COMP9447

cs

Why read source code?

- Because it is the **best** way to find the **best** bugs
- You need to do a lot of this before you can start doing the same thing in a reverse context
- So many people are resistant to actually reading the source; love to spend time playing with the program by manipulating input, no desire to understand.
- Mark Dowd, “I'll never stop to be amazed by the amount of efforts people put in not understand things”
- When writing exploits, having a precise understanding of what is going on is super important.
- CORELan coder: precise heap spraying, total waste of time.

https://www.corelan.be/index.php/2013/02/19/deps-precise-heap-spray-on-firefox-and-ie10/

^^ WILLFUL IGNORANCE ^^

C language problems

There are several classes of problems in C that we're particularly interested in. We'll be talking about:

- Bad API usage
- Integer wrap-arounds
- Type conversions
- Operators (sizeof/bitshifts/etc)
- Pointer arithmetic
- Other weird stuff

Bad API usage

- Using APIs like those in stdlib in the wrong way.
  - Not checking sizes.
  - Classic format string:

```
int
main(int argc, char *argv[])
{
    if (argc == 2) printf(argv[1]);
    printf (“\n”);
}
```

Format string example

```
$ ./fmt lol
lol
$ ./fmt %x..%x..%x..%x
a4e7b610..804842b..a4e4dff4..8048420
$ ./fmt %s
U WV1 S .
$ ./fmt %s..%s..%s..%s
U WV1 S . .. N ..|- ..U WVS O
$ ./fmt %s..%s..%s..%s..%s..%s..%s
U WV1 S . .. N ..|- ..U WVS O..(null).... $ ri
$ ./fmt %s..%s..%s..%s..%s..%s..%s..%n
Segmentation fault
$
```

Realistically, any function with a variable number of args that calculates the number of arguments from user controlled data is vulnerable!

Really old stuff – gets()

- Reads a string until it encounters newline or EOF.

```
int
main(void)
{
	char	s[4];
	gets(s);
}
```

Compiler will warn us:

```
gets.c:(.text+0x11): warning: the `gets' function is dangerous and should not be used.
```

```
$ ./perl -e "print 'A' x 100" | ./gets
Killed
```

strcpy()

There used to be a huge amount of exploitable buffer overflows from strcpy.

```
int
main(int argc, char *argv[])
{
    char destination[32];
    strcpy(destination, argv[1]);
}
```

strcpy() output

```
$ ./strcpy `perl -e "print 'A' x 32"`
$ ./strcpy `perl -e "print 'A' x 40"`
$ ./strcpy `perl -e "print 'A' x 48"`
Killed

$ dmesg | tail -n 3
PAX: terminating task: /home/dzhkh/share/uni/c/strcpy(strcpy):21488,
uid/euid: 1000/1000, PC: 41414141, SP: bd9d9aa0
PAX: bytes at PC: ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
PAX: bytes at SP-4: 41414141 00000000 bd9d9b44 bd9d9b50 b18753d0
bd9d9ae0 ffffffff b1893fc4 08048210 00000001 bd9d9b00 b1884c06
b1894a98 b18756a8 b1857ff4 00000000 00000000 bd9d9b18 b7f7d9ef
69b40ff6 00000000
```

strncpy() is safe?

From the strncpy manpage:

- char *strcpy(char *dest, const char *src);
- char *strncpy(char *dest, const char *src, size_t n);

DESCRIPTION

The strcpy() function copies the string pointed to by src, including the terminating null byte ('\0'), to the buffer pointed to by dest.

The strings may not overlap, and the destination string dest must be large enough to receive the copy.

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

strlcpy() is what you want to use, but even that's still not perfect, and is not available on all platforms!

strncpy() example

```
int
main(int argc, char *argv[])
{
    char a[16] = “ABCDEFGHIJKLMNO”;
    char b[16];
    if (argc == 2)
    {
        strncpy(b, argv[1], 16);
        printf(“%s\n”, b);
    }
}
```

What do you think this does? What’s the security impact?

strncpy() continued

```
$ ./strncpy1 abcdefghijklmno (this is 15 characters so there’s a NUL)
abcdefghijklmno
$ ./strncpy1 abcdefghijklmnop (this is 16 – no nul)
abcdefghijklmnopABCDEFGHIJKLMNO
```

- This works because the strings are next to each other in memory, and we don’t have a nul (yes, nul) byte.

