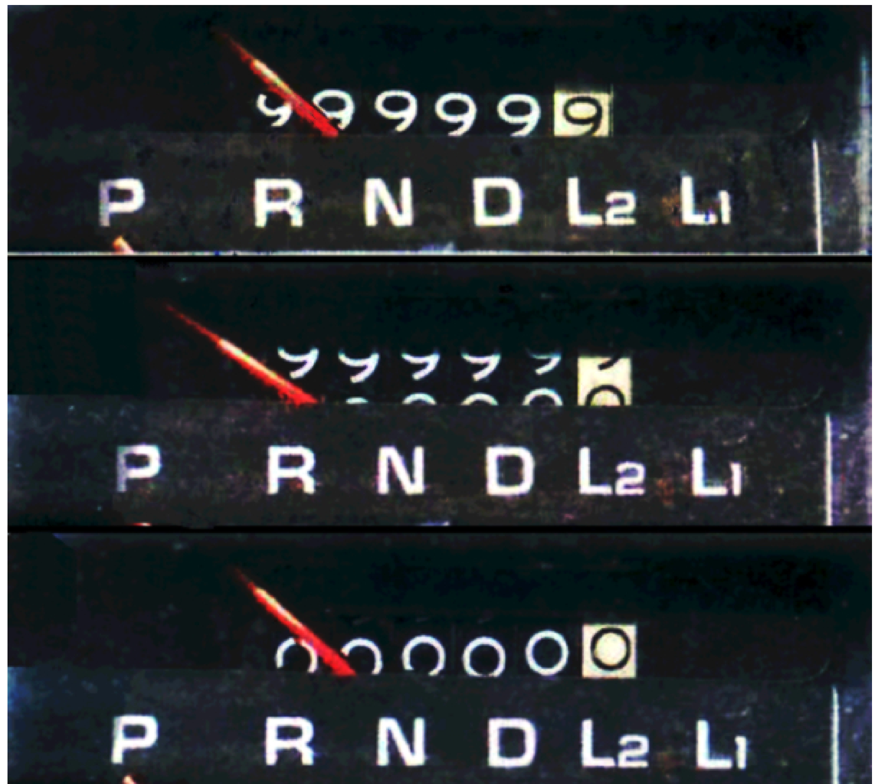
Lec03: Integer Overflow and Undefined Behavior

Taesoo Kim

Goals and Lessons

- Learn about two classes of vulnerabilities
    1. Integer overflow
    2. Undefined behavior
- Understand their security implications
- Understand best security practices
- Learn them from the real-world examples (Android, Linux, etc)

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 ->  2147483648 (INT_MAX)
    - 0x80000000 -> -2147483649 (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 positive

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

range(signe integer) = [-2^31-1, 2^31] = [-2147483649, 2147483648]
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: 0   OF: 0    SF: 0
0xffffffff + 0x00000002 = 0x00000001 (-1 + 2 = 1)
  -> CF: 1   OF: 0    SF: 0
0x80000000 + 0xffffffff = 0x7fffffff (-2147483649 + -1 =  2147483648)
  -> CF: 1   OF: 1    SF: 0
0x7fffffff + 0x00000001 = 0x80000000 ( 2147483648 +  1 = -2147483649)
  -> CF: 0   OF: 1    SF: 1
```

C’s Integer Representation

```
                          x86 (32b)     x86_64 (64b)
    char                : 1 bytes       1 bytes
    unsigned char       : 1 bytes       1 bytes
    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]
  [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, 2147483648] -> [-128, 127]
 [0x80000000, 0x7fffffff] -> [0x80, 0x7f]

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

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

Example of Precision Conversion

```
$ cd lec03-intovfl/intovfl
$ ./intovfl2
0x7fffffff ->
  (unsigned int)(unsigned char): 000000ff
  ; mov eax, 0x7fffffff
  ; movzx eax, al

  (unsigned int)(char)         : ffffffff
  ; mov eax, 0x7fffffff
  ; movsx eax, al
```

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 > 8 bytes) -> 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)

