IF2130 – Organisasi dan Arsitektur Komputer

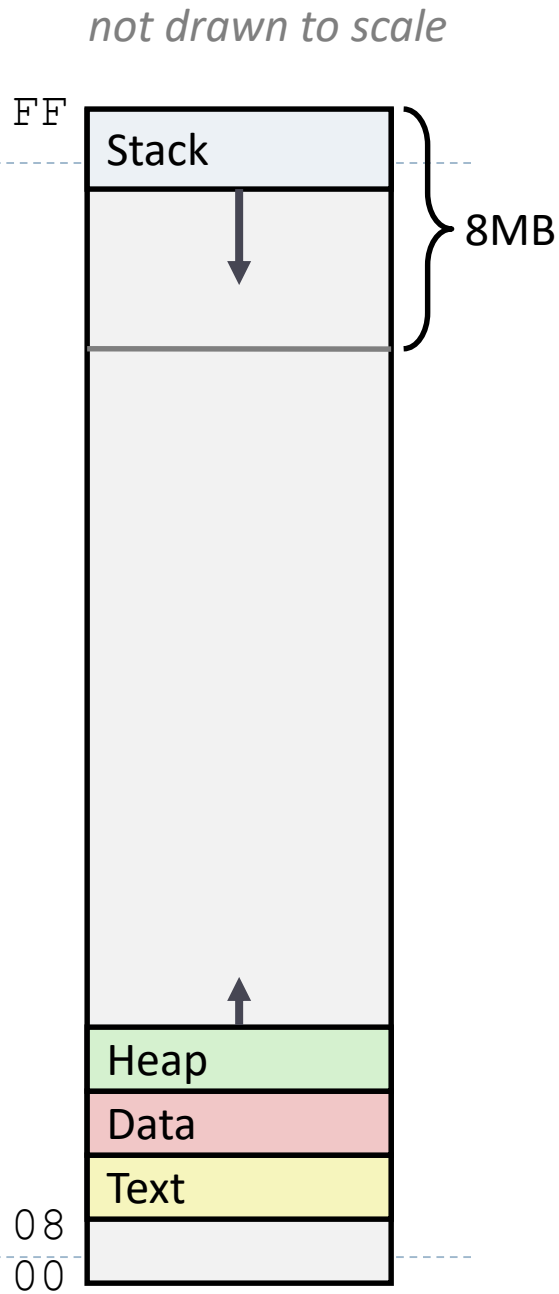
sumber: Greg Kesden, CMU 15-213, 2012

Machine-Level Programming: Memory Layout dan Buffer Overflow

Achmad Imam Kistijantoro (imam@staff.stei.itb.ac.id)
Robithoh Annur
Bara Timur
Monterico Adrian

IA32 Linux Memory Layout

- Stack
  - Runtime stack (8MB limit)
  - E. g., local variables
- Heap
  - Dynamically allocated storage
  - When call malloc(), calloc(), new()
- Data
  - Statically allocated data
  - E.g., arrays & strings declared in code
- Text
  - Executable machine instructions
  - Read-only



Memory Allocation Example

not drawn to scale

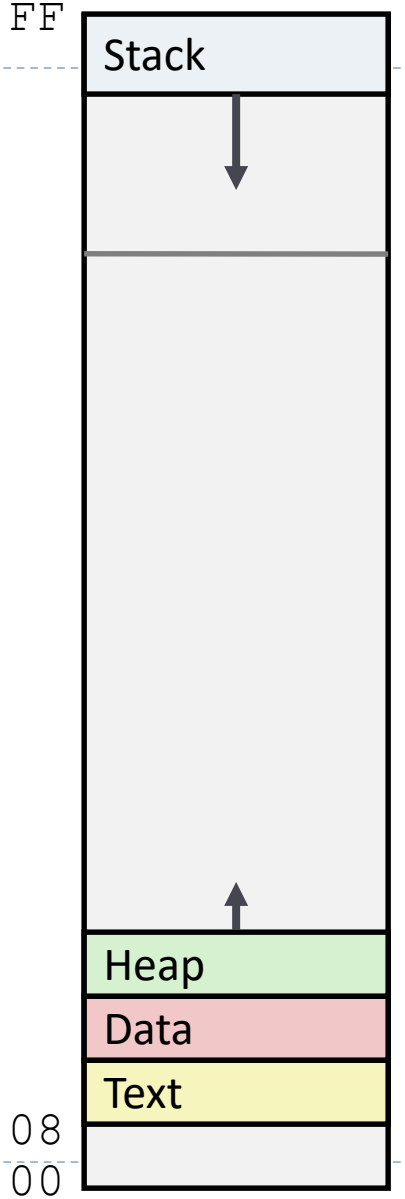
```
char big_array[1<<24];  /*  16 MB */
char huge_array[1<<28]; /* 256 MB */

int beyond;
char *p1, *p2, *p3, *p4;

int useless() {  return 0; }

int main()
{
 p1 = malloc(1 <<28);  /* 256 MB */
 p2 = malloc(1 << 8);  /* 256 B  */
 p3 = malloc(1 <<28);  /* 256 MB */
 p4 = malloc(1 << 8);  /* 256 B  */
 /* Some print statements ... */
}
```

Where does everything go?



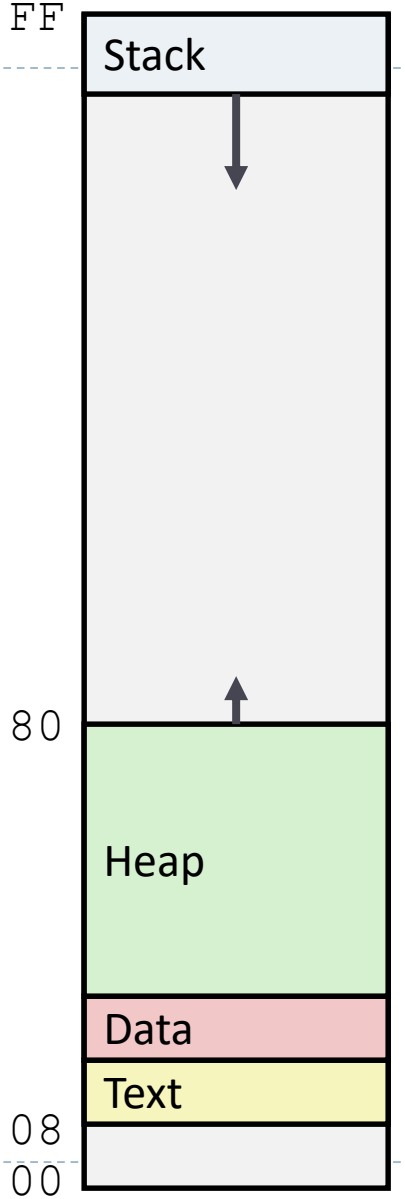
IA32 Example Addresses

not drawn to scale

address range ~2^{32}

`$esp`	`0xffffbcd0`
`p3`	`0x65586008`
`p1`	`0x55585008`
`p4`	`0x1904a110`
`p2`	`0x1904a008`
`&p2`	`0x18049760`
`&beyond`	`0x08049744`
`big_array`	`0x18049780`
`huge_array`	`0x08049760`
`main()`	`0x080483c6`
`useless()`	`0x08049744`
final `malloc()`	`0x006be166`

`malloc()` is dynamically linked
address determined at runtime



```
imam@DELL-2020:~/if2130$ more memory_alloc.c
#include <stdio.h>
#include <stdlib.h>

char big_array[1<<24];
char huge_array[1<<28];

int beyond;
char *p1, *p2, *p3, *p4;

int useless() { return 0; }

int main() {
        p1 = malloc(1<<28);
        p2 = malloc(1<<8);
        p3 = malloc(1<<28);
        p4 = malloc(1<<8);

        register long i asm("rsp");

        printf("rsp : %#010lx\n", i);
        printf("p3  : %#010lx\n", (long) p3);
        printf("p1  : %#010lx\n", (long) p1);
        printf("p4  : %#010lx\n", (long) p4);
        printf("p2  : %#010lx\n", (long) p2);
        printf("&p2 : %#010lx\n", (long) &p2);
        printf("&beyond: %#010lx\n", (long) &beyond);
        printf("big_array : %#010lx\n", (long) big_array);
        printf("huge_array : %#010lx\n", (long) huge_array);
        printf("main() : %#010lx\n", (long) main);
        printf("useless() : %#010lx\n", (long) useless);
        printf("malloc() : %#010lx\n", (long) malloc);

        getchar();
}
```

```
imam@DELL-2020:~/if2130$ ./memory_alloc
rsp         : 0x7ffe71da4d10
p3          : 0x7f72940cd010
p1          : 0x7f72a40ce010
p4          : 0x55da49fa03b0
p2          : 0x55da49fa02a0
&p2         : 0x55da38981040
&beyond     : 0x55da49981080
big_array   : 0x55da48981080
huge_array  : 0x55da38981060
main()      : 0x55da3897e188
useless()   : 0x55da3897e179
malloc()    : 0x7f72b416c260
```

```
imam@DELL-2020:~/if2130$ more /proc/412/maps
55da3897d000-55da3897e000 r--p 00000000 08:10 483032                     /home/imam/if2130/memory_alloc
55da3897e000-55da3897f000 r-xp 00001000 08:10 483032                     /home/imam/if2130/memory_alloc
55da3897f000-55da38980000 r--p 00002000 08:10 483032                     /home/imam/if2130/memory_alloc
55da38980000-55da38981000 r--p 00002000 08:10 483032                     /home/imam/if2130/memory_alloc
55da38981000-55da38982000 rw-p 00003000 08:10 483032                     /home/imam/if2130/memory_alloc
55da38982000-55da49982000 rw-p 00000000 00:00 0
55da49fa0000-55da49fc1000 rw-p 00000000 00:00 0                          [heap]
7f72940cd000-7f72b40cf000 rw-p 00000000 00:00 0
7f72b40cf000-7f72b40f4000 r--p 00000000 08:10 30676                      /lib/x86_64-linux-gnu/libc-2.31.so
7f72b40f4000-7f72b426c000 r-xp 00025000 08:10 30676                      /lib/x86_64-linux-gnu/libc-2.31.so
7f72b426c000-7f72b42b6000 r--p 0019d000 08:10 30676                      /lib/x86_64-linux-gnu/libc-2.31.so
7f72b42b6000-7f72b42b7000 ---p 001e7000 08:10 30676                      /lib/x86_64-linux-gnu/libc-2.31.so
7f72b42b7000-7f72b42ba000 r--p 001e7000 08:10 30676                      /lib/x86_64-linux-gnu/libc-2.31.so
7f72b42ba000-7f72b42bd000 rw-p 001ea000 08:10 30676                      /lib/x86_64-linux-gnu/libc-2.31.so
7f72b42bd000-7f72b42c3000 rw-p 00000000 00:00 0
7f72b42d3000-7f72b42d4000 r--p 00000000 08:10 30668                      /lib/x86_64-linux-gnu/ld-2.31.so
7f72b42d4000-7f72b42f7000 r-xp 00001000 08:10 30668                      /lib/x86_64-linux-gnu/ld-2.31.so
7f72b42f7000-7f72b42ff000 r--p 00024000 08:10 30668                      /lib/x86_64-linux-gnu/ld-2.31.so
7f72b4300000-7f72b4301000 r--p 0002c000 08:10 30668                      /lib/x86_64-linux-gnu/ld-2.31.so
7f72b4301000-7f72b4302000 rw-p 0002d000 08:10 30668                      /lib/x86_64-linux-gnu/ld-2.31.so
7f72b4302000-7f72b4303000 rw-p 00000000 00:00 0
7ffe71d86000-7ffe71da7000 rw-p 00000000 00:00 0                          [stack]
7ffe71dda000-7ffe71ddd000 r--p 00000000 00:00 0                          [vvar]
7ffe71ddd000-7ffe71ddf000 r-xp 00000000 00:00 0                          [vdso]
```

