

<b>Student Name:</b>	Rameez Ahmed
<b>Student Email:</b>	k21064605@kcl.ac.uk
<b>Student ID:</b>	K21064605
<b>Hacker Handle:</b>	r4m3z

## Instructions:

This is a **template file** for providing **explanations** for your solutions of the coursework challenges.

1. First, please **provide your details in the fields above** (name, email, ID, VM username).
2. Second, for each **solved challenge**, explain the **identified vulnerability** as well as your **exploitation method**. For more information about **what to include in your explanation**, please refer to the **example template** given below. Please provide your answers for each challenge on the corresponding page, starting with **Challenge 1** on **page 2**.
3. Finally, after typing your answers, please **save this file as PDF** (Explanations.pdf) and **include it in the submitted archive** along with your exploit files (as explained in the **Coursework Guide**).

## Example template for your answer

Challenge X	
1. Explain the Vulnerability	<p>What to include in your <b>explanation of the <u>vulnerability</u></b>?</p> <ul style="list-style-type: none"> <li>➤ name the vulnerability (eg: <b>vulnerable to XYZ attack</b>)</li> <li>➤ where is the vulnerability introduced in the source code and how did you identify it?</li> <li>➤ you <b>may</b> include other relevant information.</li> </ul>
2. Explain Your Exploit	<p>What to include in your <b>explanation of the <u>exploit</u></b>?</p> <ul style="list-style-type: none"> <li>➤ how does your exploit work?</li> <li>➤ for <b>challenges 5-10</b>, if your exploit uses <b>information obtained from debugging the program</b>, briefly explain <b>how you obtained such info</b> (eg: stack layout, offsets &amp; addresses, etc).</li> <li>➤ you <b>may</b> include <b>additional information</b>, such as</li> </ul>

	<p>interesting findings, alternative exploit methods, etc.</p> <p>➤ if your exploit uses code snippets authored by someone else, <b>make sure to cite the source</b>.</p>
--	---

Challenge 1	
1. Explain the Vulnerability	<p>Environment Vulnerability:</p> <p>" ~/.secret " the ~ means the location is dependent on the home directory saved in the environment. Therefore, if an adversary has access and ability to change the environment, then they have control on the path that the program takes to check the access for the secret. In this case, the program executes the diff command to check the difference between the secret file in /var/challenge/level1 to a secret file in the home directory. So by having access to the environment variable HOME, they could just change the variable to be the location of where the secret file already is.</p>
2. Explain Your Exploit	<p>Since we know that the ~ represents the HOME variable in the environment, if we alter the HOME variable and point it to where the secret file is currently being var/challenge/level1 so that the second part of the diff command input will be the same as the first input and status will allow us access to the shel with the priviledges to execute l33t</p>

Challenge 2	
1. Explain the Vulnerability	<p>Time Race Condition Vulnerability:</p> <p>the program has a timer for 5 seconds before outputting. Within this 5 seconds, a script.sh file is created, written to, printed then deleted within that time. that script has higher priviledges to allow users to execute the necho command. with the priviledges, if an adversary were to have access to write to this file created within the time frame, they can overwrite what is in that file and can execute whatever commands they want with those priviledges</p>
2. Explain Your Exploit	<p>The exploit leverages this behavior by starting the program (./../var/challenge/level2/2 &amp;) and introducing a brief delay (/bin/sleep 1) to ensure the program creates script.sh. During this window, the exploit overwrites script.sh with a malicious version that contains a Bash shebang (#!/bin/bash) and a command to execute the privileged binary /usr/local/bin/l33t. When the program subsequently</p>

	executes script.sh, it runs the malicious payload, achieving unauthorized command execution.
--	--

Challenge 3	
1. Explain the Vulnerability	<p>Path traversal attack:</p> <p>The vulnerability in Level 3 is due to the lack of validation when constructing a file path using user-controlled input. The program directly concatenates argv[1] into the path variable, allowing attackers to inject malicious paths that escape the intended directory structure and point to unauthorized binaries. This type of attack is known as a path traversal attack, where an attacker manipulates file paths to access or execute files outside the intended directory scope, potentially leading to privilege escalation or unauthorized actions.</p>
2. Explain Your Exploit	<p>This exploit targets the vulnerability in Level 3, where the program constructs a file path using user input from argv[1] without validation. By providing a crafted relative path (<code>../../../../../usr/local/bin/l33t</code>), the exploit manipulates the constructed path to point to the privileged binary <code>/usr/local/bin/l33t</code> instead of the intended target. When the program executes the constructed path, it runs the <code>l33t</code> command instead, achieving unauthorized command execution.</p>

