Lecture 8

<2016-04-27 Wed>

Contents

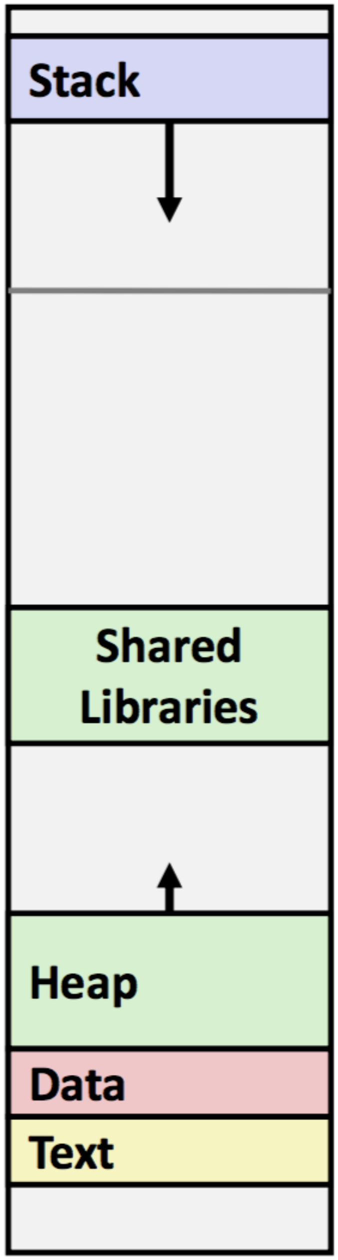
1 Memory Layout

try memory layout example

- stack

  - runtime stack (8MB limit)

- heap
  - dynamically allocated
  - `malloc`, `calloc`, `new`
- data
  - statically allocated data
  - e.g. global variables, `static` variables, string constants
- text / shared library
  - executable machine instructions
  - read-only


Stack
Shared
Libraries
Heap
Data
Text

2 Buffer Overflow

2.1 Vulnerability

- when exceeding the memory size allocated for an array
- most common form
  - unchecked lengths on string inputs
    - particularly for bounded character arrays on the stack

2.1.1 String Library Code

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

- library implementation of unix function `gets()`
  - no way to specify limit on number of characters to read
- similar problem with other library functions
  - `strcpy`, `strcat`: copy strings of arbitrary length
  - `scanf`, `fscanf`, `sscanf`: when given %s conversion specification

