



Lebanese American University

School of Arts and Sciences

Department of Computer Science and Mathematics

CSC435 (Computer Security)

Lab #02 (Buffer Overflow)

A Report Done By

Ahmad Hussein

ahmad.hussein03@lau.edu

Mahmoud Joumaa

mahmoud.joumaa@lau.edu

And presented to Dr. Ayman Tajeddine

October 2023

Table of Contents

I. Research.....	3
II. Analyzing C Code.....	5
Code A	5
Code B	5
Code C	6
Code D	6
III. Vulnerability	7
Machine MAC and IP Addresses.....	7
1. Setting Up The System	8
2. HTTP.sys Buffer Overflow & DOS Vulnerability	12
IV. Conclusion	16
V. References.....	17

I. Research

1. C and C++ are by far the two languages most commonly associated with buffer overflow attacks. Fortran and Assembly are other examples of languages prone to buffer overflow. These languages allow direct memory access and memory manipulation but lack built-in remedies (such as boundary checks) against buffer overflow.

On the other hand, languages like PERL, Javascript, and C# use built-in safety mechanisms that minimize their vulnerability towards buffer overflow attacks. Interpreted languages like Java or Python are virtually almost immune to such attacks.

A general rule of thumb would be that the lower the language, the more vulnerable it is to buffer overflow attacks.

2. As different operating systems implement different safeguards and techniques to combat security vulnerabilities, such as buffer overflow, a successful attack on an operating system may fail on another. This is highly dependent on the employed architecture as well as the program itself. However, we were not able to pinpoint anything detailed or specific to each operating system that would allow for an objective comparison between Windows, Linux, and MACOS when it comes to buffer overflows.
3. Find in the following table the C functions vulnerable against buffer overflow attacks and their corresponding safe substitutes:

C unsafe functions against buffer overflow	C safe functions against buffer overflow
<i>gets()</i> – reads input until new line or end-of-file is reached	<i>fgets()</i> – reads input until new line or end-of-file is reached or until the desired size is entered
<i>sprintf()</i> – sends formatted output to a string as a pointer	<i>snprintf()</i> – sends formatted output of desired size (excess characters are discarded) to a string as a pointer
<i>strcat()</i> – concatenate two given strings	<i>strncat()</i> – concatenates the first <i>n</i> characters of the source string into the destination string
<i>strcpy()</i> – copies one string into another	<i>strncpy()</i> – copies the first <i>n</i> characters of the source string into the destination string
<i>vsprintf()</i> – stores formatted data from variable argument list to string	<i>snprintf()</i> – sends formatted output of desired size (excess characters are discarded) to a string as a pointer

4. **Stack Canaries** are random integers placed before critical memory locations, such as the stack return pointer. Before executing the return, this value is checked. If it was manipulated in some way, then buffer overflow is detected.

ASLR (Address Space Layout Optimization) is a technique used to combat buffer overflow by randomizing the memory locations of certain critical aspects, such as the base of a stack or a heap. This can also typically be configured for a subset of the entire memory where buffer overflow is more likely to occur.

Data Execution Prevention (DEP) is a mechanism that renders parts of the memory exclusive for data. Executables cannot run in these locations, thus hindering code launches from places they are not supposed to be launching.

5. **AOL Instant Messaging (AIM)**

- a. The vulnerability basically arises from AIM's feature which allows users to invite others to play online games. When parsing the game request, users could overflow the code and consequently take control over the victim's system.
- b. As the attackers could take absolute control over a victim's system, they could execute any arbitrary code they desired. Attackers could basically issue a DOS, install cryptominers, download trojans, or execute any other form of attack they wanted.
- c. No precautions could be taken to prevent this vulnerability because the exploit does not require any previous chat or authorization. The game request could be sent anytime to anyone.
- d. A patch was announced to be issued shortly after the vulnerability was identified (AIM versions higher than 3.5 have addressed this issue). The fix was applied on the AIM servers, but the client software could still exploit something similar using techniques that incorporate a "man in the middle". Another issue is that if the client software could penetrate the filters on AIM servers, it could issue the same type of request.

6. The `%x` string formatter is used to format the output as hexadecimal values.

The `%s` string formatter is used to format the output as a string of characters.

If `%x:%x:%s` is used without specifying the variables for each formatter to format, random memory locations will be considered instead, potentially causing memory leaks since the displayed data may not have been accessible otherwise.

II. Analyzing C Code

Code A

1. A *char* array *input* is first created of size 20. The program then displays “Say something: ” on the screen and uses the *fgets()* method to read the standard user input of the same size as *input* (i.e. 20). Any excess characters are discarded by the *fgets()* function. The *main()* then calls *CopyandDisplay()* which creates a *char* array *buffer* of size 10. *strcpy()* then copies *input* into *buffer* and displays “You said: ” followed by whatever’s stored in *buffer* and then a new line.
2. *input* is stored in the global data memory.
buffer is stored in the *CopyandDisplay()* function stack.
3. This program is vulnerable to buffer overflow that could lead to a crash as a result of segmentation fault error.
4. Line (6).
5. The program attempts copying *input* which could hold as much as 20 characters into *buffer* that is only of size 10. Buffer overflow can be achieved by having *input* contain more characters than *buffer* size (i.e. *input* can contain 11 to 20 characters to achieve buffer overflow).
6. Line (5) could be adjusted to: *char buffer[sizeof(input)]*