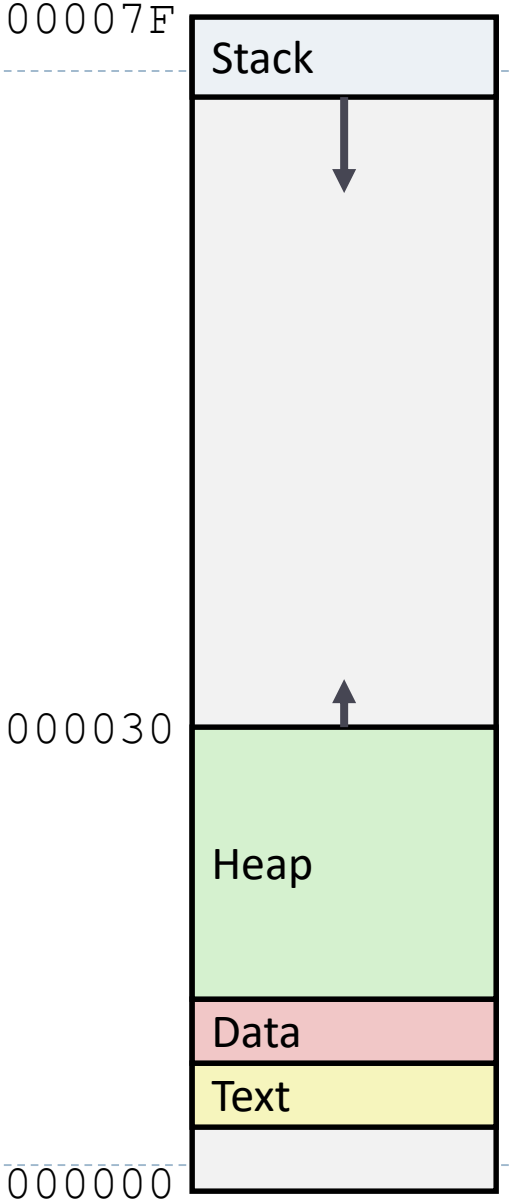
not drawn to scale

x86-64 Example Addresses

address range ~2^{47}

`$rsp`	`0x00007fffffff8d1f8`
`p3`	`0x00002aaabaadd010`
`p1`	`0x00002aaaaaadc010`
`p4`	`0x0000000011501120`
`p2`	`0x0000000011501010`
`&p2`	`0x0000000010500a60`
`&beyond`	`0x0000000000500a44`
`big_array`	`0x0000000010500a80`
`huge_array`	`0x0000000000500a50`
`main()`	`0x0000000000400510`
`useless()`	`0x0000000000400500`
final `malloc()`	`0x000000386ae6a170`

`malloc()` is dynamically linked
address determined at runtime

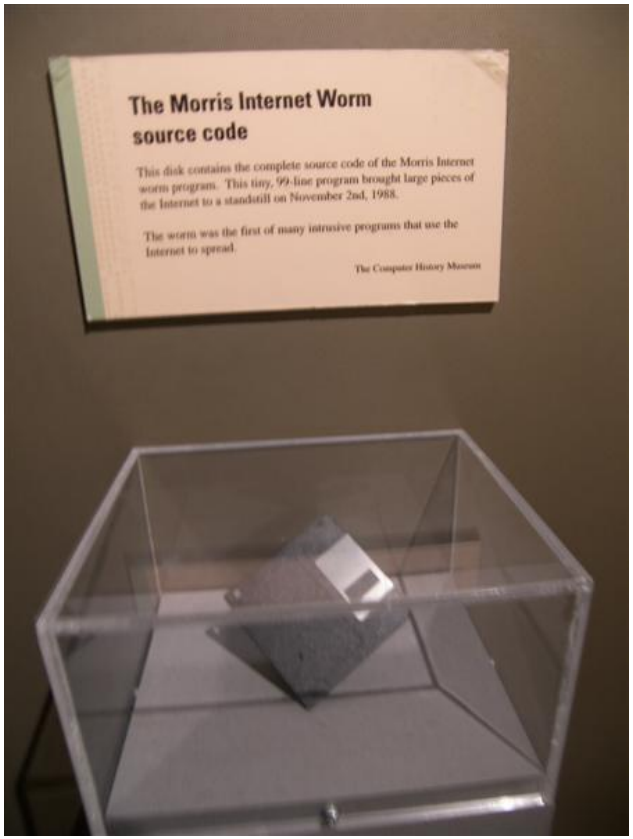


Today

- Structures
  - Alignment
- Unions
- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection

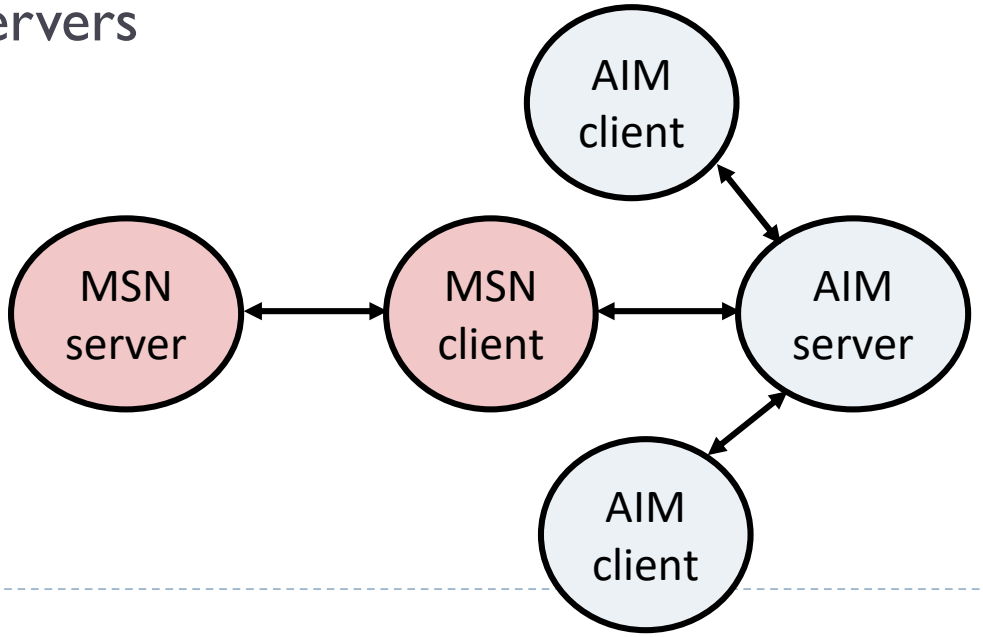
Internet Worm and IM War

- November, 1988
  - Internet Worm attacks thousands of Internet hosts.
  - How did it happen?



Internet Worm and IM War

- November, 1988
  - Internet Worm attacks thousands of Internet hosts.
  - How did it happen?
- July, 1999
  - Microsoft launches MSN Messenger (instant messaging system).
  - Messenger clients can access popular AOL Instant Messaging Service (AIM) servers



Internet Worm and IM War (cont.)

- August 1999
  - Mysteriously, Messenger clients can no longer access AIM servers.
  - Microsoft and AOL begin the IM war:
    - AOL changes server to disallow Messenger clients
    - Microsoft makes changes to clients to defeat AOL changes.
    - At least 13 such skirmishes.
  - How did it happen?

- The Internet Worm and AOL/Microsoft War were both based on *stack buffer overflow* exploits!
  - many library functions do not check argument sizes.
  - allows target buffers to overflow.

String Library Code

- Implementation of Unix function `gets()`

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

  - No way to specify limit on number of characters to read
- Similar problems with other library functions
  - **`strcpy`**, **`strcat`**: Copy strings of arbitrary length
  - **`scanf`**, **`fscanf`**, **`sscanf`**, when given **`%s`** conversion specification

Vulnerable Buffer Code

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
void call_echo() {
    echo();
}
```

```
unix>./bufdemo
Type a string:1234567
1234567
```

```
unix>./bufdemo
Type a string:12345678
Segmentation Fault
```

```
unix>./bufdemo
Type a string:123456789ABC
Segmentation Fault
```

Buffer Overflow Disassembly

echo:

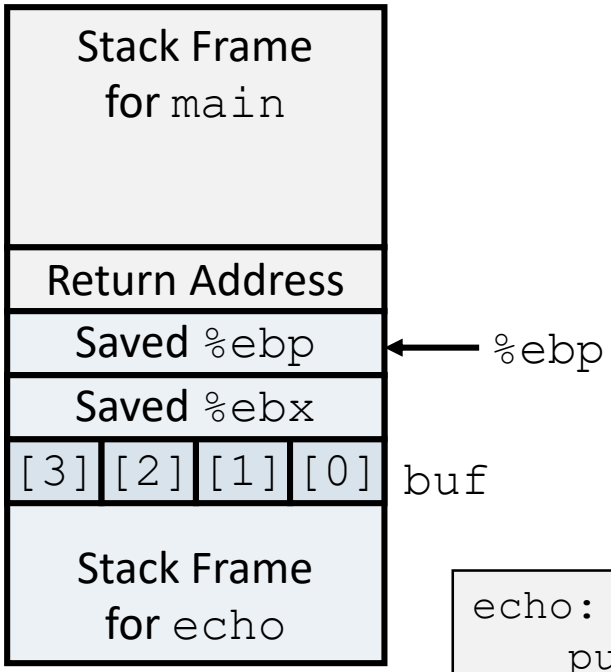
```
80485c5:  55                    push   %ebp
80485c6:  89 e5                 mov    %esp,%ebp
80485c8:  53                    push   %ebx
80485c9:  83 ec 14              sub    $0x14,%esp
80485cc:  8d 5d f8              lea 0xfffffff8(%ebp),%ebx
80485cf:  89 1c 24              mov    %ebx,(%esp)
80485d2:  e8 9e ff ff ff        call   8048575 <gets>
80485d7:  89 1c 24              mov    %ebx,(%esp)
80485da:  e8 05 fe ff ff        call   80483e4 <puts@plt>
80485df:  83 c4 14              add    $0x14,%esp
80485e2:  5b                    pop    %ebx
80485e3:  5d                    pop    %ebp
80485e4:  c3                    ret
```

call_echo:

```
80485eb:  e8 d5 ff ff ff        call   80485c5 <echo>
80485f0:  c9                    leave
80485f1:  c3                    ret
```

Buffer Overflow Stack

Before call to gets



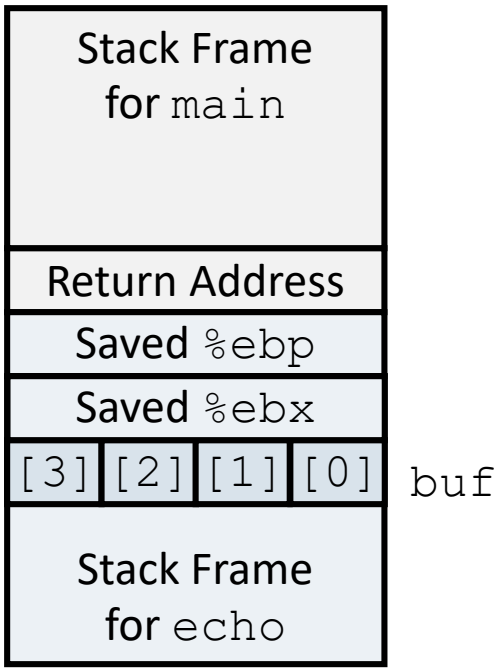

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    pushl %ebp              # Save %ebp on stack
    movl  %esp, %ebp
    pushl %ebx              # Save %ebx
    subl  $20, %esp         # Allocate stack space
    leal  -8(%ebp),%ebx     # Compute buf as %ebp-8
    movl  %ebx, (%esp)      # Push buf on stack
    call  gets              # Call gets
    . . .
```

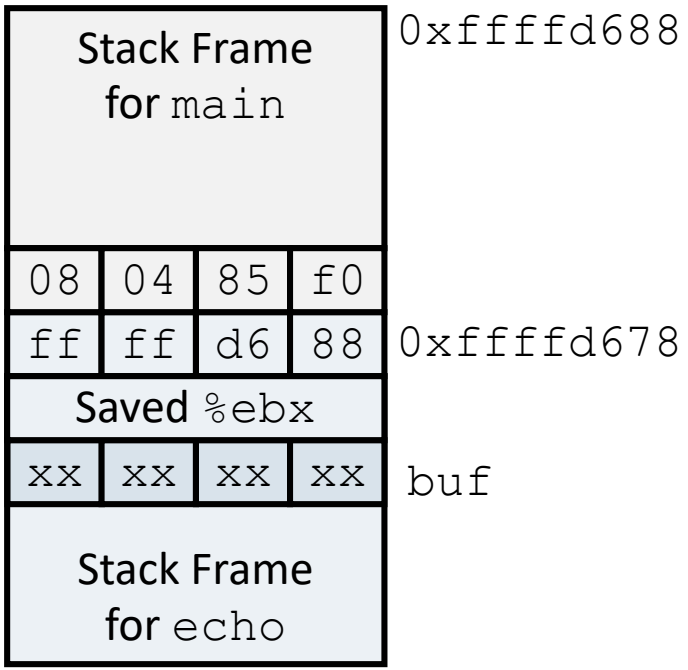
Buffer Overflow Stack Example

```
unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x80485c9
(gdb) run
Breakpoint 1, 0x80485c9 in echo ()
(gdb) print /x $ebp
$1 = 0xffffd678
(gdb) print /x *(unsigned *)$ebp
$2 = 0xffffd688
(gdb) print /x *((unsigned *)$ebp + 1)
$3 = 0x80485f0
```

Before call to gets



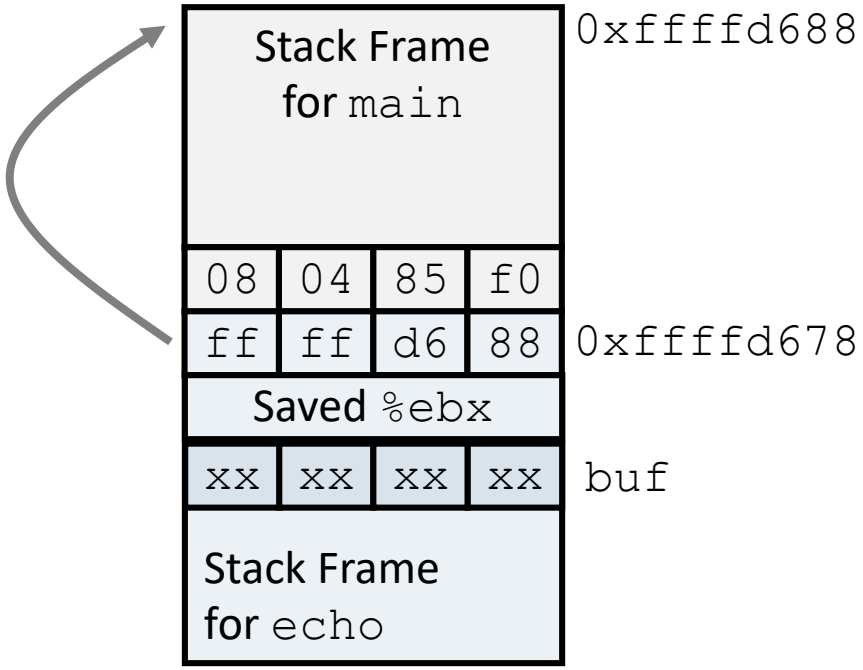
Before call to gets



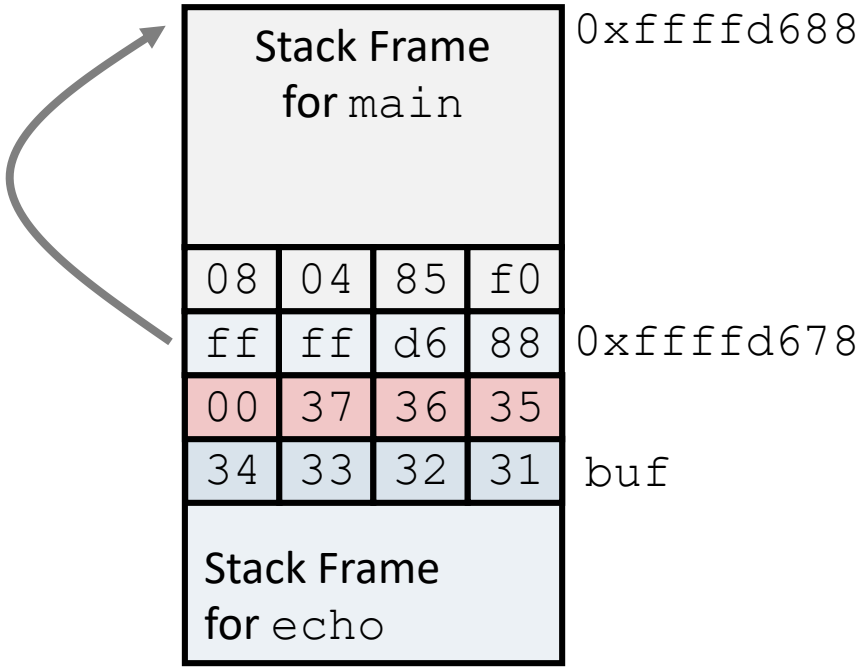

```
80485eb:  e8 d5 ff ff ff    call   80485c5 <echo>
80485f0:  c9                leave
```

Buffer Overflow Example #1

Before call to gets



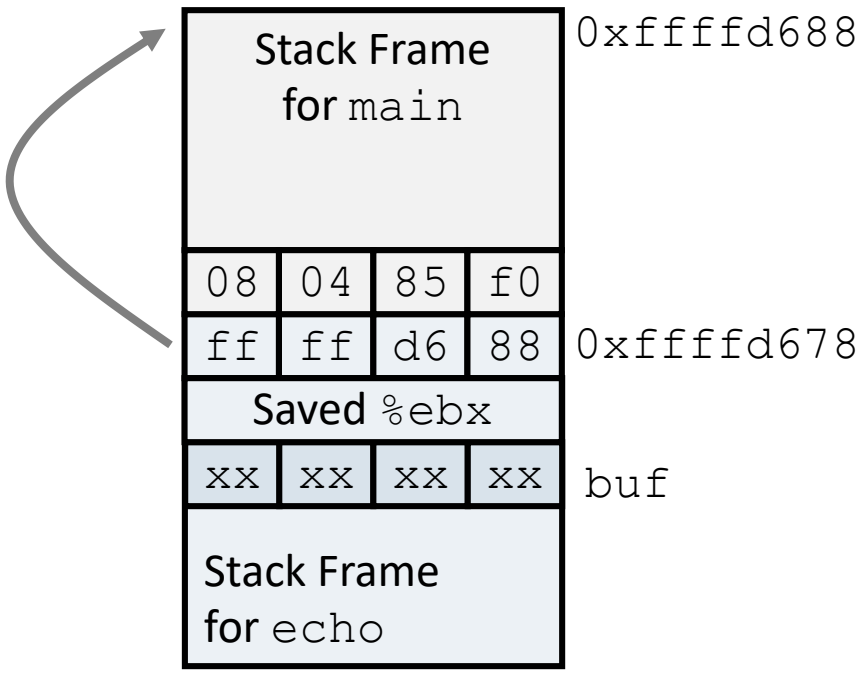
Input 1234567



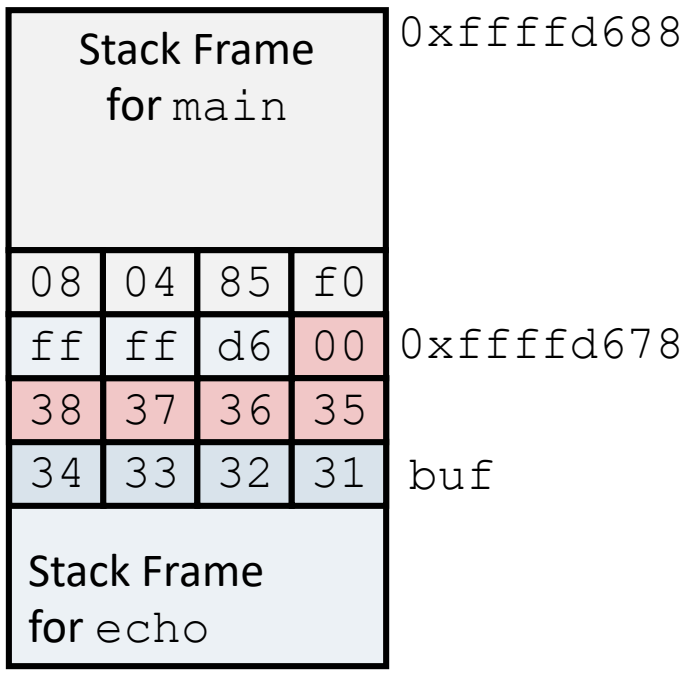
Overflow buf, and corrupt %ebx,
but no problem