2.1.2 Buffer Overflow Attack Example

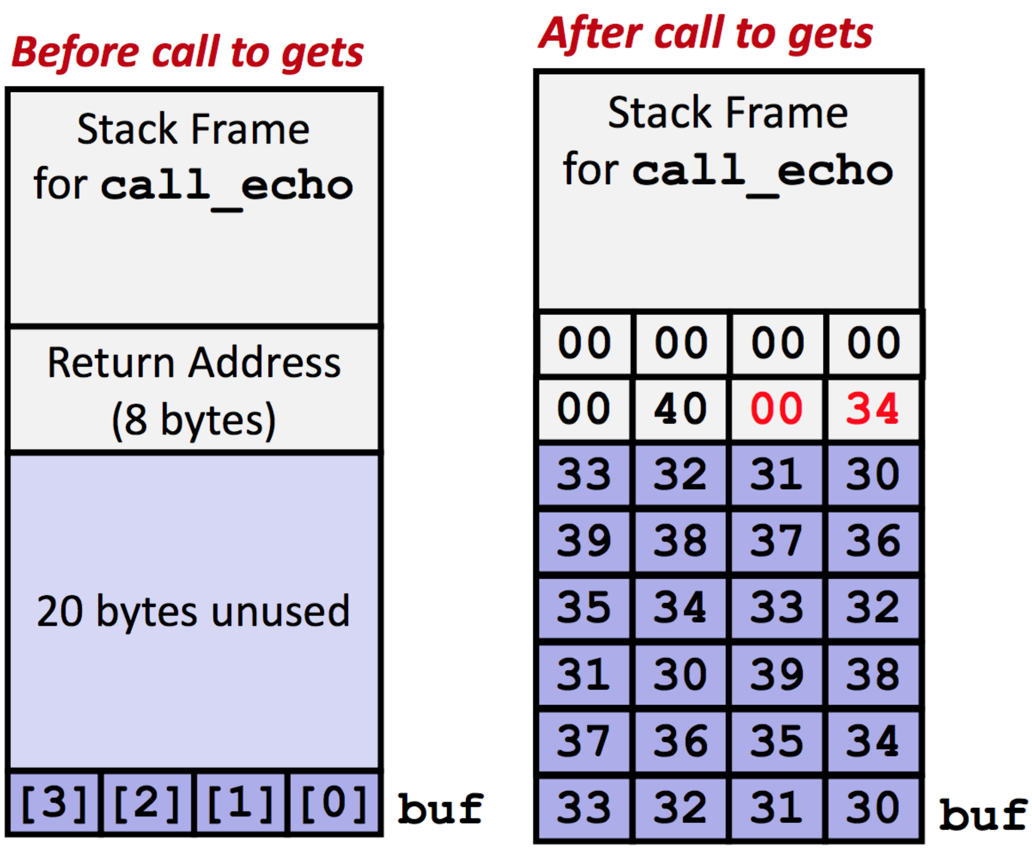
1. Buffer Overflow Stack

   - overflowed buffer
   - corrupted return pointer

   ```
   void echo() {
     char buf[4];
   ```

```
  gets(buf);
  puts(buf);
}

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

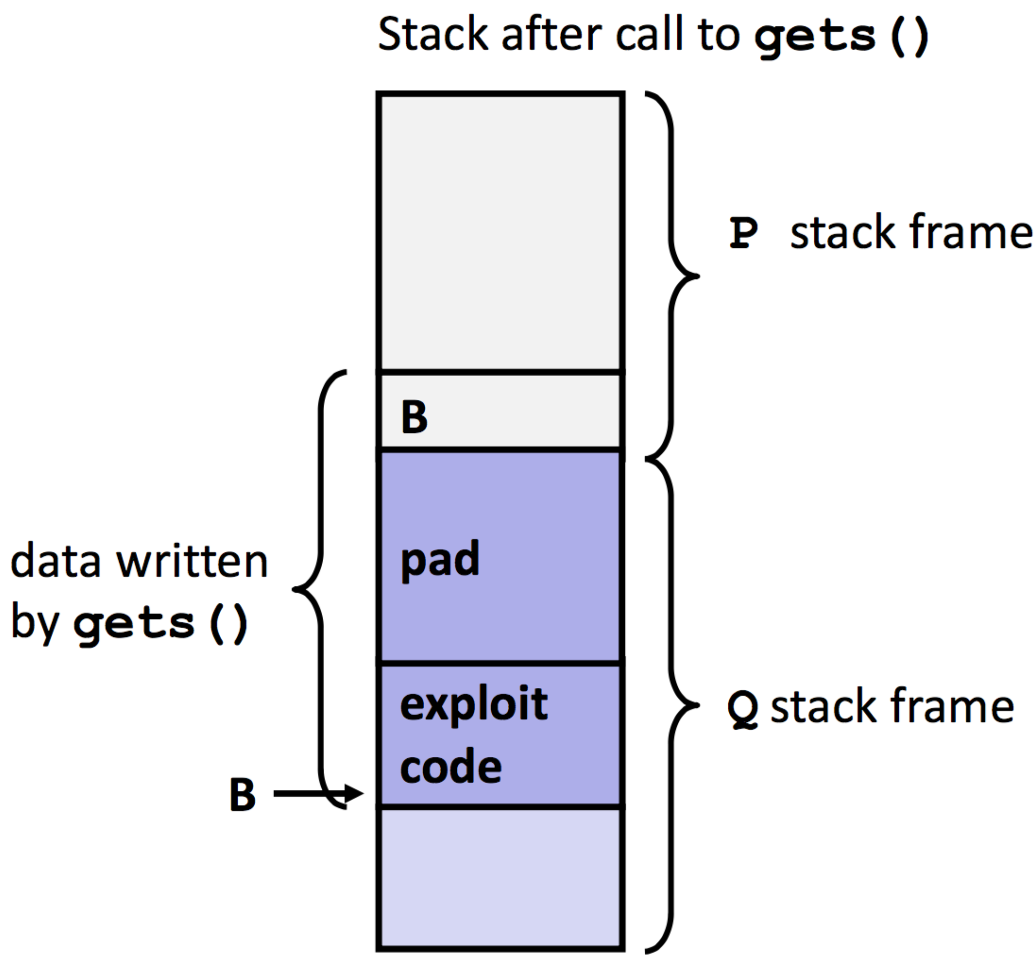


2. Code Injection Attack

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

