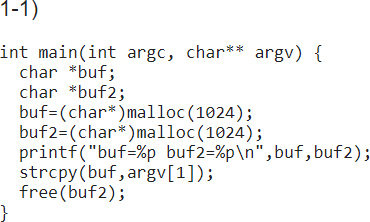
Report

Note: This is just 5 of the codes vulnerable codes showcased in OWASP Top 10 as this report is to showcase my adaptability in applying lessons learned from critical Websites that provide information on relevant cybersecurity practices. This is just some of the more relevant takes on highlighting input validation weaknesses that can be present in systems’ code.

1)

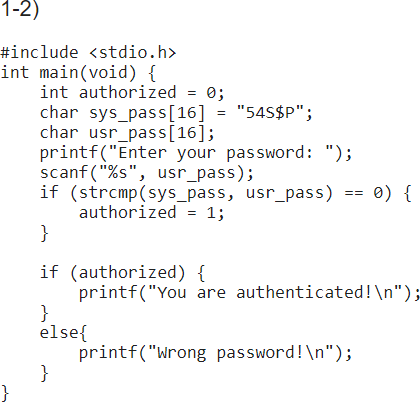


Vulnerable area: Due to copying the contents of the buffer without checking the buffers length first.

^ Vulnerability and how to exploit it: The vulnerability lies in the use of an unsafe function called strcpy. This is due to the fact that it is taking in the argument directly and copying it over to a fixed size of buffer. This means that the arguments length was not checked whether or not if it was within the size of the buffer, which means that I could exploit it by sending in a large amount of characters ‘a’s at least bigger than the memory allocated which it being 1024 bytes.

^ Solution (How to mitigate it): I would mitigate it by dropping the excess characters that surpass the limit of the buffer assigned, so that the memory allocated for the buffer cannot be overflowed. This means that I would be using a more secure function called strncpy as it would drop the excess characters out and thus, prevent me from overflowing the memory with too many characters.

2)



Strcmp also does not pay attention to the length of the inputted string during the comparison. This means that if the user inputted a character that is longer that the sys\_pass than the program may end up reading past the end of the space allocated, thus, making a memory reference to an

unintended address showcasing buffer overflow.

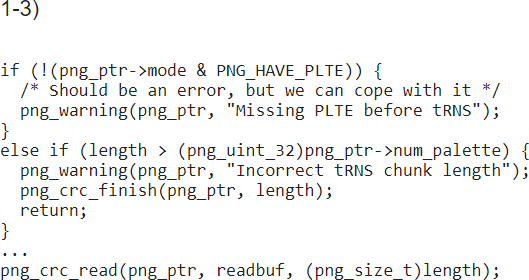
Scanf does not pay attention to

the length of the typed strings. A user can input a string longer than the allocated array (16 characters).

^Vulnerability of the code is already explained in the text boxes so I am assuming that I do not have to explain it again. I can exploit this vulnerability by injecting more than 16 characters to break the expected limit and have the program reference an unintended address. Furthermore, the program would compare the two passwords and make an unintended reference to a memory address due to the obscene length of the password that was inputted, causing buffer overflow as the comparison also does not have any length checking to ensure validity.

^Solution: you can use different functions for getting the user input such as sscanf as it provides a limit on the length of characters that can be entered which means that the user would not be able to add in a large amount of characters to cause the buffer overflow as the excess characters would just be dropped. Furthermore, I will also use the function strncmp to ensure that the comparison is only limited to an expected length and not make an unintended reference to a memory address if one of the variables is way too long.

3)

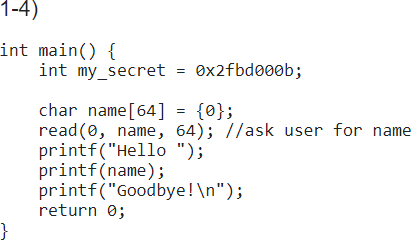


The length validation for the input is in an if-else statement which means that there is a possibility that the length of the input is not validated and that the user can input a variable that is too long.

Vulnerability and how to exploit: the vulnerability explained is in the box so I am also assuming that I do not have to state what it is again. I can exploit this vulnerability by injecting characters that would be caught by the first statement while also being long enough to overflow the allocated memory of the program. This means that the injected characters would avoid the length checking as it is in an if-else statement and successfully overflow the memory allocated to the variable.

Solution: I would remove the ‘else’ part of the code. This would ensure that the length checking occurs no matter what happens previously and that the buffer overflow attack cannot occur as the length checking will always occur, thus making sure that the input characters cannot be too long.

4)

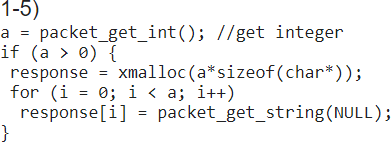


Lack of formatting in the output of the printf.

Vulnerability: The vulnerability in question is format string attack. This is due to -as shown in the text box above- the lack of formatting as a string when the program replies to the user. I can exploit this vulnerability by instead of typing in regular characters I can type out references to memory such as %p multiple times in order to have the program make a reference on an unintended memory address, thus causing a buffer overflow attack. I.e., “Kevin %p %p %p %p %p %p”.

Solution: In order to solve this critical issue I would format the variable name as a string. This would treat the output as a string entirely and not mistake for any commands. I.e., printf(“%s\n”, name);

5)



This area is vulnerable due to how if the end result for the calculation for xmalloc is too long and overflows

then the memory allocated would be too small. This is also due to the fact that char\* is 4 bytes

And if “a” is a very large number as well, then the Extra numbers from the output would flow out and the overall space allocated would be extremely small as the value simply spill out and basically “restart” and go back to 0 and increase from there.

Vulnerability and how to exploit it: The vulnerability as shown in the text box above is during the calculation where when getting the large number and multiplying it by 4 is too large, then the space allocated would be extremely small. I would exploit this vulnerability by sending in an integer so large that the space allocated for the variable response would be too small as the numbers would flow out, thus resulting in a successful buffer overflow attack. This can lead to the buffer overflow attack as getting the strings to input into the response would lead to a reference to an unallocated memory space due to the initial allocation being too small from the vulnerability pointed out and explained in the text box above.

Solution: I can solve this issue by making sure that the calculation is validated before allocating the memory space to the variable. This means that I can check whether or not the space allocated would result in values spilling over. I could also use post calculation validation to ensure that the space allocated is the required amount and not too short and compensate accordingly.