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) (unsigned long long)((int)sc + (int)uc)?
    2) (unsigned long long)sc + (unsigned 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

```
>> (a)

   (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

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

CS101: Int. Ovfl. and Undefined Behavior

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

>> (a)

   simply wrap around: 0xffffffff + 1 = 0 (CR = 1)
```

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
```

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!
```

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

CS101: Int. Ovfl. and Undefined Behavior

```
BONUS. is it possible that a / b < 0 when a < 0 and b < 0?
```

CS101: Int. Ovfl. and Undefined Behavior

BONUS. is it possible that `a / b < 0` when `a < 0` and `b < 0`?

```
a = INT_MIN && b == -1!
```

Classifying Integer-related Vulnerabilities

1. Precision (or widthness) overflows
2. Arithmetic overflows
3. Signedness bugs
4. Undefined behaviors (e.g., time bomb)

1. Precision Error: CVE-2015-1593

- Arithmetic operations b/w unsigned int and unsigned long
- Reducing ASLR entropy by four in Linux (in x86_64)

```
// @fs/binfmt_elf.c
static unsigned long randomize_stack_top(unsigned long stack_
  unsigned int random_variable = 0;
  if ((current->flags & PF_RANDOMIZE) &&
      !(current->personality & ADDR_NO_RANDOMIZE)) {
    random_variable = get_random_int() & STACK_RND_MASK;
*   random_variable <<= PAGE_SHIFT;
  }
  return PAGE_ALIGN(stack_top) - random_variable;
}
```

CVE-2015-1593: Patch

- unsigned int → unsigned long (match the precision)
- Be careful when you'd like to multiple architecture

```
// @fs/binfmt_elf.c
static unsigned long randomize_stack_top(unsigned long stack_
* unsigned long random_variable = 0;
  if ((current->flags & PF_RANDOMIZE) &&
      !(current->personality & ADDR_NO_RANDOMIZE)) {
    random_variable = get_random_int() & STACK_RND_MASK;
*   random_variable <<= PAGE_SHIFT;
  }
  return PAGE_ALIGN(stack_top) - random_variable;
}
```

2. Arithmetic Overflow: CVE-2018-6092

- Arithmetic overflows when adding two unsigned ints
- Result in *remote* code execution in Chrome (V8/WASM, 32-bit)

```
// @src/wasm/function-body-decoder-impl.h
//  count: unsigned int
//  type_list->size(): size_t
//  kV8MaxWasmFunctionLocals: size_t

* if ((count + type_list->size())
*       > kV8MaxWasmFunctionLocals) {
    decoder->error(decoder->pc()-1, "local count too large")
    return false;
  }
```

CVE-2018-6092: Patch

- Standard pattern/fix
- Avoid potential arithmetic overflows

```
// @src/wasm/function-body-decoder-impl.h
//  count: unsigned int
//  type_list->size(): size_t
//  kV8MaxWasmFunctionLocals: size_t

+ DCHECK_LE(type_list->size(), kV8MaxWasmFunctionLocals);
+ if (count
+       > kV8MaxWasmFunctionLocals - type_list->size()) {
    decoder->error(decoder->pc()-1, "local count too large")
    return false;
  }
```

3. Signedness Bugs: CVE-2017-7308

- Casting the arithmetic result of unsigned ints to sign for comparison
- Result in *remote* code execution in Linux (network stack)

```
// @net/packet/af_packet.c
//   req->tp_block_size: unsigned int
//   BLK_PLUS_PRIV(..) : unsigned int

// Q. What if we don't have (int)?
if (po->tp_version >= TPACKET_V3 &&
*   (int)(req->tp_block_size -
*         BLK_PLUS_PRIV(req_u->req3.tp_sizeof_priv)) <= 0)
    goto out;
```

3. Signedness Bugs: CVE-2017-7308

```
// @net/packet/af_packet.c
//   req->tp_block_size: unsigned int
//   BLK_PLUS_PRIV(..) : unsigned int

// Q. What if we don't have (int)?
if (po->tp_version >= TPACKET_V3 &&
//  (int)(u1 - u2) <= 0
//    => (unsigned int)100 - (unsigned int)(-200)
//    => (int)300
//    => (unsigned int)100 < (unsigned int)(-200)
*   (int)(req->tp_block_size -
*         BLK_PLUS_PRIV(req_u->req3.tp_sizeof_priv)) <= 0)
    goto out;
```

