Vulnerability Patterns & Code Auditing

CS390R - UMass Amherst

Course Information

- Project 1 assigned

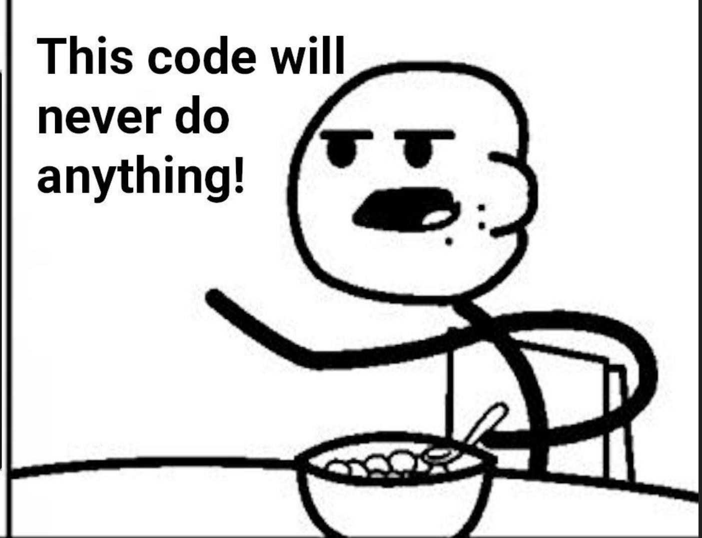
Today's Content

- Undefined Behavior
- Buffer Overflows
- Format String Vulnerabilities
- Types
- Binary Encoding
- Type Conversions
- Off-by-one errors
- Other vulnerabilities
- Auditing tips
- Cost of fixing vulnerabilities
- Let's find a 0 day!
- CVE-2022-0185

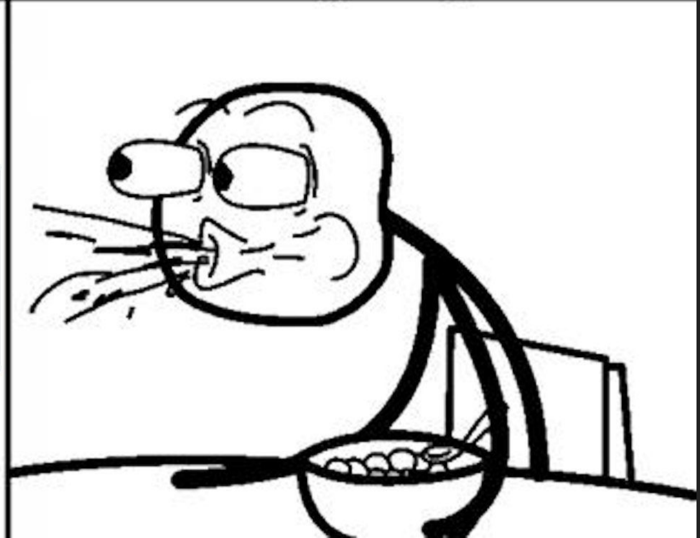
Undefined Behavior

- Any behavior that is not specifically handled by the compiler, and can therefore result in unspecified results.
- This could lead to crashes, exploitable vulnerabilities, or nothing at all

```cpp
#include <iostream>

int main() {
    while (1)
        ;
}

void unreachable() {
    std::cout << "Hello world!" << std::endl;
}
```




```
$ clang++ loop.cpp -O1 -Wall -o loop
$ ./loop
Hello world!
```



Buffer Overflow

- Overflow a buffer allocated on stack/heap/etc
- With this attackers can overwrite other data stored in the program
- Very dangerous vulnerability!

```c
/*
    1: `gets` causes a buffer overflow since it isn't bounds checked

    2: `scanf` causes a buffer overflow since it isn't bounds checked
*/
int main() {
    // 1
    char buf1[0x20];
    gets(buf1);

    // 2
    char buf2[0x20];
    scanf("%s", buf2);
}
```

