Buffer Overflow | CSC435: Computer Security - Lab 2

Instructor: Dr. Ayman Tajeddine

Group Members:

Reem Marji – 202100881

Husen Shaito – 202105705

Nadine Jaafar – 202106172

Sadek Meheiddine – 202104058

Part 1: General Questions

1. Buffer overflow is a software vulnerability where an application receives more data than it can handle, causing it to overflow into adjacent memory areas. This can lead to memory corruption, crashes, or even malicious code execution. Buffer overflow attacks are common and can be used to gain unauthorized access or escalate privileges. The buffer overflow attack targets the part of the memory that is already allocated to the process of the vulnerable object. This attack is done by overwriting some aspects of the stack or heap memory that should not be impacted, such as the frame pointer and the return address.

   In specific, the attack is done by inputting objects that are bigger in size than the parameter register allocated in the memory. To illustrate, we will utilize the following lines of code:

   ```
   char username[100];
   char password[100];
   gets(username);
   ```

   In this case, the stack is vulnerable to an attack. If we input a username of size greater than 99 chars, the password register will be impacted. This is because in a stack, the buffers that are saving data for us are stored consecutively by order. As such, when we give the username buffer more input than its allocated capacity, the excess input will be placed in the password buffer. This is called stack overflow attack that is used to impact other variables or address info. This example can be prevented by using fgets instead of gets function because fgets only takes the first 99-character input.

   On the other hand, let us take this example:

   ```
   String = (char *) malloc(sizeof(char)*size);
   strcpy(String, chars);
   ```

   In this case, the heap may be attacked. If the size of “chars” is greater than “size”, the heap will be affected by the excess values of chars. This buffer overflow could also be applied on global variables.

2. At the beginning of the execution, the main function is called to be run, the value of the address is pushed into the stack. As the code runs, variables are pushed into heap memory if they were dynamically allocated, or to the stack elsewise.

If the main function calls another function, the value of its address and the parameters are also pushed into stack. In addition, in order to get back to the function when needed, the frame pointer is also pushed into the stack. As a result, any new variables in this function, will be placed in the stack of this new corresponding frame. After the function is done executing and returned to the main, the new frame that was allocated to it will be disregarded. After that the main continues executing normally.

3. There are several functions in C that are vulnerable to buffer overflow. The following are examples of them:

   a. **strcpy()**: This function is used to copy a string from one location to another. If the source string is larger than the destination buffer, it can overflow the buffer.
      As a safe alternative, we can implement strncpy(). This function allows you to specify the maximum number of characters to copy.

   b. **strcat()**: This function is used to concatenate two strings. If the destination buffer is not large enough to hold the concatenated string, it can overflow.
      As a safe alternative, we can implement strncat() or sprintf(). strncat() allows you to specify the maximum number of characters to concatenate, while sprintf() can be used to concatenate strings with more control over the formatting.

   c. **sprintf()**: This function is used to format a string and store it in a buffer. If the buffer size is not large enough to hold the formatted string, it can overflow.
      As a safe alternative, we can implement snprintf() or asprintf(). snprintf() allows you to specify the maximum number of characters to print and automatically adds a null-terminator, while asprintf() allocates memory dynamically to store the formatted string.

   d. **gets()**: This function reads a string from standard input and stores it in a buffer. It does not perform any bounds checking, so if the input string is larger than the buffer, it can overflow.
      As mentioned earlier in the document, we can implement fgets() instead of gets(). fgets() reads a string from standard input and stores it in a buffer while ensuring that the buffer does not overflow. You must specify the maximum number of characters to read.

   e. **scanf()** : This function reads input from standard input and stores it in a buffer. If the input is larger than the buffer, it can overflow.

Instead of scanf(), use fgets() followed by sscanf(). fgets() reads the input string into a buffer, and sscanf() parses the string into the desired format, making it more secure than scanf(), which does not perform bounds checking.

f. **fread()**: This function reads data from a file and stores it in a buffer. If the amount of data read is larger than the buffer, it can overflow.
As a safe alternative, we can implement fgets() or fread() followed by bounds checking. fgets() can be used to read a line of text from a file while ensuring that the buffer does not overflow. Alternatively, you can use fread() but make sure to check the number of bytes read against the buffer size to avoid buffer overflows.

4. Brief explanations of each of the defense mechanisms against buffer overflow attacks:

   a. Compile-Time Defenses: These are mechanisms that are implemented at the time of program compilation. The goal of these defenses is to harden the program against potential buffer overflow attacks before it is even executed. Some examples of compile-time defenses include stack canaries, which are values inserted into the stack to detect buffer overflows, and address space layout randomization (ASLR), which randomizes the memory layout of the program to make it harder for attackers to find the location of vulnerable code.

   b. Run-Time Defenses: These are mechanisms that are implemented at runtime while the program is executing. The goal of these defenses is to detect and prevent buffer overflow attacks while the program is running. Some examples of run-time defenses include bounds checking, which verifies that array indices are within their expected range, and memory protection, which prevents the execution of code in certain areas of memory.

   c. Aim to Harden New Programs: These defenses are implemented with the aim of hardening newly created programs against potential buffer overflow attacks. This is typically achieved by following secure coding practices and using secure coding techniques, such as avoiding the use of vulnerable functions like gets() and using safer alternatives like fgets(), as well as properly sizing and validating input data.

   d. Aim to Detect and Abort Attacks in Existing Programs: These defenses are implemented with the aim of detecting and aborting buffer overflow attacks in existing programs. This can be achieved through the use of intrusion detection systems (IDS) and intrusion prevention systems (IPS) that can monitor network traffic and detect potential attacks, or through the use of anti-malware software that can detect and prevent the execution of malicious code. Additionally, some operating systems provide

protection mechanisms like data execution prevention (DEP) and address space layout randomization (ASLR) that can help prevent buffer overflow attacks.

e. The string "%x:%x:%s" is a standard format string that can be used with the printf() or scanf() family of functions in C to format and print or scan input/output.

