(Advanced) Computer Security!

Software Security

Buffer Overflow Vulnerabilities, Attacks, and Defenses

(part IV)

Prof. Travis Peters

Montana State University

CS 476/594 - Computer Security

Spring 2021

https://www.traviswpeters.com/cs476

Today

Reminder!
Please update your Slack, GitHub, Zoom
(first/last name, professional photo/background)

- Announcements
  - Lab 03 released -> extended deadline! (due 03/02/2021)

- Learning Objectives
  - Review the layout of a program in memory & stack layout
  - Understand buffer overflows & vulnerable code
  - Challenges in exploiting buffer overflow vulnerabilities
  - Grasp major challenges and solutions of "shellcode"
  - Countermeasures (e.g., ASLR, StackGuard, non-executable stack)

Countermeasures

Countermeasures (Overview)

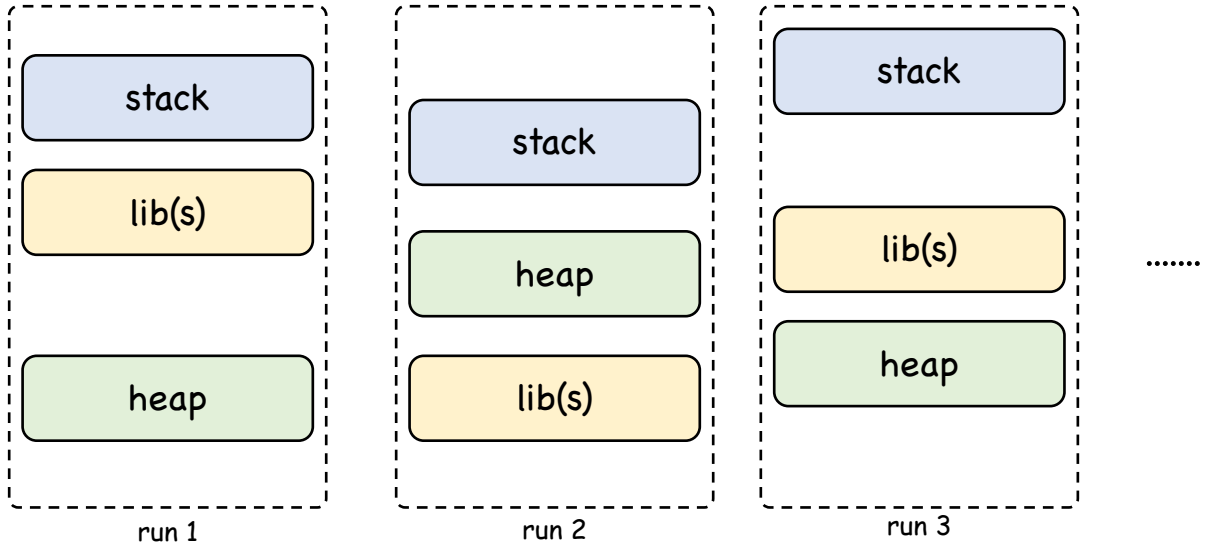
- **Developer Approaches**
  - Fix the code!!!
  - Safer functions; e.g., strncpy(), strncat(), … -> explicitly specify lengths of buffers
  - Safer dynamically linked libraries; e.g., libsafe(?) -> wrappers around vuln. routines, add boundary checks, …
- **Tools**
  - Safer SW build tools; e.g., ITS4, ARCHER, BOON, PolySpace -> static/dynamic analysis to detect buffer overflows
  - Safer languages; e.g., Java, Python -> Language provides automatic bounds checking
- **OS Approaches**
  - ASLR (Address Space Layout Randomization)
- **Compiler Approaches**
  - StackGuard
- **Hardware Approaches**
  - Non-executable stack -> can still be defeated using **ROP** / **return-to-libc** attacks

Countermeasure:

ASLR

ASLR = Address Space Layout Randomization

- Randomize the start location of the stack, heap, libs, etc.
  e.g., every time the code is loaded into memory, the stack address changes!



