Lec11: Miscellaneous Topics

Taesoo Kim

Two More Labs to Go!

Oct 28	Oct 29	Oct 30	Oct 31 **DUE:** Lab 07	Nov 01 **LEC:** Integer Overflows, Race Conditions [slides] **TUT:** Tut08: Logic Errors [video] **Assigned:** Lab08: Miscellaneous Topics
Nov 04	Nov 05	Nov 06	Nov 07 **DUE:** Lab 08	Nov 08 **LEC:** Designing Heap Allocator [slides] [note] [whiteboard] **TUT:** Tut09: Understanding Heap Bugs [video1], [video2] **Assigned:** Lab09: Exploiting Heap Bugs
Nov 11	Nov 12	Nov 13	Nov 14	Nov 15 **LEC:** Exploiting Heap Allocator [slides] **TUT:** Tut09: Exploiting Heap Allocators [video]

Administrivia

- In-class CTF: https://ctf.gts3.org/ (Open to public! Nov 22)
  - Registration: http://bit.ly/tkctf_register (#2-4 persons per team)
  - Rules: https://tc.gts3.org/cs6265/2024-fall/ctf.html
  - Submit your team's challenge by Nov 16
- NSA Codebreaker Challenge → Due: Dec 12

About CTF challenge

- Fork https://github.com/sslab-gatech/ctf-template and add our github IDs
  - Remote challenge
  - `exploit.py` and `test.py`
  - `patch.diff` for defense points
- DEMO

