Final

1. How many functions exist in this binary? Provide all their names and starting addresses in alphabetical order. Provide a minimum of 3-5 sentence description of each function. If the function is ‘simple’ (comprised of approximately 10 instructions or less), provide the starting and ending address and a brief description of what the function does. (30 pts.)

**15 total custom functions**

|  |  |  |
| --- | --- | --- |
| **Function Name** | **Starting/Ending Address** | **Description** |
|  | **All addresses are preceded with 0x0804[…]** |  |
| assignUserName | 0x0116/0x0201 | This function takes the entered password, and a length(len) which has been hard coded to 9. It prompts for the user to enter a username and scans it using %ms (prevent overflow on scan) and then writes up to 9 characters of that string to the password parameter. All copies are done with strncopy() which prevents overflows here. |
| changeManagerName | 0x0202/0x02ed | This function is redundant as it is identical to assignUserName(). |
| changeName | 0x02ee/0x035d | This function does pretty much the same thing as AssignUserName() except there is no confirmation loop. It prompts for input of length 9, scans with %ms, and then copies that string to the char\* parameter using strncpy(). One interesting detail in this function is that this adds a null byte at (char \*)Param[0x1d] (dec:29)… which is much larger than the string should be… so it’’s probably changing something it shouldn’t. it only writes one byte though and as of yet It seems inconsequential. |
| checkPassword | 0x035e/0x03dc | This function takes the generated password and the user entered password as parameters. If the strings are the same length it runs a strncmp() on them comparing at most 0x14 (dec:20) characters (or until a null terminator). If the strings match then this returns 0x1, which causes the userLevelFlag in main to identify the user as a manager. |
| checkRootPassword | 0x07c6/0x0851 | This has a local pointer to an md5 string hash which presumably is the root password hash. It calls some functions to hash the entered password and compares it to the new hash to the root hash and trips sets a return value to indicate a match or a dud. |
| **generatePassword** | 0x03dd/0x0478 | This function generates the random password. The nested loop structure populates the given string with the specified number of random values between 0x21 (dec:33) and 0x7e (dec:127) and then returns As far as I know, rand has not been seeded yet, so it will produce the same value every time it is run making the randomization predictable. We could probably generate the password, check it once in gdb and know it going forward. |
| **Main** | 0x0896/0x0c9f | Main controls the main menu and related control flow that the user interacts with, and it calls every custom function directly with the exception of menu() and promotions(). The basic structure is to ask for a password, then check that password against the generated password and the root password. This check assigns a value to a flag which identifies the user as a “user”, “manager”, or as a “superuser” – which determines which menu will be displayed, and the options available to you. Issue(?): manager menu seems to scan the user’s choice into the wrong local variable – messes with control flow (breaks it?) |
| **manager\_Budget** | 0x0479/0x04dc | This function has a local array. It prompts the user for input and then **scans to that array using %s** and then prints a failure message and exits… **overflow Opportunity!** |
| manager\_RequestPromotion | 0x04dd /0x0519 | This function takes the userRole/userLevel variable as a pass-by-reference parameter. It checks a variable (global?) that is only modifiable through the promotions() function. If that value is 0 it prints “Denied” and exits. If that value is not 0 it changes the userRole/userLevel value to 0 (super user) |
| menu – not called form main | 0x051a/0x06c4 | This function doesn’t seem to be called from anywhere. It prints a congratulatory message and then writes a seemingly random string to a file called “login” (presumably checked by the verifyFile() function) and then returns |
| passwordPrompt | 0x06c5/0x06f0 | This function prompts the user for input and then scans it into a buffer (buffer allocated by the scanf function rather than the programmer – result of the m in “%ms”) and then returns. |
| **printWelcome** | 0x06f1/0x0729 | Simple function that calls puts() 3x with the welcome message and then returns. |
| promotions – not called from main | 0x072a/ 0x0739 | Sets the (global?) variable AllowPromotions to 1 (signifying promotions are allowed) |
| user\_RequestPromotion | 0x073a/ 0x07c5 | This function takes the generated Manager Password and the userRole/userLevel Flag (pass by reference) as parameters. It then Prompts the user to enter the manager’s password, calls passwordPrompt(),  And then checkPassword() on the given string. If the strings match, it sets the user’s role to main (directly modifying the parameter passed by reference) |
| verifyFile | 0x0852/ 0x0895 | This function checks for the existence of a .login file, returining the output of the stat() function(0 means success, -1 means failure)… this makes it possible to access the congratulatory message when logged in as root (“end of the road”) |