- **Impact**
  - legit code -> Easy to run (relative to frame)
  - attacker code -> Difficult to guess the (stack) address(es) in memory
    i.e., difficult to guess frame pointer address (%ebp) and address of injected code

ASLR In Action

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


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

    printf("Address of x (in stack): 0x%x\n", (unsigned int)x);
    printf("Address of y (in mmap):  0x%x\n", (unsigned int)y);
    printf("Address of z (in heap):  0x%x\n", (unsigned int)z);

    //...
    return 0;
}
```

Disable ASLR

```
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
$ ./stack_layout3
Address of x (in stack): 0xffffd490
Address of y (in mmap):  0xf7e531d0
Address of z (in heap):  0x5655a1a0
$ ./stack_layout3
Address of x (in stack): 0xffffd490
Address of y (in mmap):  0xf7e531d0
Address of z (in heap):  0x5655a1a0
```

https://www.kernel.org/doc/html/latest/admin-guide/sysctl/kernel.html#randomize-va-space

ASLR In Action

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


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

    printf("Address of x (in stack): 0x%x\n", (unsigned int)x);
    printf("Address of y (in mmap):  0x%x\n", (unsigned int)y);
    printf("Address of z (in heap):  0x%x\n", (unsigned int)z);

    //...
    return 0;
}
```

Disable ASLR

```
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
$ ./stack_layout3
Address of x (in stack): 0xffffd490
Address of y (in mmap):  0xf7e531d0
Address of z (in heap):  0x5655a1a0
$ ./stack_layout3
Address of x (in stack): 0xffffd490
Address of y (in mmap):  0xf7e531d0
Address of z (in heap):  0x5655a1a0
```

Randomize position of stack, mmap, VDSO, and heap

```
$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
$ ./stack_layout3
Address of x (in stack): 0xffdaeba0
Address of y (in mmap):  0xf7dbf1d0
Address of z (in heap):  0x570271a0
$ ./stack_layout3
Address of x (in stack): 0xffc8b2e0
Address of y (in mmap):  0xf7e0a1d0
Address of z (in heap):  0x57c221a0
```

https://www.kernel.org/doc/html/latest/admin-guide/sysctl/kernel.html#randomize-va-space

ASLR In Action

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


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

    printf("Address of x (in stack): 0x%x\n", (unsigned int)x);
    printf("Address of y (in mmap):  0x%x\n", (unsigned int)y);
    printf("Address of z (in heap):  0x%x\n", (unsigned int)z);

    //...
    return 0;
}
```

Disable ASLR

```
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
$ ./stack_layout3
Address of x (in stack): 0xffffd490
Address of y (in mmap):  0xf7e531d0
Address of z (in heap):  0x5655a1a0
$ ./stack_layout3
Address of x (in stack): 0xffffd490
Address of y (in mmap):  0xf7e531d0
Address of z (in heap):  0x5655a1a0
```

Randomize position of stack, mmap, VDSO, and heap

```
$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
$ ./stack_layout3
Address of x (in stack): 0xffdaeba0
Address of y (in mmap):  0xf7dbf1d0
Address of z (in heap):  0x570271a0
$ ./stack_layout3
Address of x (in stack): 0xffc8b2e0
Address of y (in mmap):  0xf7e0a1d0
Address of z (in heap):  0x57c221a0
```

https://www.kernel.org/doc/html/latest/admin-guide/sysctl/kernel.html#randomize-va-space

Q: Can we still do buffer overflow when stack address in unknown (randomized)?

Defeating ASLR

Setup -> use shell w/out RUID!=EUID countermeasure + turn ASLR ON

```
$ sudo ln -sf /bin/zsh /bin/sh
$ sudo sysctl -w kernel.randomize_va_space=2
```

Compile a root-owned set-uid program

```
$ gcc -o stack-L1 -z execstack -fno-stack-protector stack.c
$ sudo chown root stack
$ sudo chmod 4755 stack
```

m32

Repeatedly run the program until we get lucky...

