Lecture 11: Code and data interactions, buffer overflows

David Hovemeyer

September 27, 2021

601.229 Computer Systems Fundamentals

Buffer overflows

A dangerous function

```
#include <stdio.h>

char *gets(char *s);
```

`gets` reads a single line of input from `stdin` and stores it in the character array pointed to by `s`

A dangerous function

```
#include <stdio.h>

char *gets(char *s);
```

`gets` reads a single line of input from `stdin` and stores it in the character array pointed to by `s`

Why is this dangerous?

A dangerous function

```
#include <stdio.h>

char *gets(char *s);
```

gets reads a single line of input from stdin and stores it in the character array pointed to by s

Why is this dangerous?

There is no way to ensure that the character array is large enough to store the input

Clicker quiz!

Clicker quiz omitted from public slides

Memory safety

- C is a *memory-unsafe* language
  - No bounds checking of array accesses
  - No restrictions on pointers:

```
uint64_t x = 0xDEADBEEF;
char *s = (char *) x;
strcpy(s, "Hello, world!");
```

- Invalid memory references are an all-too-common source of bugs in C programs
- What are the consequences of an invalid memory reference?

segfaults

- If you're *lucky*, an invalid memory reference will crash the program with a *segmentation violation*, a.k.a. segfault
- Recall (from Lecture 6) using the `pmap` program to view a running program's memory map:

```
29208:   ./art
0000562d71c36000      4K r-x-- art
0000562d71e36000      4K r---- art
0000562d71e37000      4K rw--- art
0000562d735fc000    132K rw---   [ anon ]
...etc...
```

- Memory references outside a valid region of virtual memory, or which violate access permissions (e.g., store to read-only region), result in a processor execption that is handled by the OS kernel
- Usual result is that OS sends a *signal* that terminates the running program

Memory corruption

- A much worse consequence of an invalid memory store: data is corrupted
  - A variable or array element is overwritten
  - A saved register value or temporary value is overwritten
  - A return address is overwritten (this is particularly bad, as we'll see shortly)
- In general, once a program makes an invalid memory reference, it cannot be trusted to behave correctly
  - This is why valgrind is such an important tool

A dangerous program

Based on example in textbook (code in buf.zip on course website):

```
#include <stdio.h>

void echo(void) {
  char buf[4];
  gets(buf);
  puts(buf);
}

int main(void) {
  printf("Enter a line of text:\n");
  echo();
  return 0;
}
```

A dangerous program

Based on example in textbook (code in buf.zip on course website):

```
#include <stdio.h>

void echo(void) {
  char buf[4];    <-- small buffer, safe only if string length 3 or less
  gets(buf);
  puts(buf);
}

int main(void) {
  printf("Enter a line of text:\n");
  echo();
  return 0;
}
```

Compiling and running

```
$ gcc -Og -no-pie -Wall -Wextra -fno-stack-protector -o danger danger.c
...warning about implicit declaration of gets omitted...
...warning from linker about gets being dangerous omitted...
$ ./danger
Enter a line of text:
Hi there!
Hi there!
$ echo $?
0
```

Compiling and running

```
$ gcc -Og -no-pie -Wall -Wextra -fno-stack-protector -o danger danger.c
...warning about implicit declaration of gets omitted...
...warning from linker about gets being dangerous omitted...
$ ./danger
Enter a line of text:
Hi there!
Hi there!
$ echo $?
0
```

Wait...why did the program behave correctly?

Inspect the generated code

gcc's -S option translates C code (.c file) into assembly language (.s file)

```
$ gcc -Og -no-pie -fno-stack-protector -S danger.c
...warning about implicit declaration of gets omitted...
$ head -8 danger.s
        .file         "danger.c"
        .text
        .globl         echo
        .type         echo, @function
echo:
.LFB23:
        .cfi_startproc
        pushq         %rbx
```

The echo function (assembly code)

Cleaned-up version of the echo function:

```
echo:
    pushq   %rbx
    subq    $16, %rsp
    leaq    12(%rsp), %rbx
    movq    %rbx, %rdi
    movl    $0, %eax
    call    gets@PLT
    movq    %rbx, %rdi
    call    puts@PLT
    addq    $16, %rsp
    popq    %rbx
    ret
```