Buffer Overflow Example #2

Before call to gets



Input 12345678

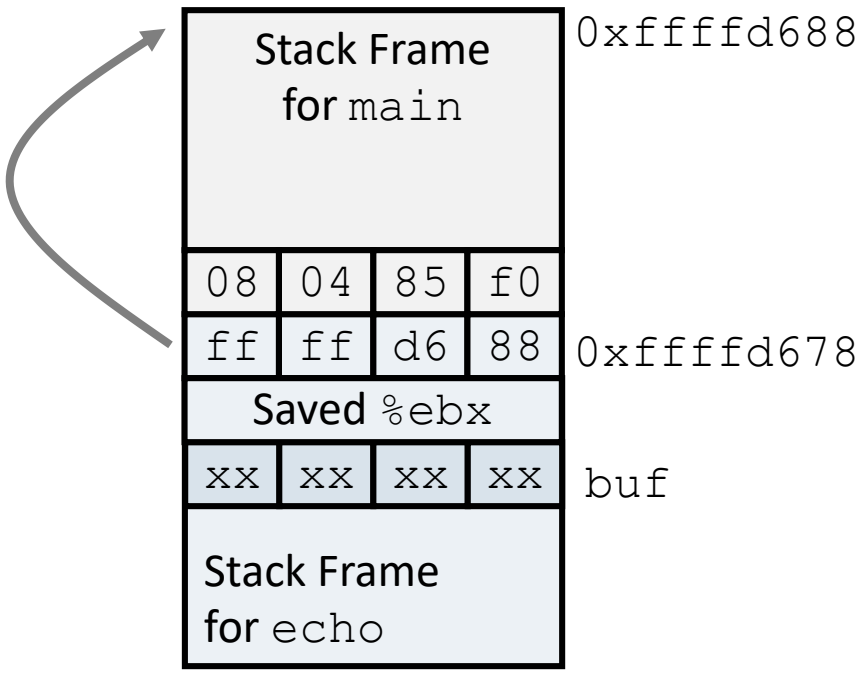


Base pointer corrupted

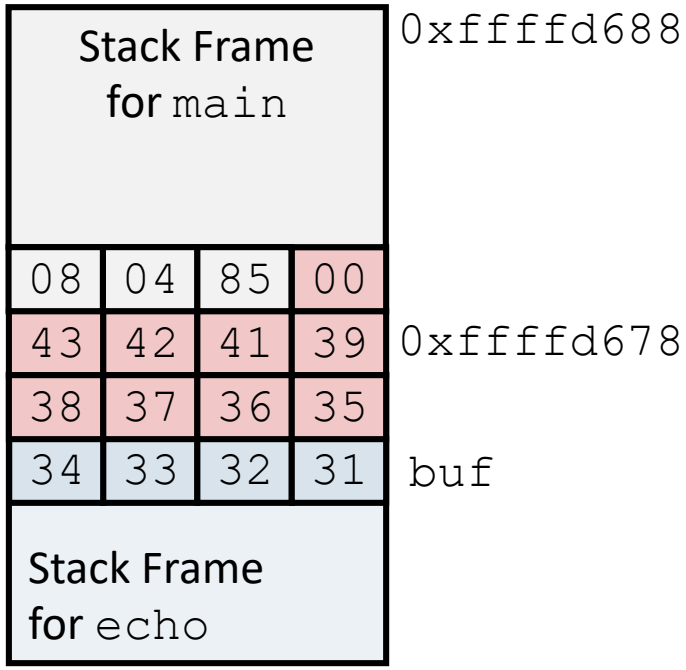
```
. . .
80485eb:    e8 d5 ff ff ff    call   80485c5 <echo>
80485f0:    c9                leave    # Set %ebp to corrupted value
80485f1:    c3                ret
```

Buffer Overflow Example #3

Before call to gets



Input 123456789ABC



Return address corrupted

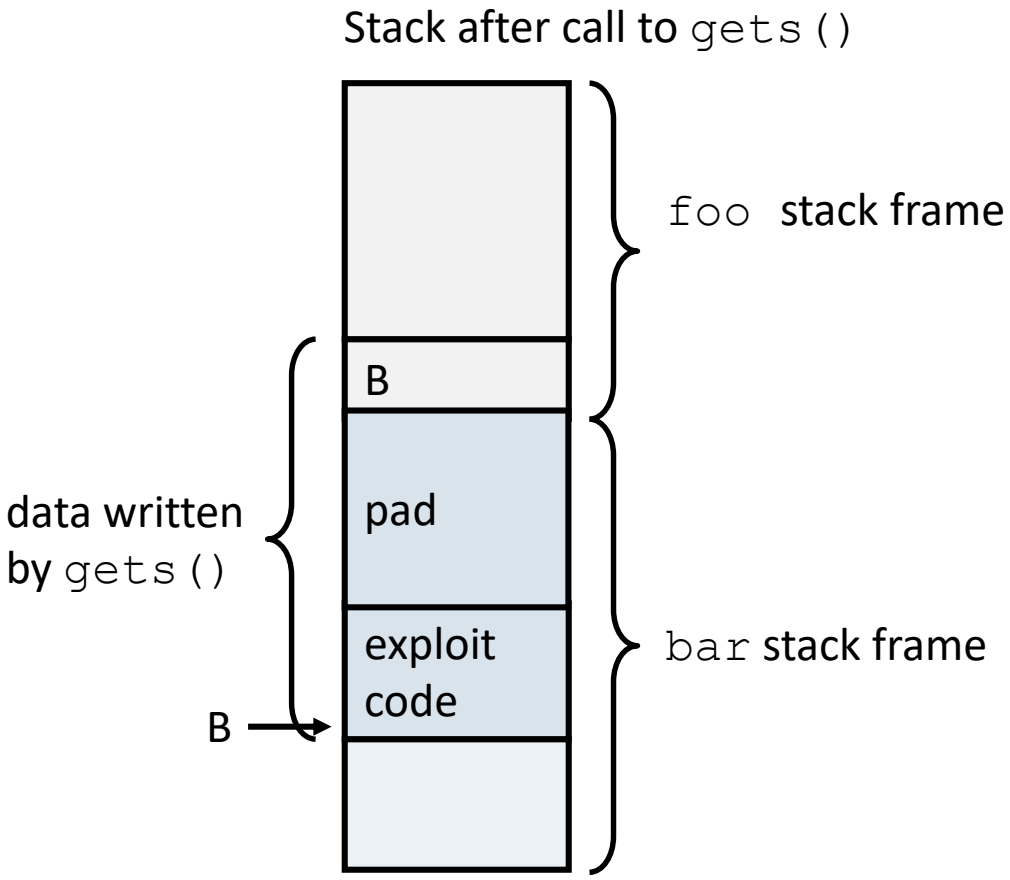
```
80485eb:    e8 d5 ff ff ff    call   80485c5 <echo>
80485f0:    c9                leave    # Desired return point
```

Malicious Use of Buffer Overflow

```
void foo(){
  bar();
  ...
}
```

return address A

```
int bar() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```



- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer B
- When `bar()` executes `ret`, will jump to exploit code

Exploits Based on Buffer Overflows

- *Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines*
- Internet worm
  - Early versions of the finger server (fingerd) used **`gets()`** to read the argument sent by the client:
    - **`finger droh@cs.cmu.edu`**
  - Worm attacked fingerd server by sending phony argument:
    - **`finger "exploit-code padding new-return-address"`**
    - exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker.

Exploits Based on Buffer Overflows

- *Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines*
- IM War
  - AOL exploited existing buffer overflow bug in AIM clients
  - exploit code: returned 4-byte signature (the bytes at some location in the AIM client) to server.
  - When Microsoft changed code to match signature, AOL changed signature location.

Date: Wed, 11 Aug 1999 11:30:57 -0700 (PDT)
From: Phil Bucking <philbucking@yahoo.com>
Subject: AOL exploiting buffer overrun bug in their own software!
To: rms@pharlap.com

Mr. Smith,

I am writing you because I have discovered something that I think you might find interesting because you are an Internet security expert with experience in this area. I have also tried to contact AOL but received no response.

I am a developer who has been working on a revolutionary new instant messaging client that should be released later this year.
...
It appears that the AIM client has a buffer overrun bug. By itself this might not be the end of the world, as MS surely has had its share. But AOL is now *exploiting their own buffer overrun bug* to help in its efforts to block MS Instant Messenger.
....
Since you have significant credibility with the press I hope that you can use this information to help inform people that behind AOL's friendly exterior they are nefariously compromising peoples' security.

Sincerely,
Phil Bucking
Founder, Bucking Consulting
philbucking@yahoo.com

It was later determined that this email originated from within Microsoft!

Avoiding Overflow Vulnerability

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```

- Use library routines that limit string lengths
  - **`fgets`** instead of **`gets`**
  - **`strncpy`** instead of **`strcpy`**
  - Don't use **`scanf`** with **`%s`** conversion specification
    - Use **`fgets`** to read the string
    - Or use **`%ns`** where **`n`** is a suitable integer

System-Level Protections

- Randomized stack offsets
  - At start of program, allocate random amount of space on stack
  - Makes it difficult for hacker to predict beginning of inserted code

```
unix> gdb bufdemo
(gdb) break echo

(gdb) run
(gdb) print /x $ebp
$1 = 0xffffc638

(gdb) run
(gdb) print /x $ebp
$2 = 0xffffbb08

(gdb) run
(gdb) print /x $ebp
$3 = 0xffffc6a8
```

- Nonexecutable code segments
  - In traditional x86, can mark region of memory as either "read-only" or "writeable"
    - Can execute anything readable
  - X86-64 added explicit "execute" permission

Stack Canaries

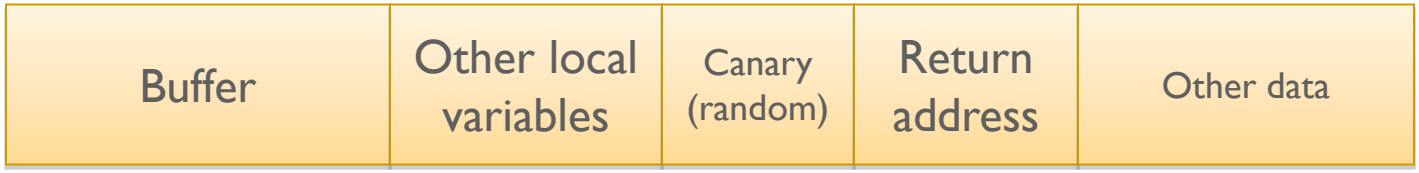
- Idea
  - Place special value ("canary") on stack just beyond buffer
  - Check for corruption before exiting function
- GCC Implementation
  - **`-fstack-protector`**
  - **`-fstack-protector-all`**

```
unix>./bufdemo-protected
Type a string:1234
1234
```

```
unix>./bufdemo-protected
Type a string:12345
*** stack smashing detected ***
```

Stack-based buffer overflow detection using a random canary

Normal (safe) stack configuration:



Buffer overflow attack attempt:

- The canary is placed in the stack prior to the return address, so that any attempt to over-write the return address also over-writes the canary.

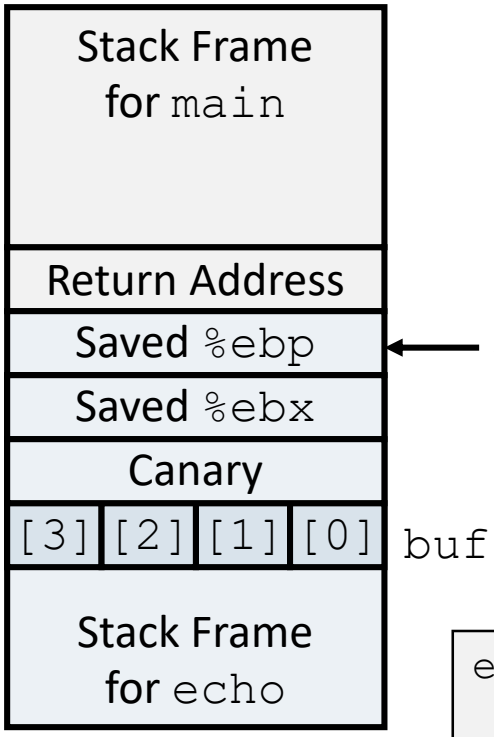
Protected Buffer Disassembly

echo:

```
804864d:   55                          push   %ebp
 804864e:  89 e5                       mov    %esp,%ebp
 8048650:  53                          push   %ebx
 8048651:  83 ec 14                    sub    $0x14,%esp
 8048654:  65 a1 14 00 00 00           mov    %gs:0x14,%eax
 804865a:  89 45 f8                    mov    %eax,0xfffffff8(%ebp)
 804865d:  31 c0                       xor    %eax,%eax
 804865f:  8d 5d f4                    lea    0xfffffff4(%ebp),%ebx
 8048662:  89 1c 24                    mov    %ebx,(%esp)
 8048665:  e8 77 ff ff ff              call   80485e1 <gets>
 804866a:  89 1c 24                    mov    %ebx,(%esp)
 804866d:  e8 ca fd ff ff              call   804843c <puts@plt>
 8048672:  8b 45 f8                    mov    0xfffffff8(%ebp),%eax
 8048675:  65 33 05 14 00 00 00        xor    %gs:0x14,%eax
 804867c:  74 05                       je     8048683 <echo+0x36>
 804867e:  e8 a9 fd ff ff              call   804842c <FAIL>
 8048683:  83 c4 14                    add    $0x14,%esp
 8048686:  5b                          pop    %ebx
 8048687:  5d                          pop    %ebp
 8048688:  c3                          ret
```

Setting Up Canary

Before call to gets

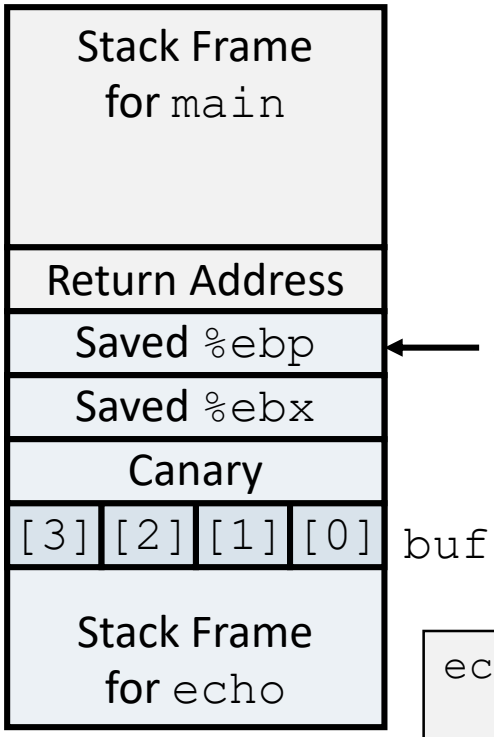



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    . . .
    movl    %gs:20, %eax      # Get canary
    movl    %eax, -8(%ebp)    # Put on stack
    xorl    %eax, %eax        # Erase canary
    . . .
```

Checking Canary

Before call to gets

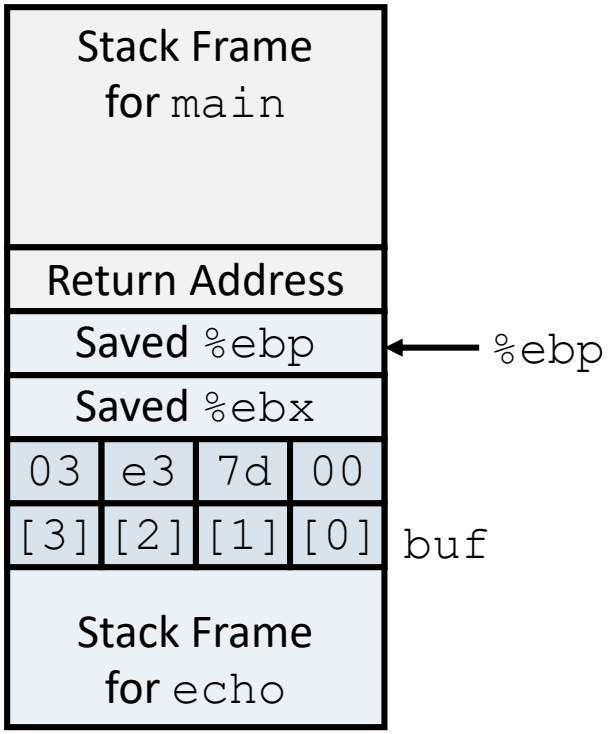



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

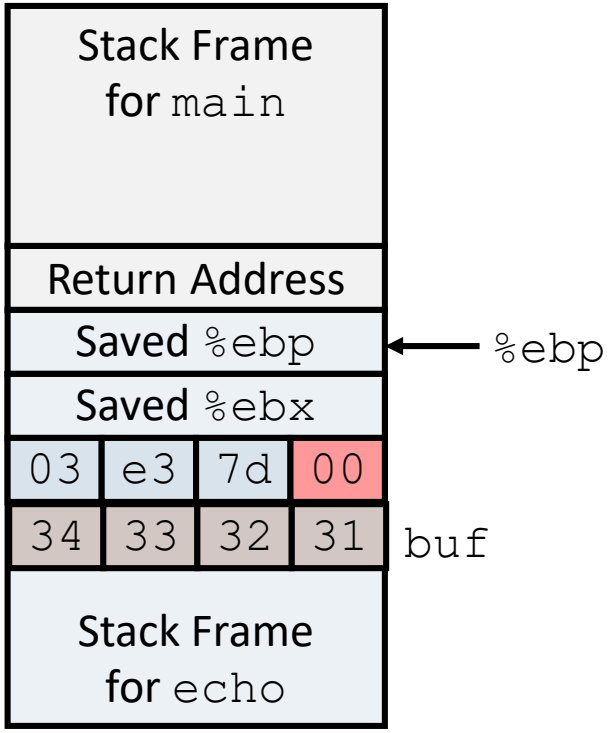
```
echo:
    . . .
    movl    -8(%ebp), %eax    # Retrieve from stack
    xorl    %gs:20, %eax      # Compare with Canary
    je      .L24              # Same: skip ahead
    call    __stack_chk_fail  # ERROR
.L24:
    . . .
```

Canary Example

Before call to gets



Input 1234




```
(gdb) break echo
(gdb) run
(gdb) stepi 3
(gdb) print /x *((unsigned *) $ebp - 2)
$1 = 0x3e37d00
```

Benign corruption!
(allows programmers to make silent off-by-one errors)

Worms and Viruses

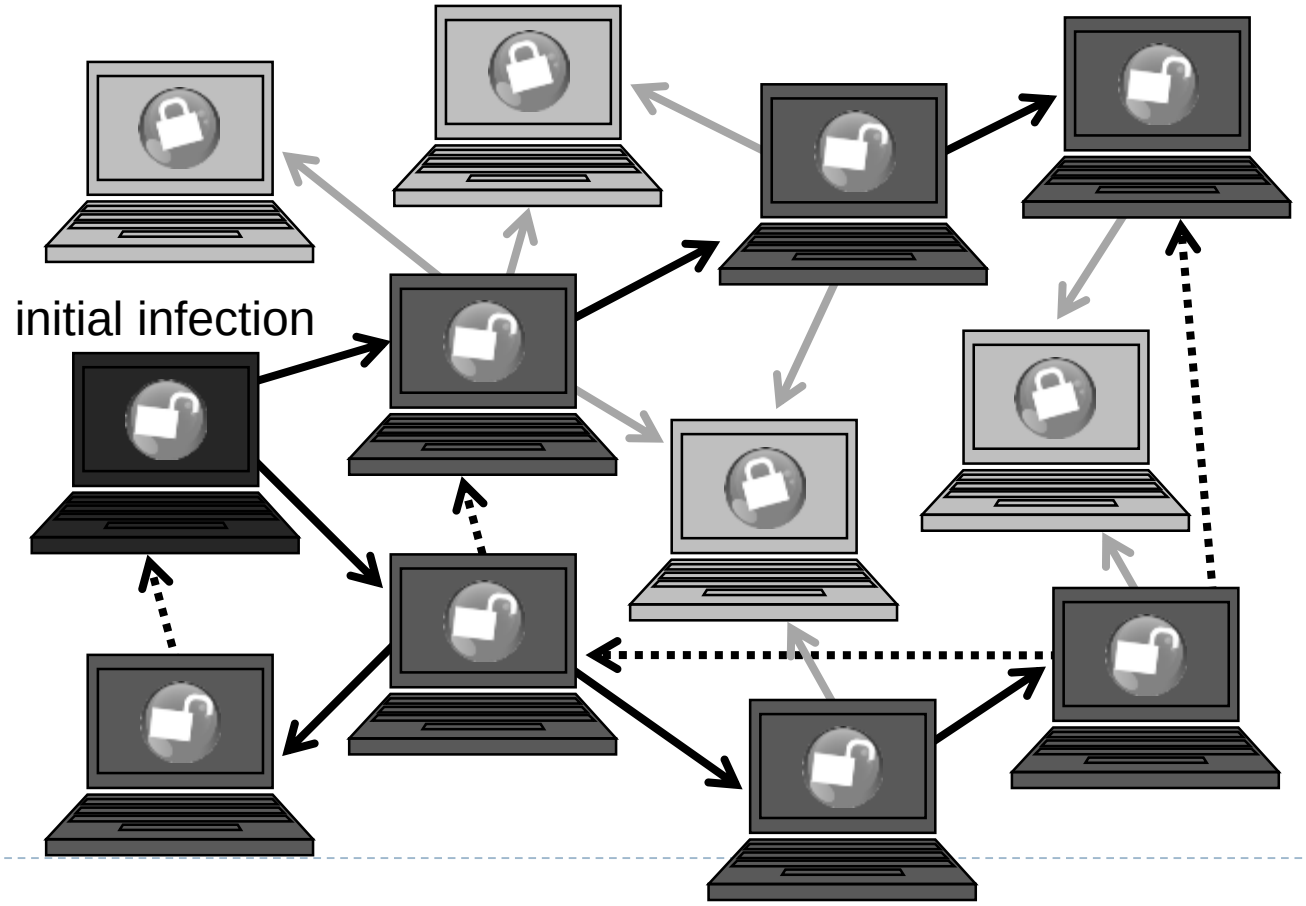
- Worm: A program that
  - Can run by itself
  - Can propagate a fully working version of itself to other computers
- Virus: Code that
  - Add itself to other programs
  - Cannot run independently
- Both are (usually) designed to spread among computers and to wreak havoc

Computer Worms

- **Computer worm** is a malware program that spreads copies of itself into other computers through computer network.
- Computer worms are technically not computer viruses (since they don't infect other programs), but some people nevertheless confuse the terms, since both spread by self-replication.
- Computer worm will carry a malicious payload, such as deleting files or installing a backdoor.

Worm Propagation

- Worms propagate by finding and infecting vulnerable hosts.
  - They need a way to tell if a host is vulnerable
  - They need a way to tell if a host is already infected.