```
#!/bin/bash

SECONDS=0
value=0

while true; do
    value=$(( $value + 1 ))
    duration=$SECONDS
    min=$(($duration / 60))
    sec=$(($duration % 60))
    echo "The program has been run $value times so far (time elapsed: $min minutes and $sec seconds)."
    ./stack-L1
done
```

```
.........
The program has been run 67679 times so far...
./brute-force.sh: line 13:  ... Segmentation fault      ./stack-L1
The program has been run 67680 times so far...
./brute-force.sh: line 13:  ... Segmentation fault      ./stack-L1
The program has been run 67681 times so far...
# id  <-- ROOT SHELL!
uid=1000(seed) gid=1000(seed) euid=0(root) ...
```

Countermeasure:

StackGuard

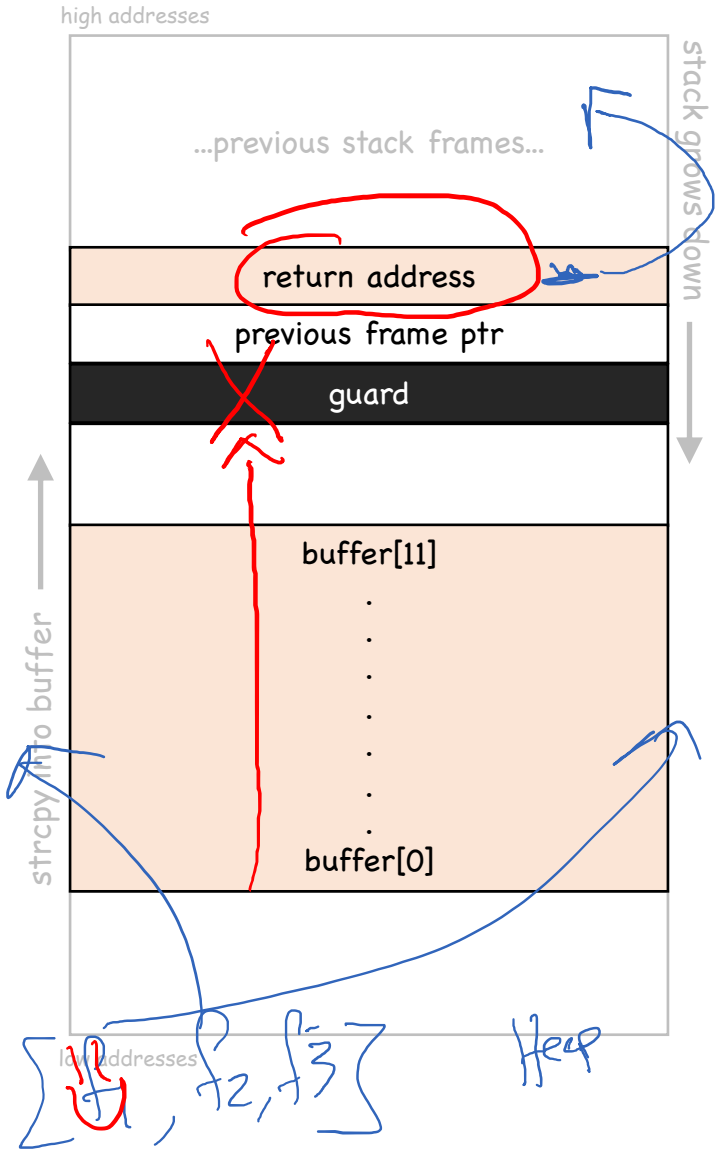
StackGuard

The main idea...

```
int foo(char *str)
{
    int guard;
    guard = secret;          // secret stored elsewhere
                             // -> copied to stack

    char buffer[12];
    strcpy(buffer, str);     // strcpy overflow?