The echo function (assembly code)

Cleaned-up version of the echo function:

```
echo:
    pushq    %rbx            <-- save %rbx (callee-saved register)
    subq     $16, %rsp
    leaq     12(%rsp), %rbx
    movq     %rbx, %rdi
    movl     $0, %eax
    call     gets@PLT
    movq     %rbx, %rdi
    call     puts@PLT
    addq     $16, %rsp
    popq     %rbx
    ret
```

The echo function (assembly code)

Cleaned-up version of the `echo` function:

```
echo:
    pushq   %rbx
    subq    $16, %rsp        <-- reserve 16 bytes of space in stack frame
    leaq    12(%rsp), %rbx
    movq    %rbx, %rdi
    movl    $0, %eax
    call    gets@PLT
    movq    %rbx, %rdi
    call    puts@PLT
    addq    $16, %rsp
    popq    %rbx
    ret
```

The echo function (assembly code)

Cleaned-up version of the echo function:

```
echo:
    pushq   %rbx
    subq    $16, %rsp
    leaq    12(%rsp), %rbx   <-- put base address of buf in %rbx
    movq    %rbx, %rdi
    movl    $0, %eax
    call    gets@PLT
    movq    %rbx, %rdi
    call    puts@PLT
    addq    $16, %rsp
    popq    %rbx
    ret
```

The echo function (assembly code)

Cleaned-up version of the `echo` function:

```
echo:
    pushq   %rbx
    subq    $16, %rsp
    leaq    12(%rsp), %rbx
    movq    %rbx, %rdi       <-- pass base address of buf to gets
    movl    $0, %eax
    call    gets@PLT
    movq    %rbx, %rdi
    call    puts@PLT
    addq    $16, %rsp
    popq    %rbx
    ret
```

The echo function (assembly code)

Cleaned-up version of the echo function:

```
echo:
    pushq   %rbx
    subq    $16, %rsp
    leaq    12(%rsp), %rbx
    movq    %rbx, %rdi
    movl    $0, %eax        <-- no vector args to gets
    call    gets@PLT
    movq    %rbx, %rdi
    call    puts@PLT
    addq    $16, %rsp
    popq    %rbx
    ret
```

The echo function (assembly code)

Cleaned-up version of the echo function:

```
echo:
    pushq   %rbx
    subq    $16, %rsp
    leaq    12(%rsp), %rbx
    movq    %rbx, %rdi
    movl    $0, %eax
    call    gets@PLT        <-- call gets
    movq    %rbx, %rdi
    call    puts@PLT
    addq    $16, %rsp
    popq    %rbx
    ret
```

The echo function (assembly code)

Cleaned-up version of the echo function:

```
echo:
    pushq   %rbx
    subq    $16, %rsp
    leaq    12(%rsp), %rbx
    movq    %rbx, %rdi
    movl    $0, %eax
    call    gets@PLT
    movq    %rbx, %rdi       <-- pass base address of buf to puts
    call    puts@PLT
    addq    $16, %rsp
    popq    %rbx
    ret
```

The echo function (assembly code)

Cleaned-up version of the echo function:

```
echo:
    pushq   %rbx
    subq    $16, %rsp
    leaq    12(%rsp), %rbx
    movq    %rbx, %rdi
    movl    $0, %eax
    call    gets@PLT
    movq    %rbx, %rdi
    call    puts@PLT        <-- call puts
    addq    $16, %rsp
    popq    %rbx
    ret
```

The echo function (assembly code)

Cleaned-up version of the echo function:

```
echo:
    pushq   %rbx
    subq    $16, %rsp
    leaq    12(%rsp), %rbx
    movq    %rbx, %rdi
    movl    $0, %eax
    call    gets@PLT
    movq    %rbx, %rdi
    call    puts@PLT
    addq    $16, %rsp        <-- de-allocate space in stack frame
    popq    %rbx
    ret
```

The echo function (assembly code)