Challenge 4	
1. Explain the Vulnerability	<p>Command Injection attacks + Environment Vulnerability:</p> <p>The vulnerable program is susceptible to a command injection attack due to its handling of user input. It takes arguments from the command line and attempts to execute a <code>find</code> command on the user's home directory using <code>execl</code>. However, it only partially sanitizes the input by checking for certain special characters like <code>;</code>, <code>&amp;</code>, and others, which can be used to chain additional commands. The program does not correctly escape or handle these characters in a safe way, leaving it open to command injection. An attacker can exploit this vulnerability by injecting shell metacharacters to execute arbitrary commands with the same privileges as the vulnerable program. Additionally, the program trusts the environment, specifically the <code>HOME</code> variable, to resolve the user's home directory path. Since the attacker has control over the environment, they can modify the <code>HOME</code> variable to point to any directory, enabling them to manipulate the execution path and gain control over the program's behavior.</p>
2. Explain Your Exploit	<p>The exploit script exploits the command injection vulnerability by</p>

	<p>leveraging an environment variable to manipulate the execution context. It sets the HOME environment variable to /usr/local/bin, which influences the path used by the vulnerable program. The script then invokes the vulnerable program with an argument designed to inject the command l33t -exec l33t {} \;, which runs the l33t command with elevated privileges. This allows the attacker to bypass the input checks, executing arbitrary code such as invoking the l33t program or any other command with elevated privileges, effectively compromising the system's security. By controlling the environment and exploiting the trust the program places in it, the attacker is able to execute arbitrary commands as a trusted user, bypassing security restrictions.</p>
--	---

Challenge 5	
1. Explain the Vulnerability	<p>Buffer Overflow:</p> <p>The vulnerable program in this case is susceptible to a buffer overflow attack due to the unsafe use of the strcpy and gets functions. Both functions do not check the length of the input being copied into the buffer and filename arrays, making them vulnerable to overflow if the input exceeds the allocated size. Specifically, the buffer is defined with a size of 192 bytes, but no bounds checking is performed before copying data into it. This allows an attacker to overwrite adjacent memory, potentially including the return address or other critical variables in the program's stack. In this case, the attacker could exploit the buffer overflow to inject malicious code or control the program's execution flow. Additionally, the program allows the execution of files based on user input and trusts the filename, which increases the risk of arbitrary code execution if an attacker can overwrite the program's flow.</p>
2. Explain Your Exploit	<p>The exploit script takes advantage of the buffer overflow vulnerability by providing an oversized input to the gets function. The script first constructs a string of 192 'a' characters, which fills the buffer array. Then, it injects a command that modifies the program's flow. By appending /usr/local/bin/l33t to the buffer content, the attacker is attempting to exploit the overflow to change the return address and redirect execution to the l33t program located in /usr/local/bin. Once the buffer overflow occurs, the program calls execlp to execute the filename specified, and the attacker can hijack the execution flow to run the l33t command with elevated privileges, potentially compromising the system. This attack is particularly effective because the program does not adequately validate or sanitize the input and</p>

	allows unchecked execution of arbitrary code.
--	---

Challenge 6	
1. Explain the Vulnerability	<p>Buffer Overflow</p> <p>The vulnerable program suffers from a critical issue with the way it handles the length argument, particularly when the input value is -1. The atoi function converts the length argument from a string to an integer, and since the length variable is an unsigned short, the value -1 is cast to the maximum unsigned short value, 65535. However, this value is later used to allocate the buffer using alloca. This casting error is problematic because the buffer size is intended to be a positive integer, but when -1 is provided, it results in an allocation of 0 bytes for the buffer due to the unsigned casting behavior. As a result, the program effectively creates a zero-length buffer, which causes the memory to be mismanaged. The program proceeds to write into this zero-length buffer, allowing the attacker to write directly into memory, potentially overwriting critical data such as the return address or other control structures.</p>
2. Explain Your Exploit	<p>The exploit script takes advantage of this vulnerability by passing -1 as the length argument, which causes the length variable to be cast to 0, resulting in a zero-sized buffer. Since the buffer has no valid size, the program writes directly to memory, allowing the attacker to exploit this by calculating the distance from the buffer to the saved return address. Through this calculation, the attacker determines that the return address is located 28 bytes from the start of their input. The attacker then uses the format of the program to write exactly to bytes 28, 29, 30, and 31, carefully overwriting the return address without disrupting other important stack data. To execute the attack, the attacker sets their shellcode as an environment variable, specifically SHELLCODE, which contains a series of NOP instructions followed by the malicious payload. Using gdb, the attacker determines the address of the shellcode in memory, ensuring the exploit targets the correct memory location. The script then passes this shellcode's address to the vulnerable program, causing it to overwrite the return address with the shellcode's location. When the buffer overflow occurs, the program executes the shellcode, allowing the attacker to run arbitrary commands such as invoking the l33t program with elevated privileges. This precise control over the buffer overflow, combined with the environment manipulation and GDB analysis, allows the attacker to</p>

