Project1 Solutions

Saturday, February 2, 2019 5:53 PM

Part 1 - 20 points

Students are expected to have found at least 3 vulnerabilities (one from each task), fixes for **three** of the vulnerabilities, and an exploit for **one**.

- For each vulnerability under 3 not found, -2pts
  - For each one over 3, +1pt extra credit
- For each fix under 3 (or incorrect fixes) -2pts
- For incorrect exploit, -8pts.

Part 2 - 40 points

- Task 4: 20 points

1. What is does the register ebp typically hold and what ebp value is pushed on the stack in a normal stack frame (x86 32 bit)? **(1 pts.)**

   The ebp register typically holds the bottom of the current execution stack frame. Ebp register is pushed onto the stack frame when a function call is made so that the bottom of the current stack frame (the frame from before the call) can be retrieved after the call returns.

2. What does the following x86 assembly instruction do (note destination address comes second here)? **(1 pts.)**
   lea -0x20(%ebp),%eax

   It treats the value stored in register ebp as a memory address, and offsets that address by -0x20. The resulting address is then stored into register eax (not the value found at the resulting address, but the address itself).

   Navigate to the folder Coding Problem 1/ enclosed with this assignment. There you will find a Linux executable file (ELF, 32bit, Optimization Level = 0) that has several vulnerabilities you will exploit, "buf overflow 1." Run the program and enter a password to familiarize yourself with the program (./buf overflow 1).
   For this problem you will need to answer the following questions using readelf and objdump or other programs of your choice.

3. In the source code below, we can see that the programmer hardcoded a password. Use one of the tools above to disassemble the binary and try to guess the password amongst the strings present. Include a screenshot of the section in the executable where the password is located. Include a screenshot entering the password and the successful authentication into the program using the password. What is the correct password? **(2 pts.)**

   corgiesarebad

4. Looking carefully at the source code or disassembled file identify a potential buffer overflow and how it can be used to bypass the password authentication code. **(3 pts.)**

   buff[] has a hardcoded size, and overflowing it will result in a non-zero number being written into variable pass, which results in the authentication check passing regardless of the password.

5. Run the objdump -D -s buf overflow 1 and navigate to the disassembled code for the authenticate function. **(0 pts)**

   No answer needed

6. Include the disassembled output of the authenticate function. **(1 pts)**

   Answer is screenshot or copy/paste of the assembly code for authenticate function.

7. In relation to ebp, where is the variable pass stored (Hint: Use the initial value of pass to find the instruction)? Explain how you figured this out. **(2 pts)**

   From the instruction:

   ```
   5bd:   c6 45 f7 00            movb   $0x0,-0x9(%ebp)
   ```

   We can see that the memory location at ebp-0x9 is initialized to zero. The only variable in function authenticate that is initialized to zero is the variable pass, so ebp-0x9 must be the memory location of pass.

8. In relation to ebp, where is the variable buff stored (Hint: Use the call to gets( ) as a reference point)? Explain how you figured this out. **(2 pts)**

   Using gets() as a reference point, we know arguments being passed to gets() are passed by putting them on the stack before the function call. The sequence of instructions:

   ```
   lea    -0x20(%ebp),%eax
   push   %eax
   call   400 <gets@plt>
   ```

   Loads the memory location ebp-0x20 onto the stack just before the call to gets(), which only takes the buffer as an argument. So ebp-0x20 must be the address of buff.

   Students don't have to use gets() as a reference (strcmp() also works), as long as they get the address correct and explain it.

9. How many bytes long is the buffer that holds the entered password? Explain how you determined this. **(1 pts)**

   Buff is at ebp-0x20, pass is at ebp-0x9. Since buff runs up to pass, we know buff must be 0x20-0x9: 23 bytes long.

10. What is the minimum number of characters a user has to enter in order to overflow the buffer and write a nonzero value to the variable pass (Hint: the null terminator in a string has a value of 0)? **(1 pts)**

    24 characters. 23 to fill up the buffer and one more to write into the address of pass.

11. Use a hexeditor to open the binary file and search for the correct password found at the start of this exercise. Change the password so that the word "bad" is turned into "good". and save the binary. Try to enter your modified password into the changed binary. Did it work? Include a screenshot of the program running with your entry attempt. **(2 pts)**

    Student should give a screenshot of giving the password "corgisaregood" and elevated privileges being granted.

12. Briefly explain how to eliminate the vulnerabilities in this program. **(2 pts)**

Fixes can include:

Not storing the password in plaintext, but a hash of it.

Stack canary's to prevent overflow.

13. Is it a good idea to store sensitive information as a plaintext character array? What are some alternatives? How does the Linux login program handle storing user passwords? (20 pts.) **(2 pts)**

    It is not a good idea to store sensitive information in plaintext ever, even in compiled executables.

    The correct way is to secure the sensitive information cryptographically, either by a secure hash function, or encryption.

    Linux keeps a cryptographic hash of the users password, and when a user attempts to login, it applies the same hash algorithm to the user's input, and compares the result against the stored hash.

Task 5: 20 points

1. Use objdump to disassemble the binary. Navigate to portion of output for function1. **(0 pts)**

   No answer needed

2. In relation to ebp, where is the beginning of the character buffer used to store the string? **(5 pts)**

   ebp-0x28 is the start of the buffer. It can be found using gets() as a reference.

   `ebp-0x28` is the start of the temp `temp_string` buffer.

3. What is the minimum number of bytes you need to write to the character buffer in order to overwrite the return address? **(5 pts)**

   Since the buffer starts at ebp-0x28, we know the buffer is length 40 bytes away from ebp on the stack. We are trying to overwrite the return value, which is located directly after the stored ebp value. This means we need to write 45 bytes to 'bleed' into the stored return address, and 48 to fully overwrite it.

   Distance from ebp to buffer: 40 bytes

   Overwriting ebp: 44 bytes

   Overwriting return address: 48 bytes

4. How many bytes are in an address for a 32 bit binary? What is the minimum number of addresses you need to write from the beginning of the character array to overwrite the return address? **(2 pts)**

   32-bit addresses use 4 bytes.

   48 bytes needed to overwrite return address: 48/4 = 12 addresses

5. What is the address of function2? **(3 pts)**

   `0x08048934`

6. Modify the file, Input Gen/main.c to rewrite the return address with the address for function2 when function1 is called. Run make to compile the binary for the input generator, Input Gen/input gen. Pipe the output of the input generator to the original program. To do this make sure your working directory is Coding Program 2/ and then run the command, Input Gen/input gen | ./buf overflow 2 . **(2 pts)**

   Input_gen/main.c should write the address 0x08048934 12 times.

7. Include your output of the programming calling function2 (Note it is ok if an error occurs after function2 runs). Include the full source code for your input generator and explain why the attack succeeded. **(3 pts)**

Explanation should be along the following lines:

The attack works because the return address defines which code segment to jump back to after completion of the function. Since we wrote over this address with another function, the code will return there. After executing function2, it crashes because the stack no longer makes sense to the program.

In the students code, count should = 11. The for loop runs until i is <= 11, which is 12 times.