Memory looks like this:

```
<-to-base-of-stack                    -to-top-of-stack->
SESPSEBPAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBB
<!--- strings go in this direction
```

SESP = SAVED ESP, SEBP, SAVED EBP.

Is this really dangerous?

This is more than just a memory leak. There’s all sorts of stuff that happens when things go wrong, and a tiny amount of control can be leveraged a long way.

Think about some situation where we get to write a nul byte one byte past the end of a string, and off-by-one sort of situation.

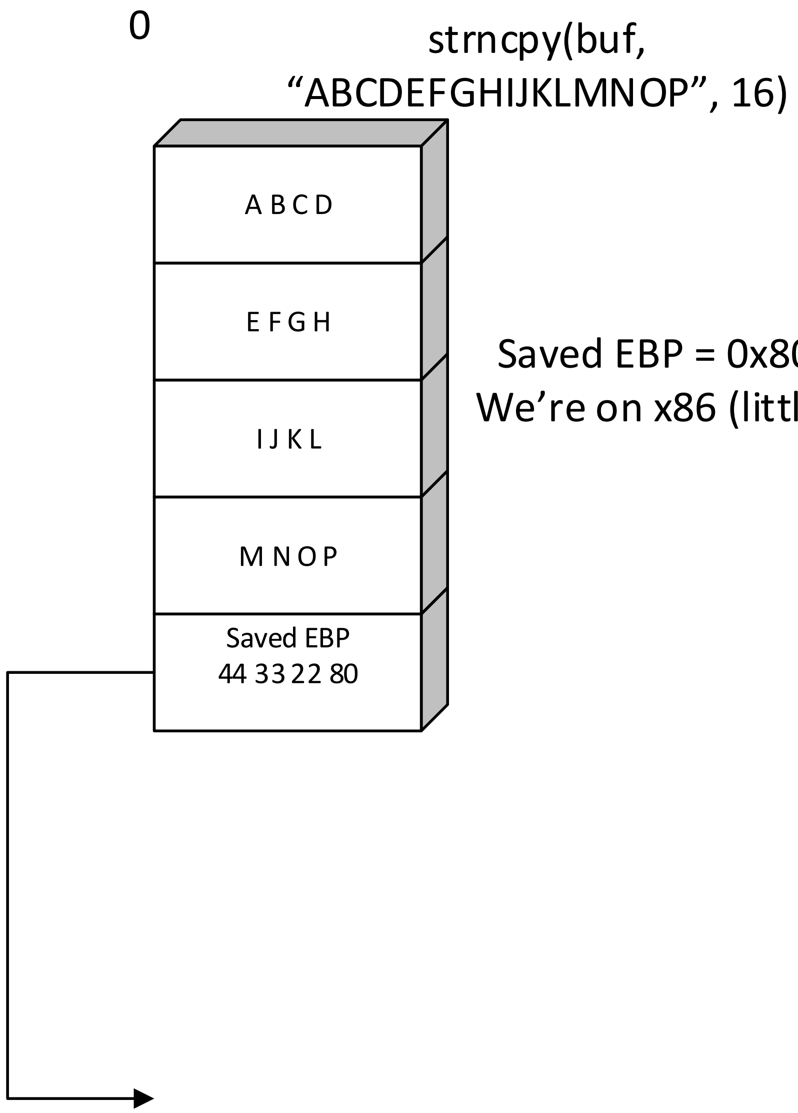
```
char[16] s = read_from_user(16);
//other stuff which makes this believable.
s[16] = ‘\0’
```

EBP partial overwrite

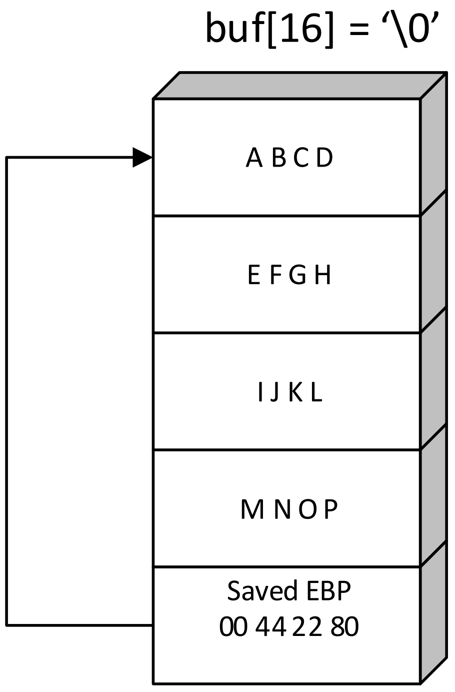
Under specific conditions, like if we're the first string in the function, and our saved ebp doesn't already end in 0x00, we can make it end in 0x00 with that nul byte! As the stack grows down (from high addresses to low addresses), ebp may now point into our data.