CVE-2017-7308: Patch

- Direct comparison of unsigned ints!
- Fix a potential overflow inside the macro as well

```
// @net/packet/af_packet.c
//   req->tp_block_size: unsigned int
//   BLK_PLUS_PRIV(..) : unsigned long long

if (po->tp_version >= TPACKET_V3 &&
*   req->tp_block_size
*     <= BLK_PLUS_PRIV((u64)req_u->req3.tp_sizeof_priv))
    goto out;
```

Testing Signedness Bugs

```
$ cd lec03-intovfl/intovfl
$ make

$ ./uintcmp
0 < 1 = 1?
  (int)(0 - 1) == -1 <= 0? -> 1
1 < 0 = 0?
  (int)(1 - 0) == 1 <= 0? -> 0
10 < 4294967196 = 0?
  (int)(10 - 4294967196) == 110 <= 0? -> 0
unsigned int a = ?
unsigned int b = ?
```

4. Undefined Behaviors: NaCL/x86

- Shifting more than the available #bits
- Result in the entire sandbox sequence no-op!

```
// @NaClSandboxAddr()
                                             +--> 32 bytes in x32
                                             |
                                             +------------------
return addr & ~(uintptr_t)((1 << nap->align_boundary) - 1);
                                   +-------------------------
                                   |
                                   +--> (1 << 32) -> 1 in gcc!
```

Ref. https://bugs.chromium.org/p/nativeclient/issues/detail?id=245

Secure Way to Add Two Ints

- SEI CERT C Coding Standard

```
void func(signed int si_a, signed int si_b) {
  signed int sum = si_a + si_b;
  /* ... */
}
```

Secure Way to Add Two Ints

```c
#include <limits.h>

void f(signed int si_a, signed int si_b) {
  signed int sum;
  if (((si_b > 0) && (si_a > (INT_MAX - si_b))) ||
      ((si_b < 0) && (si_a < (INT_MIN - si_b)))) {
    /* Handle error */
  } else {
    sum = si_a + si_b;
  }
  /* ... */
}
```

Secure Way to Multiplying Two Ints

```
void func(signed int si_a, signed int si_b) {
  signed int result = si_a * si_b;
  /* ... */
}
```

Secure Way to Multiplying Two Ints

```
void func(signed int si_a, signed int si_b) {
  signed int result;
  signed long long tmp;

  tmp = (signed long long)si_a * (signed long long)si_b;

  /* If the product cannot be represented as a 32-bit intege
     handle as an error condition. */
  if ((tmp > INT_MAX) || (tmp < INT_MIN)) {
    /* Handle error */
  } else {
    result = (int)tmp;
  }
  /* ... */
}
```

Secure Way to Multiplying Two Ints

The following portable compliant solution can be used with any conforming implementation, including those that do not have an integer type that is at least twice the precision of `int`:

```
#include <limits.h>

void func(signed int si_a, signed int si_b) {
  signed int result;
  if (si_a > 0) {  /* si_a is positive */
    if (si_b > 0) {  /* si_a and si_b are positive */
      if (si_a > (INT_MAX / si_b)) {
        /* Handle error */
      }
    } else { /* si_a positive, si_b nonpositive */
      if (si_b < (INT_MIN / si_a)) {
        /* Handle error */
      }
    } /* si_a positive, si_b nonpositive */
  } else { /* si_a is nonpositive */
    if (si_b > 0) { /* si_a is nonpositive, si_b is positive */
      if (si_a < (INT_MIN / si_b)) {
        /* Handle error */
      }
    } else { /* si_a and si_b are nonpositive */
      if ( (si_a != 0) && (si_b < (INT_MAX / si_a))) {
        /* Handle error */
      }
    } /* End if si_a and si_b are nonpositive */
  } /* End if si_a is nonpositive */

  result = si_a * si_b;
}
```