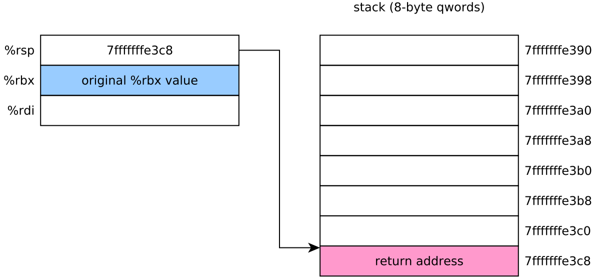
Cleaned-up version of the echo function:

```
echo:
    pushq   %rbx
    subq    $16, %rsp
    leaq    12(%rsp), %rbx
    movq    %rbx, %rdi
    movl    $0, %eax
    call    gets@PLT
    movq    %rbx, %rdi
    call    puts@PLT
    addq    $16, %rsp
    popq    %rbx             <-- restore %rbx
    ret
```

Tracing the danger program

On entry to echo function:

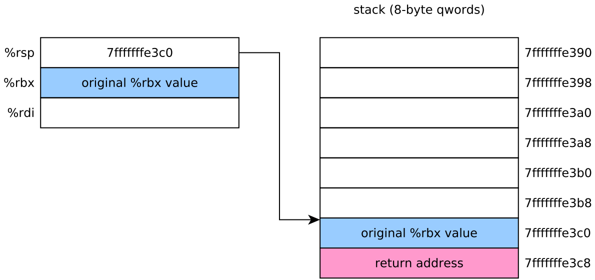
```
echo:
    pushq   %rbx
    subq    $16, %rsp
    leaq    12(%rsp), %rbx
    movq    %rbx, %rdi
    movl    $0, %eax
    call    gets@PLT
    movq    %rbx, %rdi
    call    puts@PLT
    addq    $16, %rsp
    popq    %rbx
    ret
```



Tracing the danger program

After pushing `%rbx`:

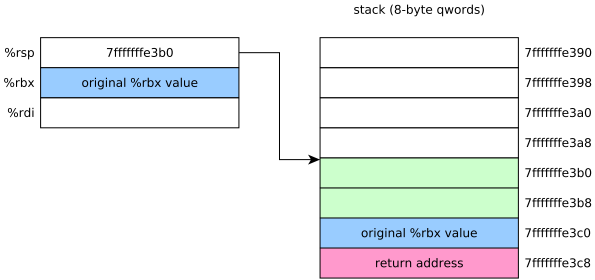
```
echo:
        pushq   %rbx
        subq    $16, %rsp
        leaq    12(%rsp), %rbx
        movq    %rbx, %rdi
        movl    $0, %eax
        call    gets@PLT
        movq    %rbx, %rdi
        call    puts@PLT
        addq    $16, %rsp
        popq    %rbx
        ret
```



Tracing the danger program

After reserving 16 bytes in stack frame:

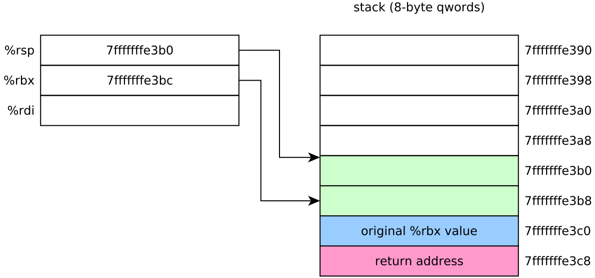
```
echo:
    pushq   %rbx
    subq    $16, %rsp
    leaq    12(%rsp), %rbx
    movq    %rbx, %rdi
    movl    $0, %eax
    call    gets@PLT
    movq    %rbx, %rdi
    call    puts@PLT
    addq    $16, %rsp
    popq    %rbx
    ret
```



Tracing the danger program

After loading base address of buf into `%rbx`:

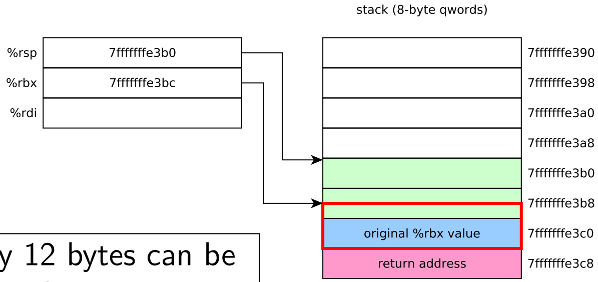
```
echo:
    pushq   %rbx
    subq    $16, %rsp
    leaq    12(%rsp), %rbx
    movq    %rbx, %rdi
    movl    $0, %eax
    call    gets@PLT
    movq    %rbx, %rdi
    call    puts@PLT
    addq    $16, %rsp
    popq    %rbx
    ret
```



Tracing the danger program

After loading base address of buf into %rbx:

```
echo:
    pushq   %rbx
    subq    $16, %rsp
    leaq    12(%rsp), %rbx
    movq    %rbx, %rdi
    movl    $0, %eax
    call    gets@PLT
    movq    %rbx, %rdi
    call    puts@PLT
    addq    $16, %rsp
    popq    %rbx
    ret
```

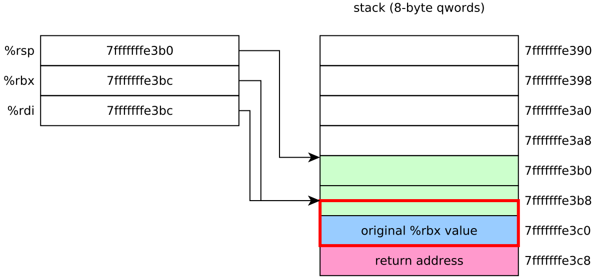


Exactly 12 bytes can be stored before overwriting the return address

Tracing the danger program

Pass base address of buf to gets:

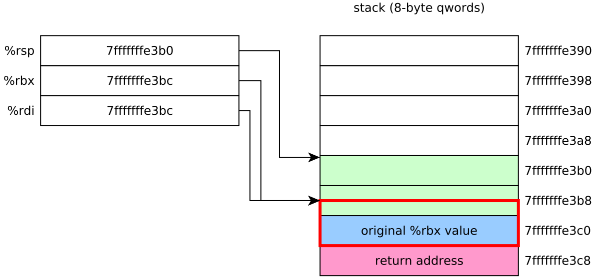
```
echo:
    pushq   %rbx
    subq    $16, %rsp
    leaq    12(%rsp), %rbx
    movq    %rbx, %rdi
    movl    $0, %eax
    call    gets@PLT
    movq    %rbx, %rdi
    call    puts@PLT
    addq    $16, %rsp
    popq    %rbx
    ret
```



Tracing the danger program

Just before call to gets:

```
echo:
    pushq   %rbx
    subq    $16, %rsp
    leaq    12(%rsp), %rbx
    movq    %rbx, %rdi
    movl    $0, %eax
    call    gets@PLT
    movq    %rbx, %rdi
    call    puts@PLT
    addq    $16, %rsp
    popq    %rbx
    ret
```



Explanation of behavior

- The `danger` program appeared to work when the input was `Hi there!` because the string only requires 10 bytes to store, and 12 bytes were available
- The saved `%rbx` value is partially overwritten, but `main` (the caller) wasn't using that register
  - Hard to know whether `main`'s caller was using it

Explanation of behavior

- The `danger` program appeared to work when the input was `Hi there!` because the string only requires 10 bytes to store, and 12 bytes were available
- The saved `%rbx` value is partially overwritten, but `main` (the caller) wasn't using that register
  - Hard to know whether `main`'s caller was using it

We got lucky

Overwriting the return address

- When the return address is overwritten, control won't return to the correct instruction when the function returns
- What could happen?

