Explanations

Sunday, September 29, 2019 7:26 PM

- **overflow**
  - What is the vulnerability?
    - The program is reading a file into a 64 byte buffer without checking that it is not going to write more than 64 bytes. In fact, it takes from the beginning of the file, the number of bytes to try to read.
    - This problem where the program could potentially, for some inputs, allow a buffer to be filled past its capacity, is called a buffer overflow vulnerability.
    - One of the most common attacks on a buffer overflow vulnerability, is to give it an input such that the buffer is overflowed precisely by content that will write into a return address, the address of malicious code. That malicious code can be written into the buffer itself and the "return address" can point back into the buffer.
  - How did I arrive at my attack
    - I noticed that that overflow.c program is compiled to have an executable stack. So I knew I should try to put my shellcode to launch /bin/SH into the buffer itself.
    - I noticed that if at the beginning of the file I put a very big number, the program will read my attacker_controlled_file.txt to the end. So I put the big number at the beginning of the file.
    - Then immediately after in the file, I put my shellcode. Which was only 30 bytes. I needed to know how far to overflow the buffer to put my return address pointing back to the shellcode.
    - I used gdb to see how the stack was layed out, and how many words after the end of the buffer was the return address. I used the "print &buf" command to see where the buffer began in memory and the "info frame" command to see where the return address was stored so I could figure out how much to overflow the buffer.
    - Then I realized that when the program is launched with gdb, the addresses are slightly offset. So i put print statements in the overflow.c file to see at what address buf begins (i.e. where does my shellcode begin, i.e. what do i need to write into the return address).
  - Vulnerability repair
    - This vulnerability could be repaired by putting a check into the overflow.c program, to make sure that no more than 64 bytes from the file are read into the buffer.
- **format1**
  - What is the vulnerability?
    - When using a format function like printf, user supplied data should never be the format string.
    - In format1, on lines 29 and 30, format functions are used. In line 29, the user input is a parameter, not the format string, which is fine.
    - But then the output of this function is used as the format string in line 30. And that is the vulnerability. It allows the attacker to give the program as input a malicious string which will result in the exploit
    - Secondly, it makes an unsafe call to echo which allows me to inject a "&&/bin/SH"
  - By what process did you determine the relevant memory addresses?
    - I knew I had to mess up the forbidden_chars table. I knew a pointer to the table was stored on the stack. So I had to see how many machine words from the buf was stored this pointer, so I could use it as the parameter matching with %n, thus clobbering the table.
    - To find out how many machine words away it was, I passed %p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p into format1 as a parameter, and it printed for me some of the stack.

        - I used gdb to see what the pointer to forbidden_chars was. So I knew the pointer was stored 11 machine words away from the buf where the format string of line 30 was stored.
    - Vulnerability repair
        - A fix for this would be to never allow a string that a user is able to modify be used as a format string. I.e. change line 30 from
            - printf(buf);
            - to
            - printf("%s", buf);
            - Another good change would be to change the call to echo to use a system call rather than use exec. Because a system call drops priviledges.
- **format2**
    - What is the vulnerability?
        - it is a format string vulnerability
        - on line 19, a buffer which contains user input is passed directly into printf. So it allows the user to inject a format string that will overwrite the forbidden_chars array
        - Secondly, it makes an unsafe call to echo which allows me to inject a "&&/bin/SH"
    - By what process did you determine the relevant memory addresses?
        - I knew I needed to overwrite the forbidden_chars array. In order to do that, I needed to find a place on the stack with the address of forbidden_chars, so that i could use %n in the format string to write something in that location, thus overwriting the forbidden chars.
        - With a hint from Dr. Henry, I figured out that as a command line parameter I was passing in an int that would go on the stack. So the solution was to pass in the integer representation of the address of forbidden_chars.
        - To find that address, I put a print statement in the source of format2.c, which would print the address of forbidden_chars.
    - Vulnerability repair
        - A fix for this would be to never allow a string that a user is able to modify be used as a format string. I.e. change line 19 from
            - printf(buf);
            - to
            - printf("%s", buf);
            - Another good change would be to change the call to echo to use a system call rather than use exec. Because a system call drops priviledges.
- **format3**
    - What is the vulnerability?
        - it is a format string vulnerability
        - on line 49, a buffer which contains user input is passed directly in to printf. So it allows the user to inject a format string that will write into the allowed_chars array, the characters it needs for the exploit.
        - Secondly, it makes an unsafe call to echo which allows me to inject a "&&/bin/SH"
    - By what process did you determine the relevant memory addresses?
        - I knew I needed to write the ascii code of the characters I needed for my attack into the allowed_chars array
        - So i inserted into the source code a print statement to print the address of that array
        - then I passed in %p%p%p%p%p%p%p%p%p%p%p%p into the program and checked to see how many pointers up the stack was the pointer to the allowed_chars array
        - Thats how I knew to use "11$"
        - To know what "number" to write into that location, I converted the string of chars I needed to hex. Then I converted that to decimal. Then i subtracted 10 because I knew the "Checking "" part of the string printed before my format string was 10 characters. The result was 623256613

- Vulnerability repair
  - A fix for this would be to never allow a string that a user is able to modify be used as a format string. I.e. change line 49 from
    - printf(buf);
    - to
    - printf("%s", buf);
    - Another good change would be to change the call to echo to use a system call rather than use exec. Because a system call drops priviledges.

- **env**
  - What is the vulnerability?
    - the problem is that env.c executes "echo" without specifying the full path to the /bin/echo program.
    - This allows me to change the PATH environment variable to make the OS look for "echo" in the working directory before in /bin/.
    - So I simply put my own version of echo in the working directory (which launches /bin/SH), and before executing env, I modify the PATH environment variable.
  - By what process did you determine the relevant memory addresses?
    - This does not apply for this question
  - Vulnerability repair
    - The repair would be to call echo with its full path /bin/echo rather than just "echo"
- **tocttou**
  - What is the vulnerability?
    - This is an improvement on env, because it checks that "echo" refers to /bin/echo.
    - The problem is that there is a gap in time between the check for that, and the actual execution of "echo".
    - in that time, an attacker can change the environment in such a way that by the time "echo" is executed, it refers to a version of echo other than the one intended by the programer.
    - My attack was to change the PATH environment variable to look in the working directory first.
    - Then I launch the tocttou program
    - While the tocttou program sleeps, I compile my echo.c program, so that by the time tocttou executes "echo", it is actually my version of echo which lauches /bin/SH.
  - By what process did you determine the relevant memory addresses?
    - doesnt apply in this question
  - Vulnerability repair
    - The best repair here would be to call echo with a full path /bin/echo.