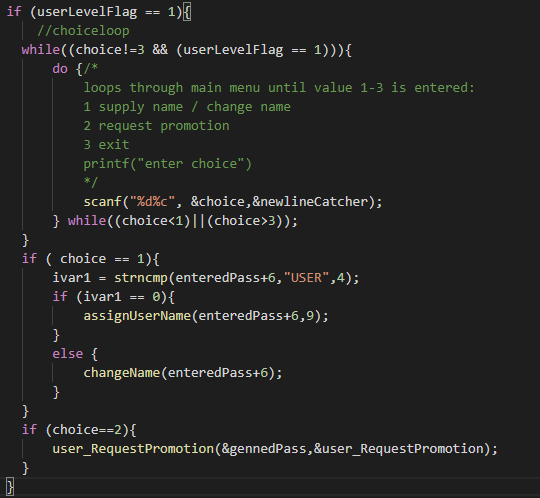
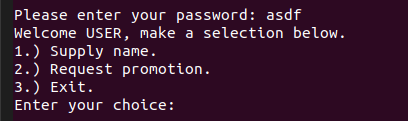
1. How many possible user roles are there? Provide screenshots and descriptions supporting your answers. (5 pts.)

**I combined my answers for #2 and #4 here:**

The screencaps of code are from my personal notes for readability – so a step removed from ghidra’s C-level overview, but I’ve been meticulous and I trust it.

**Three user roles:   
1) User:**

**Option 1:** calls either assignUserName() or changeName() depending on the status of the local flag indicating presence of a username.  
**Option2:** calls user\_RequestPromotion()

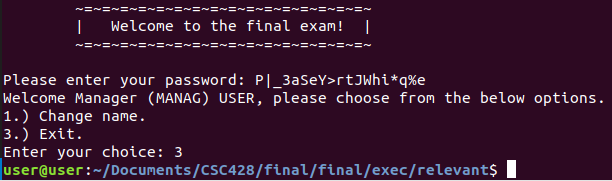
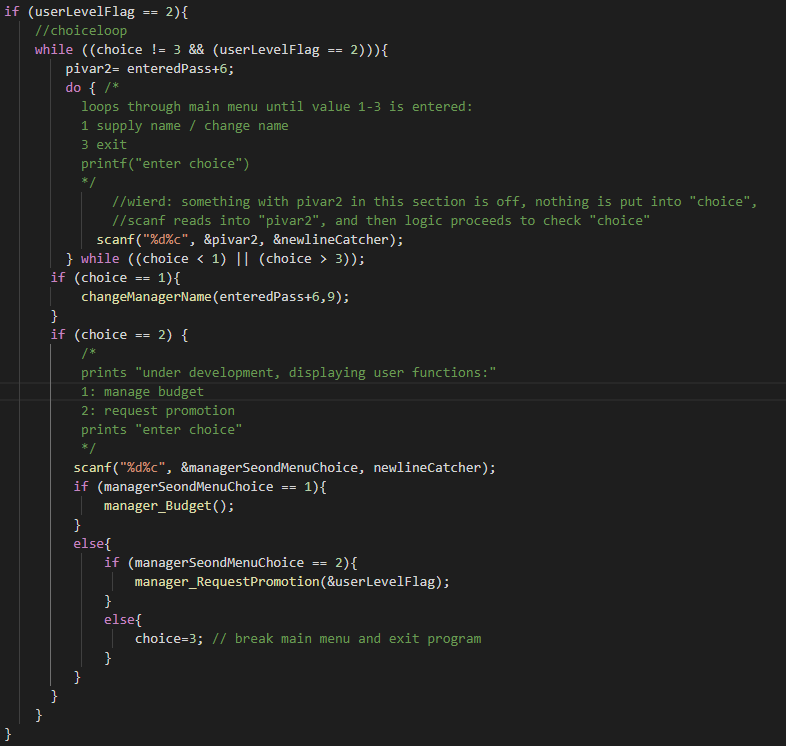
**Option3:** breaks the loop and program exits.  
  
