Lecture 07 – Integer overflow

Stephen Checkoway

Oberlin College

Unsafe functions in libc

- strcpy
- strcat
- gets
- scanf family (fscanf, sscanf, etc.) (rare)
- printf family (more about these later)
- memcpy (need to control two of the three parameters)
- memmove (same as memcpy)

Replacements

- Not actually safe; doesn’t do what you think
  - strncpy
  - strncat
- Available on Windows and C11 Annex K (the optional part of C11)
  - strcpy_s
  - strcat_s
- BSD-derived, moderately widely available, including Linux kernel but not glibc
  - strlcpy
  - strlcat

Buffer overflow vulnerability-finding strategy

1. Look for the use of unsafe functions
2. Trace attacker-controlled input to these functions

Real-world examples from my own research

- Voting machine: Sequoia AVC Advantage
  - About a dozen uses of strcpy, most checked the length first
  - One did not. It appeared in infrequently used code
  - Configuration file with fixed-width fields containing NUL-terminated strings, one of which was strcpy'd to the stack
- Remote compromise of cars
  - Lots of strcpy of attacker-controlled Bluetooth data, first one examined was vulnerable
  - memcpy of attacker-controlled data from cellular modem

Reminder: Think like an attacker

- I skimmed some source code for a client/server protocol
- The server code was full of trivial buffer overflows resulting from the attacker not following the protocol
- I told the developer about the issue, but he wasn’t concerned because the client software he wrote wouldn’t send too much data
- Most people don’t think like attackers.

Integer overflows

Integer overflow occurs when the result of the arithmetic operation doesn’t fit in the number of bits available for the result

3 different flavors of integer overflow

Usually combined with a second vulnerability (e.g., buffer overflow)

- E.g., Use an integer overflow to bypass a conditional like
  if (size <= max_size)
  that’s intended to prevent buffer overflow and then overflow the bufer

Three flavors of integer overflows

1. Truncation: Assigning larger types to smaller types

```
   int i = 0x12345678;
   short s = i;
   char c = i;
```

Truncation example (on 64-bit architecture)

```
struct s {
      unsigned int len;
      char buf[];
};

void foo(struct s *p) {
      char buffer[100];
      if (p->len < sizeof buffer)
            strcpy(buffer, p->buf);
      // Use buffer
}
```

```
int main(int argc, char *argv[]) {
      size_t len = strlen(argv[0]);
      struct s *p = malloc(len + 5);
      p->len = len;
      strcpy(p->buf, argv[0]);

      foo(p);
      return 0;
}
```

Three flavors of integer overflows

2. Arithmetic overflow
    - This occurs when performing arithmetic operations produces a value which is too large to fit in a variable
    - Ex.
      ```
      int product = a * b;
      int sum = a + b;
      int difference = a - b;
      ```
    - These are frequently combined with the third type

Three flavors of integer overflow

3. Signedness bugs
   - Compare two signed integers, assuming nonnegativity
     ```
     if (x < 100)
       do_something();
     ```
   - Compare a signed and unsigned integer
     ```
     if (size < sizeof buffer)
       do_something();
     ```
   - Treating a signed negative number as unsigned
     ```
     void *p = malloc(size); // size < 0
     ```

Exploiting integer overflow

- Attacker controls the value of an integer n which gets used in multiple ways
  - Comparisons as signed/unsigned, 4 bytes/2 bytes, etc.
  - Arithmetic with positive n produces negative result
  - Arithmetic with negative n produces positive result
- Two’s complement integers don’t model mathematical integers well
  - Mathematical integers: If x > 0 and y > 0, then x*y > 0
  - Two’s complement integers: 15000000*500 = -1089934592
  - Programmers are used to thinking about mathematical integers

OpenSSH integer overflow

```
nresp = packet_get_int();
if (nresp > 0) {
     response = xmalloc(nresp*sizeof(char*));
     for (i = 0; i < nresp; i++)
          response[i] = packet_get_string(NULL);
}
```

- nresp is attacker-controlled and set to 0x40000000
- sizeof(char *) is 4 (on ILP32 machines)
- nresp*sizeof(char*) is 0 and xmalloc succeeds

Boeing 787 integer overflow

“We have been advised by Boeing of an issue identified during laboratory testing. **The software counter internal to the generator control units (GCUs) will overflow after 248 days of continuous power**, causing that GCU to go into failsafe mode. If the four main GCUs (associated with the engine mounted generators) were powered up at the same time, after 248 days of continuous power, all four GCUs will go into failsafe mode at the same time, resulting in a loss of all AC electrical power regardless of flight phase.”

https://s3.amazonaws.com/public-inspection.federalregister.gov/2015-10066.pdf

Defending against integer overflow

- Use appropriate types:
    - Need a size or a count? Use size_t
    - Need a specific bit-width? Use uint8_t, uint16_t, uint32_t, uint64_t, etc.
    - Need an integer to hold a pointer? Use intptr_t

Integer overflow checking in C is difficult

```c
#include <stdlib.h>

int safe_add(int a, int b) {
    if (a > 0 && b > 0) {
        if (a + b <= 0)
            abort();
    } else if (a < 0 && b < 0) {
        if (a + b >= 0)
            abort();
    }
    return a + b;
}
```

```asm
safe_add:
        lea eax, [rdi+rsi]
        ret
```

Undefined behavior

- C (and C++) have a wide variety of undefined behavior
- Signed (but not unsigned) integers have undefined behavior on overflow
- The compiler gets to assume undefined behavior doesn't happen!
- Compiler removes dead code

```c
#include <stdlib.h>

int safe_add(int a, int b) {
    if (a > 0 && b > 0) {
        if (a + b <= 0)
            abort();
    } else if (a < 0 && b < 0) {
        if (a + b >= 0)
            abort();
    }
    return a + b;
}
```

Correct implementation (I hope)

```c
#include <limits.h>
#include <stdlib.h>

int safe_add(int a, int b) {
    if (a > 0 && b > INT_MAX - a)
        abort();
    if (a < 0 && b < INT_MIN - a)
        abort();
    return a + b;
}
```

Compiler flags

- -fwrapv — Treat signed integers as having two’s complement behavior
- -ftrapv — Trap on overflow, broken on older compilers and constants
- -fsanitize=undefined — Undefined behavior sanitizer, not on old compilers