The code could crash

```
$ ./danger
Enter a line of text:
Hello, world!
Hello, world!
Segmentation fault (core dumped)
```

The code could crash

```
$ ./danger
Enter a line of text:
Hello, world!
Hello, world!
Segmentation fault (core dumped)
```

- The string `Hello, world!` requires 14 bytes to represent, so the first two bytes of the return address are overwritten
- Control returns to a zeroed region of memory
- The bytes `00 00` encode the instruction `add %al,(%rax)`
- `%rax` contains the return value of `puts`, which is 14
- No memory is mapped at address 14, so a segmentation fault occurs

Vulnerability to untrusted data

- Let's assume that the input sent to the program is *untrusted*
  - I.e., we should assume that it was generated by a malicious user who wants to take control of our computer and do nefarious things
  - For many kinds of programs — especially network applications — most or all input data is untrusted
- Because of the buffer overflow, the input sent to the program can change the `echo` function's return address to an arbitrary value
- This means the malicious user has (some) control over which code executes when the function returns
  - This is extremely bad!
- If a malicious actor ("attacker") knows that a buffer overflow bug exists, what does it allow them to do?

Executing arbitrary code from the stack

- In the previous (32-bit) x86 architecture, any region of memory marked as readable is also *executable*
- The attacker can send code that will be written onto the stack
  - The malicious data must overwrite the return address with the location of the exploit code (on the compromised stack)
  - This requires knowing (or guessing) the stack pointer's value (so that control "returns" to the code on the stack)

Nop sleds

- To make arbitrary code execution more feasible, attacker can construct a "nop sled": a long series of `nop` (do nothing) instructions leading to exploit code
  - As long as forged return address hits the nop sled, the exploit code will execute
  - This allows the exploit to work (with some probability) even if the exact stack pointer value isn't known (the guess just has to be "close enough")

Exploiting existing code

- Another way of exploiting a buffer overflow is to overwrite the return address with the address of an instruction in the running program
- If the target instruction is chosen carefully, it may be able to cause the execution of an arbitrary function with arbitrary arguments
- For example, if the return address is overwritten with a code address leading to the execution of the `system` function, an arbitrary program could be executed
  - The exploit must somehow manage to forge argument(s): `pop` instructions are useful for this

The costs of buffer overflow vulnerabilities

- Security compromises of computer systems cost the U.S. economy many *billions* of dollars anually
- Buffer overflows are an important category of security vulnerability
  - But there are many other types of vulnerabilities!

Mitigations for buffer overflows

Mitigations for buffer overflows

- What can we do about buffer overflows?
  - Write code that doesn't have bugs
  - Use memory-safe programming languages
  - Make stack non-executable
  - Address space randomization
  - Detect stack smashing

Write code that doesn't have bugs

- There are lots of things we can do to improve code quality:
  - Thorough testing
  - Code reviews
  - Static analysis
- These are all good ideas, and they will help
  - None of these techniques will catch all bugs

Use memory-safe programming languages

- There are programming languages which guarantee memory safety: Java, Rust (except for "unsafe" code), etc.
  - Memory references are checked at compile time and/or runtime to ensure that only valid memory locations are accessed by the program
- These languages can (in principle) eliminate the possibility of buffer overflows
  - Other kinds of security vulnerabilities are still possible
- Choose the right language for the job

Make stack non-executable

- x86-64 systems allow regions of memory to be marked as non-executable
  - Attempt to execute code from non-executable regions results in a processor exception which can be handled by the OS kernel
- This can eliminate the possibility of a buffer overflow resulting in arbitrary code execution from the stack
- Recall example memory map from Lecture 6 (stack is not executable):

  ```
  00007fff84484000     132K rw---     stack
  ```
- This does not eliminate the possibility of security vulnerabilities, but it makes them harder to implement

Address space randomization

- For exploits which depend on knowing the current (approximate) stack pointer value, the OS kernel can randomly choose where to place the stack in memory
- Code and data in *position-independent* executables can be loaded into memory at arbitrary addresses
  - Exploits depending on a return address jumping to a specific instruction become less likely to succeed
- Address space randomization techniques make exploits more difficult, but don't make them impossible

Detect stack smashing

- Compiler can generate code to detect improper modification of stack memory:
  - On procedure entry, store a "stack canary" value near the return address
  - Prior to return, check the canary value
  - If canary was modified, terminate program
- Canary value generated randomly, cannot easily be guessed
- Return address (in theory) can't be overwritten without also overwriting canary value
- Small runtime overhead incurred on instrumented function calls
- Enabled by default in recent Linux/gcc

Not actually a canary