Format String Vulnerabilities

- user controlled argument is passed to a printf-like function without format specifiers, so the argument itself can have format string specifiers

```
→ a bat format.c
File: format.c

#include <stdio.h>

int main() {
    char buf[32];

    fgets(buf, sizeof(buf), stdin);

    printf(buf);
}

→ a ./format
%p %d %s %x
0x207325206425207 0  a8b04e80
```

Types

- Character Types:
  - char, signed char, unsigned char (usually default to signed)
  - guaranteed to take up 1 byte of storage, but may not always be 8 bits
  - sizeof(char) is always one
- Integer Types
  - 4 signed integer types: short int, int, long int, long long int
  - Each of these has a corresponding unsigned type that takes up the same amount of storage
  - Signed integer types can represent both positive and negative values
  - unsigned integer types can only represent positive values
- Floating types
  - 3 real floating types float, double, long double
- Bit fields
  - Specified number of bits in an object
  - signed or unsigned depending on the declaration
  - example: ‘unsigned int id:1;’ 1 bit unsigned value

Binary Encoding

- Unsigned Integers
  - pure binary form, base-two numbering system
  - 00011011 = $2^4 + 2^3 + 2^1 + 2^0 = 27$
  - 11111111 = $2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 255$
- Signed
  - Sign bit - sign stored in the sign bit
  - Ones Complement - sign stored in sign bit, for negative numbers all bits are inverted
  - Twos Complement - Most used implementation
    - Sign bit is 1 if number is negative and 0 if its positive
    - Positive values can be read directly
    - For negative values, negate entire number and add 1, removes ambiguity of having a positive and negative zero

What to generally expect

- No padding bits in integer types
- Twos complement for everything
- Bytes are 8 bits long
- Little endian or Big endian
- Char type is 1 byte signed
- Short type is 2 bytes
- Int type is 4 bytes
- Long type is 4 bytes
- Long long is 8 bytes
- Pointers depend on the system, 8 bytes on 64-bit, 4 bytes on 32-bit

Integer Max/Min values

	8-bit	16-bit	32-bit	64-bit
Min (signed)	-128	-32768	-2147483648	-9223372036854775808
Max (signed)	127	32767	2147483647	9223372036854775807
Min (unsigned)	0	0	0	0
Min (unsigned)	255	65535	4294967295	18446744073709551615

Integer Over/Underflows

- Ariane 5 rocket self-destructed 37s after launch, resulting in ~$370million in damages
- Caused by data conversion from a 64-bit float to a 16-bit signed integer

Integer Over/Underflows

- Not usually exploitable on their own, but they frequently lead to unexpected program states that could be exploitable
- When looking for integer issues, pay special attention to any place where a user controlled value is added/subtracted/multiplied with other values
- The same vulnerability applies to multiplication via bit shifting

```c
unsigned int a;
a = 0;              // a = 0
a = a - 1;          // a = 4294967295

int b;
b = 2147483648;     // b = 2147483648
b = b + 1;          // b = -2147483648
```

```c
/*
    Real-world vulnerability in OpenSSH 3.1
    Output of packet_get_int() is user-controlled
    and used to determine how many responses to expect.

    Used to allocate the response array and fill it with data.
    If nresp is large enough, 'nresp * sizeof(char*)' can cause
    an integer overflow, resulting in a tiny number. This would
    make malloc return a very small memory buffer and lead to
    a buffer overflow.

    This vulnerability ended up being a critical remote code
    execution vulnerability.
*/
void input_userauth_response() {
    unsigned int nresp;
    ...
    nresp = packet_get_int();
    if (nresp > 0) {
        response = xmalloc(nresp * sizeof(char*));
        for (int i = 0; i < nresp; i++) {
            response[i] = packet_get_string(NULL);
        }
    }
    packet_check_eom();
}
```

Type Conversions /1

- C lets you do pretty much anything via casting
  - eg. ‘unsigned char * long int + char*’ <- valid way to setup a pointer with proper casting
- Explicit Type Conversions
  - Programmer explicitly requests type conversion by casting
- Implicit Type Conversions
  - Hidden transformations performed by compiler
  - Happens when eg. 2 numbers of different types are compared
- The rules for type conversions are very subtle and can lead to very hard to spot bugs
- Value preserving conversion
  - The new type can represent all possible values of the old type
  - eg. char -> Int
- Value changing conversion
  - Old type contains values that can’t be presented by the new type
  - int -> unsigned int