So... when the function finishes, it uses this saved value as the stack base of the next function. When the caller then calls another function, this is then used as the new stack pointer, as well as the base pointer for local variables. We now control data in the calling function!

Let’s look at a picture of it



Saved EBP = 0x80223344
We’re on x86 (little endian)



Heap problems

Heap overflows are a huge issue.

Although lots of these are standard overflows as per *cpy or in a loop, some are related to specifics of heap implementation.

Use after free():

```
{
    char *s = malloc(n);
    free(malloc);
    function(s);
}
```

- Not really this simple – The lines may be thousands/millions of instructions apart.

NULL pointers and malloc

Not checking if malloc() worked is bad!

- malloc() will return NULL, which almost everywhere is 0. Then we'll try to access the page at address 0x0! With some tricks, this can be exploitable. Most of the time, just a DoS.
- Always make sure people check!

```
if ((s = malloc(1024)) == NULL)
    error();
```

Double frees() can be exploitable, but they're somewhat trixy and depends on the heap library:

```
{
    char *s=malloc();
    free(s);
    free(s);
}
```

More API errors

Due to the way C is compiled/typed, APIs can be used incorrectly without preventing compilation. This doesn’t mean you shouldn’t use the APIs, lots of overflows etc. are in loops meant to replace things like strncpy().

- You have to use APIs properly..

Let’s look at the manpage for memset:

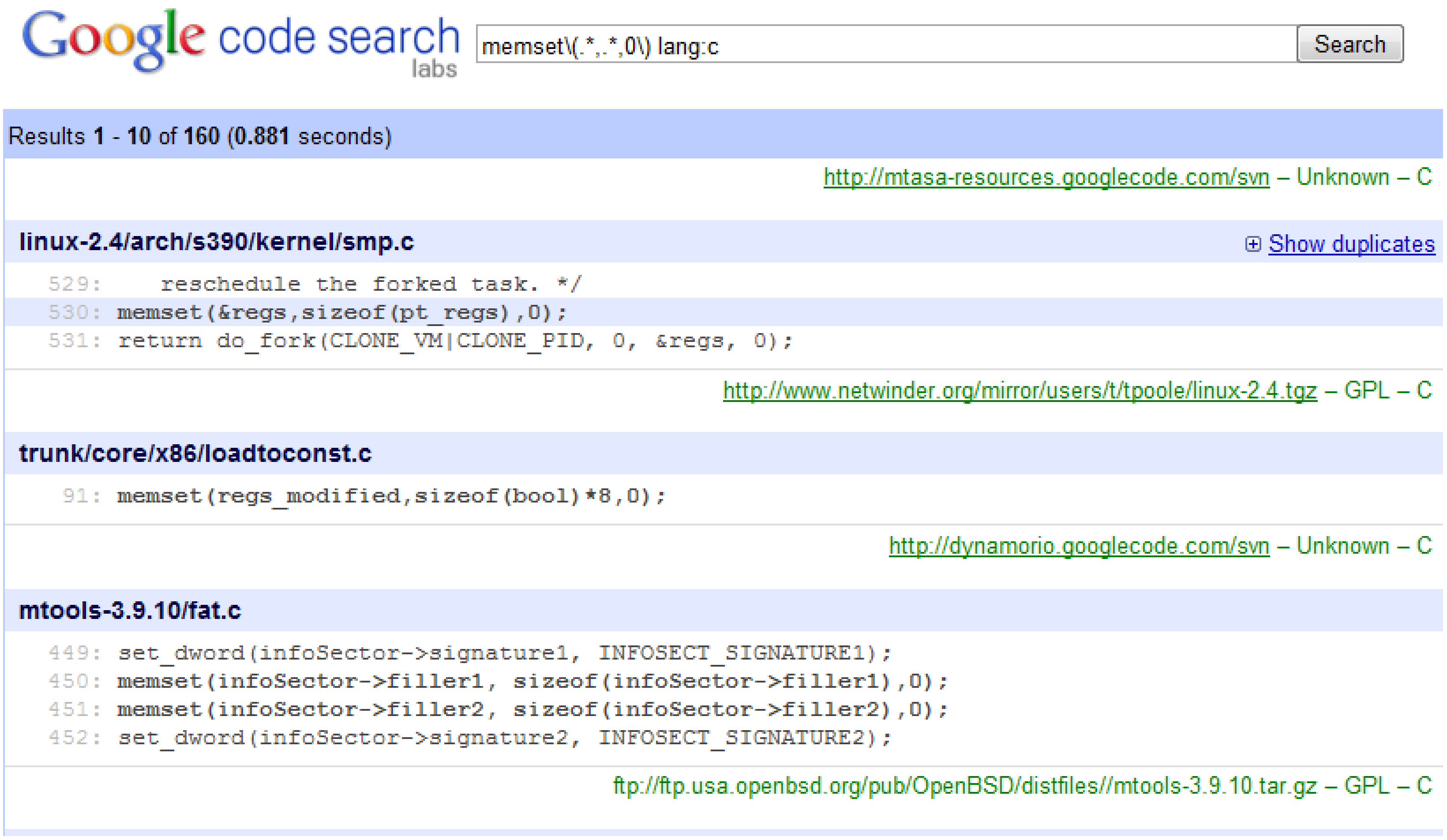
void *memset(void *s, int c, size_t n);