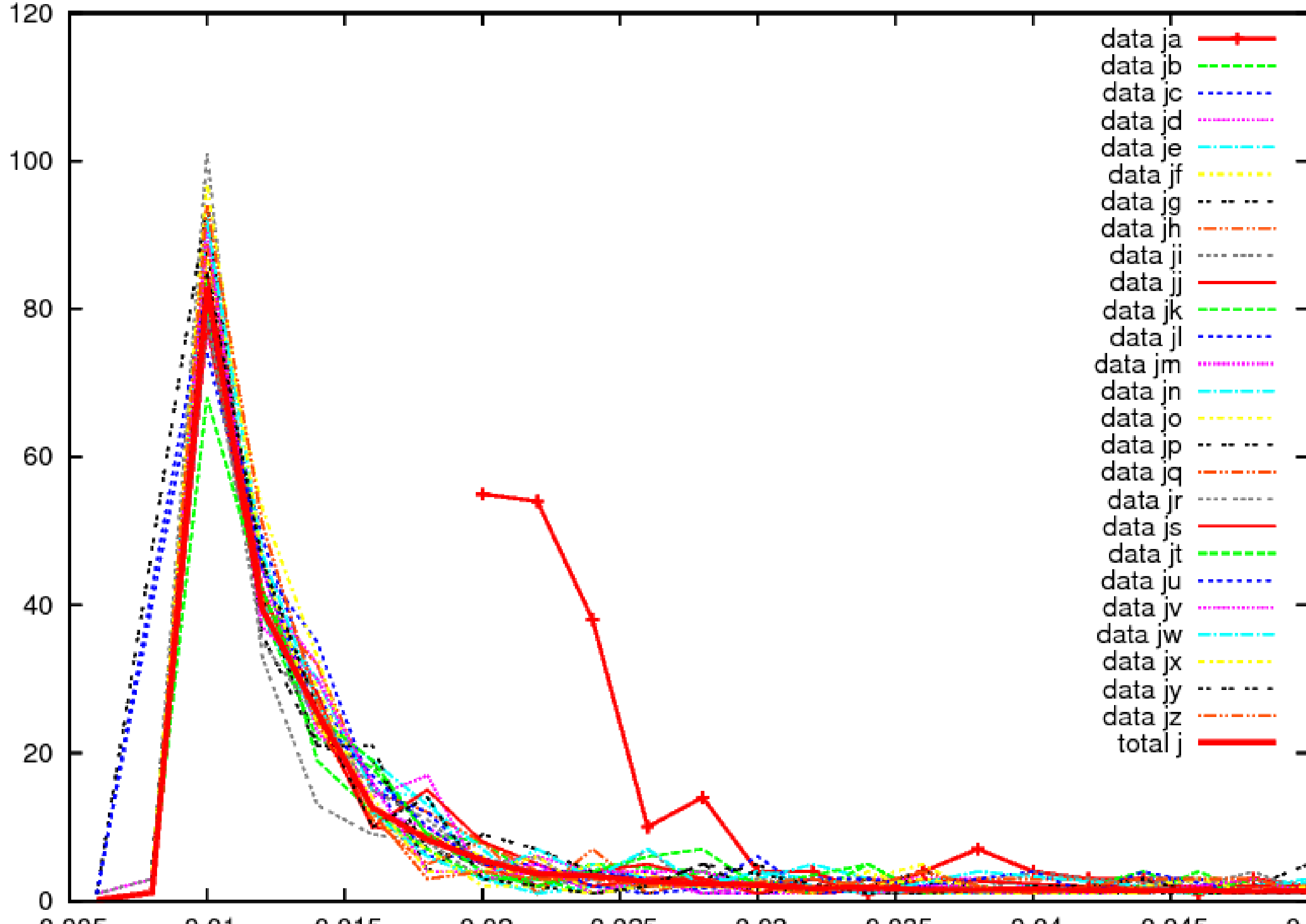
Summary of Lab07

- Remote environments impose unique challenges:
  - **Side-channels**: passwd (timing channel)
  - **Command injection**: mini-shellshock (via cgi params)
  - **Weak defense**: diehard (stack canary)
  - **Insufficient info**: 2048_game (guessable)
  - **Time-of-check-time-of-use**: memo (file size/read)
  - **Common attack vectors**: obscure (on ARM), array, fmtstr-heap2, 2kills, return-to-dl

Discussion: passwd

```
for (; cur < end; cur ++) {
  int c = fgetc(stdin);
  if (c == '\n')
    break;
  /* short circuit */
  if (*cur != c) {
    break;
  }
  /* NOTE. make it easlier */
  usleep(10000);
}
```

Discussion: passwd



Discussion: diehard

- What was the problem?

Discussion: diehard

- Problem: fork() does not change canary
- Exploit: change the last byte of canary one at a time
  - if correct, executed normally
  - if wrong, terminated
- 2^64 → 2^8 x 8 (now, tractable!)
- e.g., Apache stack overflow

Lab08: Miscellaneous

- **Integer overflow**
- Web
- Race condition
- Interesting exploit techniques, so miscellaneous

CS101: Integer Representation

Ref. https://en.wikipedia.org/wiki/Integer_overflow

CS101: Two's Complement Representation

The value w of an N-bit integer $a_{N-1}a_{N-2}\dots a_0$

$$w = -a_{N-1}2^{N-1} + \sum_{i=0}^{N-2} a_i 2^i.$$

```
e.g., in x86 (32-bit, 4-byte):
  - 0x00000000 ->  0
  - 0xffffffff -> -1
  - 0x7fffffff ->  2147483647 (INT_MAX)
  - 0x80000000 -> -2147483648 (INT_MIN)
```

Ref. https://en.wikipedia.org/wiki/Two's_complement

Arithmetic with Two's Complements

- One instruction works for **both** sign/unsigned integers (i.e., add, sub, mul)
  - e.g., add reg1, reg2 (not distinguishing signedness of reg1/2)
- Properties:
  - Non-symmetric representation of range, so single 0
  - MSB represents signedness: 1 means negative, 0 means non-negative

Arithmetic with Two's Complements

- One instruction works for **both** sign/unsigned integers (i.e., add, sub, mul)
  - e.g., add reg1, reg2 (not distinguishing signedness of reg1/2)
- Properties:
  - Non-symmetric representation of range, so single 0
  - MSB represents signedness: 1 means negative, 0 means non-negative

```
0x00000001 + 0x00000002 = 0x00000003 ( 1 + 2 = 3)
0xffffffff + 0x00000002 = 0x00000001 (-1 + 2 = 1)
0xffffffff + 0xfffffffe = 0xfffffffd (-1 +-2 =-3)

range(signed integer) = [-2^31, 2^31-1] = [-2147483648, 2147483647]
range(unsigned integer) = [0, 2^32-1] = [0, 4294967295]
```

Question!