 **2) Manager:**

**Option 1:** this calls changeManagerName()

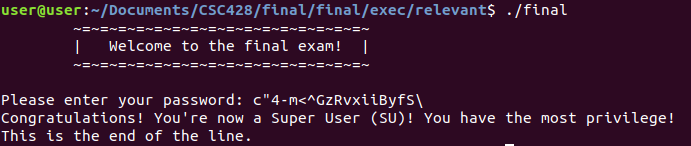
**Option 2(hidden):** this will print a second menu selection with the choice between :  
 1) manage budget ------- calls manager\_Budget()

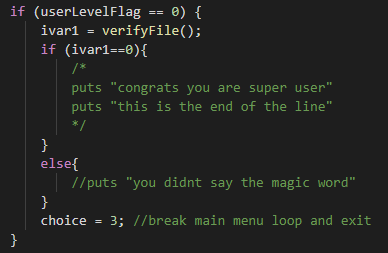
2) request promotion --- calls manager\_RequestPromotion()

**Option 3:** breaks loop and exits program.

**3) SuperUser:** no options





1. What unique format string is being used by scanf? Describe how it works and how it could affect buffer overflow attempts (5 pts.)

**%ms prompts scanf to allocate a buffer appropriately sized for the scanned string. This should prevent buffer overflows in the local memory space since it will probably (I haven’t confirmed by testing it, just inferring from what I’ve read) be allocating space from the heap.**

1. What options are presented to each user role? Provide screenshots of each user roles’ options. If none are presented, provide a screenshot of the program’s output (10 pts.)

**I put this answer together with #2 (see #2) with roles/options/function calls/caps**

1. For every option presented in question 4, provide what function is called by the option. In the ‘**User Role Option**’ list the role followed by the option (ex: role 0xa: Change Password”) (10 pts.)  
   **I put this answer together with #2 (see #2) with roles/options/function calls/caps**
2. Locate and document the vulnerabilities in this program: there are 2 at minimum. Your documentation must include (20pts):

* Screenshots documenting the before and after-effects on memory of the vulnerability being exploited. Describe the screenshots in a minimum of 2-3 sentences.

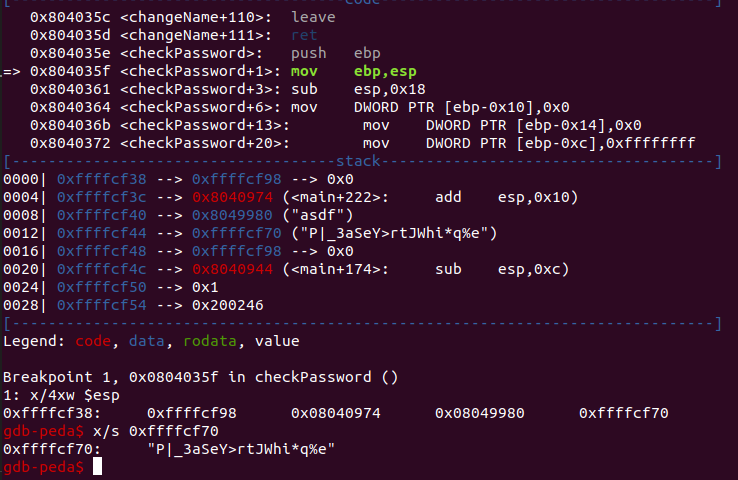
**The two vulnerabilities that I found and exploited are detailed thoroughly below including extensive descriptions and screencap documentation, so I will leave that out from here (I hope that is okay). The two vulnerabilities I’m talking about are a buffer overflow in manager\_Budget that gives me control of EIP, and an unseeded rand() which results in a predictable password.**