DESCRIPTION

The memset() function fills the first n bytes of the memory area pointed to by s with the constant byte c.

Commonly used by people to zero out memory to prevent sensitive data from being left there

Is anyone really that dumb



memset\(.*,.*,0\) lang:c

Results **1** - **10** of **160** (**0.881** seconds)

http://mtasa-resources.googlecode.com/svn – Unknown – C

linux-2.4/arch/s390/kernel/smp.c ⊞ Show duplicates

```
   reschedule the forked task. */
memset(&regs,sizeof(pt_regs),0);
return do_fork(CLONE_VM|CLONE_PID, 0, &regs, 0);
```

http://www.netwinder.org/mirror/users/t/tpoole/linux-2.4.tgz – GPL – C

trunk/core/x86/loadtoconst.c

```
memset(regs_modified,sizeof(bool)*8,0);
```

http://dynamorio.googlecode.com/svn – Unknown – C

mtools-3.9.10/fat.c

```
set_dword(infoSector->signature1, INFOSECT_SIGNATURE1);
memset(infoSector->filler1, sizeof(infoSector->filler1),0);
memset(infoSector->filler2, sizeof(infoSector->filler2),0);
set_dword(infoSector->signature2, INFOSECT_SIGNATURE2);
```

ftp://ftp.usa.openbsd.org/pub/OpenBSD/distfiles//mtools-3.9.10.tar.gz – GPL – C

Converting between types

You can convert between types explicitly and automatically, may produce a warning:

```
char c;
int i;
char *p = &c;
```

Explicit conversion:

```
if ((int) c == i)
```

Automatic conversions:

```
if (c == i) printf("%d\n", p); //-1262761449
```

Automatic example that warns(but not errors, because this is allowed… weak typing!) when compiled:

```
if (p == c)
```

```
ptr.c:7: warning: comparison between pointer and integer
```

Integer overflows and underflows

Numeric overflow:

```
unsigned int a;
a=0xe0000020;
a=a+0x20000020;
```

Numeric underflow:

```
unsigned int a;
a=0;
a=a-1;
```

Puzzle this:

```
u_char * make_table(unsigned int width, unsigned
int height, u_char *init_row)
{
    unsigned int n;
    int i;
    u_char *buf;
    n = width * height;
    buf = (char *)malloc(n);

    if (!buf) return (NULL);

    for (i=0; i< height; i++)
        memcpy(&buf[i*width], init_row, width);
    return buf;
}
```

