Comparisons

Comparisons

- Pay attention to comparisons protecting allocation, array indexing and copy operations
- Keep track of variables and their data types
- Watch for unsigned values being compared with their signed buddies as the signed operands will be promoted, potentially creating issues
- Watch for unsigned values being compared in a “negative” fashion
  - Eg: if (uint_potato < 0) or if (uint_potato <= 0)
  - The latter comparison will not cause the compiler to emit a warning, which may be missed by developers

Comparisons

- Bugs can occur when comparisons are done across integers of different types due to promotions regardless of whether it's a widening or narrowing operation
- What is the problem?

```c
#define MAX_SIZE 1024

int read_packet(int sockfd)
{
    short length;
    char buf[MAX_SIZE];

    length = network_get_short(sockfd); //user controlled

    if(length - sizeof(short) <= 0 || length > MAX_SIZE){
        error("bad length supplied\n");
        return -1;
    }
    if(read(sockfd, buf, length - sizeof(short)) < 0){
        error("read: %m\n");
        return -1;
    }
    return 0;
}
```

length is a signed short
It will be promoted to a signed int, then to an unsigned int because sizeof() returns size_t

Therefore the comparison will never be negative

If length is a –ve number
The first check will still pass, and the second check will also pass as length will be promoted to a signed int, and
(-ve > +ve) is not true

Comparisons

- Trimmed down version of previous program
- Length – sizeof(short) == 1 – 2 == -1
- But “here” is not printed

```c
// Online C compiler to run C program online
#include <stdio.h>

int main() {
    short length = 1;
    int s = sizeof(short);

    if (length - sizeof(short) <= 0)
        printf("here %d\n", length);
    return 0;
}
```

Comparisons

- Modification: sizeof() is assigned to int s, and comparison is done as (length – s)
- Result of sizeof is implicitly casted to int
- short is promoted to int, so the comparison returns -1, therefore “here” is printed

```c
// Online C compiler to run C program online
#include <stdio.h>

int main() {
    short length = 1;
    int s = sizeof(short);

    if (length - s <= 0)
        printf("here %d\n", length);
    return 0;
}
```

```
/tmp/Ol
here 1
```

Comparisons

- Will “here” be printed?

Yes

Even though length is now an unsigned short, it is lower rank than s, which is a signed int

Therefore length is promoted to an int, and the promotion stops there

```c
#include <stdio.h>

int main() {
    unsigned short length = 1;
    int s = sizeof(short);

    if (length - s <= 0)
        printf("here %d\n", length);
    return 0;
}
```

```c
// Online C compiler to run C program online
#include <stdio.h>

int main() {
    unsigned short length = 1;
    int s = sizeof(short);

    if (length - s <= 0)
        printf("here %d\n", length);
    return 0;
}
```

```
/tmp/Olo
here 1
```

Comparisons

- What is the problem?

max and length are short, therefore they are both promoted to signed int when compared

Note that the promotion stops at signed int because signed int can represent all values in both short and unsigned short

If length is –ve, it will be pass the (length > max) check

When it gets passed to read(int, void*, size_t), the –ve value gets converted into an unsigned int and becomes a large +ve

```
int read_data(int sockfd)
{
    char buf[1024];
    unsigned short max = sizeof(buf);
    short length;

    length = get_network_short(sockfd);

    if(length > max){
        error("bad length: %d\n", length);
        return -1;
    }
    if(read(sockfd, buf, length) < 0){
        error("read: %m");
        return -1;
    }
    return 0;
}
```

Comparisons

- What is the problem?

atoi returns int, but is assigned to unsigned int n

n < 0 will never be true as n is unsigned

The “n > 1024” check will catch any overflows and return -1 as expected

get_int returns signed, but is assigned to unsigned long n

unsigned long n will never be -1, so the check will always pass

```c
int get_int(char *data)
{
    unsigned int n = atoi(data);

    if(n < 0 || n > 1024)
        return -1;
    return n;
}
int main(int argc, char **argv)
{
    unsigned long n;
    char buf[1024];

    if(argc < 2)
       exit(0);

    n = get_int(argv[1]);
    if(n < 0){
        fprintf(stderr, "illegal length specified\n");
        exit(-1);
    }
    memset(buf, 'A', n);
    return 0;
}
```

Operators

sizeof

- Used regularly for buffer allocations, size comparisons, and size parameters to length-oriented functions
- One of the most common mistakes is accidentally using it on a pointer instead of its target

sizeof

- What is the problem?

