Alonso` veurem com se sobreescriu la memòria convenientment per a situar en la posició que volem el valor introduït.

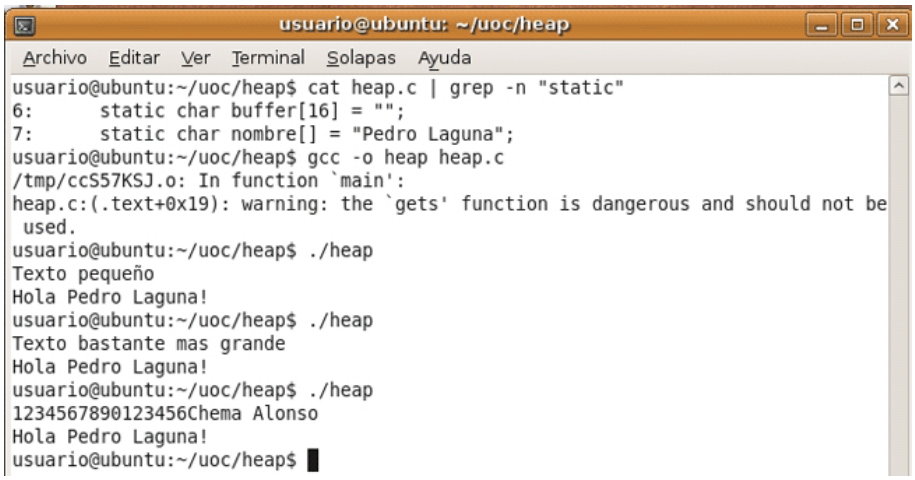


Programa `heap.c` després de la modificació del *heap*

Com es pot observar en la línia 00401086, el valor al qual es referencia ara s'ha modificat i conté actualment el nom `Chema Alonso`.

3.1.2. Exemple en Linux

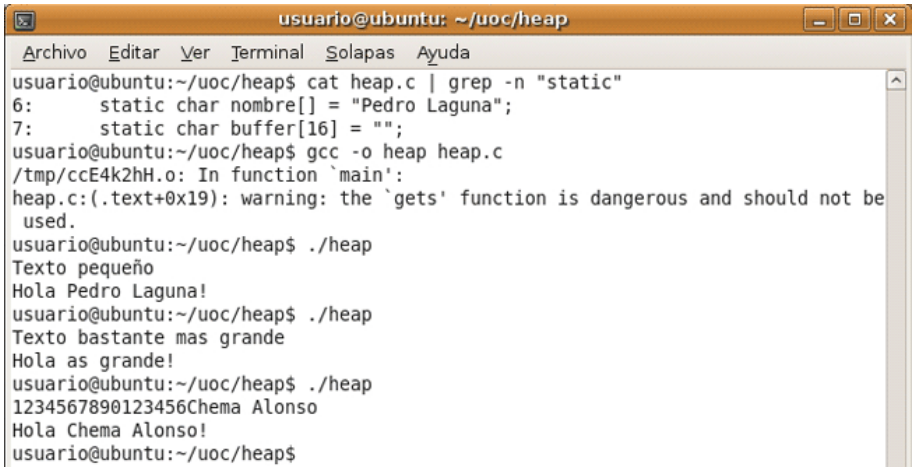
Canviarem ara d'escenari. Executarem l'Ubuntu 8.10 amb un compilador GCC 4.3.2 i veurem les subtils diferències que acompanyen el canvi de versió del compilador. Per començar, es procedirà a executar exactament el mateix programa que en l'exemple anterior.



Execució de `heap.c` en un sistema Linux

En aquest exemple que s'observa en la figura anterior hem volgut incloure la línia de la compilació del programa perquè es vegi com el GCC mateix ens adverteix que l'ús de la funció `gets()` pot donar lloc a algun tipus d'error i que l'hauríem d'evitar. De tota manera, i malgrat usar una funció no recomanada, el nostre programa sembla segur, no hi ha desbordament de la variable `buffer` que ens permeti modificar el valor de la variable `nom`.

És cert això? Doncs, lamentablement, no. En el GCC 4.3.2 les variables en el *heap* s'organitzen de manera inversa, això és, primer s'emmagatzemaria el valor de *nom* i posteriorment el de `buffer`, per la qual cosa mai no en podríem arribar a sobreescriure el valor. Això és, si canviem l'ordre en el qual es declaren les variables, el GCC, en compilar el codi, els reservarà la memòria de tal manera que la variable *buffer* quedi davant de la variable *nom*, per la qual cosa en podrem tornar a sobreescriure el valor igual que vam fer en el primer exemple.

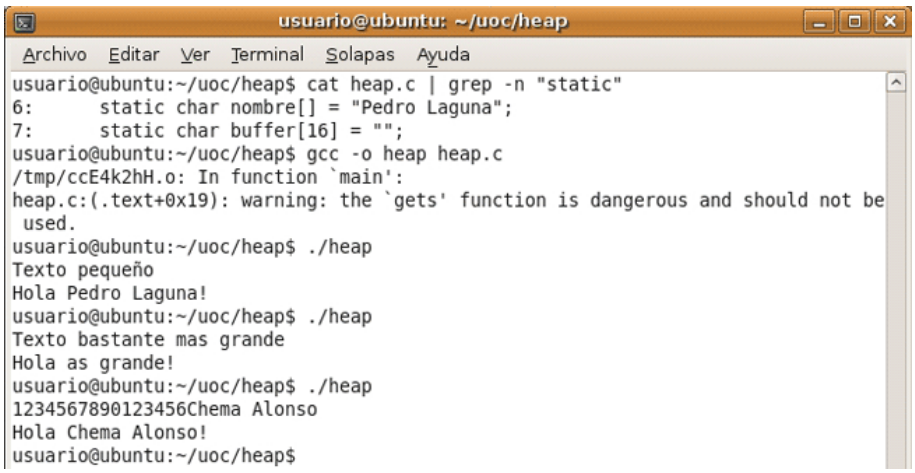


Execució del programa *heap.c* modificant l'ordre de les variables

De nou el nostre codi torna a ser vulnerable, i permet la sobreescriptura de la variable *nom* com es veu en la figura anterior.

Per a analitzar aquests canvis que es produeixen en Linux tornarem a usar el programa GDB i a aprendre alguns trucs nous. Per a això és necessari que compilem el programa `heap.c` amb l'opció `-g`, la qual cosa generarà els símbols de depuració que el GDB entendrà, interpretarà i usarà. La manera de fer això ja l'hem vista en mòduls anteriors.

Mitjançant el GDB disposem d'ordres per a localitzar les variables en memòria, per exemple mitjançant *info scope main*, que ens retornarà les variables declarades dins del mètode `main`. Amb aquesta ordre podem saber exactament on hem de localitzar els valors que busquem sense haver de recórrer la memòria a la recerca d'una cadena coneguda.



Anàlisi de les variables en memòria mitjançant el GDB

En la figura anterior hem fet una anàlisi mitjançant el GDB en què hem localitzat les posicions de memòria de les variables `nom` i `buffer` i, com es pot observar, la posició de *buffer* és més baixa que la de *nom*. El buidatge següent

de la zona de memòria on es troben les podivionrd demostra inequívocament com el creixement (recordem, al contrari que en la pila) excessiu de la variable *buffer* pot donar lloc a una sobreescriptura de la variable *nom*.