Integer safety crash course

Read this chapter, it's from a great book you should buy:
http://taossa.com/index.php/the-vault/chapter-6-c-language-issues/

Signedness

- **1111111111111111111111111111111***1*
- *Most significant bit*
- *Least significant bit*

- int a = -1;

Two's Complement

- **1111111111111111111111111111111***1*
- *Most significant bit*
- *Least significant bit*
- ~negate → **0000000000000000000000000000000***0*
- add 1 → **0000000000000000000000000000001***1*

- int a = -1, b;
- b = -a;
  printf("%d\n",a);
- b = ~a + 1;
  printf("%d\n",b);

Declaring variables

- int c;
- char c;
- long c;
- short c;
- #define X 4

- Signed or unsigned?
- Signed or unsigned?
- Signed or unsigned?
- Signed or unsigned?
- Signed or unsigned?

http://www.cdecl.org/

Declaring variables

I. int c;

II.char c;

III.long c;

IV.short c;

V. #define X 4

I. **Signed** or unsigned?

II. Often signed- depends

III. Signed

IV. Signed

V. Signed

http://www.cdecl.org/

Signedness in a security context

- Bounds checking
- Bounds checking
- Bounds checking

- void dumpcash(unsigned int amt) { … }

  ```
  if (amt < acct_bal)
    dumpcash(amt);
  ```

- #define SIZE 64

  ```
  if (index < SIZE)
     buffer[index] = data;
  ```

Signedness Concept → implicit funk

- int a = read_int(sockfd);
- if (a < sizeof(buf)) {

  }

- If a = -1?

Signedness Concept 1 → implicit conversion (coercion)

- signed vs unsigned
- Converted to unsigned arithmetic
- **Unsigned takes precedence**

- int a;
- a < sizeof(buf)
- == false
- 0xffffffff < sizeof(buf)

Signedness Concept 2 → sign extension

- a = 1000000*0*
- b = 1**11111111111111111111111**1000000*0*

- char a = 0x80;
- unsigned int b = a;
- printf("%u\n",b);

- **This slide might remind you of a lab exercise advisory you might be stuck on**

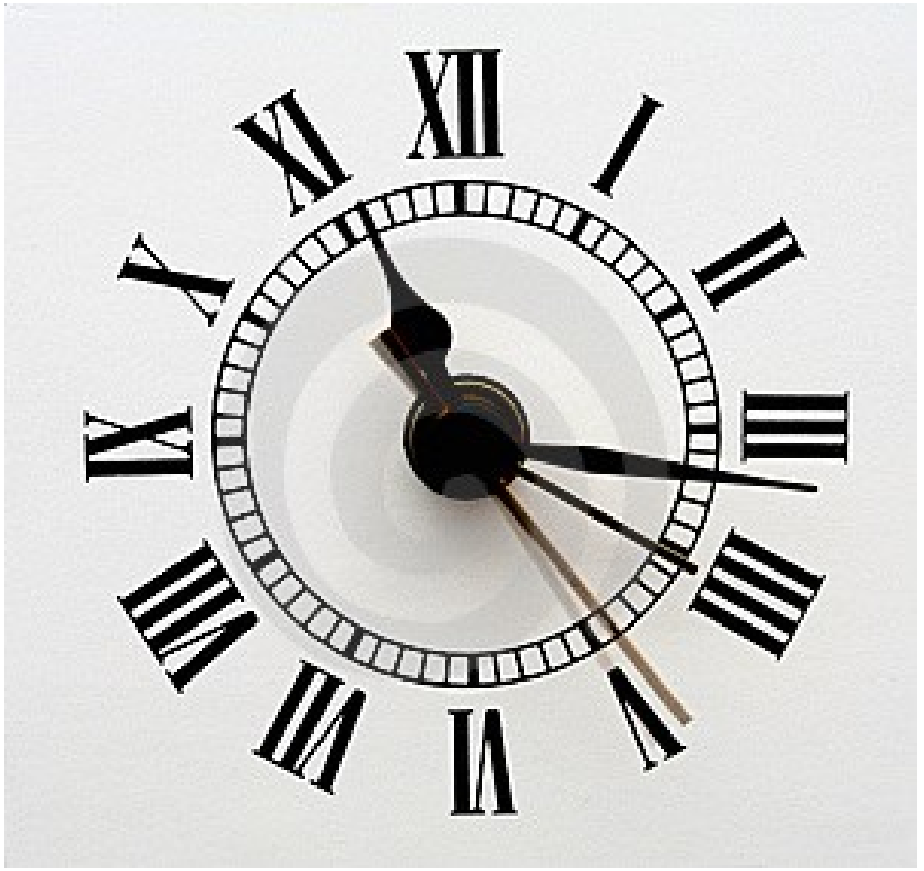
Signedness Concept 3 → integer promotion

- Well covered in taossas
- Learned this from fellow RPI student Roy Wellington in 2008; Roy read the C spec
- **Default is int conversion**

- unsigned short a, b;
- a = strtoul(argv[1],0,0);
- b = strtoul(argv[2],0,0);
- if(a*b < 0){
  printf("Roy is the man\n");
  }

Considerations for the future

- Signedness bugs are here to stay
- 64-bit won't make them go away



Esoteric signedness fun

- 10000000000000000000000000000000
- *Most significant bit*
- *Least significant bit*

- int a = -2147483648;

  printf("%d %d\n", -a, ~a+1);

Guess the next topic

- What are the boundary conditions for len?