Computer Viruses

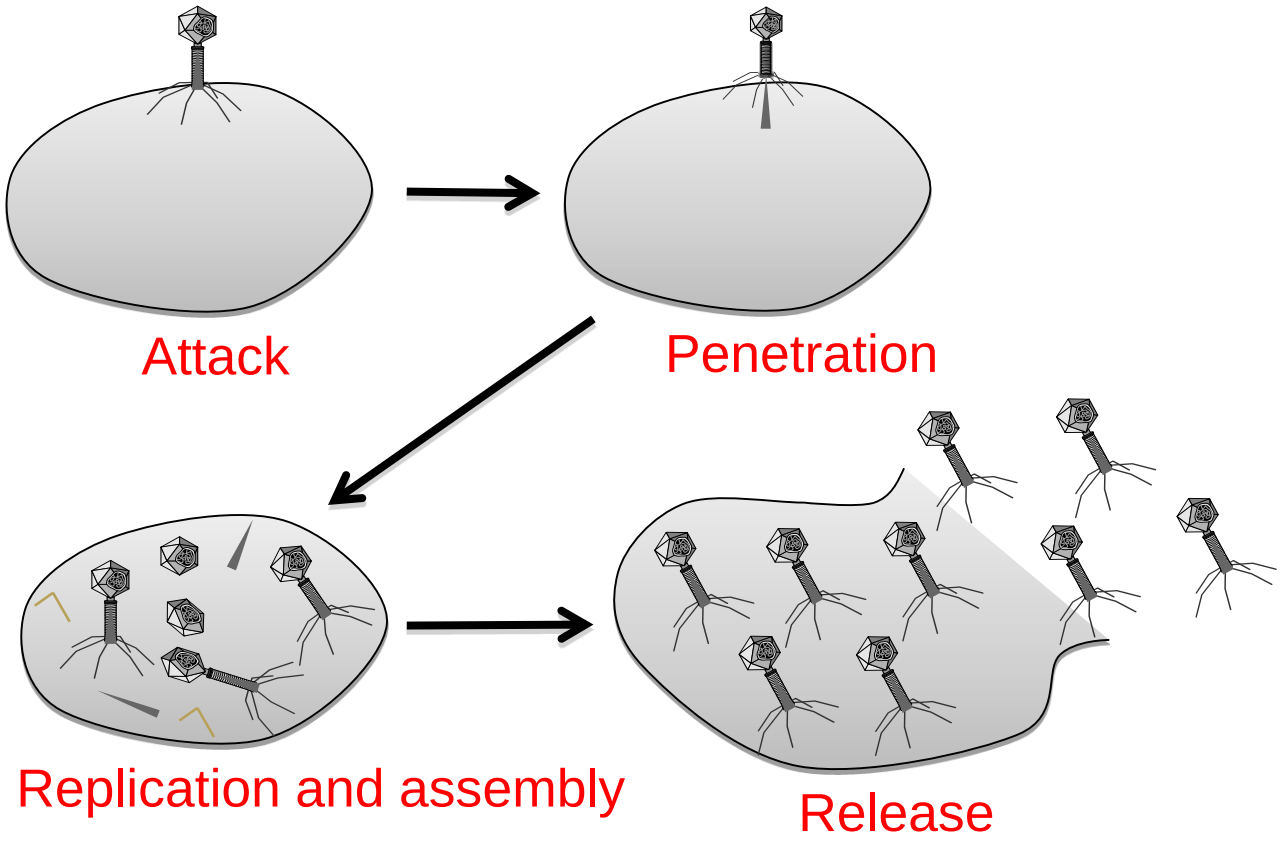
- A **computer virus** is computer code that can replicate itself by modifying other files or programs to insert code that is capable of further replication.
- This self-replication property is what distinguishes computer viruses from other kinds of malware, such as logic bombs.
- Another distinguishing property of a virus is that replication requires some type of **user assistance,** such as clicking on an email attachment or sharing a USB drive.

Computer Viruses

- A **computer virus** is computer code that can replicate itself by modifying other files or programs to insert code that is capable of further replication.
- This self-replication property is what distinguishes computer viruses from other kinds of malware, such as logic bombs.
- Another distinguishing property of a virus is that replication requires some type of **user assistance,** such as clicking on an email attachment or sharing a USB drive.

Biological Analogy

- Computer viruses share some properties with Biological viruses

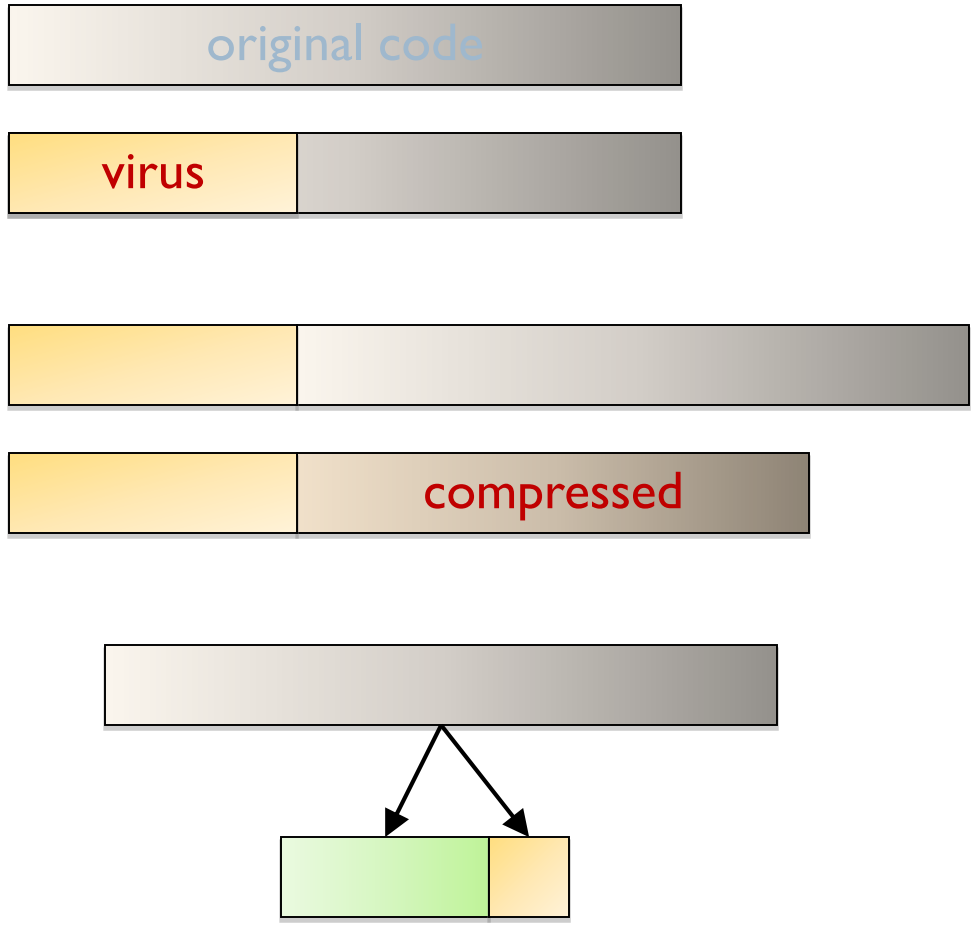


Virus Phases

- **Dormant phase.** During this phase, the virus just exists—the virus is laying low and avoiding detection.
- **Propagation phase.** During this phase, the virus is replicating itself, infecting new files on new systems.
- **Triggering phase.** In this phase, some logical condition causes the virus to move from a dormant or propagation phase to perform its intended action.
- **Action phase.** In this phase, the virus performs the malicious action that it was designed to perform, called **payload.**
  - This action could include something seemingly innocent, like displaying a silly picture on a computer's screen, or something quite malicious, such as deleting all essential files on the hard drive.

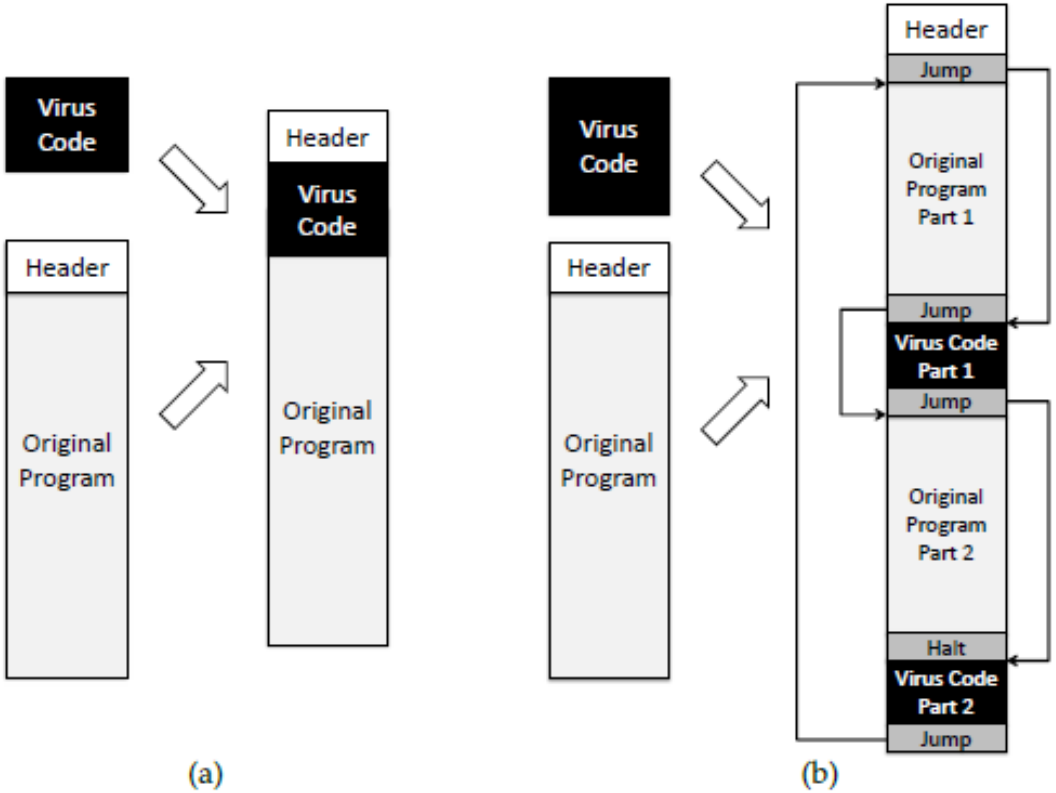
Infection Types

- Overwriting
  - Destroys original code
- Pre-pending
  - Keeps original code, possibly compressed
- Infection of libraries
  - Allows virus to be memory resident
  - E.g., kernel32.dll
- Macro viruses
  - Infects MS Office documents
  - Often installs in main document template



Degrees of Complication

- Viruses have various degrees of complication in how they can insert themselves in computer code.