Code B

1. The *main()* function simply calls *foo()*.
(Note that the method is expected to return an int but returns nothing. This may be handled by the compiler in more recent versions of C.)
The *foo()* function creates a *char* called *ch* and an array of *chars* called *buffer* of size 5. It then initializes an *int i* to zero. “Say something: ” is then displayed on the screen. The function then keeps on looping to read user input using the *getchar()* method and saves each character into *ch* then saves *ch* into the next index in *buffer*. The loop terminates when a new line (*\n*) is entered. After terminating the loop, the final element in the buffer is set to the terminating null character *\0* and the function displays “You said: ” followed by the contents of the buffer and a new line before returning zero.
2. *ch*, *buffer*, and *i* are stored in the *foo* function stack.
3. This program is vulnerable to buffer overflow that could lead to a crash as a result of segmentation fault error.
4. Line (9).
5. By inputting more than 4 characters, the program attempts to store them all in addition to the character ‘*\0*’ in the buffer which is only of size 5.
6. Adjust line (8) to: *while ((ch = getchar()) != ‘\n’ && i < 5)*

Code C

1. The *main()* function initializes *fake_fd* integer then calls *readAndCopyData()* before returning zero. The *readAndCopyData()* function first initializes an *int len* and reads its size (4 bytes in this case) from the file descriptor and stores it in *len*. The function then allocates enough memory for *len* characters as a pointer *buf*. If *buf* is successfully initialized, *len* of the file descriptor is read and stored in *buf* before deallocating the memory used by *buf* via *free()*.
2. *fake_fd* is stored in the global data memory.
fd, *len*, and *buf* are stored in the *readAndCopyData()* function stack.
3. The program is vulnerable to a buffer overflow that could lead to memory leak.
4. Line (10).
5. If *len* is greater than the actual size of the file descriptor, data beyond the file descriptor would be read into *buf* consequently enabling access to data that may not be available otherwise.
6. The following could be a viable substitute of the vulnerable code:

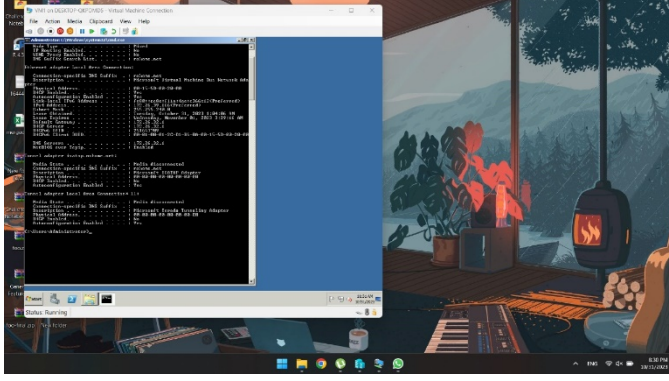
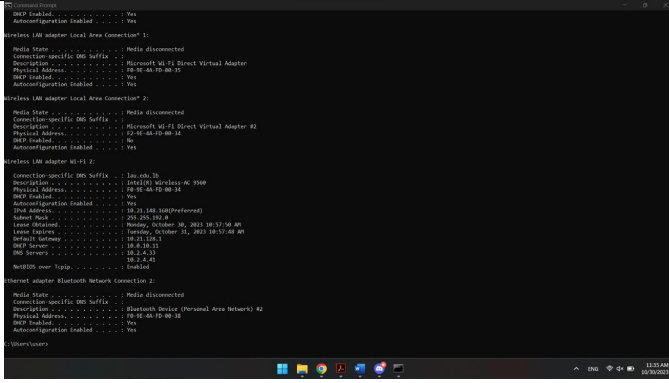
```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
int main() {
    int fd = open("test.txt", O_RDONLY);
    off_t len;
    len = lseek(fd, 0, SEEK_END);
    lseek(fd, 0, SEEK_SET);
    char* buff = malloc(len+1);
    read(fd, buff, len);
    printf("%s", buff);
    return 0;
}
```

Code D

1. The *main()* checks if there are at least two arguments passed as input. If that is the case, it passes the second argument to the *printInput()* function before returning zero. The *printInput()* function then simply prints the char array passed as an argument.
2. *argc*, *argv*, and *input* are stored in the stack.
3. This program is vulnerable to a buffer overflow attack that may cause a memory leak.
4. Line (4).
5. If the input is a formatter (such as *%s* or *%x*) without any variables to format, certain memory locations, that would otherwise be inaccessible, will be formatted and displayed on the screen.
6. Line (4) can be adjusted to: *printf("%s", input);*

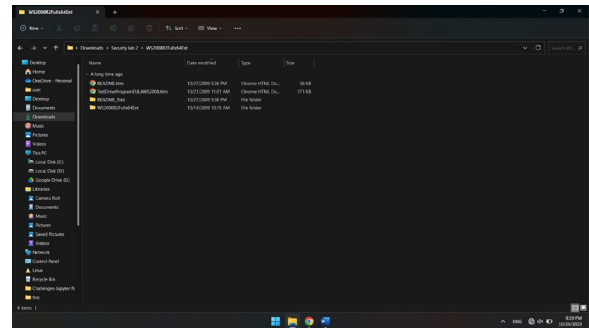
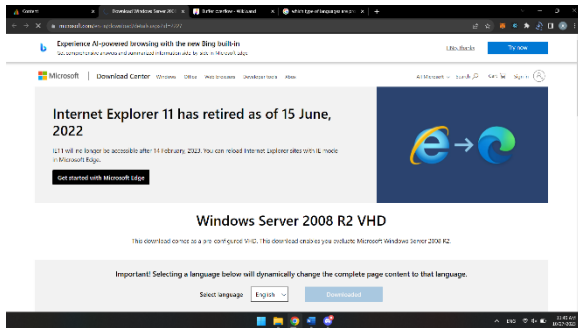
III. Vulnerability