In this particular format string, %x is a conversion specifier that indicates that an unsigned hexadecimal integer value should be printed or scanned, and %s is a conversion specifier that indicates that a string of characters should be printed or scanned.

The issue lies as follows. If this format string is used with printf() or fprintf() functions, it could potentially cause a memory leak if the %s conversion specifier is used with a pointer to a dynamically allocated memory buffer that has not been freed. If the %s specifier is used without specifying the maximum number of characters to print, it could keep printing the contents of the buffer until it encounters a null-terminator character, which could be outside the allocated memory buffer. This could lead to a memory leak, where the program continues to consume memory indefinitely.

For example, we can take a look at the following lines of code:

```
char* buff = malloc(256);
strcpy(buff, "An example of possible memory leak");
printf("%x:%x:%s", 2468, 1357, buff);
```

In this example, the malloc() function is used to allocate a memory buffer of size 256 and the strcpy() function is used to copy a string into the buffer. The printf() function is then used with the format string "%x:%x:%s" to print the values of two integer variables and the string stored in the buffer. If the buffer is not freed before the program terminates, it can cause a memory leak.
This is can be simply avoided by calling the free() function on the buffer variable, the memory that was allocated by the malloc() function is released and can be reused by the program or the operating system.

Note: the ":" character is a literal character that is printed as is.

Part 2: Vulnerable C

A:

1. The main function is run and calls the function foo(); This function initiates a char variable called ch and a char buffer of size 5 called 'buffer'. It then prompts the user to "Say Something:" and input a char to be saved in the ch. It keeps taking input, and terminates when the user enters '\n'(ENTER). Then, char by char, ch is saved into 'buffer'. After that the function prints the 'buffer' and returns to the main function that will then be finished.
2. The variables of this functions are stored in the stack memory.
3. The type of attack that this program is vulnerable to is stack overflow.
4. The vulnerability can be observed in the fifth line: "`buffer[i++] =ch;`"
5. The buffer overflow is achieved by inputting more than 5 characters.
   When we attempted running the code with an input greater than 5, the error "Stack Smashing Detected: Terminated" was observed. In the description of this error found online, it is said that this is displayed when there is a chance of buffer overflow in the current input. We strongly believe that this warning is possible to attain due to an implementation of canary values.

6. We can update the condition of the while loop as follows:

```
while((ch=getchar()) !='\n' && i<5))
```

B:

1. These lines of code will first save a character array named buf, and two integers i and len. Moreover, the code will read bytes of size(len) from the file descriptor fd and save them in len. Then the code will allocate memory of size len for buf. Finally, the code will read from fd string of size len and save them in buf.
2. The integers and pointer are saved in the stack, whereas the array of chars is allocated in the heap.
3. The type of the attack is buffer overflow.
4. The vulnerability can be observed in the following lines:

```
read(fd,&len, sizeof(len));
buf = malloc(len);
```

5. Buffer overflow could be achieved here because the value of len is not checked, whereby the value read from the file and saved in len could be of negative value. Applying malloc with a negative value parameter will achieve buffer overflow.
6. We can add a condition that validates that the value of "len" is greater than 0:

```
read(fd, &len, sizeof(len));
if(len>0){
buf = malloc(len);
read(fd,buf,len);}
```

C:

1. The code will merely print the value given through the command line.
2. The variables are stored in the arg vector in the stack.
3. This type of attack is a buffer overflow attack by format string.
4. The vulnerability can be observed in the following line of code:

```
printf(argv[1]);
```

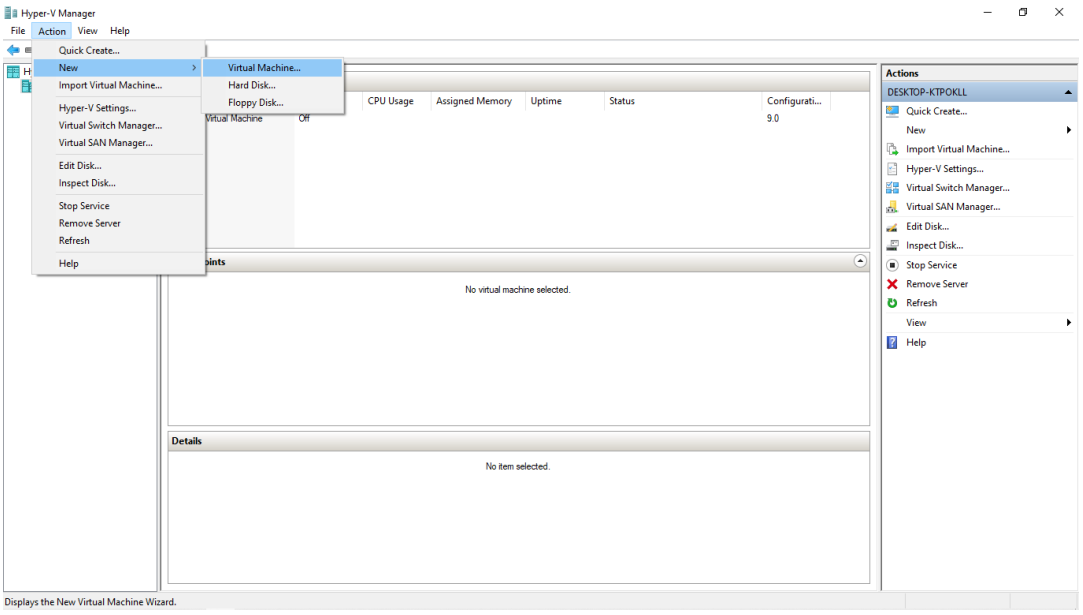
5. The overflow can be easily achieved by giving the value of argv[1] as "%s%s", this will make the program display data presented in the memory.
6. As a mean of prevention, we can replace the vulnerable code by what follows:

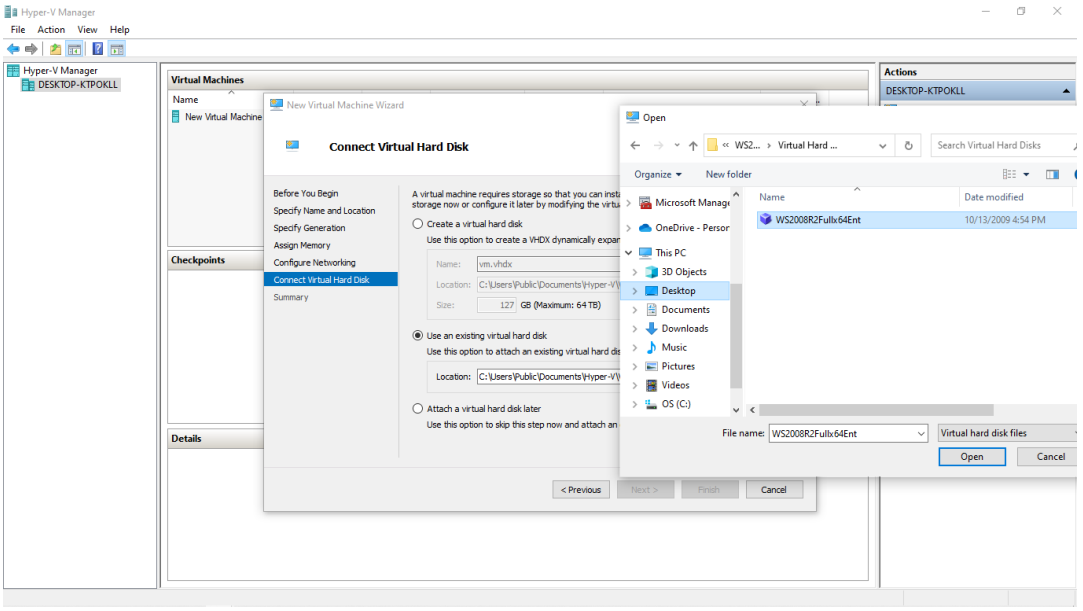
```
printf("%s", argv[1])
```

Part 3: Lab – Vulnerability

- Download the virtual machine of winserver 2008 R2.

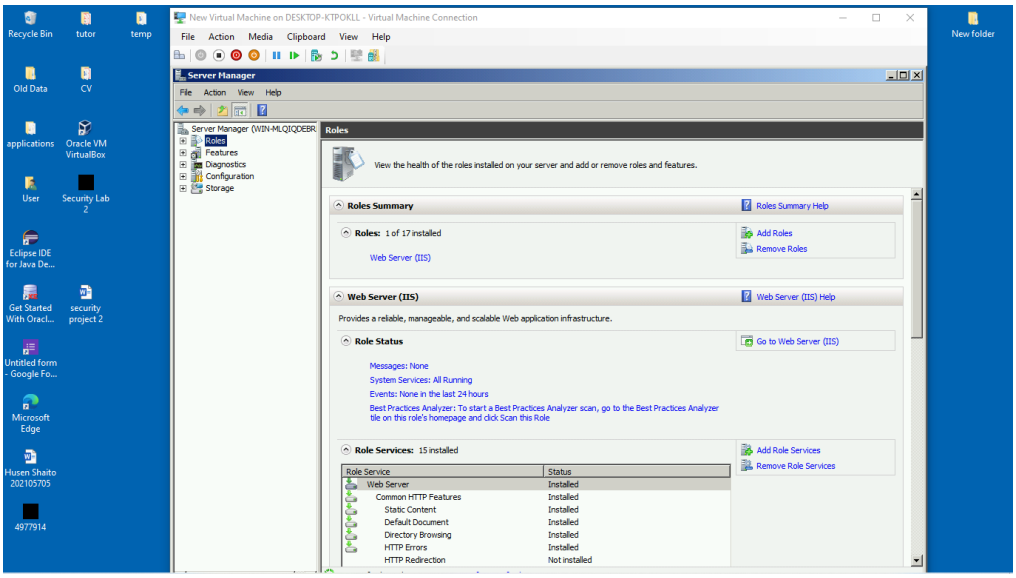
After completing the supplementary information and downloading what is required, our virtual machine was ready for running.





- Enable IIS.

We then enabled IIS accordingly.



- Research a vulnerability related to Buffer overflow and DoS.
- What is the name of this vulnerability? How does it work? Explain.

After thorough research, we discovered that the name of the vulnerability is CVE-2015-1635. It is also known as "HTTP.sys Remote Code Execution Vulnerability" or "HTTP.sys RCE" for short. The vulnerability was first discovered in March 2015 by a security researcher at Google's Project Zero team.