Case Study: Chackra Core (Multiplying Ints)

```
// Returns true if we overflowed, false if we didn't
bool Int64Math::Mul(int64 left, int64 right, int64 *pResult)
#if defined(_M_X64)
  // (I)MUL (Q/64) R[D/A]X <- RAX * r/m64
  int64 high;
  *pResult = _mul128(left, right, &high);
  return ((*pResult > 0) && high != 0) \
           || ((*pResult < 0) && (high != -1));
#else
  *pResult = left * right;
  return (left != 0 && right != 0 \
           && (*pResult / left) != right);
#endif
}
```

Case Study: glibc (Multiplying UInts)

```
static inline bool
check_mul_overflow_size_t(size_t left, size_t right,
                          size_t *result) {
  /* size_t is unsigned
     so the behavior on overflow is defined. */
  *result = left * right;
  size_t half_size_t \
    = ((size_t) 1) << (8 * sizeof (size_t) / 2);

  if ((left | right) >= half_size_t) {
    if (right != 0 && *result / right != left)
      return true;
  }
  return false;
}
```

New Proposals: __builtin_*_overflow()

- Ref. https://gcc.gnu.org/onlinedocs/gcc/Integer-Overflow-Builtins.html

```
bool __builtin_add_overflow (type1 a, type2 b, type3 *res);
bool __builtin_sub_overflow (type1 a, type2 b, type3 *res);
bool __builtin_mul_overflow (type1 a, type2 b, type3 *res);

bool __builtin_uadd_overflow (unsigned int a, unsigned int b,
                              unsigned int *res);
...
```

Taking Advantage of E/RFLAGS in x86

```
bool __builtin_add_overflow (type1 a, type2 b, type3 *res);

11bb:   mov    ecx,0x0
11c0:   add    eax,edx
11c2:   jno    11c9 <main+0x70>
11c4:   mov    ecx,0x1
llc9:   ...
```

Example: New calloc()

```c
void * calloc (size_t x, size_t y) {
  size_t sz;
  if (__builtin_mul_overflow (x, y, &sz))
    return NULL;
  void *ret = malloc (sz);
  if (ret)
    memset (ret, 0, sz);
  return ret;
}
```

Integer Overflow Beyond Security

- 1996: Ariane 5 rocket crashed
- 2015: FAA requested to reset Boeing 787 every 248 days
- 2016: a Casino machine printed a prize ticket of $42,949,672!

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

Not Abusing Int-related BU in Optimizer

- Option: -fwrapv (gcc/clang)

```c
// $ cd lec03-intovfl/intovfl
// $ make check-fwrapv

int base;
scanf("%d", &base);

// Q. base = INT_MAX?
if (base < base + 1)
  printf("base < base + 1 is true!\n");
```

Exercise: Real-world Examples

- Ex1. Android Stagefright (CVE-2015-1538, CVE-2015-3824)
- Ex2. Linux Keyring (CVE-2016-0728)

Exercise: Real-world Examples

- Ex1. Android Stagefright (CVE-2015-1538, CVE-2015-3824)
  - → Remote exploit on Android via MMS
- Ex2. Linux Keyring (CVE-2016-0728)
  - → Local privilege escalation

CVE-2015-1538: Android (Stagefright)

- Result in out-of-bound writes after integer overflows

```
// @edd4a76eb4747bd19ed122df46fa46b452c12a0d
// CVE-2015-1538
// uint32_t mTimeToSampleCount;

    mTimeToSampleCount = U32_AT(&header[4]);
    mTimeToSample = new uint32_t[mTimeToSampleCount * 2];
    ...
    for (uint32_t i = 0; i < mTimeToSampleCount * 2; ++i)
      mTimeToSample[i] = ntohl(mTimeToSample[i]);
    ...
```