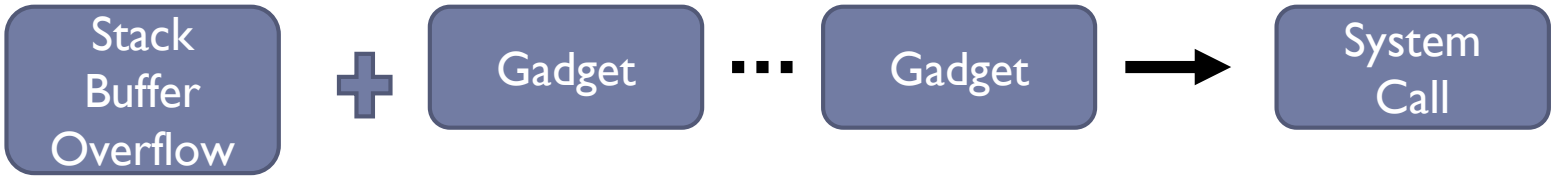


Behavior Based viruses
Monitored behaviors can include:

- Attempts to open, view, delete, and/or modify files;
- Attempts to format disk drives and other unrecoverable disk operations;
- Modifications to the logic of executable files, scripts of macros;
- Modification of critical system settings, such as start-up settings;
- Disadvantage : cannot predict future behavior

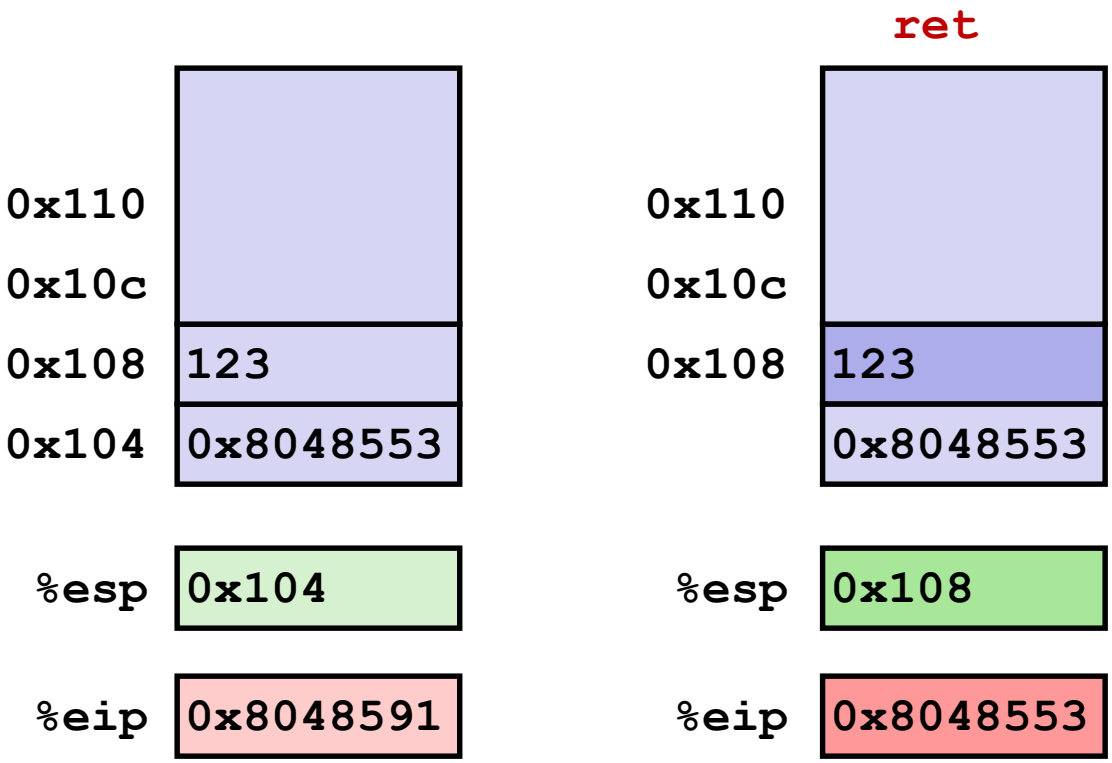
Return Oriented Programming

- Serangan dengan memanfaatkan standard library yang tersedia, tanpa harus menginjeksi kode



Procedure Return Example

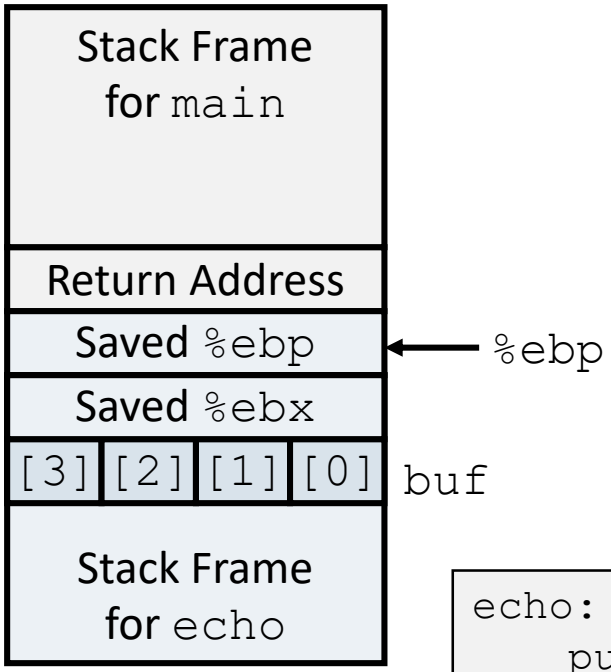
```
8048591:      c3                          ret
```



%eip: program counter

Buffer Overflow Stack

Before call to gets




```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    pushl %ebp              # Save %ebp on stack
    movl  %esp, %ebp
    pushl %ebx              # Save %ebx
    subl  $20, %esp         # Allocate stack space
    leal  -8(%ebp),%ebx     # Compute buf as %ebp-8
    movl  %ebx, (%esp)      # Push buf on stack
    call  gets              # Call gets
    . . .
```

Return oriented programming

- Gadget: potongan sekuens instruksi yang berakhiran dengan ret (return) => menjalankan instruksi yg ditunjuk stack

- Contoh gadget:

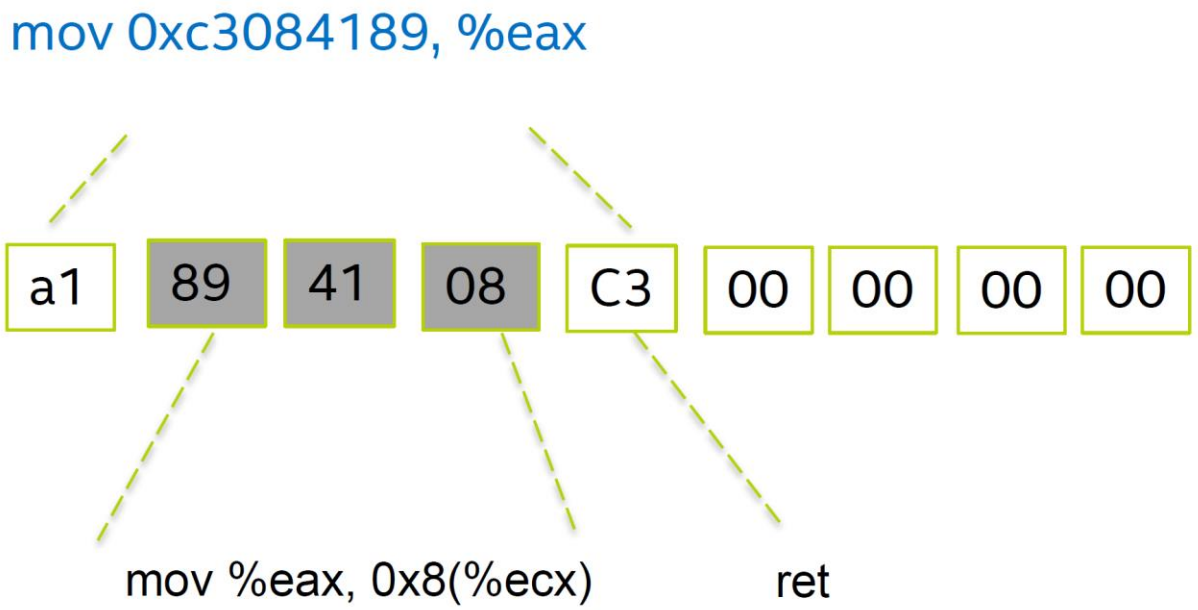
```
pop %ecx
pop %eax
ret
```

```
mov %eax, (%ecx)
ret
```

- Stack setup:

```
| <address of mov %eax, (%ecx)>        |
| <value to write>                     |
| <address to write to>                |
| <address of pop %ecx; pop %eax; ret> |
```

Contoh gadget lain



ROP Attack

- Memanfaatkan call ke libc (return to libc)

any sufficiently large program codebase

arbitrary attacker computation and behavior,
without code injection

Teknik: kumpulkan db gadget dengan mencari ret (c3) pada kode, digunakan sebagai sekuens instruksi serangan

Today

- Structures
  - Alignment
- Unions
- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection
  - Return Oriented Programming

Buffer overflow

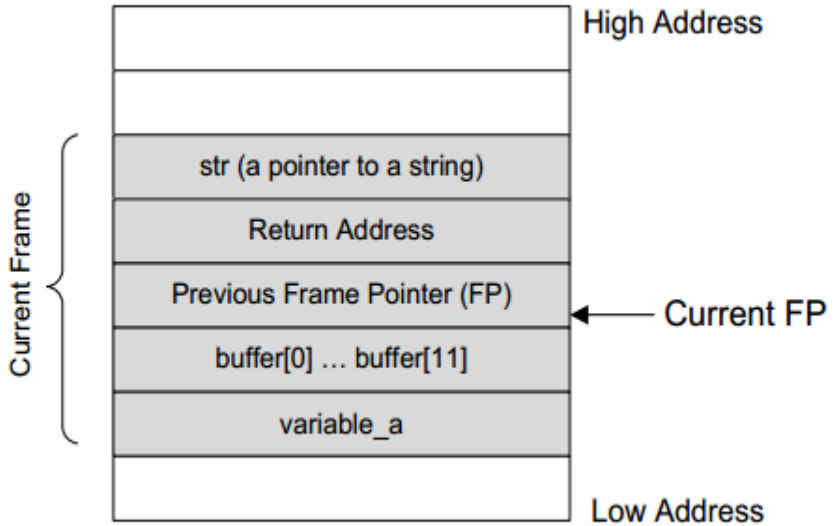
- An attempt to store too much information into an allocated space in a computer's memory.
  - A hacker can exploit a buffer overflow to overwrite the return address to point to a new program, rather than original correct program
  - This new program is inserted by the hacker into the system which allows them to access the OS and perform malicious attacks

char buffer2[8] = {0};strcpy(buffer2,"hellooooooo hooooooow are youuuuuuuu");

```
void func (char *str) {
   char buffer[12];
   int  variable_a;
   strcpy (buffer, str);
}

Int main() {
   char *str = “I am greater than 12 bytes”;
   func (str);
}
```

(a) A code example



(b) Active Stack Frame in func()

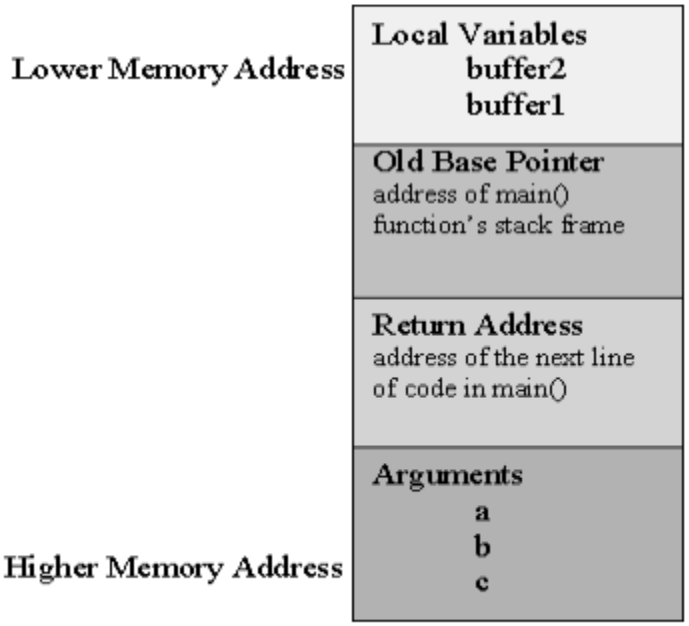
```
void func (char *str) {
   char buffer[12];
   int  variable_a;
   strcpy (buffer, str);
} if (canary == secret)

Int main() {
   char *str = “I am greater than 12 bytes”;
   func (str);
}
```