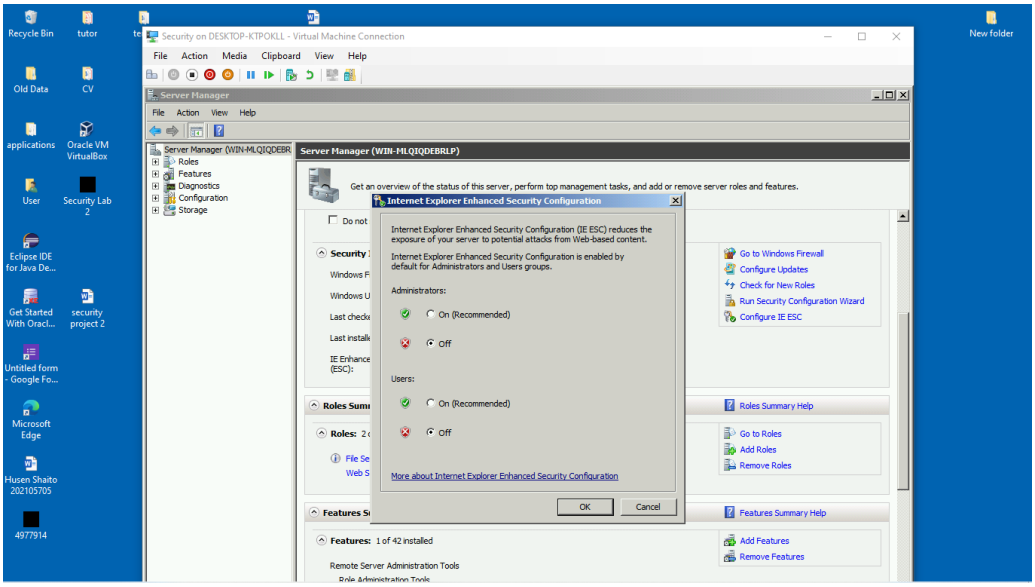
The vulnerability was related to a buffer overflow in the HTTP.sys driver, which is responsible for handling HTTP requests in Microsoft IIS web servers. The buffer overflow was caused by a flaw in the way HTTP.sys parsed specially crafted HTTP requests. An attacker can exploit the vulnerability by sending a specially crafted HTTP request to a vulnerable server. The request includes a "Range" header that specifies a large number of bytes, but the final byte range is set to a value that is much smaller than the total number of bytes specified. This causes the HTTP.sys driver to allocate a buffer that is much larger than necessary to handle the request.

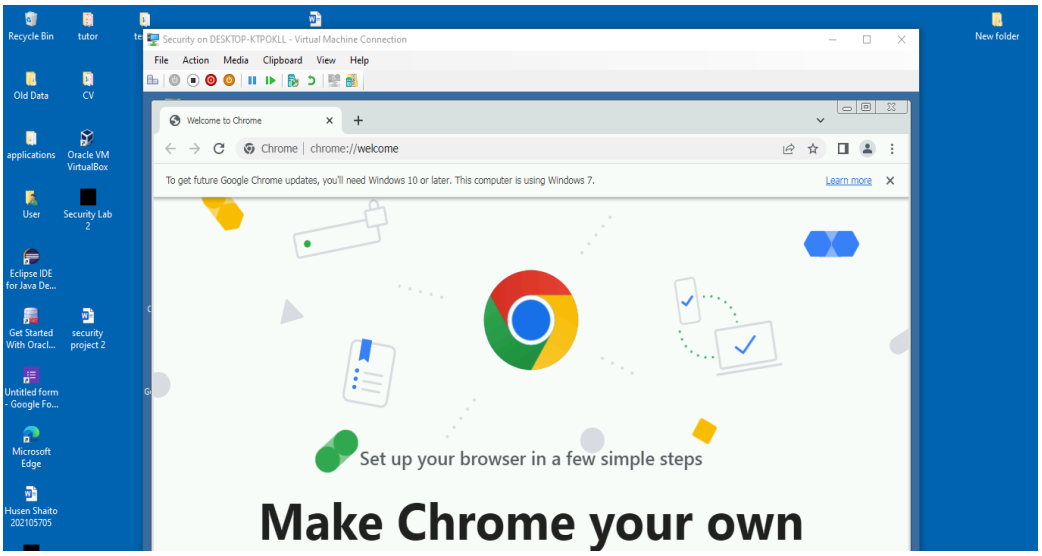
This vulnerability could also be used to launch denial-of-service (DoS) attacks against web servers, as well as to gain unauthorized access to sensitive data or to take control of the affected system.

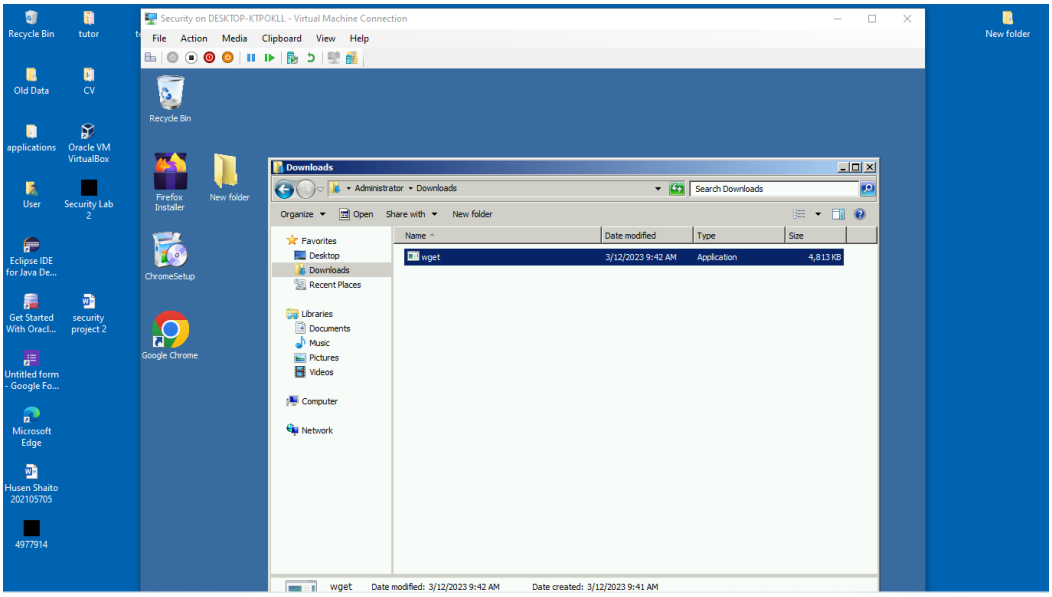
The focal idea is that the end of the range given is exactly the highest value that can fit into an unsigned long (64-bit) value, which is 18446744073709551615. (2 to the power of 64, minus 1). The initial value of the range seems to just be marginally significant; if it is 0, an unpatched server will respond with "Requested Range Not Satisfiable," rather than completely crashing. However, any value larger than 0 and below the file's size will cause an entire server meltdown.