	hijack the program's execution and escalate privileges.
--	---

## Challenge 7

<b>1. Explain the Vulnerability</b>	<p>Of By One + Buffer Overflow:</p> <p>The main vulnerability in the program is an off-by-one error in the buffer size validation. The condition <code>strlen(argv[1]) &gt; sizeof(username)    strlen(argv[2]) &gt; sizeof(password)    strlen(argv[3]) &gt; sizeof(hostname)</code> should be using <code>&gt;=</code> rather than <code>&gt;</code>. This is because <code>strlen()</code> counts the number of characters in the string, excluding the null terminator, while <code>sizeof()</code> includes the total buffer size, including the null byte. As a result, the program fails to properly account for the null terminator when validating the input length. By using <code>&gt;</code> instead of <code>&gt;=</code>, the program allows a string that exactly fills the buffer (including the null byte) to be copied without triggering an error. This creates an off-by-one vulnerability, where if the user provides input that fills the buffer exactly (e.g., 64 bytes for username), it overflows the buffer because the null byte isn't considered in the comparison, potentially corrupting adjacent memory. When this happens, the program continues to copy input into the next buffer, which may overwrite critical data, such as the return address, leading to potential exploitation. Once the buffers begin to overflow and spill into the next buffer, if enough data is copied, it will eventually overflow the result buffer as well, which is directly adjacent in memory. This allows the attacker to overwrite important parts of the stack, including the return address.</p>
<b>2. Explain Your Exploit</b>	<p>The exploit script takes advantage of this off-by-one error by carefully crafting input that will overflow the result buffer and overwrite the return address. The script starts by constructing large inputs for the username, password, and hostname arguments, each filled with 64 bytes of NOP instructions (<code>\x90</code>). This ensures that the buffers are filled to their maximum size. In the final input for the hostname, the attacker uses a carefully calculated offset to overwrite the return address. The shellcode, stored in the <code>SHELLCODE</code> environment variable, is injected into the input at the correct location to replace the return address. The hostname input is constructed to write 11 bytes of shellcode (<code>\x90</code> padding followed by the payload) before inserting the address of the shellcode, represented as <code>\x27\xfe\xff\xbf</code>. This precise offset allows the attacker to control the return address and redirect the program's execution to the injected shellcode. By calculating the exact number of bytes needed to reach the saved return address on the stack, the</p>

	attacker is able to execute arbitrary code, such as invoking the l33t program. The off-by-one error, combined with the failure to properly validate the null byte, enables this successful buffer overflow exploitation, ultimately allowing the attacker to overwrite the return address and hijack the program's execution.
--	---

Challenge 8	
<b>1. Explain the Vulnerability</b>	<p>Format string attack</p> <p>The vulnerability in this program arises from the improper use of the fprintf function without a format string. Specifically, the fprintf function is called with log_entry and NULL as the arguments, but the format string is not properly specified for the function. This allows a user to control the format string, leading to a format string vulnerability. The user can inject arbitrary format specifiers, such as %n, to control the program's execution flow and manipulate memory. In this case, the attacker can overwrite specific locations on the stack, such as return addresses or function pointers, by using the %n format specifier. In this case the GOT was targetted. The vulnerability exists because the program trusts user input without properly sanitizing it, enabling the attacker to exploit the program's logic to execute arbitrary commands. The lack of proper validation and format string handling allows an attacker to bypass security controls and execute arbitrary commands with elevated privileges.</p>
<b>2. Explain Your Exploit</b>	<p>The first step was to find the offset from the user input to where it is stored in memory. Done this by creating a program to find the instances of 41414141 when AAAA is input. This needed padding as they were split so the input was BBAAAA and the offset was 68. knowing this I found my GOT addresses and chose fclose as that was called after overwriting the GOT address. Then I crafter my write 4 primitive, where the first input was the padding found from before then it would be my 4 addresses in little endian of the GOT address fclose. The next part would be overwriting the address with the address of my shellcode that I put in the environment, the address being 0xbffffe27. The first part of the address being 27, I calculated the width by counting all the bytes which was 6(uid: )+2(padding:AA)+16bytes = 24 and did the decimal value of 27 being 39 and took that away from 24 giving me 15. The next was FE being 254-39 = 215, then the rest were calculated as 257 and 192. This width was put in the write 4 primitive and the offset as well, offset increasing by one for each GOT address. This then executed my</p>

	shellcode and performed l33t
--	------------------------------

<b>Challenge 9</b>
--------------------

<b>1. Explain the Vulnerability</b>	
-------------------------------------	--

<b>2. Explain Your Exploit</b>	
--------------------------------	--

<b>Challenge 10</b>
---------------------

<b>1. Explain the Vulnerability</b>	
-------------------------------------	--

<b>2. Explain Your Exploit</b>	
--------------------------------	--