- Then, how to interpret the arithmetic result?

```
; 0xffffffff + 0xfffffffe = 0xfffffffd (-1 +-2 =-3)

mov eax, 0xffffffff     ; eax = 0xffffffff
mov ebx, 0xfffffffd     ; ebx = 0xfffffffe
add eax, ebx            ; eax = 0xfffffffd
; eax = 0xfffffffd
; 1) is it -3?
; 2) is it 4294967293 (== 0xfffffffd)?
```

Idea: Using Status Flags (E/RFLAGS)

- **CF:** overflow of **unsigned** arithmetic operations
- **OF:** overflow of **signed** arithmetic operations

```
0x00000001 + 0x00000002 = 0x00000003 ( 1 + 2 = 3)
  -> CF: ?   OF: ?    SF: ?
```

Idea: Using Status Flags (E/RFLAGS)

- **CF:** overflow of **unsigned** arithmetic operations
- **OF:** overflow of **signed** arithmetic operations

```
0x00000001 + 0x00000002 = 0x00000003 ( 1 + 2 = 3)
  -> CF: 0   OF: 0    SF: 0
0xffffffff + 0x00000002 = 0x00000001 (-1 + 2 = 1)
  -> CF: ?   OF: ?    SF: ?
```

Idea: Using Status Flags (E/RFLAGS)

- **CF:** overflow of **unsigned** arithmetic operations
- **OF:** overflow of **signed** arithmetic operations

```
0x00000001 + 0x00000002 = 0x00000003 ( 1 + 2 = 3)
 -> CF: 0   OF: 0    SF: 0
0xffffffff + 0x00000002 = 0x00000001 (-1 + 2 = 1)
  -> CF: 1   OF: 0    SF: 0
0x80000000 + 0xffffffff = 0x7fffffff (-2147483648 + -1 =  2147483647)
  -> CF: ?   OF: ?    SF: ?
```

Idea: Using Status Flags (E/RFLAGS)

- **CF:** overflow of **unsigned** arithmetic operations
- **OF:** overflow of **signed** arithmetic operations

```
0x00000001 + 0x00000002 = 0x00000003 ( 1 + 2 = 3)
  -> CF: 0   OF: 0    SF: 0
0xffffffff + 0x00000002 = 0x00000001 (-1 + 2 = 1)
  -> CF: 1   OF: 0    SF: 0
0x80000000 + 0xffffffff = 0x7fffffff (-2147483648 + -1 =  2147483647)
  -> CF: 1   OF: 1    SF: 0
0x7fffffff + 0x00000001 = 0x80000000 ( 2147483647 +  1 = -2147483648)
  -> CF: ?   OF: ?    SF: ?
```

Idea: Using Status Flags (E/RFLAGS)

- **CF:** overflow of **unsigned** arithmetic operations
- **OF:** overflow of **signed** arithmetic operations

```
0x00000001 + 0x00000002 = 0x00000003 ( 1 + 2 = 3)
  -> CF: 0   OF: 0    SF: 0
0xffffffff + 0x00000002 = 0x00000001 (-1 + 2 = 1)
  -> CF: 1   OF: 0    SF: 0
0x80000000 + 0xffffffff = 0x7fffffff (-2147483648 + -1 =  2147483647)
  -> CF: 1   OF: 1    SF: 0
0x7fffffff + 0x00000001 = 0x80000000 ( 2147483647 +  1 = -2147483648)
  -> CF: 0   OF: 1    SF: 1
```

C's Integer Representation

		x86 (32b)	x86_64 (64b)
	char	: 1 byte	1 byte
	unsigned char	: 1 byte	1 byte
	short	: 2 bytes	2 bytes
	unsigned short	: 2 bytes	2 bytes
	int	: 4 bytes	4 bytes
	unsigned int	: 4 bytes	4 bytes
(*)	long	: 4 bytes	8 bytes
(*)	unsigned long	: 4 bytes	8 bytes
	long long	: 8 bytes	8 bytes
	unsigned long long	: 8 bytes	8 bytes
(*)	size_t	: 4 bytes	8 bytes
(*)	ssize_t	: 4 bytes	8 bytes
(*)	void*	: 4 bytes	8 bytes

Thinking of C's Type/Precision Conversion

- Lower → higher precision

```
        char -> int
[-128, 127] -> [-128, 127]
```

Thinking of C's Type/Precision Conversion

- Lower → higher precision

```
          char -> int
   [-128, 127] -> [-128, 127]
  [0x80, 0x7f] -> [0xffffff80, 0x0000007f]
                     ------> sign extended (e.g., movsx)

 unsigned char -> unsigned int
      [0, 255] -> [0, 255]
```

Thinking of C's Type/Precision Conversion

- Lower → higher precision

```
         char -> int
  [-128, 127] -> [-128, 127]
 [0x80, 0x7f] -> [0xffffff80, 0x0000007f]
                     ------> sign extended (e.g., movsx)

unsigned char -> unsigned int
     [0, 255] -> [0, 255]
    [0, 0xff] -> [0, 0x000000ff]
                        ------> zero extended (e.g., movzx)
```

