Security Engineering 7CCSMSEN
Semester 1 (2023-2024)
Coursework Submission

Student Name:	**Tyler Supersad**
Student Email:	**tyler.supersad@kcl.ac.uk**
Student ID:	**K23139090**
Hacker Handle:	**tyl3r**

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	**What to include in your explanation of the vulnerability?** ➤ name the vulnerability (eg: vulnerable to XYZ attack) ➤ where is the vulnerability introduced in the source code and how did you identify it? ➤ you **may** include other relevant information.
2. Explain Your Exploit	**What to include in your explanation of the exploit?** ➤ how does your exploit work? ➤ for **challenges 5-10**, if your exploit uses **information obtained from debugging the program**, briefly explain **how you obtained such info** (eg: stack layout, offsets & addresses, etc). ➤ you **may** include **additional information**, such as interesting findings, alternative exploit methods, etc. ➤ if your exploit uses code snippets authored by someone else, **make sure to cite the source**.

Challenge 1	
1. Explain the Vulnerability	The vulnerability within the code depicted in 1.c was initially identified through active exploration of the program's behavior. During this exploration, a crucial observation surfaced: the program invoked the shell exclusively when the two files being compared were determined to be equivalent. This observation sparked an investigative focus on the comparison process and its reliance on the "diff" command to discern the contents of the ".secret" file in the system's challenge directory against a file in the user's home directory, signified by the tilde symbol (~) [line 11 of 1.c]. Further scrutiny uncovered that the program's vulnerability lies in its implicit trust in the environment variable to reliably denote the user's home directory. This trust, however, becomes a security risk, as manipulating the "~" variable emerges as a potential exploit path, hypothetically prompting the program to unintentionally compare the ".secret" file with a file under the attacker's control. This identified susceptibility exemplifies a vulnerability to an environment attack.
2. Explain Your Exploit	The exploit targets the blind trust placed in the environment variable $HOME, which the program relies on to locate a file for comparison. By manipulating $HOME to point to the directory containing the program, as demonstrated by setting *HOME=/var/challenge/level1* in the command-line, the “diff” command inadvertently matches the same file against itself, falsely confirming their equality. This manipulation deceived the authentication logic, enabling the program to grant shell access, thereby allowing the execution of the l33t command.

Challenge 2	
1. Explain the Vulnerability	The vulnerability present in the 2.c program stems from a critical race condition that emerges within the process flow involving the creation and subsequent execution of the script.sh file. This program operates through a sequential routine: generating a script, writing its content, a brief waiting period, execution, and finally, removing the script file. A race condition denotes a scenario where a program's behavior hinges on the timing and sequence of events. In this context, vulnerability arises due to a temporal window - a short interval between script creation and execution. This interval, specifically demonstrated by the for loop waiting five seconds before initiating the execution of script.sh on line 31, presents an exploitable opportunity for a malicious actor. During this critical time window, an attacker can exploit the vulnerability by manipulating or replacing the contents of the script.sh file, leading to unauthorized execution of arbitrary code [RACE CONDITON ATTACK].
2. Explain Your Exploit	*Before delving into the exploitation process, it's important to note that my progression to the next challenge wasn't immediately apparent. There wasn't a clear indication of passing the level until I checked the scoreboards and discovered my advancement to level 3. This realization occurred a few days after passing, leaving me perplexed as I attempted to decipher the methods that facilitated my success. My approach varied considerably throughout, making it challenging to pinpoint the exact technique responsible. However, here's what I believe contributed to my successful completion of level 2:* To exploit the race condition vulnerability, I utilized two terminals: one within the '/var/challenge/level2' directory to execute '2.c' and another in the HOME directory for easy access to the soon-to-be-created 'script.sh'. Upon executing '2.c', the 'script.sh' file emerged in the HOME directory, providing a narrow five-second window for me to modify its contents. Realizing that 'rm script.sh' was swifter than purging the contents within 'script.sh', I promptly removed the file. Subsequently, I generated 'script.sh' via the 'nano script.sh' command and inserted the command 'echo "/usr/local/bin/l33t"' into the file. I then saved the file's contents, ensuring that upon execution, 'script.sh' would trigger the 'l33t' command. *To optimize efficiency and guarantee execution within the five-second timeframe, I pre-saved these commands, facilitating rapid execution with convenient access using the arrow keys.*