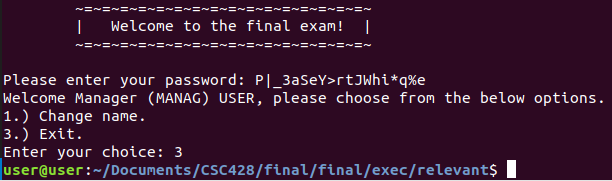
* A description of how the exploitation of the vulnerability affects the execution of the program and what functionality, if any, you gain from exploiting it.

***The Buffer overflow:* this vulnerability gives me control of EIP, so it’s a critical vulnerability that basically lets me go/do whatever I want. It is locked behind the manager user role however. Unfortunately this user role is exposed to unauthorized access due to the randomizer on the password generator being broken.**

***The unseeded rand():* this vulnerability allows me to inspect the memory address of the stored plaintext password which \*should\* be random each go… however because rand is not seeded, it starts from the same page each time, and after looking at this string once I know the manager password going forward… this grants me access to the manager user role, and the buffer overflow hidden in it’s options.**

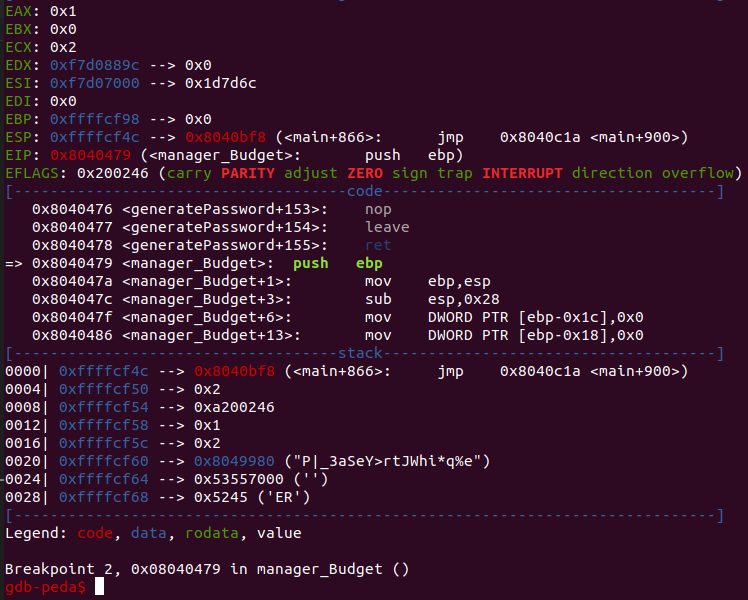
1. What are the passwords used by the binary to enforce user roles? (HINT: They will be human readable characters.) Provide screenshots and descriptions as to how you found the passwords. (5 pts.)

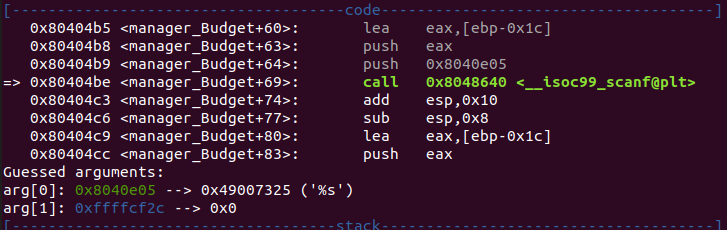
**Manager Role Password:** P|\_3aSeY>rtJWhi\*q%e  
*details:*   
the rand() function used to generate the password in generatePassword() is not seeded, so it generates the same sequence every time. In GDB I set a breakpoint in checkpassword(), and checked the stack for the char\* parameter corresponding to the generated string and popped it out.  