Solution

What happens if you specify a width of 0x400 and a height of 0x1000001

result is 0x40000400

modulo 0x10000000 == 0x400 or 1024

So 1024 bytes would be allocated but much much much more be written

Another puzzle

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

Challenge-Response Integer overflow – **openssh 3.1**

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

sizeof() *?

```c
#include <stdio.h>
int
main(void)
{
    printf("%d:%d\n", sizeof(char), sizeof(char*));
}
```

```
[dzhkh@meth ~]$ gcc lol.c -o lol
[dzhkh@meth ~]$ ./lol
1:4
```

sizeof() – samp2.c

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

int
check(char *animal, char *buf)
{
    printf("%s has %d bytes of space\n", animal, sizeof(buf));
}

int
main(void)
{
    char *zebra = malloc(1024);
    char leopard[1024];
    check("zebra", zebra);
    check("leopard", leopard);
    printf("Local zebra has %d bytes of space\n", sizeof(zebra));
    printf("Local leopard has %d bytes of space\n", sizeof(leopard));
}
```

```
[dzhkh@oversized-dildos ~]$ make samp2
cc    samp2.c   -o samp2
[dzhkh@oversized-dildos ~]$ ./samp2
zebra has 8 bytes of space
leopard has 8 bytes of space
Local zebra has 8 bytes of space
Local leopard has 1024 bytes of space
```

sizeof()

Sizeof is very misunderstood. Poor thing.

```
int
zebra(char *buf)
{
    printf("zebra has %d stripes\n", sizeof(buf));
}

int
main(void)
{
    char *pass = malloc(1024);
    zebra(pass);
}
```

```
zebra has 4 stripes
```

Puzzle 3

```
char *read_data(int sockfd)
{
    char *buf;
    int length = network_get_int(sockfd);
    if(!(buf = (char *)malloc(MAXCHARS)))
        die("malloc: %m");
    if(length < 0 || length + 1 >= MAXCHARS){
        free(buf);
        die("bad length: %d", value);
    }
    if(read(sockfd, buf, length) <= 0){
        free(buf);
        die("read: %m");
    }
    buf[value] = '\0';
    return buf;
}
```

Solution

```
char *read_data(int sockfd)
{
    char *buf;
    int length = network_get_int(sockfd);
    if(!(buf = (char *)malloc(MAXCHARS)))
        die("malloc: %m");
    if(length < 0 || length + 1 >= MAXCHARS){
        free(buf);
        die("bad length: %d", value);
    }
    if(read(sockfd, buf, length) <= 0){
        free(buf);
        die("read: %m");
    }
    buf[value] = '\0';
    return buf;
}
```

Explaination

The problem is in the +1 in:

```
if(length < 0 || length + 1 >= MAXCHARS){ }
```

0x7FFFFFFF is > 0

0x80000000 < 0

```
thoth:~/test $ cat test.c
#include <stdio.h>
int main(void) {
    printf("%d and %d\n", 0x80000000, 0x7FFFFFFF);
    return 0;
}
thoth:~/test $ gcc test.c -o test
thoth:~/test $ ./test
-2147483648 and 2147483647
thoth:~/test $
```

Do you crazy +-1 dodginess to values the user doesn't control!

Type conversions

Usually most important in function calls:

```
int copy(char *dst, char *src, unsigned int len)
{
    while (len--)
    *dst++ = *src++;
}

int
main(void)
{
    int f = -1;
    copy(mydst, mysrc, f);
}
```

Really important in calls to libc routines like snprintf, strncpy, memcpy, read, strncat whatEVERZ

Sign extension

unsigned char → int

```
5  → 5
05 → 00 00 00 05
```

signed char → int

```
-5 → -5
FB → FF FF FF FB
```

signed char -> unsigned int

```
-5 → 4294967291
FB → FF FF FF FB
```

signed Int -> unsigned short int

```
-1000000    → 48576
FF F0 BD C0 → BD C0
```

What happens to jim + bob here?

```
int main(void)
{
    unsigned char jim = 255;
    unsigned char bob = 255;
    if ((jim + bob) > 300)
        printf("yes\n");
    else
        printf("no\n");
}
```

What happens to jim + bob here?

```
int main(void)
{
    unsigned char jim = 255;
    unsigned char bob = 255;
    if ((jim + bob) > 300)
        printf("yes\n");
    else
        printf("no\n");
}
```

The answer is “yes”.

What happens here?