Challenge 3	
1. Explain the Vulnerability	The vulnerability in the code (3.c) originates from its sensitivity to a path traversal attack. Upon thorough examination, the program expects user input to be either `cat` or `ls`, subjecting it to security checks. It then dynamically allocates memory to create a command path with the user input, intending to execute one of the binaries in the `/devel/bin` directory based on the provided command (`/devel/bin/cat` or `/devel/bin/ls`). Despite the program's attempt to bolster security by scrutinizing prohibited characters, the persistent path traversal vulnerability stems from insufficient sanitization. Exploiting this weakness, external entities could redirect from `/var/challenge/level3/devel/bin` to `/usr/local/bin/l33t`. This flaw exposes the code from mishandling user input.
2. Explain Your Exploit	In a path traversal attack, an attacker can manipulate input by introducing "../" sequences, enabling them to navigate up the directory tree. When executed, 3.c is designed to run within the /var/challenge/level3/devel/bin directory. In this scenario, a malicious actor could provide input crafted to prompt the program to execute a binary outside this directory, such as "/usr/local/bin/l33t." To determine the necessary traversal steps, I calculated the levels between `/var/challenge/level3/devel/bin` and the target `/usr/local/bin/l33t` directory. By using "cd .." iteratively, they identify the number of steps required to reach the root ("/") and subsequently access the "usr" directory, ultimately reaching "l33t." In this case, my analysis revealed it was five levels up. Consequently, the executed command becomes: ./3 ../../../../../usr/local/bin/l33t.

Challenge 4	
1. Explain the Vulnerability	The code in 4.c exhibits a susceptibility to command injection attacks, where the program, designed to build and execute a find command for searching files in the HOME directory based on specified patterns, lacks robust handling of special characters in user input despite implemented checks. Consequently, the vulnerability is primarily attributed to the use of the snprintf function in line 42, which directly interpolates user-provided input (argv[i]) into the command string. This creates a command injection risk, allowing an attacker to manipulate the constructed command and inject additional commands to be executed. Special characters such as ;, |, and & can be used maliciously.
2. Explain Your Exploit	With the program expecting user input to build a `find` command for searching files within the HOME directory, I strategically created a simple script file (`malicious.sh`) within the HOME directory. The content of the script file is inconsequential; its sole purpose was to exist within the directory. The exploitation takes place by navigating to the directory containing the program, in this case, /var/challenge/level4 where 4.c is located. Instead of providing a typical search term as an argument in the command line, I supply the following input: ./4 "malicious.sh -exec l33t {} \;". This input manipulates the find command by appending the -exec option with the l33t command to be executed on the file found (`malicious.sh`). As a result, the intended l33t command is executed.

Challenge 5	
1. Explain the Vulnerability	The program utilizes two global character arrays, `buffer` and `filename`, each allocated with a size of 192 characters and residing in the BSS segment. When executed, the program expects two command-line arguments: the first denotes the program to execute (`sort` or `uniq`), and the second serves as an input parameter for that program. Input acquisition employs different strategies based on the presence of the second argument; if provided, `strcpy()` is used to copy it into `buffer`, and in its absence, the less secure `gets()` function is employed to directly acquire user input. Both `buffer` and `filename` are stored in contiguous memory within the BSS segment and are not stack-allocated. The absence of proper size checking during input processing makes the program susceptible to a buffer overflow attack. Should the input data exceed the buffer's 192-character limit, overflow occurs into the adjacent `filename` buffer. This overflow presents the risk of unpredictable consequences, potentially resulting in the overwriting of `filename` contents with data from `buffer`. In practice, an attacker can exploit this vulnerability by manipulating such input, ultimately gaining unauthorized control over the program's execution flow.
2. Explain Your Exploit	The program was exploited by manipulating a buffer overflow vulnerability during the processing of command-line arguments. The vulnerable code failed to enforce proper size checking when copying the second command-line argument into the buffer variable, which was allocated a fixed size of 192 bytes. The provided exploit input took advantage of this oversight by crafting a command that, when executed, exceeded the buffer size. Specifically, the exploit input used the python print command to generate a string consisting of 192 A's followed by "/usr/local/bin/l33t". This string, when copied into the buffer, overflowed into the adjacent filename variable, overwriting its contents. Due to the lack of adequate bounds checking, the manipulated `filename` variable now contained the injected string "/usr/local/bin/l33t". Subsequently, the program executed this manipulated filename using the execlp() function. As a result, the l33t command was invoked.