- Use “wget” or “curl” or just your “browser”, forge the request and perform this attack on the server.

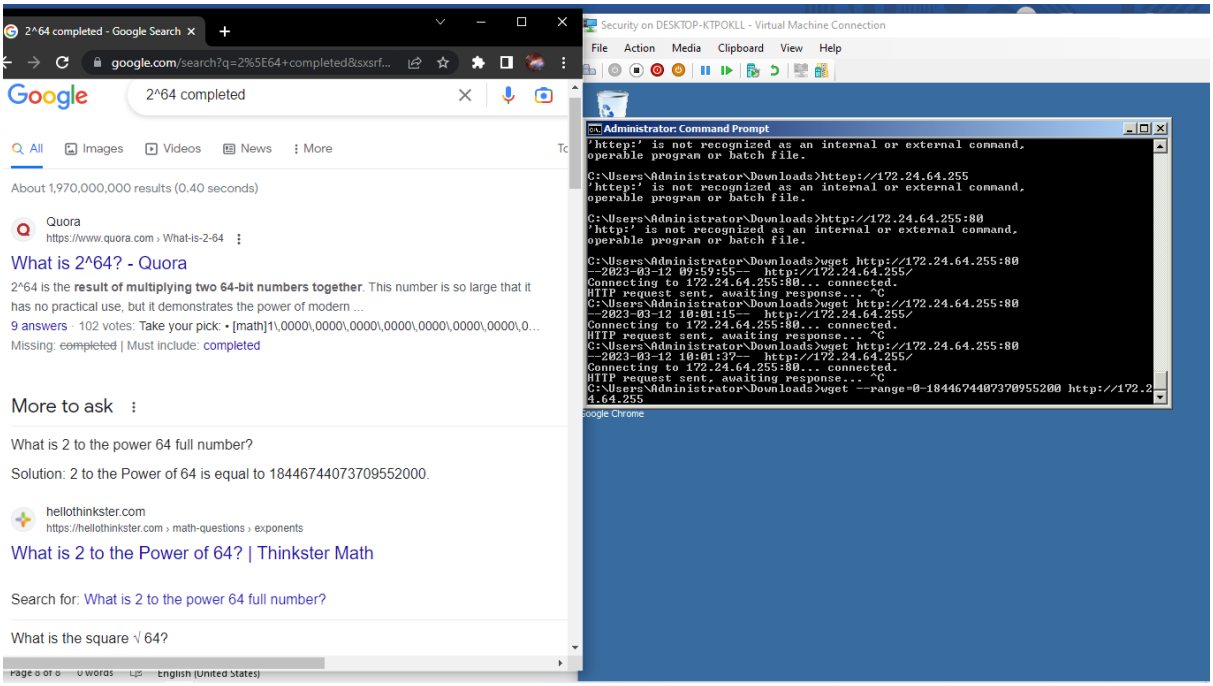
We encountered several issues when downloading wget. As such, we turned off the Explorer Enhanced Security Configuration and downloaded chrome in order to install wget from it.