Machine MAC and IP Addresses

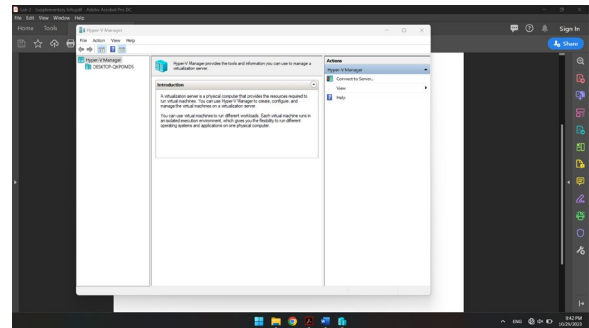
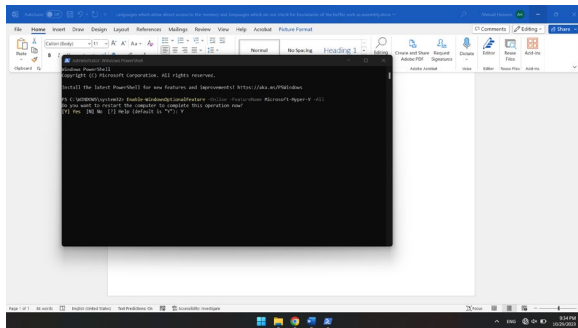
	<p>Virtual Machine</p> <p>MAC Address: 00-15-5D-00-20-00</p> <p>IP Address: 172.26.39.166</p>
	<p>Host Machine</p> <p>MAC Address: F0-9E-4A-FD-00-34</p> <p>IP Address: 10.21.148.160</p>

1. Setting Up The System

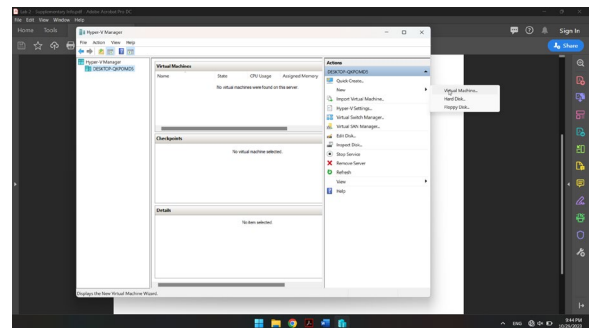
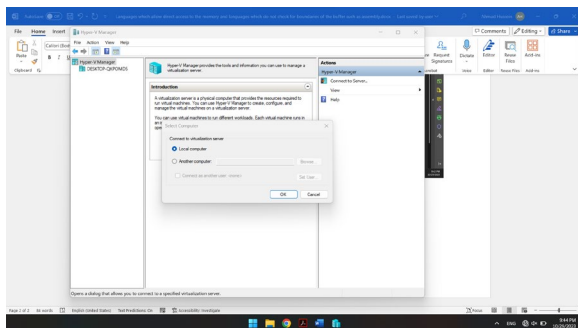
First, we installed Windows 2008 R2 as shown in the following two screenshots:

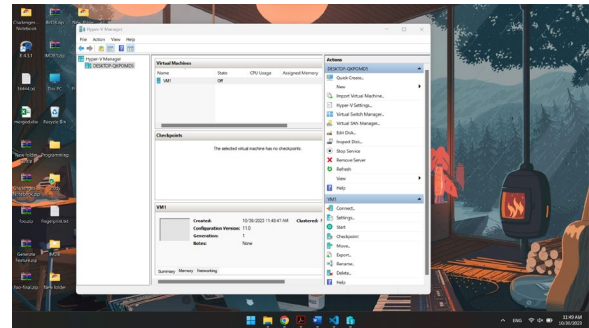
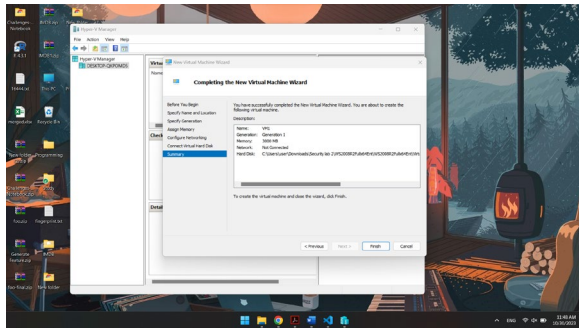
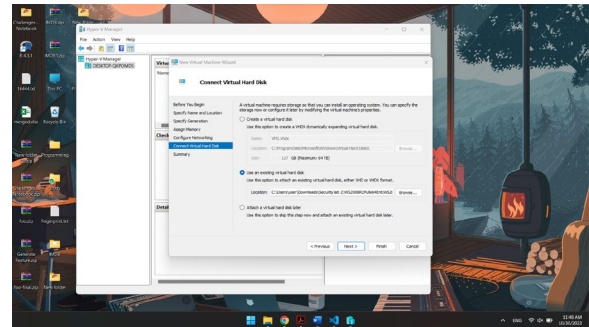
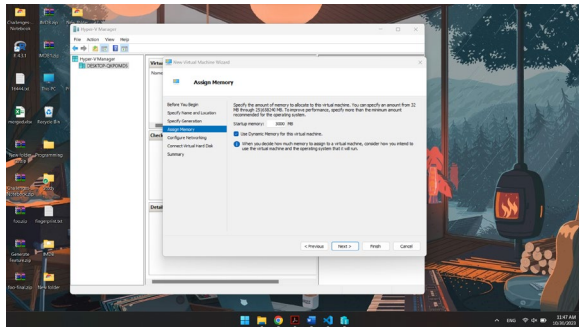
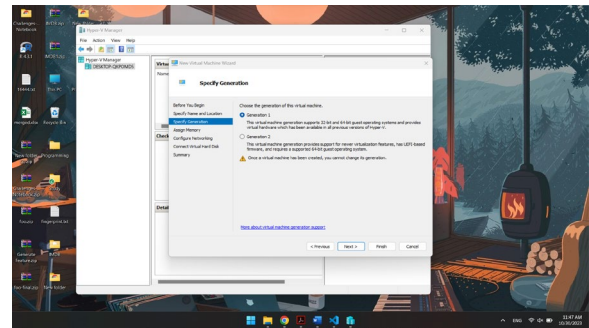
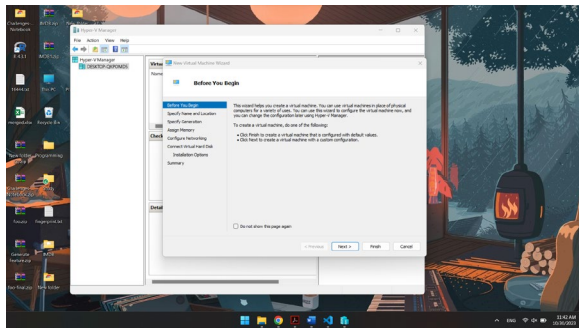


We then activated Hyper-V in our windows by running the command `Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Hyper-V -All` in powershell as administrator.

