Lec 7: Integer Overflow

IS561: Binary Code Analysis and Secure Software Systems

Sang Kil Cha

Integer Overflow

What is Integer Overflow?

Logically,
`0xffffffff + 1 = 0x100000000.`

But, in reality, on x86 (32-bit),
`0xffffffff + 1 = 0.`

What is Integer Overflow?

Logically,
`0xffffffff + 1 = 0x100000000.`

But, in reality, on x86 (32-bit),
`0xffffffff + 1 = 0.`

Why does it happen?

Width Overflow

- On x86
  - Unsigned integer 4,294,967,295 + 1 = 0 (4294967295 = `0xffffffff`)
- On x86-64,
  - Unsigned integer 18,446,744,073,709,551,615 + 1 = 0 (18446744073709551615 = `0xffffffffffffffff`)

Signedness Overflow

- On x86,
  - `MAX_INT = 2,147,483,647 = 0x7fffffff`
  - `MIN_INT = -2,147,483,648 = 0x80000000`
  - So, `(int) 2147483647 + 1 = - 2147483648`

Integer Underflow

Integer underflow is the same as integer overflow except that it happens when the result is less than the minimum value[1].

```
int i = 0x80000000; // -2147483648
i = i - 1;
printf("%d\n", i); // prints 2147483647
```

[1] We will simply use the term integer overflow to refer to both cases.

Subtlety in C

Some integer overflows are undefined in C, meaning that their behaviors may vary across different platforms.

For example, on a 32-bit CPU, `1 << 32` is undefined. It could be 0 or not.

Signed Overflows are Undefined

Unsigned overflows are defined, but signed overflows are not.

For example, `INT_MAX + 1` is undefined, but `UINT_MAX + 1` is defined (is zero).

Undefined Behavior Example[2]

"`INT_MAX + 1`" is undefined in C, although "`UINT_MAX + 1`" is defined.

```
int foo(int x) { return (x + 1) >  x; }
int main(void)
{
  printf("%d\n", (INT_MAX+1) > INT_MAX);
  printf("%d\n", foo(INT_MAX));
  return 0;
}
```

The above program prints out 0 and 1. Why?

[2] Understanding Integer Overflow in C/C++, ICSE 2012

Why `foo` returns 1?

Disassembly of `foo`

```
mov   eax,0x1
ret
```

Compilers can silently break things, but is this a compiler bug?

More Subtle Example

```
int addsi(int lhs, int rhs)
{
    if ((((lhs + rhs) ^ lhs) & ((lhs + rhs) ^ rhs))
        >> (sizeof(int) * 8 - 1)) {
        abort(); // detect integer overflow
    }
    return lhs + rhs;
}
```

Is this safe? Is this buggy?

Are Integer Overflows Exploitable?

- Not always.
- But sometimes, integer overflows can cause an ***unexpected*** memory corruption (thus, it could lead to a control flow hijack!)

Example 1

```
int catvars(char *buf1, char *buf2, unsigned len1, unsigned len2)
{
    char mybuf[256];

    if((len1 + len2) > 256){    // len1=0x104, and len2=0xfffffffc ?
        return -1;
    }

    memcpy(mybuf, buf1, len1);  // 0x104 = 260 (overflow already)
    memcpy(mybuf + len1, buf2, len2);

    do_some_stuff(mybuf);

    return 0;
}
```

Example 2 (OpenSSH)

```
void input_userauth_info_response(int type, unsigned int seq, void *ctxt)
{
  int i;
  unsigned int nresp;
  char **response = NULL;
  ...
  nresp = packet_get_int(); // user input
  if (nresp > 0) {
    response = malloc(nresp * sizeof(char*));
    for (i = 0; i < nresp; i++)
      response[i] = packet_get_string(NULL);
  }
  packet_check_eom();
  ...
}
```

Example 3 (gzip)

```
unsigned w, d; // offsets
if (w - d >= e) // to avoid overlap
{
  memcpy(ptr + w, ptr + d, e);
  w += e;
  d += e;
}
```

Can you spot the problem?

Example 3 (gzip)

```
unsigned w, d; // offsets
if (w - d >= e) // to avoid overlap
{
  memcpy(ptr + w, ptr + d, e);
  w += e;
  d += e;
}
```

Can you spot the problem? What if `d` $\geq$ `w`?

The fix was: `if (d < w && w - d >= e)`

Takeaways So Far

Integer overflows help trigger unexpected behaviors, such as memory corruption.

Integer Overflows Could be Disastrous

Some integer overflows *per se* (without memory corruption) can be exploitable under a special environment, such as ***smart contracts***.

Integer Overflows in Smart Contracts

Ethereum Smart Contracts

Ethereum smart contracts are a program written in Solidity or Vyper. A smart contract defines a business logic or a set of promises and protocols between parties.

Integer Types in Solidity

- `int` (= `int256`): 256-bit signed integer.
- `uint` (= `uint256`): 256-bit unsigned integer.
- `int8`, `int16`, `int32`, ...
- `uint8`, `uint16`, `uint32`, ...

Transactions

Any Solidity function that is marked as `public` can be invoked by any other users. To invoke a function, one needs to make a ***transaction***.

Simple Solidity Example[3]

```
contract Counter {
    uint public count;

    // Function to get the current count
    function get() public view returns (uint) {
        return count;
    }

    // Function to increment count by 1
    function inc() public {
        count += 1;
    }

    // Function to decrement count by 1
    function dec() public {
        // This function will fail if count = 0
        count -= 1;
    }
}
```

[3] https://solidity-by-example.org/first-app/

Example: Banking dApp

```
function transfer(address _to, uint256 _amount) {
  require (balanceOf[msg.sender] >= _amount);
  balanceOf[msg.sender] -= _amount;
  balanceOf[_to] += _amount;
}
```

Example: Coin

```
function buyTokensPresale() public payable onlyInState(State.PresaleRunning) {
  require(msg.value >= (1 ether / 1 wei));
  uint newTokens = msg.value * PRESALE_PRICE;

  require(presaleSoldTokens + newTokens <= PRESALE_TOKEN_SUPPLY_LIMIT);

  balances[msg.sender] += newTokens; // !
  supply += newTokens;
  presaleSoldTokens += newTokens;
  totalSoldTokens += newTokens;

  LogBuy(msg.sender, newTokens);
}
```

One can generate coins exceeding the `PRESALE_TOKEN_SUPPLY_LIMIT`.

Integer Errors

Casting Error: Loss of Precision

Similarly to integer overflows, casting to a smaller type can be problematic.

CWE-197

```
int i;
short s;
i = (int)(~((int)0) ^ (1 << (sizeof(int) * 8 - 1)));
s = i;
printf("Int MAXINT: %d\nShort MAXINT: %d\n", i, s);
```

Implicit Casting Error: `memset`

```
void *memset(void *s, int c, size_t n)
```

The second parameter of `memset` indicates what byte to store. If the value is beyond the range of a `signed char`, then its higher order bit will be truncated.

Implicit Casting Error: `memset`

```
void *memset(void *s, int c, size_t n)
```

The second parameter of `memset` indicates what byte to store. If the value is beyond the range of a `signed char`, then its higher order bit will be truncated.

For example, `memset(array, 4096, n);` will write `n` bytes of zeros to `array`.

Question?

Exercise

Implement a C program to demonstrate "the year 2038 problem". Discuss how one can solve this problem.