Attempt1 to Fix Stagefright

```
// @edd4a76eb4747bd19ed122df46fa46b452c12a0d
// CVE-2015-1538
// uint32_t mTimeToSampleCount;

     mTimeToSampleCount = U32_AT(&header[4]);
+    uint64_t allocSize \
+      = mTimeToSampleCount * 2 * sizeof(uint32_t);
+    if (allocSize > SIZE_MAX) {
+      return ERROR_OUT_OF_RANGE;
+    }
     mTimeToSample = new uint32_t[mTimeToSampleCount * 2];
     ...
```

Attempt2 to Fix Stagefright

- RHS: (unsigned int)mTimeToSampleCount * 2 * sizeof(uint32_t)!

```
// @e2e812e58e8d2716b00d7d82db99b08d3afb4b32
     mTimeToSampleCount = U32_AT(&header[4]);
+    uint64_t allocSize \
+      = mTimeToSampleCount * 2 * (uint64_t)sizeof(uint32_t)
+    if (allocSize > SIZE_MAX) {
+      return ERROR_OUT_OF_RANGE;
+    }
     mTimeToSample = new uint32_t[mTimeToSampleCount * 2];
     ...
```

Attempt3 to Fix Stagefright

- RHS: (uint64_t)sizeof(uint32_t) is constant, again!

```
// @a105482ae577852ffd08ce88ae5d1ba81db875ac
         mTimeToSampleCount = U32_AT(&header[4]);
+        uint64_t allocSize \
+          = (uint64_t)mTimeToSampleCount * 2 * sizeof(uint32_t)
+        if (allocSize > SIZE_MAX) {
+          return ERROR_OUT_OF_RANGE;
+        }
         mTimeToSample = new uint32_t[mTimeToSampleCount * 2];
         ...
```

Attempt4 to Fix Stagefright

- RHS: allocSize > SIZE_MAX in 64-bit!

```
// @a105482ae577852ffd08ce88ae5d1ba81db875ac
     mTimeToSampleCount = U32_AT(&header[4]);
+    uint64_t allocSize \
+      = (uint64_t)mTimeToSampleCount * 2 * sizeof(uint32_t)
+    if (allocSize > UINT32_MAX) {
+      return ERROR_OUT_OF_RANGE;
+    }
     mTimeToSample = new uint32_t[mTimeToSampleCount * 2];
     ...
```

CVE-2015-1538: Finally!

```
// @a3630a418b4f65277a42cd4018cd3b0b7e134d0c

+ if (mTimeToSampleCount >
+         UINT32_MAX / (2 * sizeof(uint32_t))) {
+     // Choose this bound because
+     // 1) 2 * sizeof(uint32_t) is the amount of memory neede
+     //     for one time-to-sample entry in the time-to-sample
+     //     table.
+     // 2) mTimeToSampleCount is the number of entries of the
+     //     time-to-sample table.
+     // 3) We hope that the table size does not exceed
+     //     UINT32_MAX.
          return ERROR_OUT_OF_RANGE;
    }
    ...
```

CVE-2015-3824: Android (Stagefright)

```
// @463a6f807e187828442949d1924e143cf07778c6
// CVE-2015-3824
-    uint8_t *buffer = new uint8_t[size + chunk_size];
+    if (SIZE_MAX - chunk_size <= size) {
+      return ERROR_MALFORMED;
+    }
+
+    uint8_t *buffer = new uint8_t[size + chunk_size];
```

CVE-2016-0728: Linux Keyring

- Forcing the refcnt to be overflowed, resulting in privlege escalation!

```
long join_session_keyring(const char *name) {
  // NOTE. refcnt is incremented on "success"!
  keyring = find_keyring_by_name(name, false);
  if (PTR_ERR(keyring) == -ENOKEY) { ... }
  else if (IS_ERR(keyring)) { ... }
  } else if (keyring == new->session_keyring) {
    ret = 0;
    goto error2;
  }
  ...
  key_put(keyring);
 error2:
  return ret;
}
```

Summary

- Two classes of vulnerabilities:
    1. Integer overflow
    2. Undefined behavior
- They are likely to have serious security implications when happened!
- Common patterns and known ways to mitigate/elliminate them!

References

- Stagefright Bugs
- Basic Integer Overflows
- Integer Rules
- CVE-2017-7308
- CVE-2018-6092
- CVE-2015-1593