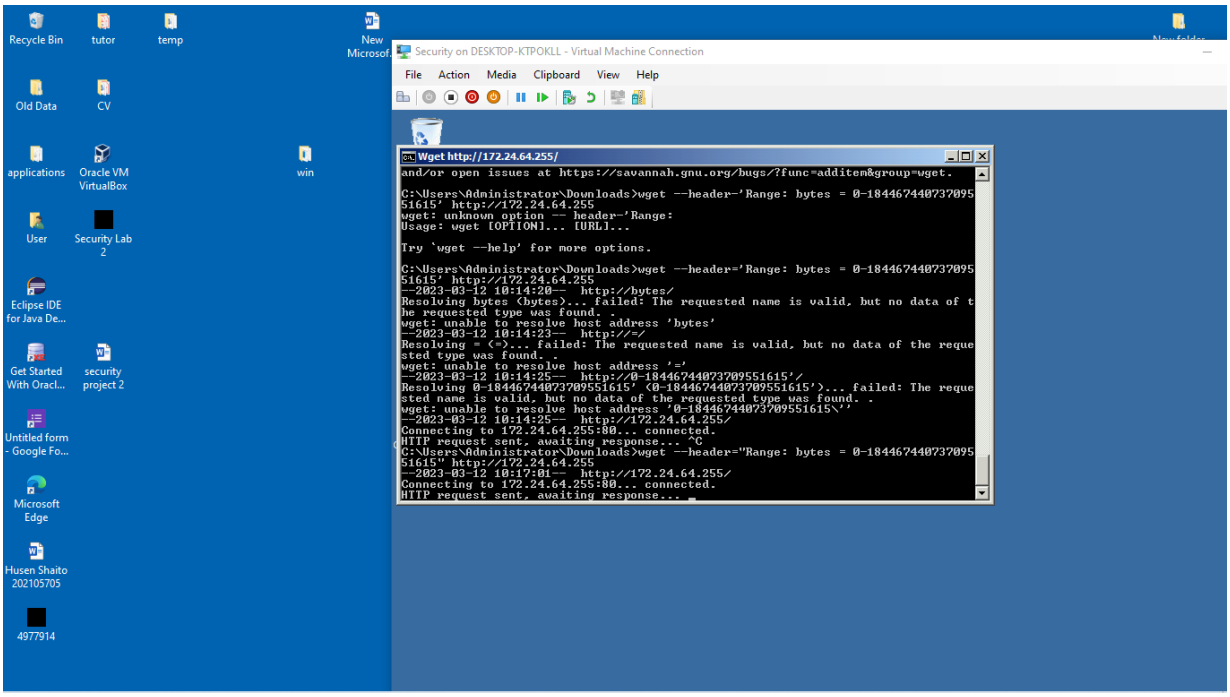




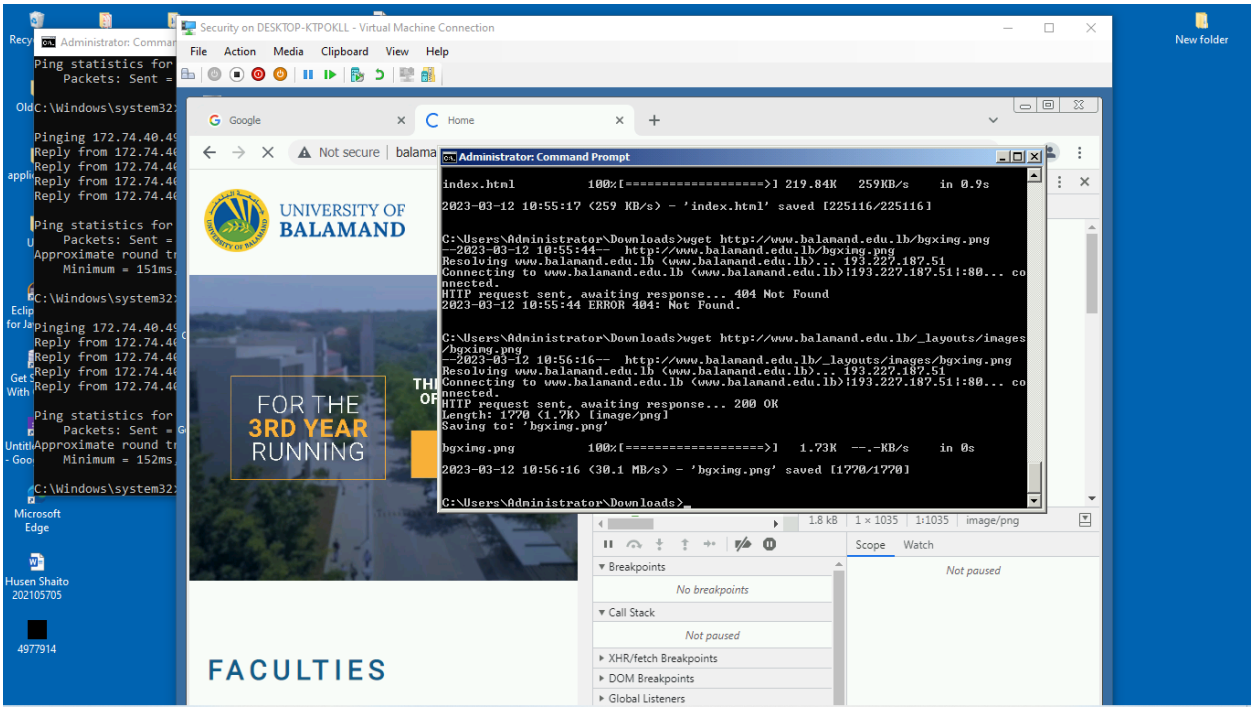


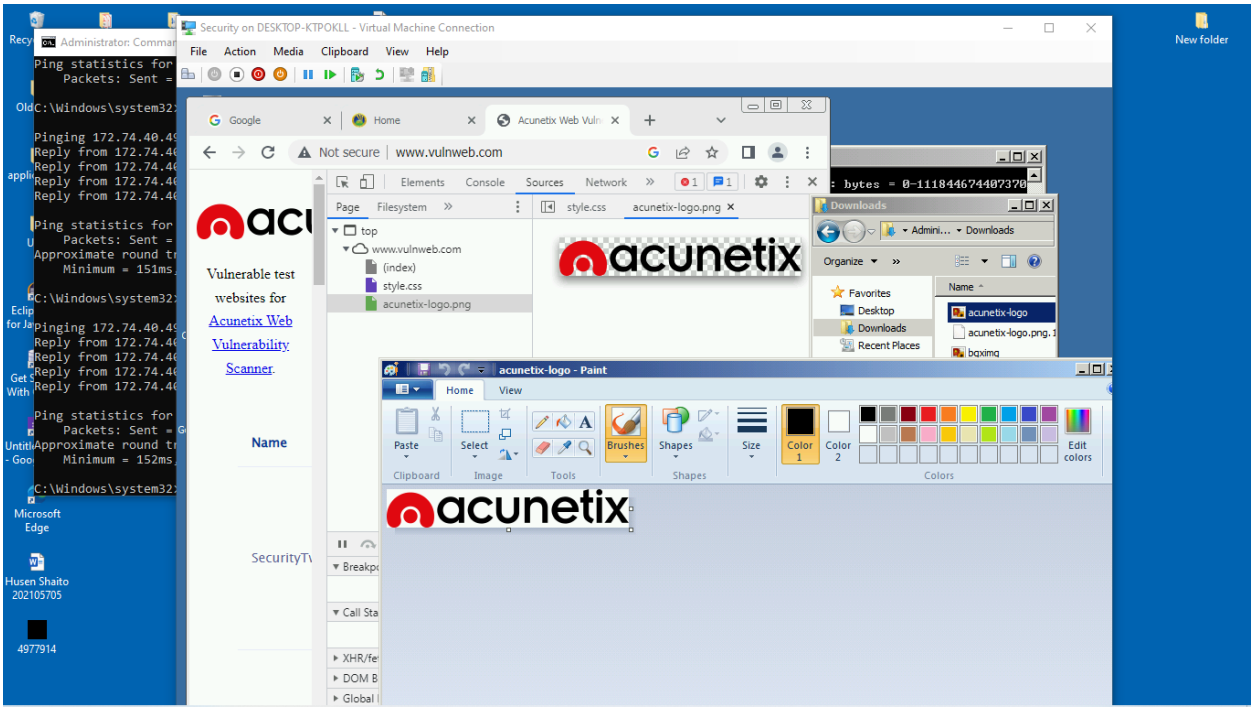
We attempted forging the request on multiple IIS websites that we created with certain IP addresses. Nonetheless, we didn't achieve our end goal.