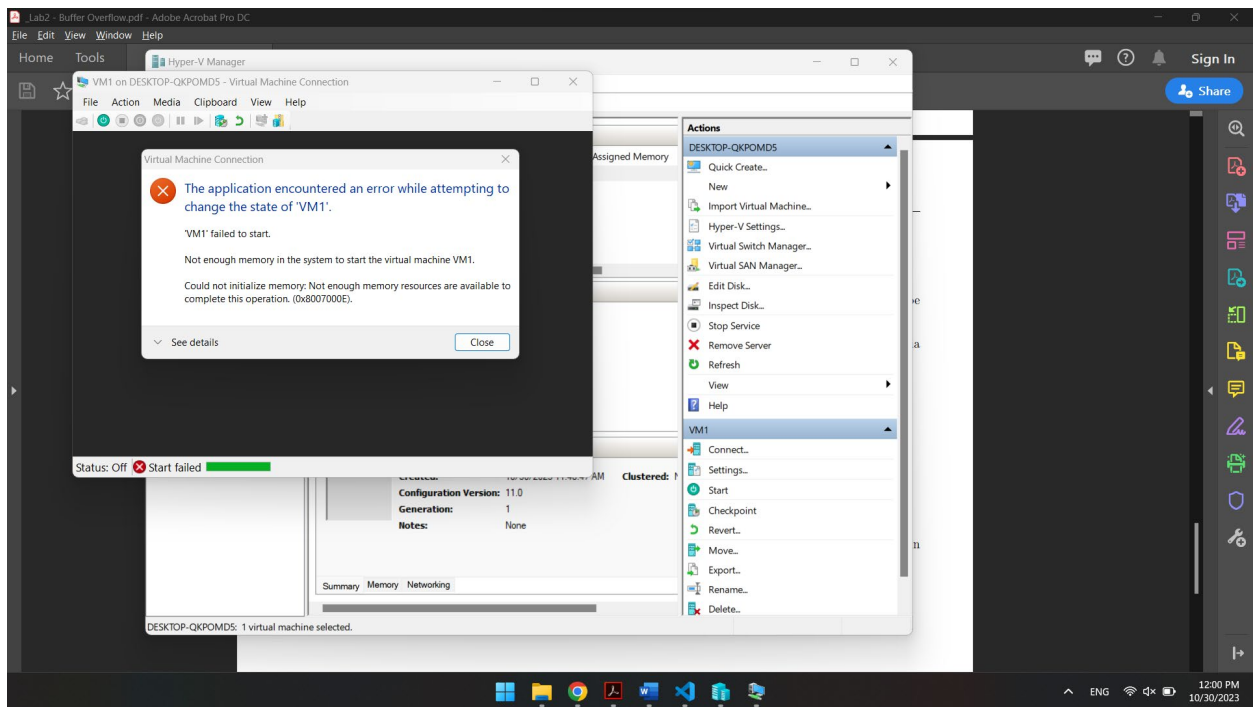


We then continued to open the Hyper-V manager and connect it to the server by choosing “Local Computer” and set up the virtual machine (i.e. specify name, startup memory, and virtual hard disk):