Thinking of C's Type/Precision Conversion

- Higher → lower precision (what's correct mappings?)
- Mathematically complex, but architecturally simple (truncation!)

```
                     int -> char
[-2147483649, 2147483647] -> [-128, 127]
 [0x80000000, 0x7fffffff] -> [0x80, 0x7f]

          unsigned int -> unsigned char
         [0, 4294967295] -> [0, 255]
         [0, 0xffffffff] -> [0, 0xff]

        both simply, eax -> al  (by processor)
```

CS101: Question?

```
char c1 = 100;
char c2 = 3;
char c3 = 4;

c1 = c1 * c2 / c3;
```

CS101: Question?

```
char c1 = 100;
char c2 = 3;
char c3 = 4;

c1 = c1 * c2 / c3;
     ------- Q1?
     ------------ Q2?

  1) 300 / 4 = 75
  2) 300 (0x12c, which is > 1 byte) -> 0x2c / 4 = 11
```

Basic Concept: Integer Promotion

- Before any arithmetic operations,
- All integer types whose size is smaller than sizeof(int):
    1. Promote to int (if int can represent the whole range)
    2. Promote to unsigned int (if not)

Basic Concept: Integer Promotion

- Before any arithmetic operations,
- All integer types whose size is smaller than sizeof(int):
    1. Promote to int (if int can represent the whole range)
    2. Promote to unsigned int (if not)

```
e.g.,

c1 = (int)c1 * (int)c2 / (int)c3;
   = 100 * 3 / 4
   = 300 / 4
   = 75
```

CS101: Comparing un/signed ints

```
int si = -1;
unsigned int ui = 1;

if (si < ui) {
  return true; // Q1?
} else {
  return false; // Q2?
}
```

Example: char/unsigned char Addition

- Promote to int (if int can represent the whole range)

```
// by rule 1. -> (1)
char sc = SCHAR_MAX;
unsigned char uc = UCHAR_MAX;

long long sll = sc + uc;

    1) (long long)((int)sc + (int)uc)?
    2) (long long)sc + (long long)uc?
```

Example: int/unsigned int Comparison

- Promote to unsigned int (if not)

```
// by rule 2. -> (2)
int si = -1;
unsigned int ui = 1;

printf("%d\n", (int)(si < ui));
             1) ui promotes to int
                = -1 < 1
                = 1
             2) si promotes to unsigned int
                = 0xffffffff < 1
                = 0
```

Remark: Undefined Behaviors

- Overflow of **unsigned integers** are **well-defined** (i.e., wrapping)
- Overflow of **signed** integers are **undefined**
  - But well-defined to the processor (i.e., just wrapping in x86)
  - Optimization takes advantages of this, making it hard to understand

CS101: Int. Ovfl. and Undefined Behavior

1. (in x86_64) what does the expression `1 > 0` evaluate to?
   (a) 0 (b) 1 (c) NaN (d) -1 (e) undefined

CS101: Int. Ovfl. and Undefined Behavior


```
1. (in x86_64) what does the expression  1 > 0  evaluate to?
    (a) 0    (b) 1    (c) NaN    (d) -1    (e) undefined

>> (b)

   (int) 1 > (int) 0
```

CS101: Int. Ovfl. and Undefined Behavior

2. `(unsigned short)1 > -1?`

   (a) `1`   (b) `0`   (c) `-1`   (d) `undefined`

CS101: Int. Ovfl. and Undefined Behavior


```
2. (unsigned short)1 > -1?
    (a) 1   (b) 0   (c) -1    (d) undefined

>> (a)

   unsigned short can be represented by int
   (int)(unsigned short)1 > (int)-1
```

CS101: Int. Ovfl. and Undefined Behavior

3. `-1U > 0?`

   (a) `1`   (b) `0`   (c) `-1`   (d) `undefined`

CS101: Int. Ovfl. and Undefined Behavior

```
3. -1U > 0?
    (a) 1   (b) 0   (c) -1    (d) undefined

>> (a)

   unsigned int can't be represented by int,
     so promote to unsigned int
   (unsigned int)(-1U) = 0xffffffff > 0
```

CS101: Int. Ovfl. and Undefined Behavior


```
5. abs(-2147483648), abs(INT_MIN) in x86_32?
    (a) 0  (b) < 0  (c) > 0  (d) NaN
```

CS101: Int. Ovfl. and Undefined Behavior


```
5. abs(-2147483648), abs(INT_MIN) in x86_32?
    (a) 0  (b) < 0  (c) > 0  (d) NaN

>> (b)
   Undefined, but the way the processor works:

   int abs (int i) {
     return i < 0 ? -i : i;
   }

   Q. What about in x86 (64-bit)?
```

CS101: Int. Ovfl. and Undefined Behavior

6. `1U << 0`?