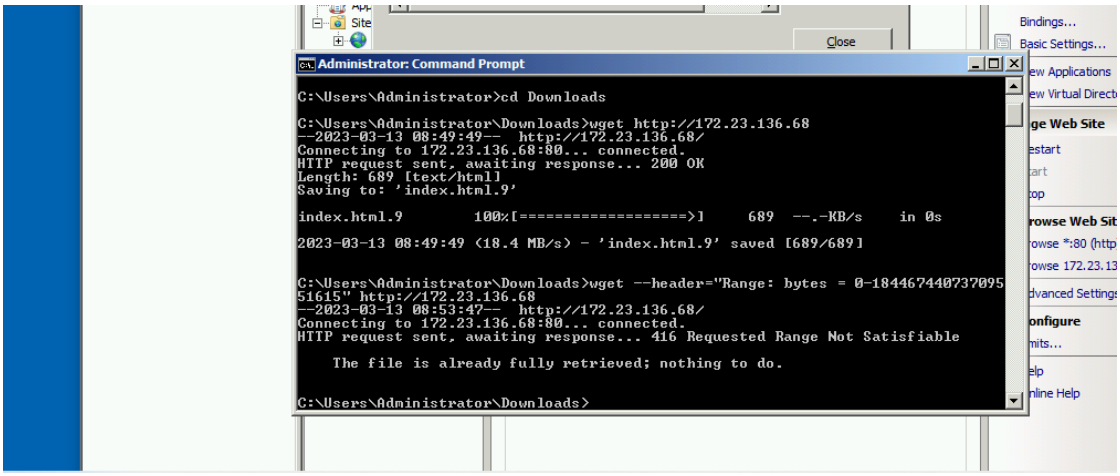
In attempt to test what were we doing inaccurately, we tried the request on different online http websites such as the Balamand university and acunatix. However, the error was still not achievable despite being able to download files and images related to the website.



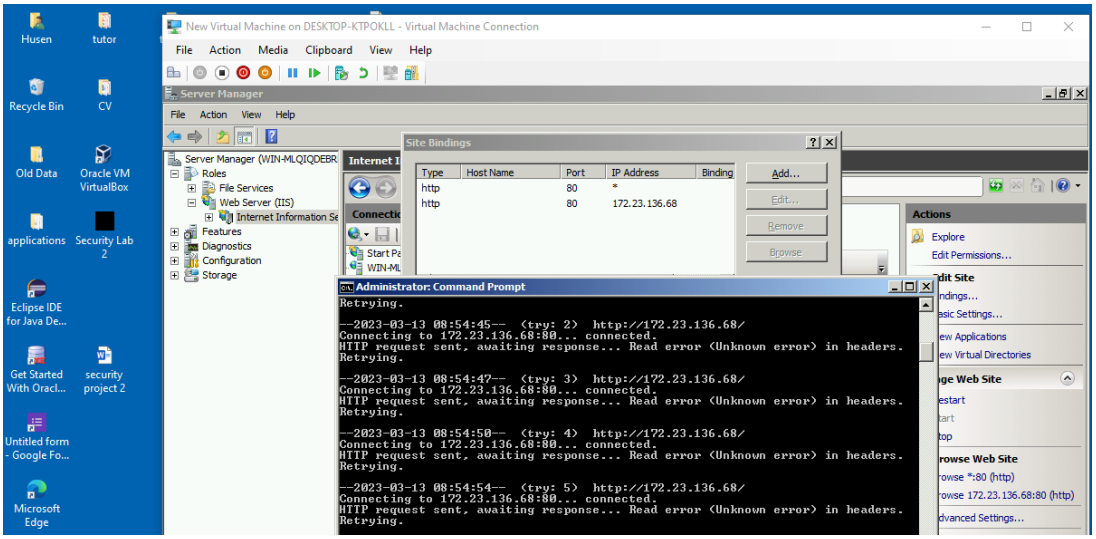


After some trial and error, we figured we needed to make an IIS website with the IP address of our machine. Thus, we first configured the IP address, and we discovered it to be 172.23.136.68 .

When asking a request with 0 to 2^64 bytes the following range error got displayed.