Challenge 6

1. Explain the Vulnerability	The 6.c program features a vulnerability arising from unchecked user input validation during the allocation of a buffer using the alloca function in line 32. The program is designed to construct a string of a specified length based on character-index pairs provided as command-line arguments. The critical flaw lies in the lack of validation for the user-defined length (buffersize), allowing potential exploitation. Specifically, an attacker can intentionally set a large buffer size, such as 65535, leading to an unsigned integer wraparound. Consequently, the buffer size becomes effectively zero, and the subsequent allocation using alloca may result in inadequate space for data storage. This lack of buffer capacity introduces a potential avenue for exploitation, particularly in scenarios involving control flow hijacking. As the buffer size becomes zero due to the wraparound, attempts to write to the buffer (buffer[index] = argv[i][0];) may lead to unexpected behaviors. These could include endeavors to write to memory locations beyond the intended buffer, driven by miscalculations or undefined behavior. For example, an attacker might seek to overwrite the saved EIP. This manipulation can be strategically achieved through input arguments that, even within the constraints of a zero-sized buffer, trigger the execution of a specific "l33t" command, facilitating the attacker's control over the program's behavior.

2. Explain Your Exploit	The vulnerability in the program allows for exploitation to execute injected shellcode, specifically invoking the program 'l33t', by manipulating the saved return address (saved EIP) on the stack. To identify the location of the saved EIP on the stack, I used debugging techniques, specifically through the GNU Debugger (GDB). To initially trigger this exploit, the buffer needed to have a size of zero, which was achieved by providing a length of 65535 (unsigned integer wraparound). So, by inputting the command `starti 65535 A0 A1 A2 A3` in GDB, I tested the positioning of the A's on the stack concerning the saved EIP. Through analysis, it was determined that the saved EIP (0x40051e91) was saved at the 28th, 29th, 30th, and 31st bytes from the beginning of the input buffer. Additionally, the address of the shellcode was calculated (unsigned int addr = 0xc0000000 - 8 - strlen(VULN) - 1 - strlen(shellcode) – 1) and determined to be 0xbfffffab or \xAB\xFF\xFF\xBF in little-endian. By crafting an exploit program that utilizes the execve() system call, the vulnerable program was invoked with specific command-line arguments, where the 28th, 29th, 30th, and 31st bytes represented each byte in the address of the shellcode. For instance, using '65535 AB28 FF29 FF30 BF31' as command-line arguments helped to ensure that the address of the shellcode was properly placed in the expected positions on the stack. This careful manipulation of input, buffer size, stack layout, and shellcode placement allowed the exploit to successfully overwrite the saved return address on the stack with the address of the shellcode, effectively redirecting the program's control flow and enabling execution of the injected shellcode ('l33t').

Challenge 7	
1. Explain the Vulnerability	The 7.c program exhibits a buffer overflow vulnerability, primarily stemming from insufficient bounds checking during string copying operations. The vulnerability is situated in lines 17-19, where the `strcpy` function is employed to copy command-line arguments into fixed-size character arrays designated for `username`, `password`, and `hostname`. The critical issue arises due to the absence of validation on the length of the command-line arguments (`argv[1]`, `argv[2]`, and `argv[3]`) against the predefined buffer sizes. In the event that any of these arguments surpass the allocated 64-character limit, a buffer overflow occurs. The `strcpy` function, lacking checks or limitations on the number of characters copied, can potentially overwrite adjacent memory regions.
2. Explain Your Exploit	The exploitation of the buffer overflow vulnerability in 7.c necessitates a comprehensive understanding of the program's memory layout, including the locations of the shellcode and the saved EIP. Therefore, I crafted a malicious input with the intent to trigger the overflow, manipulating the program's control flow. To execute this exploit, an intricate program was devised, leveraging the execve() system call. The primary objective was to substitute the saved EIP with the address of the shellcode, thereby initiating its execution. Employing a technique reminiscent of a previous level, the shellcode's address was calculated using the formula: unsigned int addr = 0xc0000000 - 8 - strlen(VULN) - 1 - strlen(shellcode) – 1, yielding 0xbfffffab or \xAB\xFF\xFF\xBF (little-endian format). During debugging in GDB, the vulnerable program exhibited a segmentation fault when provided with three 64-byte input strings (AAAA..., BBBB..., CCCC...). To circumvent this challenge, rather than specifying distinct values for the third argument (CCCC...), I adopted a pragmatic approach. I lazily repeated the use of the shellcode address (\xAB\xFF\xFF\xBF...). This simplified the exploit, heightening the probability of overwriting the saved EIP with the shellcode address. When the vulnerable program reached the point of EIP restoration, it inadvertently redirected to the shellcode address. Consequently, the program executed the shellcode/l33t command.