    if (guard == secret)
        return;              // stack appears to be OK
    else
        exit(1);             // overflow detected!
}
```



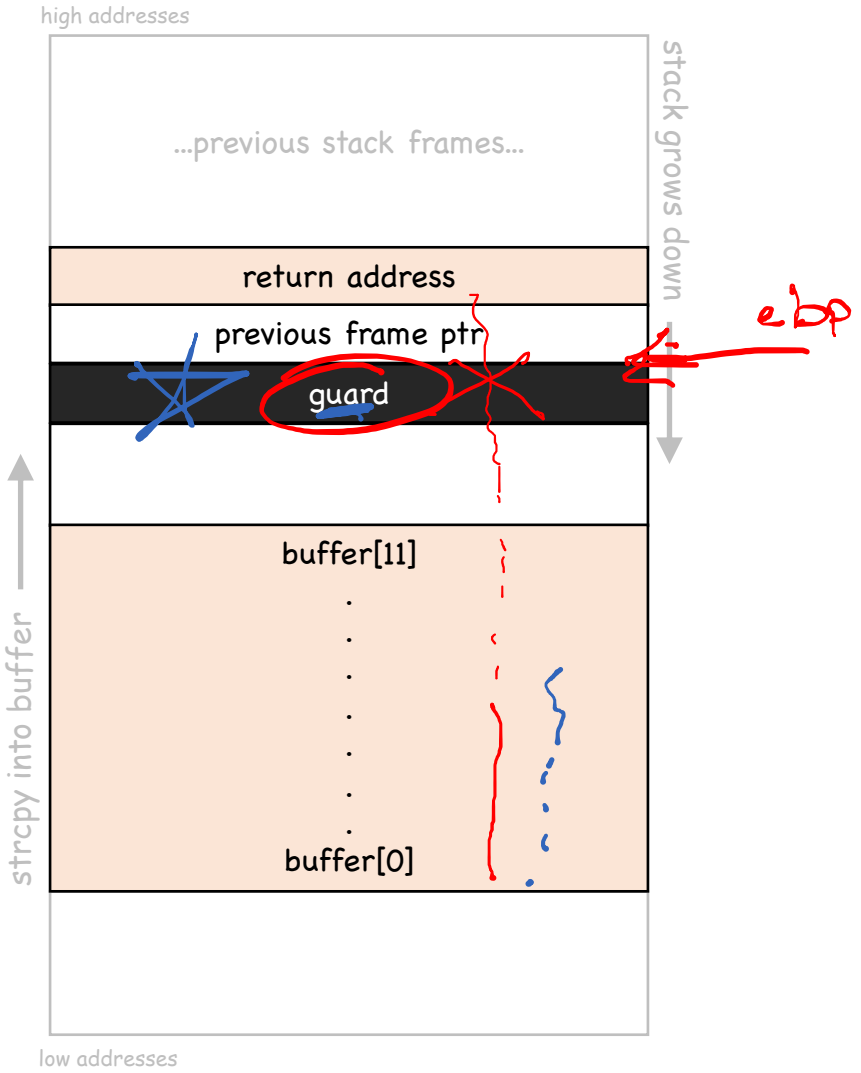
StackGuard In Action

```
$ gcc -m32 -ggdb vuln.c -o vuln

$ ./vuln hello
Returned Properly

$ ./vuln hitherethisisamuchlongerstring
*** stack smashing detected ***: terminated
Aborted
```

format strings



StackGuard In Action (cont.)

Canary set/check is handled for us **by the compiler**!

```
# Recall:
$ gcc -m32 -ggdb vuln.c -o vuln32
```

-> default assembly syntax on our VMs is AT&T
mnemonic src, dst

```
$ objdump -d -S vuln
...snipped...