- weirdcopy(char *buf, char *data, unsigned int len){
  do {
    buf[len] = *data++;
  } while(--len);
  }

Thanks to A^2 for pointing out an error


000
FreeFoto.com

Underflow

- Aside: everything but 0 is true in C
-
- len = 0;
- len → len = -1
- len = -2
- …
- len = 1
- len = 0

- while(--len){
  do_stuff
  }

A common pattern for underflows

- This bad code is <u>everywhere</u>

- buf[strlen(buf)-1] = 0;

- sz = 0
  buf = malloc(sz);
  for(i = 0; i < sz-1; i++)
  buf[i] = read_char(fd);

- strlen(buf) can be 0

- malloc 0 still allocates memory

- ***These are exploitable***

Overflow on addition

- int x = 0xdeadbeef;
- x + 0xbadc0ded = ?

http://www.youtube.com/watch?v=y4GDcvweo14

Addition under the covers

- int x = 0xdeadbeef;
- x + 0xbadc0ded = ?
- 1101111010101101101111110111011111
  1011101011011100000011011111011011
- ----------------------------------
  ±100110011000100111001100110111100

Multiplication too!

- uint sz = read_int();
- Buf = malloc(sizeof(int) * sz);

- Numeric overflow by multiplication,
- Can be used to trigger buffer underflows and overflows depending on signedness.

Overflow thoughts

- Numeric overflows have become more difficult to exploit because of 64-bit
- :-(
- X86 + other archs can detect overflows, compilers dont

Don't forget about truncation

- Look for lots of these in 64 bit code; anywhere you have variable integer sizes

- uint64_t sz = read_bytes(sizeof(uint64_t));
- uint32_t limit = sz;
- if(limit < MAX) { … }

2 – Format strings

- printf(buf)
- sprintf,snprintf,syslog,asprintf,v*rintf, custom debug functions, ...
- Newer gcc versions warn about unspecified format strings

Deep thoughts on fmt strings

- Easy to find
- Direct parameter access not available on windows
- Latest VC++ disables %n entirely

3 – off by ones

- char buf[256]; int i;
- for(i = sizeof(buf); i >= 0; i—)
- for(i = 0; I <= sizeof(buf); i++)
- for(i = 0; sizeof(buf) > i; i++)
- char *buf = malloc(256);
  buf[256] = 0;

NUL termination

- “The strncpy() function is similar, except that at most n bytes are copied. Warning: If there is no null byte among the first of src, the string placed in dest will not be null terminated.”

  “The functions snprintf() and vsnprintf() write at most size bytes (including the trailing null byte ('\0')) to str.”

float != double

- Float
  8 bit exponent
  23 bit mantissa
  About 7 decimal digits precision
- Double
  11 bit exponent
  53 bit mantissa
  About 16 decimal digits precision

Floating point woes

```
#include <stdio.h>
#include <unistd.h>

double GetTime();

int main(int argc, char* argv[])
{
float a = GetTime();
usleep(5000 * 1000);
float b = GetTime();
printf("a =  %f\nb =  %f\ndt = %f\n", a, b, b-a);
return 0;
}
```

Output

- a = 1265603328.000000
- b = 1265603328.000000
- dt = 0.000000

Spot the bug

```
int verify(char* in, char* pass)
{
if(strcmp(in,pass) == 0)
  return STATUS_PASSWORD_OK;
return STATUS_PASSWORD_ERROR;
}
```

buf = new T[x]

- buf = malloc(x*sizeof(T));
  for(i=0; i<x; i++)
    call constructor on buf[i]
- What happens when x*sizeof(T) > UINT_MAX?

delete x != delete[] x

- delete x
  call destructor on *x
  free(x)
- delete[] x
  ask memory manager for size of block
  for(i=0; i<size / sizeof(T); i++)
    call destructor on x[i]
  free(x)

Double free / use after free

- Member functions are just functions with a hidden parameter “this”
- Calling on an invalid pointer will sometimes succeed
- Data corruption, especially static / global variables, typically results

Reference counting

- Double free = drop reference counter twice
- We have one object and free it twice
- Refcount is now... ?

Exception handling

- Exception records are usually on the stack
- Corrupt these and throw an exception

Don't trust your compiler

```
#include <stdio.h>
int main(int argc, char* argv[])
{
   unsigned int* a;
   unsigned int i;
   for(i=0; i<1; i++)
      printf("*a = %d\n",*a);
   return 0;
}
```

Here's a harder one

```
#include <stdio.h>
int main(int argc, char* argv[])
{
    unsigned int a;
    unsigned int b = a;
    unsigned int i;
    for(i=0; i<1; i++)
    {
        if(i > 0xdead)
        {
            printf("Initializing a\n");
            a = 999;
        }

        printf("a = %3d, b = %3d. ",a, b);
        if(a == b)
            printf("Equal\n");
        else
            printf("Not equal\n");
    }
    return 0;
}
```

Citations, Good reading

- http://www.ruxcon.org.au/files/2006/unusual_bugs.
- http://taossa.com/index.php/the-vault/chapter-6-c-la
- http://cr.yp.to/2004-494.html