Challenge 8	
1. Explain the Vulnerability	The source code in 8.c presents a program that revolves around the execution of user-provided commands. However, a comprehensive analysis of the source code revealed a critical vulnerability — a potential format string vulnerability. This vulnerability arises from the absence of sanitation of user inputs, specifically those incorporated within format specifiers in functions such as `fprintf` in the `sudoexec` function. This posed a significant security risk, potentially exposing the program to exploitation by adversaries who could manipulate format strings to induce memory corruption. Hence, the locus of this vulnerability is primarily identified within the `sudoexec` function, where the user-provided command interfaces with the `system` function, and during the logging process, where the command is embedded within a format string for writing to the *sudolog* file in line 20.

2. Explain Your Exploit	In the experimental exploration of the program, its execution from the home directory was necessitated, employing an absolute path (e.g., ./../../var/challenge/level8/8.c). Subsequently, the fprintf function output, directed to the "sudolog" file in the home directory, became a focal point for analysis. To probe the program's response to input, the command "AAAA" was issued, with an anticipation of observing the hexadecimal representation '41414141' logged in the "sudolog" file. Employing direct parameter access, the dollar sign qualifier (%N$x) facilitated navigation to the nth parameter. Following a pattern misalignment observation, the input was adjusted to "BBAAAA," resulting in a successful alignment with the anticipated steps and the identification of an offset of 68. Subsequently, leveraging a format string vulnerability, the Write4primitives technique was employed to inscribe four arbitrary bytes within the process's address space, enabling the compromise of the program's execution flow and initiation of the l33t shellcode. The retloc of the Global Offset Table (GOT) entry for fclose() was ascertained using the **readelf -r 8 \| grep -w fclose** command, yielding the value 0x0804a010. The choice of fclose as the GOT entry was strategic, given that It is called after the source of the vulnerability in the fprintf function found at line 20. With the retloc in possession, the subsequent step involved fragmenting the write for the return address at four incremental retlocs through Write4primitives, incrementing each rightmost byte successively (\x10\xa0\x04\x08, \x11\xa0\x04\x08, \x12\xa0\x04\x08, \x13\xa0\x04\x08). The process of determining the precise address of the shellcode involved leveraging established methods from prior levels, leading to the computation of the address as \xAB\xFF\xFF\xBF. With this crucial address in hand, the subsequent step involved the meticulous construction of a format string, considering the bytes written to the "sudolog" file post-execution. Taking into account the additional bytes written after execution (specifically, 2133:<space>BB, which included two characters to realign the stack), alongside the four different incremental retlocs, the total offset amounted to 24 bytes. To align the calculated offset with the desired memory locations, a series of arithmetic computations ensued. Subtracting 24 from the initial offset \xAB, the resulting value of 147 was utilized to formulate the first format string: %147u%68$n, where 68 denotes a specific location in the memory. Subsequent calculations involved

	further subtractions: subtracting \xAB from \xFF resulted in 84 (%84u%69$n, where 69 is another memory location), subtracting \xFF from \xFF resulted in 0 (%70$n, representing yet another memory location), and subtracting \xFF from \xBF resulted in -64. To circumvent a segmentation fault, 256 was added to the negative width field, yielding the positive complement value 192 (%192u%71$n, with 71 designating a distinct memory location). These calculations, seamlessly integrated into the constructed exploit program, culminated in the formulation of a strategically crafted string. The resulting program, when executed with the provided arguments, invoked the shellcode, ultimately executing the l33t command and demonstrating the successful exploitation of the format string vulnerability.