Type Conversions /2

- Widening (eg. short to int)
  - Copy bit pattern from old type to new type and zero/sign extend depending on type
  - Zero Extension: propagate 0 to all high bits (used for unsigned values)
  - Sign Extension: propagate the sign bit to all high bits (used for signed values)

```
Type:          unsigned char    ->    int
Value:         5                ->     5
Repr:          "\x05"           ->    "\x00\x00\x00\x05"
("zero extended, result is as expected")

Type:          signed char      ->    int
Value:         -5               ->     -5
Repr:          "\xFB"           ->    "\xFF\xFF\xFF\xFB"
("sign extended, result could cause surprises if not careful")

Type:          char             ->    unsigned int
Value:         -5               ->    4294967291
Repr:          "\xFB"           ->    "\xFF\xFF\xFF\xFB"
("sign extended, then treated as an unsigned value resulting \
    in a very large value")
```

Type Conversions /3

- Narrowing (eg. int to short)
  - Value is truncated, bits that don’t fit in narrower new type are dropped
  - Information is always lost

```
Type:          int                  ->    unsigned short
Value:         -1000000             ->    48576
Repr:      "\xFF\xF0\xBD\xC0"       ->    "\xBD\xC0"

Type:          int                  ->    signed char
Value:         -1                   ->     -1
Repr:      "\xFF\xFF\xFF\xFF"       ->    "\xFF"
```

Type Conversions /4

- Integer Promotions
  - If an integer type is narrower than an int, it is promoted to an integer for certain operations
  - example types: char, short, unsigned char/short
  - example ops: +, -, ~, <<, >>, switch statements, etc

```
/*
   You would expect an overflow to occur since the
   max value a char can hold is 255, however, due
   to integer promotion, both jim and bob are
   promoted to integers prior to the addition and
   the check passes.
*/
unsigned char jim = 255;
unsigned char bob = 255;
if ((jim + bob) > 300) {
    function();
}
```

```
/* function is called because a is
   converted to an integer due to
   integer promotion and thus does
   not underflow */
unsigned short a = 1;
if ((a - 5) < 0) { function(); }

/* function is not called because
   the unsigned value underflows and
   results in a very large value */
unsigned short a = 1;
a = a - 5;
if (a < 0) { function(); }
```

Some Examples

Left Operand	Right Operand	Result	Common Type
int	float	Left op converted to float	float
double	char	Right op converted to double	double
unsigned int	int	Right op converted to unsigned int	unsigned int
unsigned short	int	Left op converted to int	int
unsigned char	unsigned short	Left op converted to int	int
unsigned int	long int	Left op converted to unsigned long int	unsigned long int
unsigned int	long long int	Left op converted to long long int	long long int

Let’s look at some examples

Pointer Arithmetic

- Pointers can be freely converted between types using casts
- Operations done relative to the size of the pointer target
- This can easily lead to vulnerabilities due to miscounting buffer sizes

```c
short *j;
j = (short *)0x1234;
j = j + 1 // j is now 0x1236
```

```c
/*
   b < buf + sizeof(buf) is meant to
   prevent b from advancing beyond buf[1023],
   but since its an int pointer, it actually
   prevents b from going beyond buf[4092] and
   causes a buffer overflow.
*/
int buf[1024];
int *b = buf;
while (b < buf + sizeof(buf)) {
    *b++ = get_number();
}
```

Off by One Errors

- Can be caused very easily my miscalculating length of an array or string
- What does it matter, it’s only 1 byte?
  - Overwrite least significant byte of a pointer stored in memory after the buffer
  - Stack: Overwrite least significant byte of the frame pointer thus moving stack
  - Heap: Overwrite heap metadata

```c
/*
    An attempt was made to prevent a buffer overflow in the
    for loop condition, but it was done incorrectly so 1
    byte can be written out of bounds. Since arrays start at
    0, this array index is only valid 0-31. Because the
    condition is '<=' instead of '<' it indexes one past that.
*/
void process_string(char *src) {
    char dest[32];

    for (int i = 0; src[i] && (i <= sizeof(dest)); i++) {
        dest[i] = src[i];
    }
}
```

```c
/*
    In this case, everything looks fine at first glance,
    the strlen() function however returns the number of
    characters in a C string without accounting for the
    NULL terminator. In this case if a string with length
    1024 is passed into the function, it would pass the
    check, and a nullbyte would overflow.
*/
int get_user(char *user) {
    char buf[1024];

    if (strlen(user) > sizeof(buf)) {
        die(“error: user string too long\n”);
    }
    strcpy(buf, user);
}
```