0000120d <foo>:
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

void foo(char *str)
{
    120d:   f3 0f 1e fb             endbr32
    1211:   55                      push   %ebp
    1212:   89 e5                   mov    %esp,%ebp
    1214:   53                      push   %ebx
    1215:   83 ec 24                sub    $0x24,%esp
    1218:   e8 97 00 00 00          call   12b4 <__x86.get_pc_thunk.ax>
    121d:   05 b3 2d 00 00          add    $0x2db3,%eax
    1222:   8b 55 08                mov    0x8(%ebp),%edx
    1225:   89 55 e4                mov    %edx,-0x1c(%ebp)
    1228:   65 8b 0d 14 00 00 00    mov    %gs:0x14,%ecx      // load value into register
    122f:   89 4d f4                mov    %ecx,-0xc(%ebp)    // store value on stack
    1232:   31 c9                   xor    %ecx,%ecx          // zero out register
        char buffer[12];
        strcpy(buffer, str); // buffer overflow vuln!
    1234:   83 ec 08                sub    $0x8,%esp
    1237:   ff 75 e4                pushl  -0x1c(%ebp)
    123a:   8d 55 e8                lea    -0x18(%ebp),%edx
    123d:   52                      push   %edx
    123e:   89 c3                   mov    %eax,%ebx
    1240:   e8 5b fe ff ff          call   10a0 <strcpy@plt>
    1245:   83 c4 10                add    $0x10,%esp
}
    1248:   90                      nop
    1249:   8b 45 f4                mov    -0xc(%ebp),%eax    // move stack value into reg.
    124c:   65 33 05 14 00 00 00    xor    %gs:0x14,%eax      // xor with original canary
    1253:   74 05                   je     125a <foo+0x4d>    // if equal, do normal return
    1255:   e8 e6 00 00 00          call   1340 <__stack_chk_fail_local>  // fail!
    125a:   8b 5d fc                mov    -0x4(%ebp),%ebx
    125d:   c9                      leave
    125e:   c3                      ret

...snipped...
```

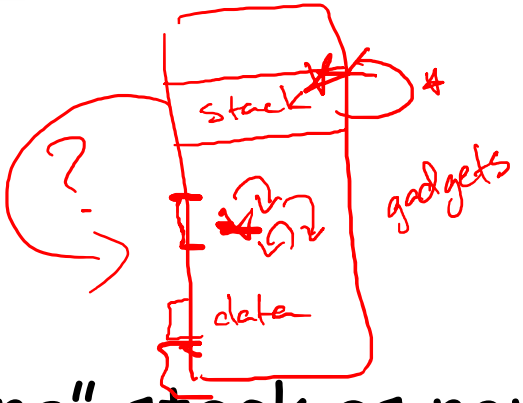
Set Canary!

Canary Check!

Countermeasure:

Non-Executable Stack

Non-Executable Stack



- Stack-based exploit can be prevented by "marking" stack as non-executable ("NX" bit on page table entry)

- Limitations:
  - Does not defend against ROP (return-oriented programming)
    - e.g., return-to-libc attacks
  - Does not defend against general overflow
    - e.g., overflow on heap -> overwrite function pointers
  - Some apps need executable stack (e.g., LISP interpreters)

Summary

Summary

- Buffer Overflows
  - primary issue -> user-controlled inputs can impact execution /control flow!
  - attack goal: overwrite return address w/ new address
    - new address = attacker code injected into process address space
    - or... new address = other places? (e.g., libc -> return-to-libc attacks!)
    - or... overflow/overwrite other things (e.g., function pointers)
  - cause other code (e.g., shellcode) to be executed

- Countermeasures
  - Better SW mitigations
    e.g., use safer/explicit functions, safer libraries, compiler-injected checks
  - Better system mitigations
    e.g., ASLR, non-executable stack

You Try!

- Try the "Narnia" challenges on **Over The Wire** -> http://overthewire.org/wargames/narnia/
- In Listing 1, how are the addresses decided for the variables a and x;
  i.e., during runtime, how does the program know the address of these two variables?
- In List 2, in which memory segments are the variables in the code located?

```
// Listing 1
void foo(int a)
{
    int x;
}
```

```
// Listing 2
int i = 0;
void func(char *str)
{
   char *ptr = malloc(sizeof(int));
   char buf[1024];
   int j;
   static int y;
}
```

- A student proposes to change how the stack grows. Instead of growing from higher addresses to lower addresses, the student proposes to let the stack grow from lower addresses to higher addresses. This way, the buffer will be allocated above the return address, so overflowing the buffer will not be able to affect the return address. Please comment on this proposal?
- Why does ASLR make buffer-overflow attacks more difficult?
- Why does a stack guard/canary make buffer-overflow attacks more difficult?
- To write a shellcode, we need to know the address of the string "/bin/sh". If we have to hardcode the address in the code, it will become difficult if ASLR is turned on. Shellcode solved that problem without hardcoding the address of the string in the code. Please explain how the shellcode in exploit.c (Listing 4.2) achieved that.