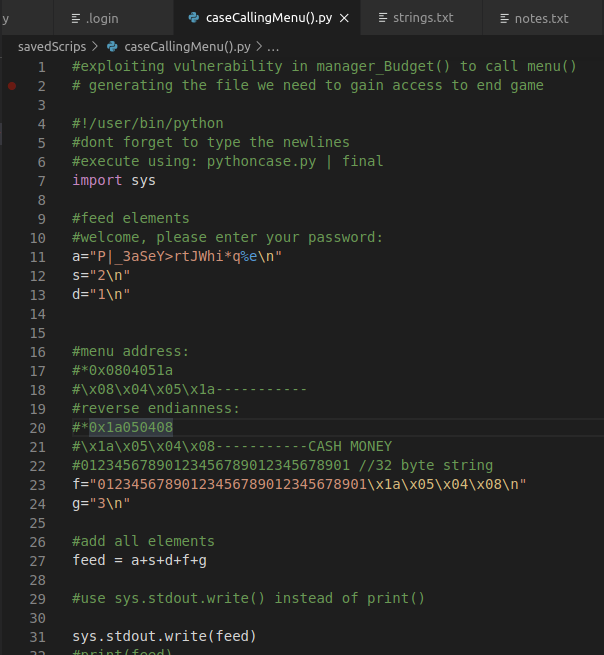
**SuperUser Role Password:** c"4-m<^GzRvxiiByfS\

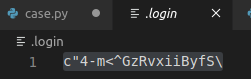
*Details:* this password is provided in the file (.login) produced by the menu() function, which is not normally accessible.

In memory, as I said: break in manager\_Budget. Noting that EIP is located at stack address: **0xffffcf4c.** 

Then we continue until just before the call to scanf, let gdb do the register math for me. It tells me then that [ebp-0x1] c is **0xffffcf2c**   
0xffffcf4c- 0xffffcf2c=0x20(dec:32) so there is a distance of 32 bytes between the head of the string, and the stored EIP.

To access it I had to exploit the buffer overflow in manager\_Budget, overwriting eip to jump to the start of menu(). I could have written the file myself, since each character code is visible in ghidra, as well… but I wanted to make sure the vulnerability was legit and confirm for myself how to actually exploit it by making it work. So I used this script to fiddle with it. at the appropriate prompt it enters 32 bytes of gibberish, and then the memory address of the start of menu()