Challenge 9	
1. Explain the Vulnerability	Upon initial analysis, the program appears to be primarily designed for number base conversion, featuring string manipulation functions. However, a critical concern arises with an optional separator derived from the environment, a potential vulnerability accentuated in the `find_separator` function in line 90. In this function, a fixed-size buffer of 256 bytes is used to store an environment variable substring. Notably, the main function's buffer spans 384 bytes, but it leverages the separator variable, without proper consideration for the bytes present within it. A significant risk surfaces when a malicious actor fills the main buffer to capacity and then injects an excessive amount of bytes into the separator environment variable, potentially causing a stack-based buffer overflow. This overflow scenario creates a potential path for exploitation, underscoring the importance of explicit bounds checking to mitigate associated risks.
2. Explain Your Exploit	In navigating this traditional buffer overflow challenge, my initial approach was to calculate the shellcode address using a technique proven effective in previous exploits. The identified shellcode address materialized as 0xbfffffab. With this crucial information, my objective was to strategically overwrite the saved return address with the shellcode address. To accomplish this, I delved into locating the saved return address on the stack and gauging its distance from the buffer. Leveraging the debugging capabilities of GDB, I meticulously observed my input adhering to the program's defined rules. Specifically, I provided the requisite two integers as argv[1] and argv[2], coupled with a sequence of 1's in argv[3]. The subsequent examination of the stack during this debug session unveiled my input of '1's prominently displayed. The saved EIP was pinpointed as 0x40051e91 at the address 0xbffff5ec. Analyzing the positioning of my '1's (31313131 on address line 0xbffff44c), I determined they were precisely 412 bytes away from the saved EIP. Equipped with this critical insight, I devised a strategy to exploit the buffer limitations of 384 bytes in the main function with my meticulously crafted argv[3]. Allocating 32 bytes to the separator variable, I strategically embedded the shellcode address in the last four bytes of this variable, formatted in little-endian (\xAB\xFF\xFF\xBF). Executing this crafted input through an exploit program proved successful, allowing me to advance to the next level by executing the l33t command embedded in the shellcode.

Challenge 10	
1. Explain the Vulnerability	Upon examination, the program appears to facilitate client-server communication. In particular, a potential buffer overflow caught my attention during a closer look at the manage_tcp_client function, starting on line 27 in the code. Within this function, data is read into the buffer from standard input (read(0, &buffer[index], 1)), with a one-byte-at-a-time approach in a loop. The loop terminates when a newline character (0x0a) is encountered. The vulnerability lies in the absence of checks regarding the size of the input data relative to the buffer's capacity (defined as CHUNK_SIZE). Should the input data exceed the available space in the buffer, the program may overwrite adjacent memory, leading to the execution of the coveted l33t command.
2. Explain Your Exploit	*Before I delve into the specifics of how I believe my exploit succeeded, I was in the dark about my challenge status until I casually checked the scoreboards and discovered that I had already passed the challenge several hours earlier. Throughout that period, I experimented with numerous methods, leaving me uncertain about the exact approach that led to my success—similar to my experience with Challenge 2. Nevertheless, I'll outline what I believe to be the correct method that resulted in my progression to the next level.* Embarking on my strategy, my first goal was to pinpoint the shellcode's address, a pursuit that led to the discovery of the target address: 0xbfffffa9. Armed with this crucial detail, I temporarily reserved it for subsequent steps. Initiating the debugging phase, I utilized the command *starti -t tcp -p <port>*. Subsequently, I intercepted the fork method, pivotal for the program's continuous execution. To facilitate this, I dispatched a sequence of A's back to the server by executing *nc 127.0.0.1 <port>* in a separate terminal, creating a pivotal catchpoint. Transitioning into the child process by configuring the *follow-fork-mode* to child, I delved into the program's functions using info functions. My scrutiny pinpointed the *read@plt* as the source affecting the stack. Setting a breakpoint at its address 0x08048730 initiated a new process. Shifting to *inferior 2* and invoking *layout asm*, I meticulously filled the stack with my A's. This intentional step provided insight into

	the input buffer's start at address 0xbffdeadc, with the saved EIP (0x0804925e) residing at 0xbffeead8. Possessed with this data, I calculated the offset, totaling 65548 bytes. Equipped with the calculated offset, the shellcode's address, and the shellcode itself, I intricately constructed an injection vector designed to replace the saved EIP's address with the shellcode's location. This meticulously crafted vector took the form of NOP sleds, followed by the shellcode and the address of the shellcode, concluding with a new-line character—an essential element to gracefully terminate the for-loop within the *manage_tcp_client* function, preventing the reading of additional bytes. [*NOP sleds + shellcode + address of shellcode + new-line character]* After intricately computing the payload, I crafted a streamlined shell script that seamlessly orchestrates the entire process. This script adeptly carries out three pivotal steps: 1) runs the program in the background, 2) executes the exploit (constructed using Python), and 3) channels its output to the initiated server, utilizing netcat. Consequently, the script's execution unfolded flawlessly, directing the client to initiate the l33t shellcode.