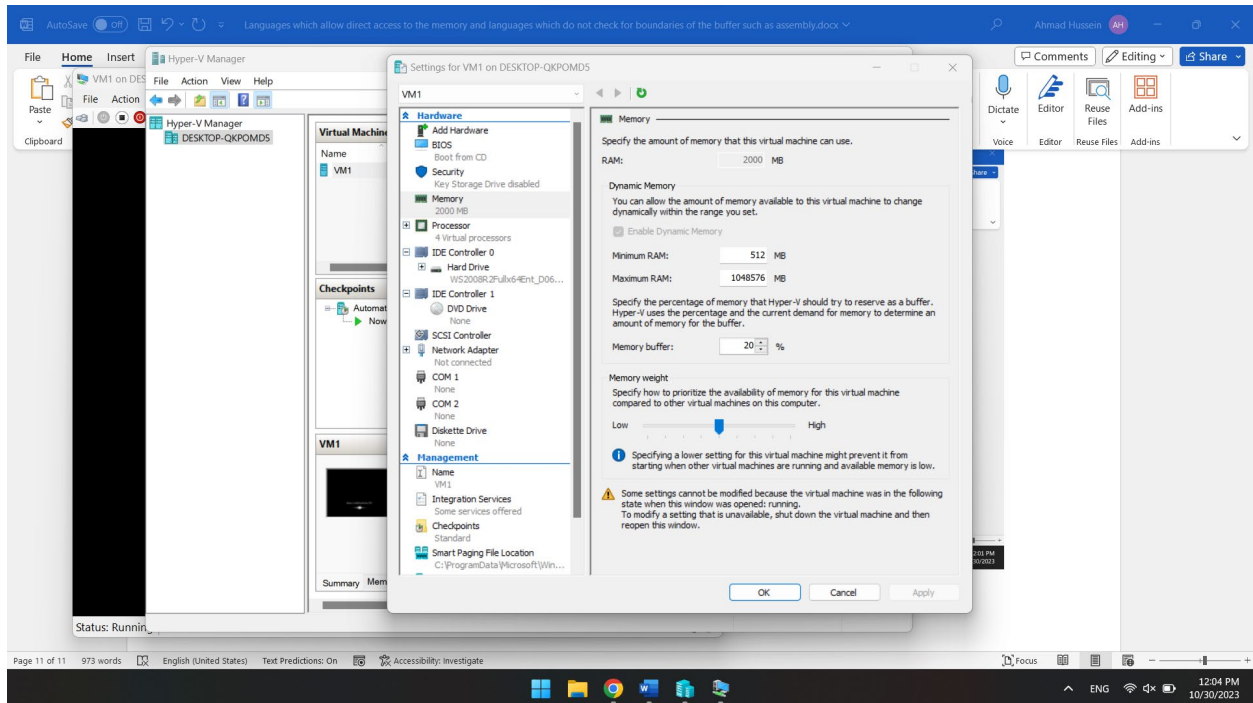




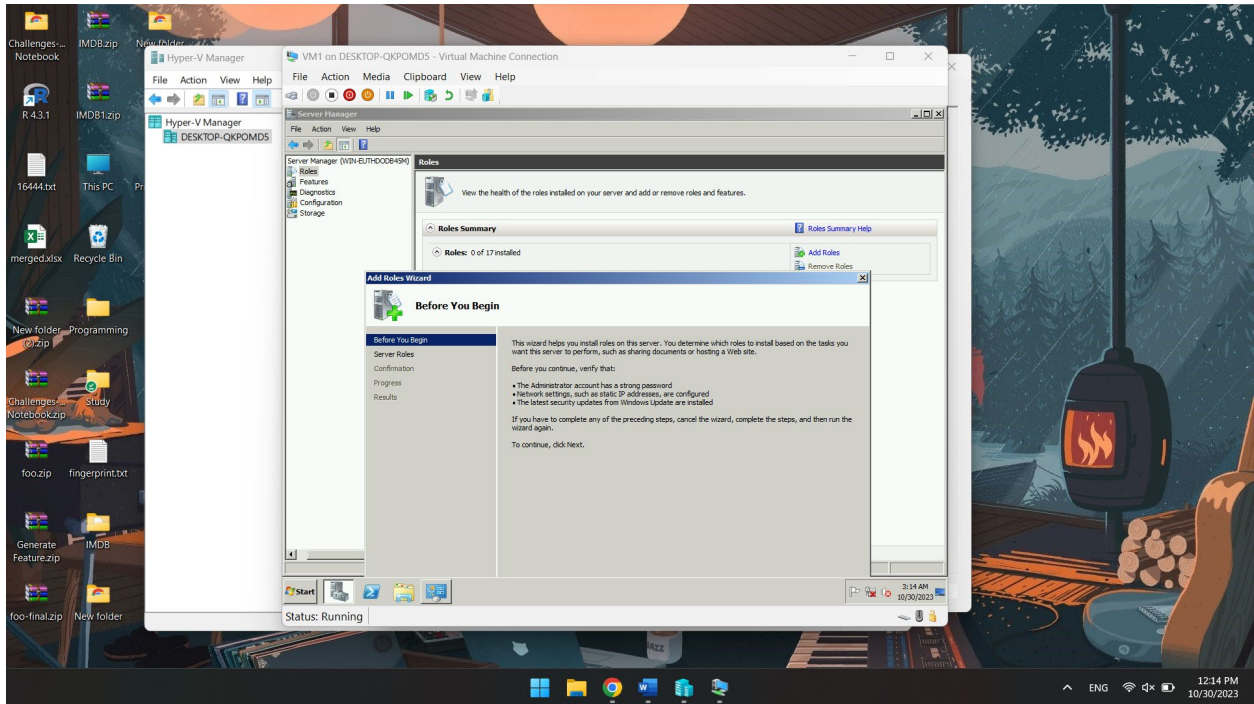
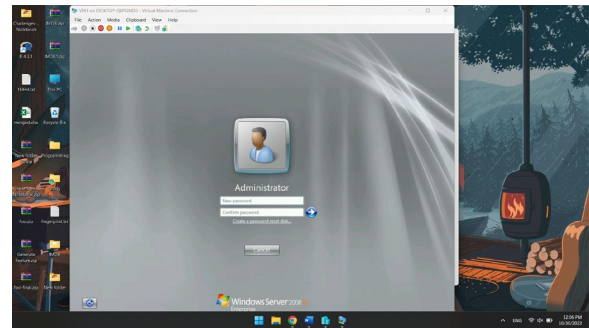
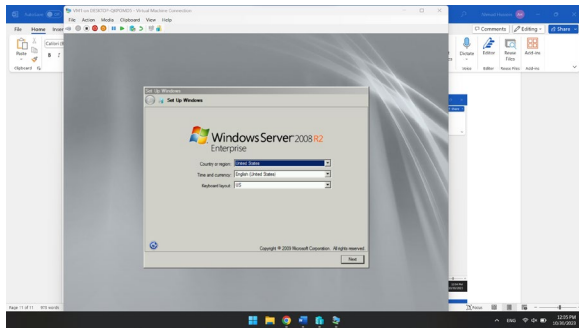
However, the following error popped up when we tried to run the virtual machine:



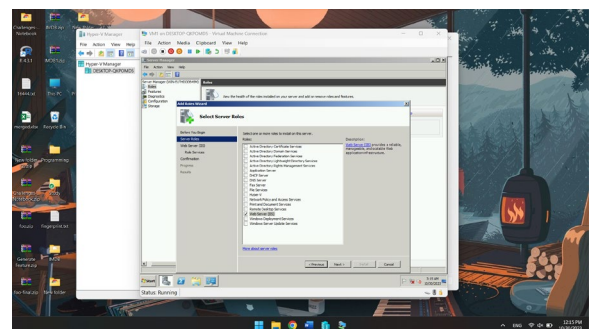
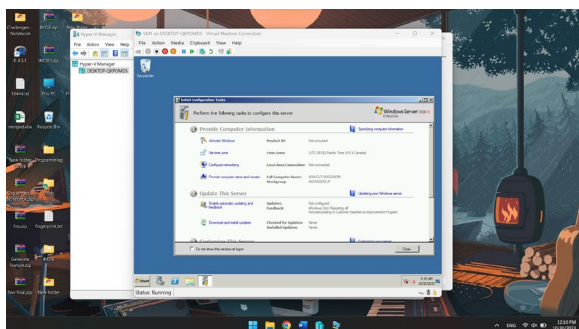
We then decreased the specified RAM to 2000MB as 3000MB were not available (free) at the time of solving this lab:

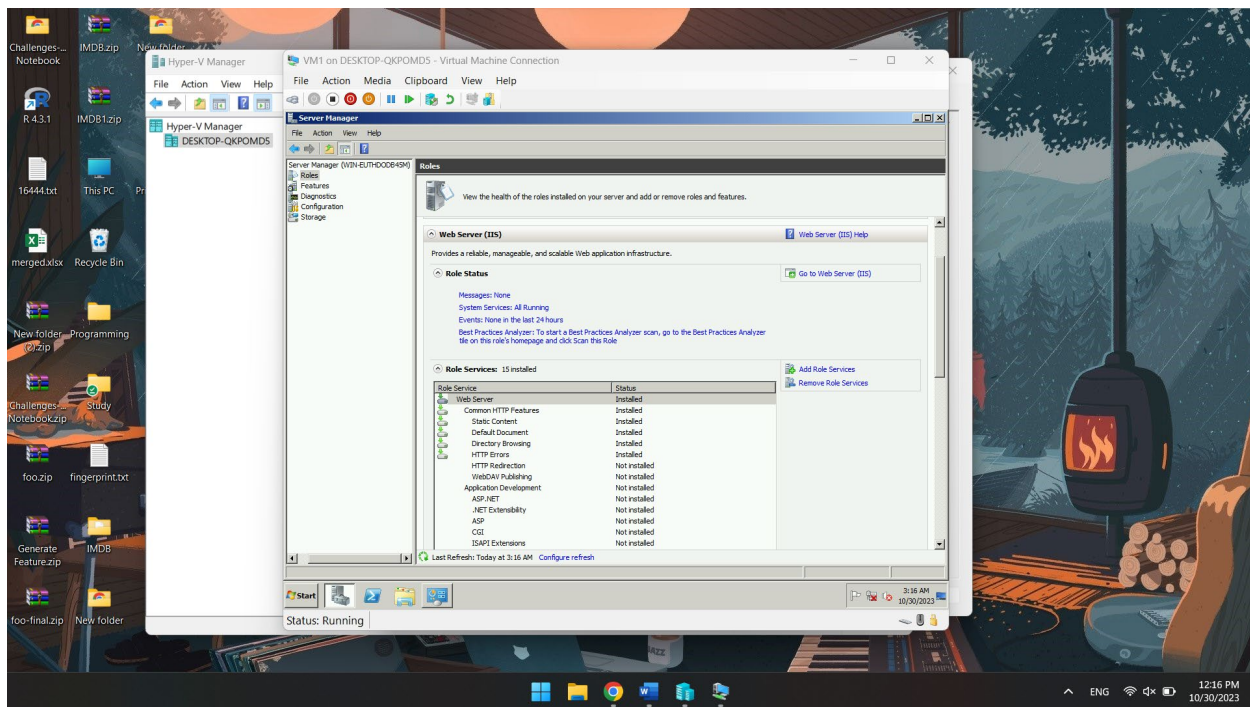


The Virtual Machine then started normally as expected and we were prompted to create a password for the administrator account of the Windows Server 2008 R2:



We then opted to enable IIS by navigating to the server manager as follows:

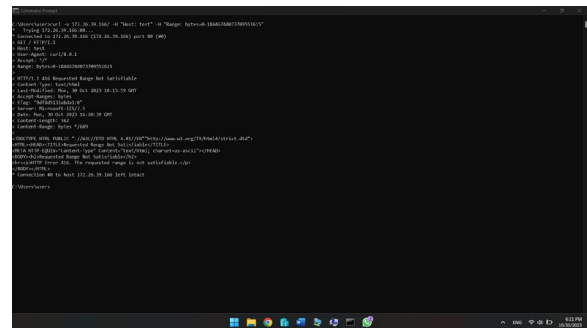
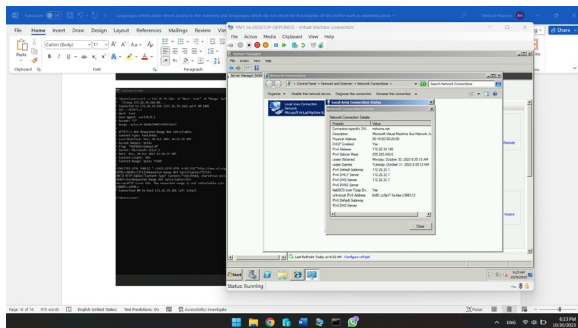