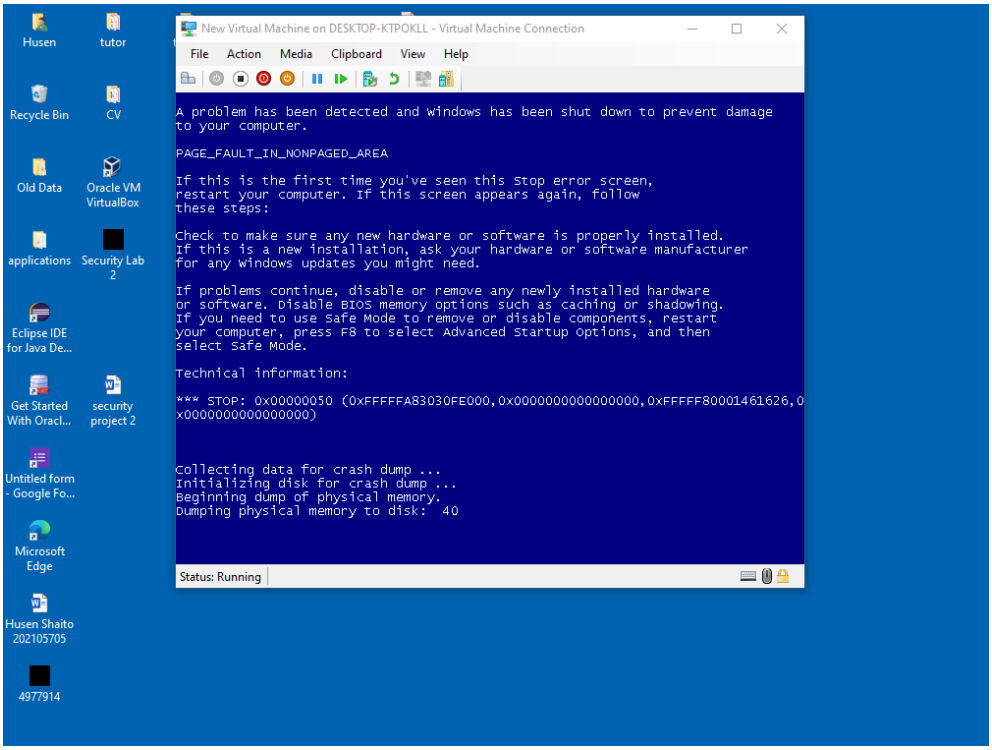


Now when we attempted the request with a range of 18-2^64, we observed the following error. Based on what we have read, the system was supposed to crash upon such input.



We thus attempted to retrieve a file from the website, specifically a default welcome.png image, by requesting the following command.

Following this attempt, the system crashed and displayed “The Blue Screen of Death”.



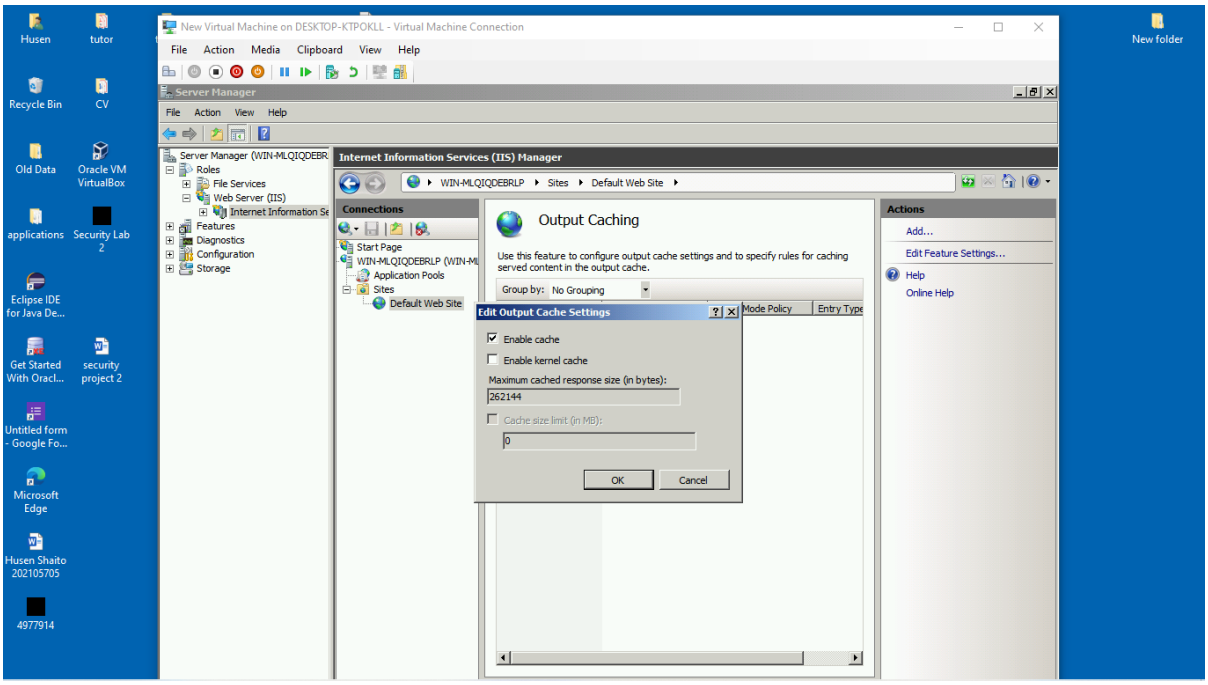
- What is the result of this attack on the server?

From what we observed, the result of this attack on the server is that it caused it to crash. It is a Denial of Service (DoS) attack, achieved by buffer overflow. The vulnerability is related to a buffer overflow in the HTTP.sys driver. As such, when HTTP.sys crashes, the entire computer crashes, blue screens, and becomes unusable until you totally restart it.

- How can we remediate the vulnerability without patching?

Upon research, we discovered that while patching is the most effective way to remediate this vulnerability, there are other methods we could implement to reduce the risk of exploitation.

We were able to remediate the vulnerability without patching by disabling IIS Kernel Cashing, as seen below. This is because disabling IIS kernel caching may reduce the amount of traffic that passes through HTTP.sys and therefore reduce the attack surface for the vulnerability.



References:

1. https://www.beyondsecurity.com/scan-pentest-network-for-microsoft-windows-http-sys-code-execution-vulnerability
2. https://nvd.nist.gov/vuln/detail/CVE-2015-1635
3. https://www.softwire.com/insights/explaining-a-security-vulnerability-the-iis-range-header-attack-cve-2015-1635/