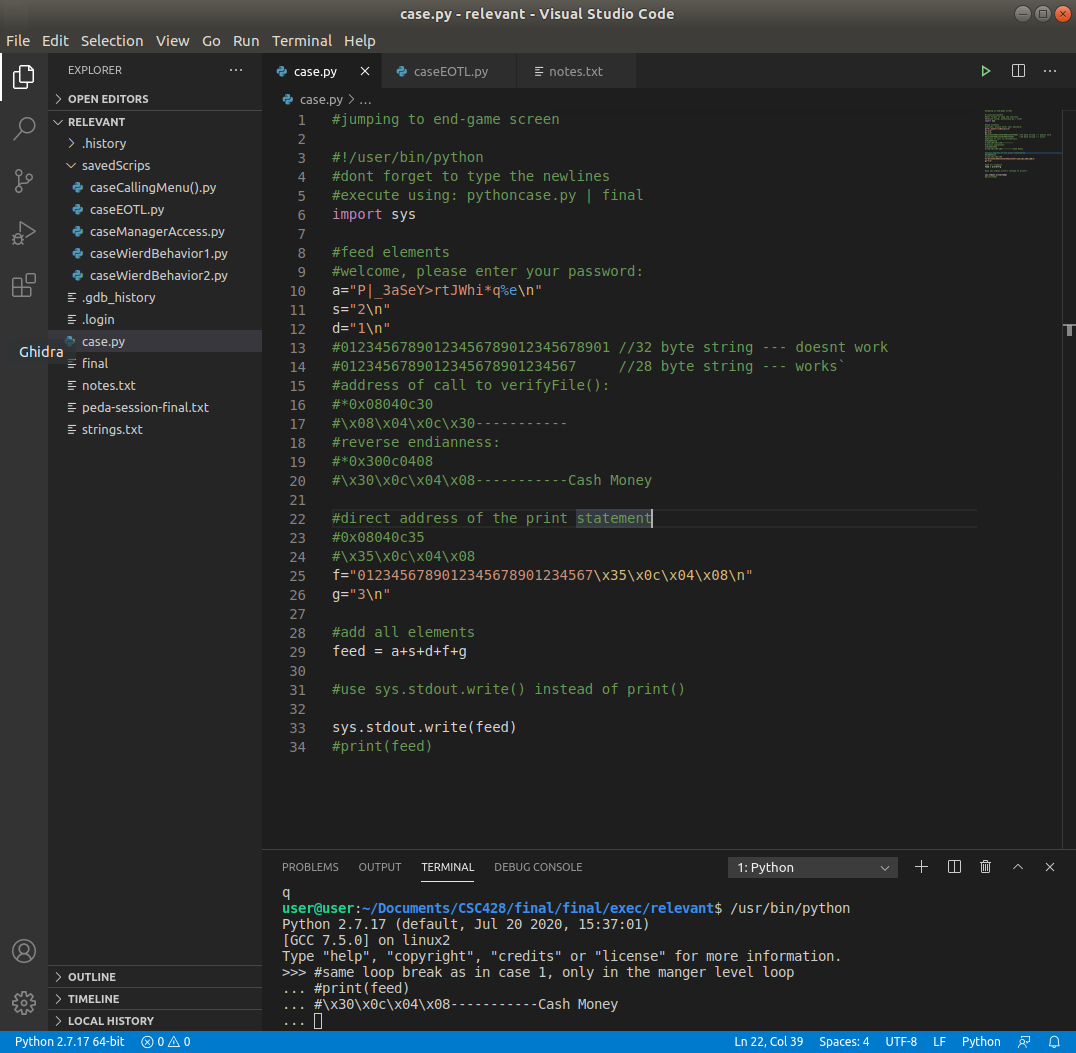




1. Using Python, write an exploit that will enable you to login as the user with most privilege. Take screenshots documenting what your exploit does. Write a 2-3 paragraph summary (4-5 sentences each) of what your exploit does, what vulnerabilities it uses, and how it alters the expected flow of the program to achieve your goal. Name your script “exploit.py” and submit it alongside your answer document in D2L. (10 pts)

**(I’m using the same vulnerability as used above to access main, which I detailed at decent length in regards to the before/after of memory, including screencaps, etc. so I’m going to keep it a little shorter rather than being redundant.)**

**As a prerequisite, this assumes that the user has the manager role. In my case, I plucked the manager password from the broken randomizer sequence.**

**This specific script does not use the password, but instead exploits the vulnerability in manager\_Budget to directly jump into the super-user’s restricted branch. It works exactly the same as the one mentioned above, only with a different address… actually… It’s not exactly the same. It only works if a reduce the gibberish buffer from 32 bytes to 28 bytes (and I don’t understand why.)** 

1. Is there any unusual side effect of the program? Document your findings. (5 pts.)

**I was pretty involved in using ghidra to map everything out before jumping in with gdb, so nothing really caught me off guard. If I was just hopping right in the file generated by menu might have slipped under my nose… if it wasn’t the file, then It must have gone under my nose.**