2. HTTP.sys Buffer Overflow & DOS Vulnerability

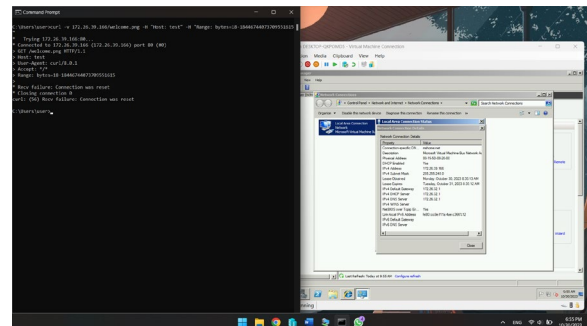
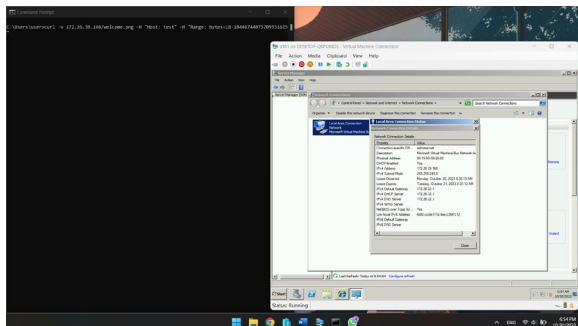
The name of the vulnerability is “*HTTP.sys Remote Code Execution*”, more technically known as *CVE-2015-1635*. This vulnerability allowed attackers, through carefully crafted HTTP requests, to most commonly DOS the vulnerable windows system, but more generally, execute arbitrary code on the victim’s system. The crafted HTTP request consists of requesting the *welcome.png* file from the server while specifying the byte range which exploits that when the server receives a byte range header, and in order to compute the size of the range, the program adds 1 to the upper bound of the range value. Thus, when the upper bound is the maximum unsigned long value, adding 1 to it would overflow the value to return back to 0 which crashes the system (the lower bound would then be greater than the upper bound).

In an attempt to simulate the DOS attack on the Windows Server 2008 R2, we first retrieved the server's IP address and attempted the *curl* command shown in the below screenshot to test whether this windows server really suffers from the CVE-2015-1635 vulnerability. The byte range is from 0 to 18446744073709551615 (which represents the maximum value of an unsigned long).

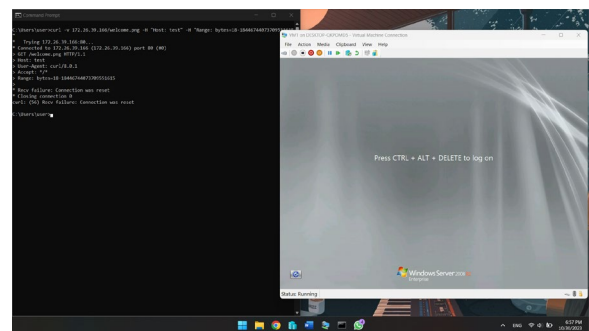
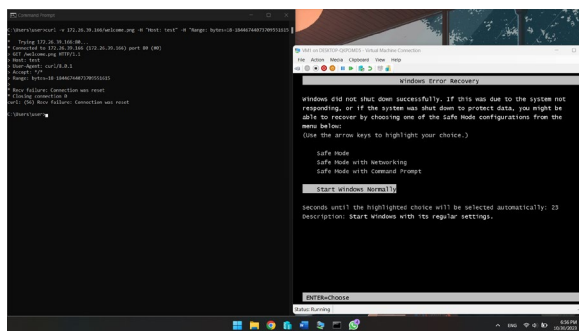
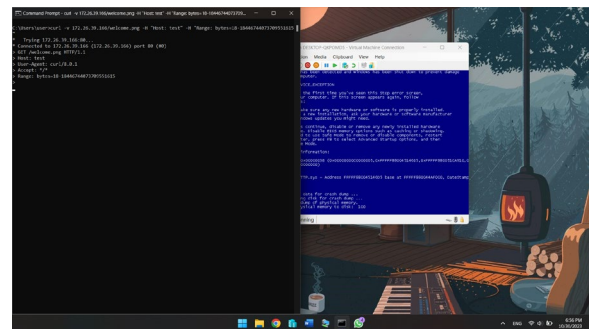
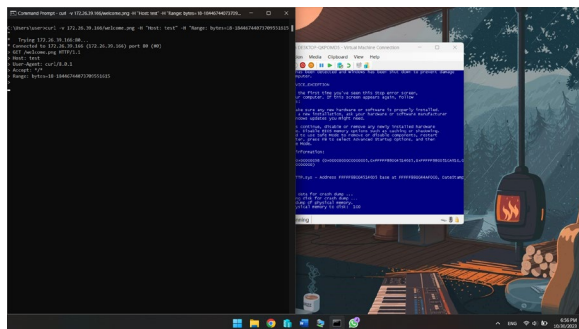
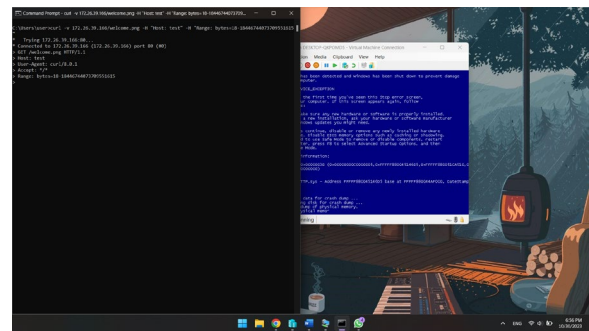
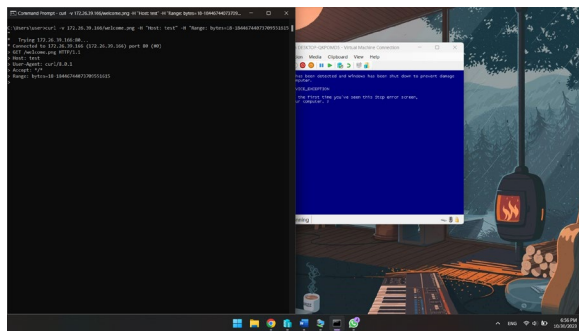


The “Requested Range Not Satisfiable” output confirms that Windows Server 2008 R2 is vulnerable.

