Integer overflows

- Integer overflow types
- Signed vs unsigned integers
- Casting between different size integer types
- Arithmetic wraparound
- Pointer arithmetics
- Why is it bad?
- How to detect and prevent integer overflows
Integer overflow types

- Signed vs unsigned confusion
- Casting between different size integer types
- Arithmetic wraparound
- Pointer arithmetics
Signed vs unsigned confusion

- Same size integer can be either signed or unsigned
- Sign bit
- Two’s complement representation of negative numbers
- Example: signed short, unsigned int
- Special case: char (signed by default on some platforms, unsigned on most platforms)
- Do understand integer conversion rules!
  - Upcast (cast to larger type)
  - unsigned to larger signed if fits
Signed vs unsigned

#define MAX_LEN 256

int datalen;
char buf[MAX_LEN];

datalen = get_int_from_socket();
if (datalen > MAX_LEN) {
    printf("invalid input - data field too large");
    exit(-1);
}

if (read(sock, buf, datalen) < 0) {
    perror("read");
    exit(errno);
}
Signedness again

```c
char* processNext(char* strm) {
    char buf[512];

    short len = *(short*) strm;
    strm += sizeof(len);
    if (len <= 512) {
        memcpy(buf, strm, len);
        process(buf);
        return strm + len;
    } else {
        return -1;
    }
}
```
Casting between different size integer types

- Example: C/C++ integer types
  - char (8+ bits)
  - short (16+ bits)
  - int (16+ bits)
  - long (32+ bits)
  - pointers

- Exact sizes depend on platform and compiler
  - char is almost universally 8 and short 16 bits
  - Different programming models, ILP32 and LP64 are most widespread
## Integer sizes in programming models

<table>
<thead>
<tr>
<th>Type</th>
<th>LP32</th>
<th>ILP32</th>
<th>LLP64</th>
<th>LP64</th>
<th>ILP64</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>int32</td>
<td></td>
<td></td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>16</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>long</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>long long</td>
<td></td>
<td></td>
<td>64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pointer</td>
<td>32</td>
<td>32</td>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
</tbody>
</table>
Size changes

- Manual cast to different size
- Automatic cast to different size
- Sign extension on upcast
- Truncation on downcast
Size confusion

```c
short int bytesRec = 0;
char buf[SOMEBIGNUM];

while(bytesRec < MAXGET) {
    bytesRec += getFromInput(buf + bytesRec);
}
```
Arithmetic wraparound

- What happens when computation result does not fit into result variable?
  - Saturation (maximum value stays)
  - Wraparound (defined to wrap and discard carry bits)
  - Undefined (low-level implementation usually results in wraparound here too)

- Underflow — below 0

- Does the programmer expect the wraparound?
Wraparound: addition

```c
unsigned int len;
char *str;

len = get_user_length();

if (!(str = (char *)malloc(len + 1))) {
    perror("malloc");
    exit(errno);
}

memcpy(str, user_data, len);
```
Wraparound: subtraction

```c
#define HEADER_SIZE 32
#define MAX_PACKET 256

int len;
char buf[MAX_PACKET], data[MAX_PACKET];

if ((len = read(sock, buf, sizeof(buf) - 1)) < 0) {
    perror("read");
    exit(errno);
}

memcpy(data, buf + HEADER_SIZE, len - HEADER_SIZE);
```
Wraparound: multiplication

```c
int num, i;
object_t *objs;

num = get_user_num();

if(!(objs = (object_t *)malloc(num * sizeof(object_t)))){
    perror("malloc");
    exit(errno);
}

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

- Closely related to integer overflows but happen directly in pointer expressions
- Sometimes when performing pointer arithmetic, miscalculations can be made as to how much space is left in a buffer. This can allow for attackers to sometimes write outside the bounds of the destination buffer.
Why is it bad

• Can result in
  – Crashes
  – Infinite loops
  – Simple data corruption
  – Buffer overflows

• Usually medium to hard to exploit
Trampoline to buffer overflows

- Can lead to buffer overflow when overflowed result is used for
  - Memory allocation
  - String operations
  - Index into a buffer
Where can it happen

- Most languages can have integer overflows
- Range checking and overflow detection can be done at run time but causes performance penalties
- Affects memory management on low level languages only — like C, C++, FORTRAN, assembly
- Safer languages can have two kinds of safety:
  - Memory safety — ultimate goal, does not touch any memory it is not intended to touch
  - Type safety — prevents random variables from being used as memory addresses
- Managed code — what can we do with it?
- Safe dialects of unsafe languages (CCured, Cyclone)
What about Java?

- Memory safety limits the damage to the variables themselves
- JNI (Java Native Interface) bypasses managed code validation and can break anything
How to detect and prevent integer overflows

- Use unsigned types
- Always specify the signedness you expect
- Restrict numeric user input
- Sanity-check values used to allocate and access memory
- Respect compiler warnings
  - gcc: -Wconversion, -Wsign-compare in -Wall/-Wextra
- Runtime trapping for debug builds
- Overflow-safe number classes on some platforms