```
int
main(void) {
    unsigned short a = 1;
    unsigned short b = 1;
    if ((a-5) < 0)
        printf(“apples\n”);
    b = b - 5;
    if (b < 0)
        printf(“oranges\n”);
}
```

Some words I stole

It prints APPLES only.

Integer types smaller than int are promoted when an operation is performed on them. If all values of the original type can be represented as an int, the value of the smaller type is converted to an int; otherwise, it is converted to an unsigned int.

A bit unexpected huh?

More…

```
int
main(int argc, char *argv[])
{
    char len;
    if (argc == 2)
        len = atoi(get_len_field());
    printf(“len is %x\n”, len);
}
```

```
[user@system ~]$ ./sign 1
len is 1
[user@system ~]$ ./sign 50
len is 32
[user@system ~]$ ./sign 128
len is ffffff80
```

Architecture differences…

```
#include <stdio.h>
int main(void) {
   char p = 0xff;
   unsigned int x = (unsigned int) p;

   printf("Len is %x\n", len);
   return 0;
}
```

Prints

x86:

ARM:

Architecture differences…

```
#include <stdio.h>
int main(void) {
    char p = 0xff;
    unsigned int x = (unsigned int) p;

    printf("Len is %x\n", len);
    return 0;
}
```

Prints

x86: `0xFFFFFFFF`

ARM:

Architecture differences…

```
#include <stdio.h>
int main(void) {
   char p = 0xff;
   unsigned int x = (unsigned int) p;

   printf("Len is %x\n", len);
   return 0;
}
```

Prints

x86: `0xFFFFFFFF`

ARM: `0xFF`

Chars are default UNSIGNED on ARM

A contrived truncation example

```
void assume_privs(unsigned short uid)
{
    seteuid(uid);
    setuid(uid);
}

int become_user(int uid)
{
    if (uid == 0)
        die("root isnt allowed");
    assume_privs(uid);
}
```

Contrived trunc, solution

```
void assume_privs(unsigned short uid)
{
    seteuid(uid);
    setuid(uid);
}

int become_user(int uid)
{
    if (uid == 0)
        die("root isnt allowed");
    assume_privs(uid);
}
```

Become_user takes an int, but assume_privs takes a short…

If shorts are two bytes and ints are 4, any value ending in 0x0000, and beginning with anything other than 0x0000 will meet this criteria. For example 65536 is 0x00010000, but once truncated to a short becomes 0.

One more

```
#define MAX_SIZE 1024
int read_packet(int sockfd)
{
    short length;
    char buf[MAX_SIZE];
    length = network_get_short(sockfd);
    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;
}
```

Sign preserving right shifts

signed char c = 0x80;

c >>= 4;

`1000 0000` value before right shift

`1111 1000` value after right shift

Division operator also is sign aware!

What’s j?

```
int
main(void)
{
    short *j;
    j = (short *)0x1234;
    j = j + 1;
}
```

What’s j?

```
int
main(void)
{
      short *j;
      j = (short *)0x1234;
      j = j + 1;
}
```

j is 0x1236

Other nuances

Order of eval:

```
printf("%d %d\n", i++, i++);
```

or

```
if (check_password(getstr(), getstr())
```

Structure padding

```
struct bob {
    int a;
    unsigned short b;
    unsigned char c;
};

int main(void)
{
    struct bob bobby;
    printf("one: %d, two: %d\n“, (sizeof(bobby.a)
+ sizeof(bobby.b) + sizeof(bobby.c)),
    sizeof(bobby));
}
```

sizeof(bobby.a) + sizeof(bobby.b) + sizeof(bobby.c) == 7
sizeof(bobby) == ?

Sometimes computers are just bullshit ;-)

basename, dirname - parse pathname components

Synopsis

```
#include <libgen.h>
char *dirname(char *path);
char *basename(char *path);
```

Description

Warning: there are two different functions basename() - see below.

```
os.execve('/usr/bin/passwd', [], {})
Segmentation fault
```

Source Code Auditing Strategies

1) Go Deep – Code Comprehension
2) Trace user Input – Focus heavily on trying to attack data points
3) Candidate point – identify every instance of a vulnerability

- There is a chapter in this on the “The Art Of Security Software Assessment” that is required reading

OK

- Can be very confusing
- There is a chapter in this on the “The Art Of Security Software Assessment” that is required reading