   (a) 1 (b) 4 (c) UINT_MAX (d) 0 (e) undefined

CS101: Int. Ovfl. and Undefined Behavior


```
6. 1U << 0?
    (a) 1   (b) 4  (c) UINT_MAX  (d) 0  (e) undefined

>> (a)
```

CS101: Int. Ovfl. and Undefined Behavior


```
7. 1U << 32?
    (a) 1   (b) 4  (c) UINT_MAX  (d) INT_MIN  (e) 0  (f) undefined
```

CS101: Int. Ovfl. and Undefined Behavior

```
7. 1U << 32?
    (a) 1   (b) 4  (c) UINT_MAX  (d) INT_MIN  (e) 0  (f) undefined

>> (f) in C

   x86 (32-bit), 1U << 32 == 1!
   shl edx,cl

   Q. 1U << -1?
```

CS101: Int. Ovfl. and Undefined Behavior

```
8. -1L << 2?
    (a) 0   (b) 4  (c) INT_MAX  (d) INT_MIN   (e) undefined
```

CS101: Int. Ovfl. and Undefined Behavior

```
8. -1L << 2?
    (a) 0   (b) 4  (c) INT_MAX  (d) INT_MIN   (e) undefined

>> (e) in C

   shift operations on sign integers are undefined

   x86 (32-bit), -1L << 2 == -4!
   edx = 0xffffffff
   cl = 2
   shl edx,cl
   vs.
   sal (signed shift, arithmatic shift)
```

CS101: Int. Ovfl. and Undefined Behavior


```
9. INT_MAX + 1?
    (a) 0   (b) 1  (c) INT_MAX  (d) UINT_MAX  (e) undefined
```

CS101: Int. Ovfl. and Undefined Behavior


```
9. INT_MAX + 1?
    (a) 0   (b) 1  (c) INT_MAX  (d) UINT_MAX  (e) undefined

>> (e) in C

   overflow in sign integers are undefined!

   x86 (32-bit), 0x7fffffff + 1 = 0x80000000
   eax = 0x7fffffff
   ecx = 1
   add eax, ecx
```

CS101: Int. Ovfl. and Undefined Behavior

- Q. How does the compiler take advantage of undefined behaviors?

```
int a = atoi(argv[1]);
if (a > 0) {
  if (a + 1 > 0) {
    printf("a+1 > 0\n");
  } else {
    printf("?!\n");
  }
}
```

CS101: Int. Ovfl. and Undefined Behavior


```
10. UINT_MAX + 1?
    (a) 0   (b) 1  (c) INT_MAX  (d) UINT_MAX  (e) undefined
```

CS101: Int. Ovfl. and Undefined Behavior


```
10. UINT_MAX + 1?
    (a) 0   (b) 1  (c) INT_MAX  (d) UINT_MAX  (e) undefined

>> (a)
```

CS101: Int. Ovfl. and Undefined Behavior

11. `-INT_MIN`?

    (a) `0` (b) `1` (c) `INT_MAX` (d) `UINT_MAX` (e) `INT_MIN`

    (f) undefined

CS101: Int. Ovfl. and Undefined Behavior


```
11. -INT_MIN?
    (a) 0   (b) 1  (c) INT_MAX  (d) UINT_MAX  (e) INT_MIN
    (f) undefined

>> (f) in C but reuslts in (e)
```

CS101: Int. Ovfl. and Undefined Behavior

12. `-1L > 1U?` on `x86_64` and `x86`

    (a) (0, 0) (b) (1, 1) (c) (0, 1) (d) (1, 0)

    (e) undefined

CS101: Int. Ovfl. and Undefined Behavior

```
12. -1L > 1U? on x86_64 and x86
    (a) (0, 0)  (b) (1, 1)  (c) (0, 1)  (d) (1, 0)
    (e) undefined

>> (c)

   x86_64: sizeof(long) > sizeof(unsigned int)
     -> (long)-1L > (long)1U
   x86: sizeof(long) == sizeof(unsigned int)
     -> (unsigned int)-1L > (unsigned int) 1U
```

Today's Tutorial

- In-class tutorial:
  - Logical vulnerability
  - Race condition and commnadline injection

```
$ ssh lab08@3.95.14.86
Password: <password>

$ cd tut08-logic-bugs
```