```
}

void P() {
  Q();
}
```



- input string contains byte representation of executable code
- overwrite return address A with address of buffer B
- when `Q` executes `ret`, will jump to exploit code

2.2 Protection

2.2.1 Avoid Overflow Vulnerabilities in Code

- `fgets` instead of `gets`
- `strncpy` instead of `strcpy`
- don't use `scanf` with `%s` as format string
  - use `fgets` to read the string
  - or use `%ns` as format string provided to `scanf`, where n is a suitable integer

2.2.2 System Level Protection

- randomized stack offsets
  - at start of program, allocate random amount of space on stack
  - shifts stack address for entire program
  - makes it difficult for hackers to predict beginning of inserted code
- nonexecutable code segment
  - in traditional x86, can mark region of memory as either "read-only" or "writable"
    * can execute anything readable
  - x86-64 added explicit "execute permission"
  - stack marked as non-executable
- stack canaries
  - place special value canary on stack beyond buffer
  - check for corruption before exiting function
  - GCC implementation
    * enable with flag `-fstack-protector`

2.2.3 Return-Oriented Programming Attacks

- challenge
  - stack randomization makes it hard to predict buffer location
  - marking stack nonexecutable makes it hard to insert binary code
- alternative strategy
  - use existing code
    - library code from stdlib
  - string together fragments to achieve overall desired outcome
  - does not overcome stack canaries
- construct program from gadgets
  - sequence of instructions ending in ret
    - encoded by single byte `0x3c`
  - code positions fixed from run to run
  - code is executable

1. Gadget Example

   - use tail end of existing functions

```
long ab_plus_c(long a, long b, long c) {
  return a*b + c;
}
```

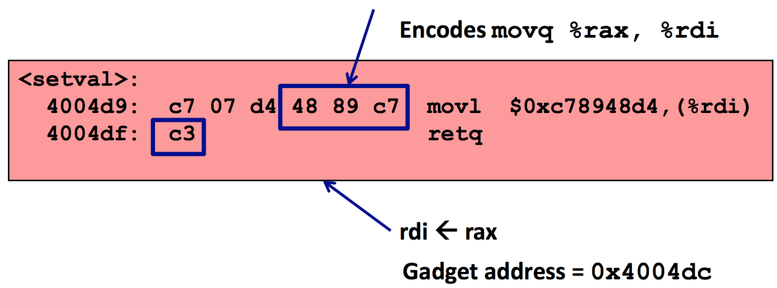
```
00000000004004d0 <ab_plus_c>:
  4004d0:  48 0f af fe  imul %rsi,%rdi
  4004d4:  48 8d 04 17  lea (%rdi,%rdx,1),%rax
  4004d8:  c3           retq
```

rax ← rdi + rdx

Gadget address = 0x4004d4

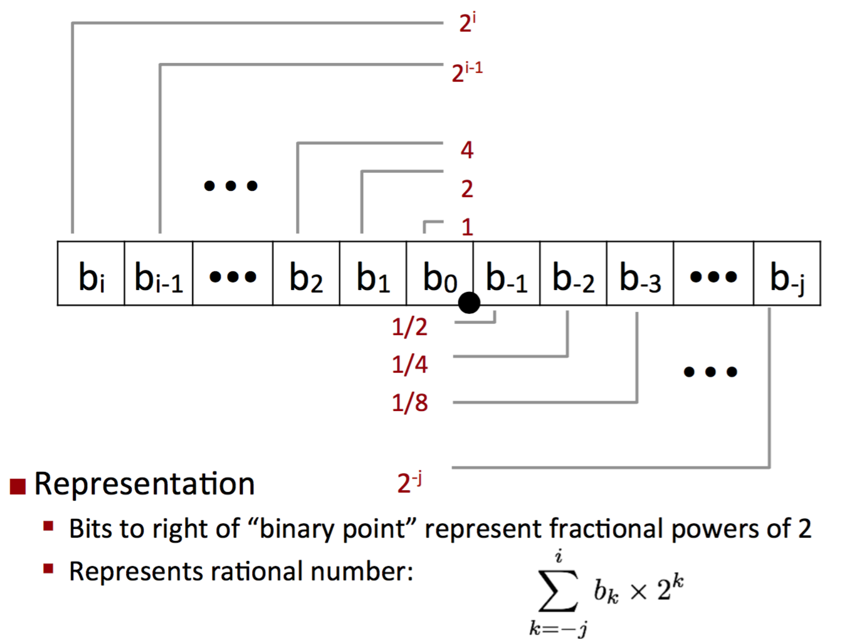
   - repurpose byte codes

```
void setval(unsigned *p) {
  *p = 3347663060u;
}
```



3 Float

3.1 Fractional Binary Numbers



- bits to right of 'binary point' represent fractional powers of 2
- representation of rational numbers $\sum_{k=-j}^{i} b_k \times 2^k$

3.1.1 example