Other

- Heap vulnerabilities
  - use after free
  - double free
  - heap overflow
- Race Conditions
- Type confusions (very relevant in JIT engines such as browsers)
- General common programming mistakes
  - sizeof() on a pointer (just returns 4/8 instead of the size of the object)
  - modulo operator on negative value returns negative result
  - Various mistakes can easily occur during shift operations

Auditing Tips

- Write simple test programs for specific cases or study the assembly directly to make sure that the expected code is output
- Pay special attention to all operations involving reading/writing a buffer (fgets, strncpy, read, etc)
- Look at all comparisons and verify that no vulnerabilities are possible due to differing types
- Do the same for all operating such as addition/subtraction/shifts/… on various numbers
- Watch out for unsigned integer values that cause peer operands to be promoted to unsigned integers (sizeof(), strlen(), etc)
- Verify precedence in complicated expressions lacking parentheses
- Pay attention to code indentation and possible typos & missing brackets/symbols
- Verify that no uninitialized memory is used. Since data is generally not zero'd, these could leak information

5 Vulnerabilities
→ Find Them!

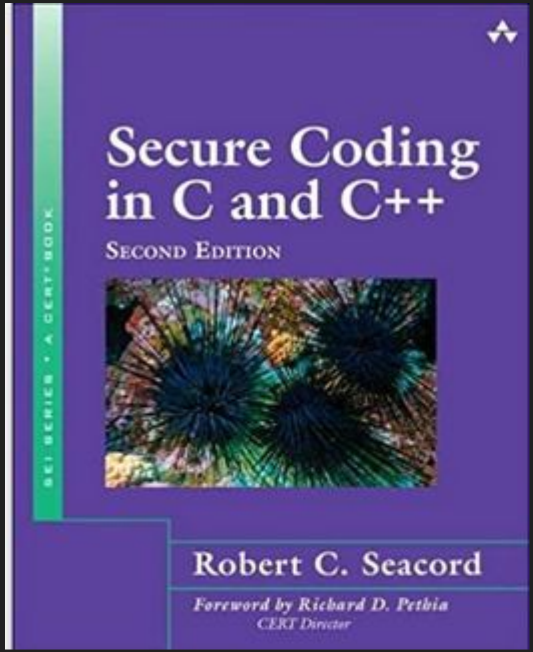
```c
#define MAX_BUF 64
void read_data() {
    char buf[MAX_BUF];
    int length;

    scanf("%d", &length);
    strncpy(buf, "Hello World", 12);

    if (length + 12 >= MAX_BUF) exit(-1);

    fgets(buf, length, stdin);
}
void main() {
    char buf[MAX_BUF];
    int choice;
    int *ptr;

    printf("What you you want to say?\n");
    gets(buf);

    printf(buf);

    read_data();

    while (1) {
        scanf("%d", &choice);
        switch(choice) {
            case 1:
                ptr = malloc(10*sizeof(int));
                break;
            case 2:
                free(ptr);
                break;
            case 3:
                exit(-1);
        }
    }
}
```

The cost of fixing vulnerabilities

- Vendors often say that memory corruption bugs aren’t exploitable / not a serious issue worth investing money in
- History has shown many examples where attackers exploited seemingly unexploitable bugs: “Where there’s a will, there’s a way”
- Fixing a bug always has a price for the company, so many want to ignore security
- Eg. embedded device, may require hardware modifications on every distributed device
- This is why being able to produce poc’s is an important skill

Good Resources/References

Secure Coding in C and C++



Art of Software Security Assessment