STACK BASICS

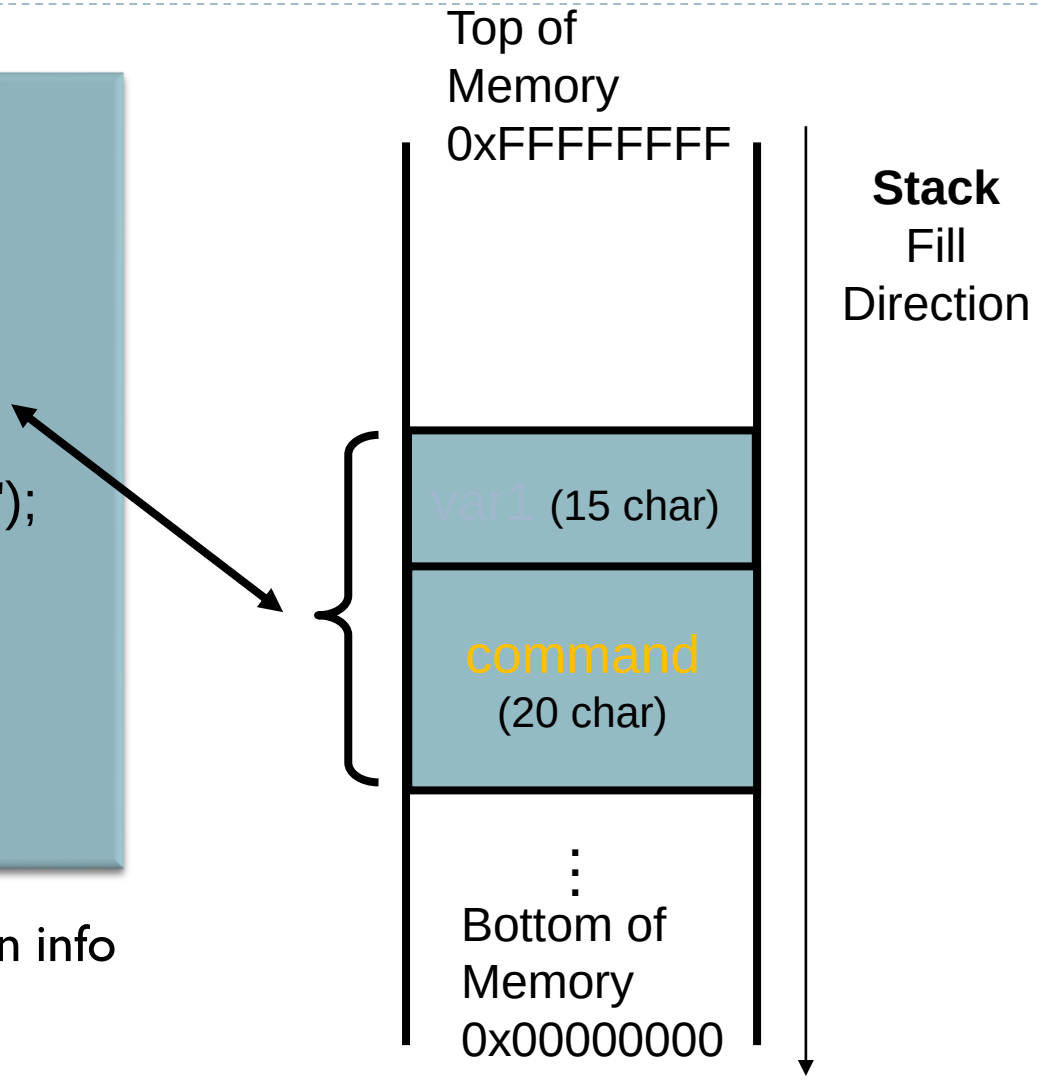
EXAMPLE

```
void function( int a, int b , int c){
      char buffer1[5];
      char buffer2[10];
}
void main(){
      function(1,2,3);
}
```



Buffer Overflow

```
domain.c
Main(int argc, char *argv[ ])
/* get user_input */
{
    char var1[15];
    char command[20];
    strcpy(command, “whois ");
    strcat(command, argv[1]);
    strcpy(var1, argv[1]);
    printf(var1);
    system(command);
}
```

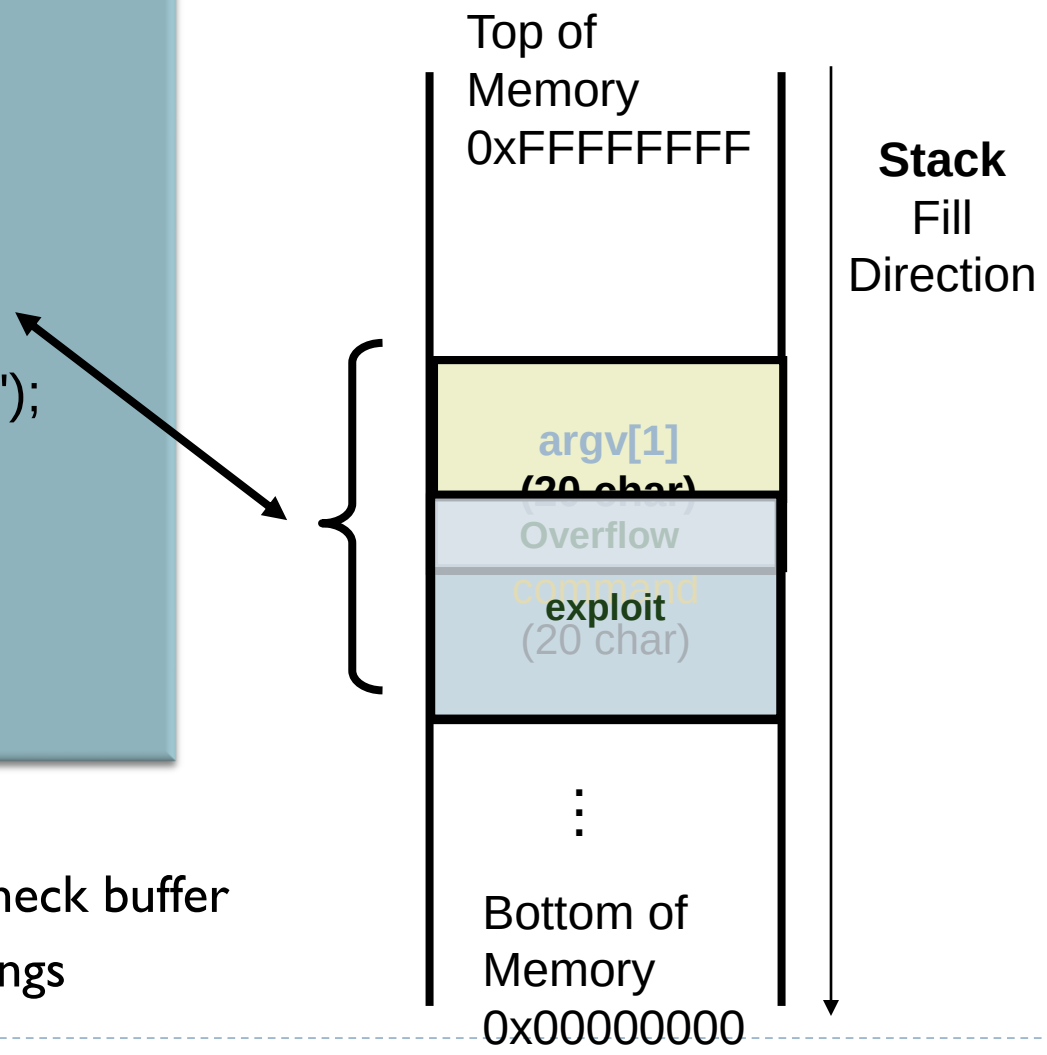


- Retrieves domain registration info
- e.g., domain brown.edu

strcpy() Vulnerability

domain.c

```
Main(int argc, char *argv[])
/*get user_input*/
{
    char var1[15];
    char command[20];
    strcpy(command, “whois ");
    strcat(command, argv[1]);
    strcpy(var1, argv[1]);
    printf(var1);
    system(command);
}
```



- argv[1] is the user input
- strcpy(dest, src) does not check buffer
- strcat(d, s) concatenates strings

Buffer overflow

Name	Amount owing
John Smith	4500

Name: John Smith
Amount Owing: 4500

Name	Amount owing
John Smithxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	x9999999990

Name: John Smithxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx9999999999
Amount Owing: 1000
(the last 0 from 1000 was not written over.)

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

int main(int argc, char *argv[])
{
  char name[5] = "";
  char lastname[10] ="";

  printf("Enter your name: ");
  scanf("%s", name);

  printf("My name is %s \n", name);
  printf("And my lastname is %s \n", lastname);

  return 0;
}

0123456789abcdef ghijklmnopqrstuvxyz
```

```
abc123@RandomHash:~$ gcc -o overflow overflow.c
abc123@RandomHash:~$ ./overflow
Enter your name: kalle
My name is kalle
And my lasname is
abc123@RandomHash:~$ ./overflow
Enter your name: kkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkk
My name is kkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkk
And my lasname is kkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkk
*** stack smashing detected ***: ./overflow terminated
Aborted (core dumped)
abc123@RandomHash:~$ ./overflow
Enter your name: 0123456789abcdefghijklmnopqrstuvxyz
My name is 0123456789abcdefghijklmnopqrstuvxyz
And my lasname is ghijklmnopqrstuvxyz
abc123@RandomHash:~$ ./overflow
Enter your name: 0123456789abcdefOhlsson
My name is 0123456789abcdefOhlsson
And my lasname is Ohlsson
abc123@RandomHash:~$
```

Stack-based buffer overflow detection using a random canary

Normal (safe) stack configuration:

Buffer overflow attack attempt:

- The canary is placed in the stack prior to the return address, so that any attempt to over-write the return address also over-writes the canary.

```
/* This program has a buffer overflow vulnerability. */
/* However, it is protected by StackGuard */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int func (char *str)
{
  int canaryWord = secret;
  char buffer[12];

  /* The following statement has a buffer overflow problem */
  strcpy(buffer, str);

  if (canaryWord == secret) // Return address is not modified
     return 1;
  else // Return address is potentially modified
     { ... error handling ...  }
}
```

- Use strong type language, e.g. java, C#, etc. With these languages, buffer overflows will be detected.
- Use safe library functions.
  - Functions that could have buffer overflow problem: `gets`, `strcpy`, `strcat`, `sprintf`, `scanf`, etc.
  - These functions are safer: `fgets`, `strncpy`, `strncat`, and `snprintf`.