```
value     |   representation
----------------------------
5 + 3/4   ==>  101.11
2 + 7/8   ==>   10.111
1 + 7/16  ==>    1.0111
```

- observations
  - divide by 2 by shifting right (unsigned)
  - multiply by 2 by shifting left
  - number of the form 0.11111_2 are just below 1.0
    * $\sum \frac{1}{2^i}$ goes to 1.0
    * use notation 1.0 - ϵ

3.1.2 limitations

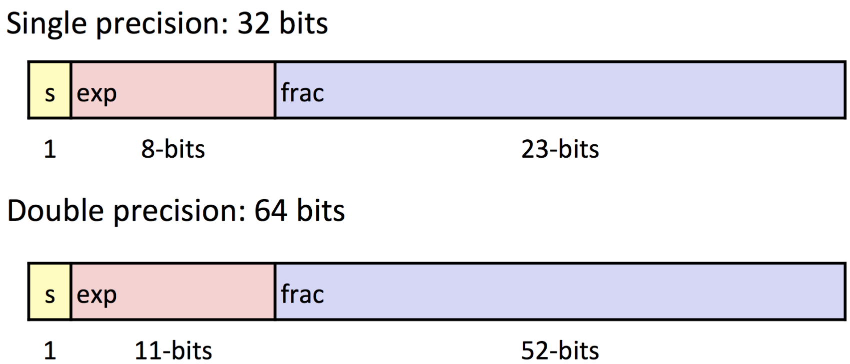
- can only reprsent numbers of the form x/2^{k}
  - other rational numbers have repeating bit representations
- just 1 setting of binary point within w bits
  - limited range of numbers

3.2 Floating Point Representation (IEEE Standard 754)

- numerical form

$$(-1)^s M 2^E$$

  - sign bit `s`, determines whether number is negative or positive
    * most significant bit is sign bit `s`
  - exponent `E`, weights value by power of 2
    * exp field encodes E (**but is not equal to E**)
  - significand `M`, is normally a fractional value $1.0 \leq x < 2.0$
    * frac field encodes M (**but is not equal to M**)



3.3 Normalized Values

- when exp $\neq$ 00...0 and exp $\neq$ 11...1
- exponent coded as a biased value: E = Exp - Bias
  - Exp: unsigned value of exp field
  - Bias = 2^{k-1} - 1, where k is number of exponent bits
    * single precision: Bias = 127
      · Exp: 1~254, E: -126~127
    * double precision: Bias = 1023
      · Exp: 1~2046, E: -1022~1023
- significand coded with implied leading 1: M = $1.xxx\ldots x_2$
  - xxx...x: bits of frac field
  - minimal when frac = 000...0
    * M = 1.0
  - maximal when frac = 111...1
    * M = 2.0 - ϵ
- get extra leading bit for free

3.3.1 example

15213_{10}

- as an integer 11101101101101_2

- as a float $1.1101101101101_2 \times 2^{13}$
  - significand
    - `M` $= 1.1101101101101_2$
    - `frac` $= 11011011011010000000000_2$
  - exponent
    - `E` $= 13$
    - `Bias` $= 127$
    - `Exp` $= 140 = 10001100_2$
  - result
    - 0 10001100 11011011011010000000000

```
15213

 11101101101101
1.1101101101101 * 2^13

Significand
M    = 1.1101101101101
frac =   11011011011010000000000

Exponent
E    = 13
Bias = 127
Exp  = 140 = 10001100

Result
0 10001100 11011011011010000000000
```

3.4 Denormalized Values

- when exp = 000…0
- exponent value: E = 1 - Bias (**instead of E = 0 - Bias**)
- significand coded with implied leading 0: M = $0.\text{xxx}\ldots\text{x}_2$
  - xxx…x: bits of frac field
- exp = 000…0, frac = 000…0

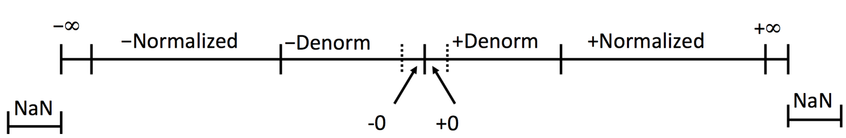
- represents zero value
- +0 (positive 0) : 0 00000000 00000000000000000000000
- -0 (negative 0) : 1 00000000 00000000000000000000000

- exp = 000...0, frac ≠ 000...0
  - numbers closest to 0.0
  - equispaced

3.5 Special Values

- when exp = 111...1
- exp = 111...1, frac = 000...0
  - represents value ∞ (infinity)
  - operation that overflows
  - both positive and negative
  - e.g. $1.0/0.0 = -1.0/-0.0 = +\infty, 1.0/-0.0 = -\infty$
- exp = 111...1, frac ≠ 000...0
  - Not-a-Number (NaN)
  - representation case when no numeric value can be determined
  - e.g. $\sqrt{-1}, \infty - \infty, \infty \times 0$

3.6 Visualization



$$(-1)^s M 2^E$$