Read up to 1023 characters from socket into userstring
If userstring has :, style contains the characters up to :

snprintf(char* buffer, size_t size_in_buffer, char* format)

If style is not null,
Print into buffer:
size_in_buffer = buffer should be at least as big as this
style=username

The intent is to get the buffer's size (1024), however, sizeof(buffer) will return 4 as buffer is a char pointer, which is 4 bytes on 32 bit systems

```c
char *read_username(int sockfd)
{
    char *buffer, *style, userstring[1024];
    int i;

    buffer = (char *)malloc(1024);

    if(!buffer){
        error("buffer allocation failed: %m");
        return NULL;
    }

    if(read(sockfd, userstring, sizeof(userstring)-1) <= 0){
        free(buffer);
        error("read failure: %m");
        return NULL;
    }

    userstring[sizeof(userstring)-1] = '\0';
    style = strchr(userstring, ':');
    if(style)
        *style++ = '\0';

    sprintf(buffer, "username=%.32s", userstring);
    if(style)
        snprintf(buffer, sizeof(buffer)-strlen(buffer)-1,
                 ", style=%s\n", style);
    return buffer;
}
```

sizeof

- buffer is a char* pointer, hence 8 is printed (64 bit program)
- test is a char array, hence 1024 is printed

```c
// Online C compiler to run C program online
#include <stdio.h>

int main() {
    char *buffer = malloc(1024);
    char test[1024];

    printf("%d %d\n", sizeof(buffer), sizeof(test));

    return 0;
}
```

```
/tmp/Olo
8 1024
```

sizeof

- Intent: While keeps looping until there's no more data or when pointer b has exceeded buf
- What is the problem here?

```
int buf[1024];
int *b;
while (havedata() && b < buf + sizeof(buf))
{
    *b++ = parseint(getdata());
}
```

However, sizeof(buf) is problematic

sizeof(buf) returns 1024 * sizeof(int) = 4096

But pointer arithmetic already takes the size of the data type into account

Eg
int *buf;
buf++; ← increments buf by sizeof(int)

Therefore the check (b < buf + sizeof(buf)) will cause b to be incremented past buf

>>, /, % for signed operands

- For >>, check if left operand is signed
- For % and /, check if the operands are signed
- Bugs can occur if users can specify –ve values

>> for signed operands

- Left shift: x<<y, bits of x shifted left by y positions → multiply by powers of 2
  - 1<<4 = 1 * 2^4, 2<<4 = 2 * 2^4
- Right shift: x>>y, bits of x shifted right by y positions → divide by powers of 2
  - 1>>4 = 1 / 2^4 (0! Shifted to nothingness), 10000>>4 = 10000 / 2^4 (625)
  - 64 >> 2 → 0100 000, the 1 is moved right twice and becomes 0001 0000
- If the left operand of >> is signed, unexpected results can happen
- The asm for >> is shr

```
signed char c = 0x80;
c >>= 4;

1000 0000 – value before right shift
1111 1000 – value after right shift
```

>> for signed operands

- This function prints the high word (msb 2 bytes) of an int
- What is the problem here?

btw, what would sizeof("65535") return?
"65535" is a char[] → therefore returns 6

If number = -1, what is number >> 16?

$-1 / 2^{16} = -1$
0xFFFFFFFF shifted to the right 2^{16} times is still 0xFFFFFFFF

What would be this value if it's interpreted as unsigned (%u)?

```c
int print_high_word(int number)
{
    char buf[sizeof("65535")];
    printf("%u\n", number >> 16);
    sprintf(buf, "%u", number >> 16);
    return 0;
}
int main()
{
    print_high_word(-1);
    return 0;
}
```

>> for signed operands

- -ve >> 2 or –ve >> 16, doesn't matter cos it will overflow regardless
- number contains 0xFFFFFFFF after being shifted
- When interpreted as an unsigned, the value is max unsigned int, when interpreted as signed, the value is as expected

```
1  // Online C compiler to run C program online
2  #include <stdio.h>
3
4  int main() {
5      int number = -1 >> 2;
6      printf("%u %d\n", number, number);
7      return 0;
8  }
```

```
/tmp/Ol0MP9HCB>
4294967295 -1
```

/ for signed operands

- What is the problem here?

bitlength is signed, a division by a -ve number will result in a -ve

Therefore if bitlength is -8, the result of the division will be -1 + 1 = 0
malloc(0) will allocate a small amount of memory

read(int, void*, size_t) will receive a –ve number, which will get converted to a large +ve

```
int read_data(int sockfd)
{
    int bitlength;
    char *buffer;

    bitlength = network_get_int(length); // user controlled

    buffer = (char *)malloc(bitlength / 8 + 1);
    if (buffer == NULL)
        die("no memory");
    if(read(sockfd, buffer, bitlength / 8) < 0){
        error("read error: %m");
        return -1;
    }
    return 0;
}
```

% for signed operands

- % operations can result in a negative result if % against a –ve dividend operand
- Eg -6 % 4 = -2, 6 % -4 = 2

```
struct session
{
    struct session *next;
    int session_id;
}
#define SESSION_SIZE = 10

struct session *new_session(int session_id)
{
    struct session *new1;

    new1 = malloc(sizeof(struct session));
    if (!new1) die();

    new1->session_id = session_id;
    new1->next = NULL;

    // add session to our sessions array, % to wrap around
    if (sessions[session_id%(SESSION_SIZE-1)])
    {
        sessions[session_id%(SESSION_SIZE-1)] = new1;
        return new1;
    }
}
```

If session_id is –ve, the sessions array will be indexed with an invalid index and can cause OOB write