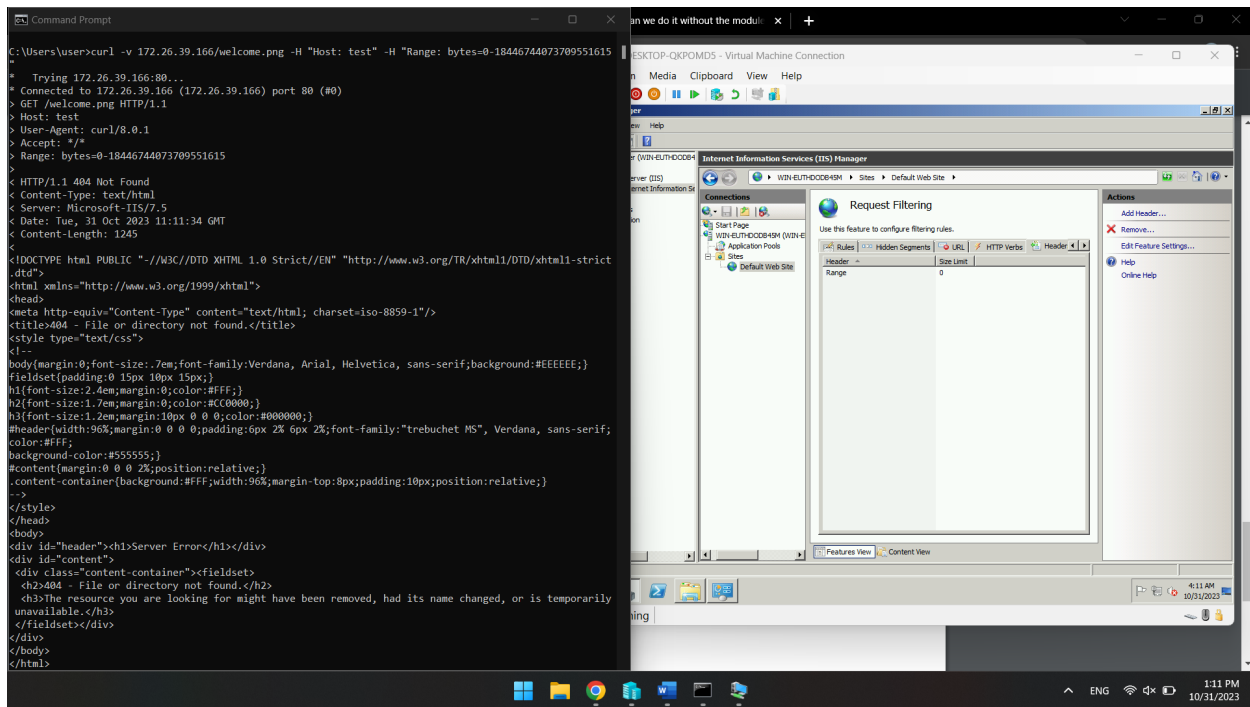
Now that we have confirmed that the server is vulnerable, we attempted to DOS the server by issuing the same command but by changing the byte range (screenshot on the left) from (0 → 18446744073709551615) to (18 → 18446744073709551615). The command did not DOS the server from the first attempt. The following output was displayed (screenshot to the right):



However, after a couple of failed trails, the command did in fact invoke a crash on the server side, thus resulting in denial of service (DOS), as shown in the below screenshots:



We attempted to remediate the vulnerability (without patching) by preventing the range in the request header to have the value of 18446744073709551615 (max unsigned long value). However, this was unsuccessful as we did not find a way to achieve that. Therefore, we opted to reject any request which might have the byte range header by creating a request filter and preventing the range header of having more than 0 bytes. Below is a screenshot illustrating the resulting output:



This vulnerability was, however, patched by Microsoft in April 2015 under the name MS15-034. This patch changed the way HTTP stack handles requests.

IV. Conclusion

Throughout this lab, we both worked together on all aspects of the report. We have both done our own research and aggregated our results together.

The practical part of this lab (part 3 – vulnerability) was solved using one laptop only (MAC & IP of which presented previously in the report). The main challenge we dealt with when tackling this lab is the limited, or rather specific, resources that we could use when researching. Not all articles were as detailed as we would have expected them to be. The most fun we had this lab was issuing commands and actually causing the server to crash, especially after a few failed attempts. It was definitely interesting to provoke and witness an actual DOS attack after just reading about it.

V. References

Languages Prone to Buffer Overflow Attacks

<https://www.imperva.com/learn/application-security/buffer-overflow/#:~:text=current%20runtime%20operations,-.What%20Programming%20Languages%20are%20More%20Vulnerable%3F,in%20C%20and%20C%2B%2B.>

<https://www.fortinet.com/resources/cyberglossary/buffer-overflow>

C unsafe and safe functions against buffer overflow

gets() and fgets()		https://www.geeksforgeeks.org/fgets-gets-c-language/	
sprintf()	https://www.tutorialspoint.com/c_standard_library/c_function_sprintf.htm	snprintf()	https://cplusplus.com/reference/cstdio/snprintf/
strcat()	https://cplusplus.com/reference/cstring/strcat/	strncat()	https://cplusplus.com/reference/cstring/strncat/
strcpy()	https://cplusplus.com/reference/cstring/strcpy/	strncpy()	https://cplusplus.com/reference/cstring/strncpy/
vsprintf()		https://cplusplus.com/reference/cstdio/vsprintf/	

Stack Canary

https://www.wikiwand.com/en/Stack_buffer_overflow

ASLR

<https://www.ibm.com/docs/en/zos/2.4.0?topic=overview-address-space-layout-randomization>

DEP

[https://support.microsoft.com/en-us/topic/what-is-data-execution-prevention-dep-60dabc2b-90db-45fc-9b18-512419135817#:~:text=Data%20Execution%20Prevention%20\(DEP\)%20is,from%20those%20areas%20of%20memory.](https://support.microsoft.com/en-us/topic/what-is-data-execution-prevention-dep-60dabc2b-90db-45fc-9b18-512419135817#:~:text=Data%20Execution%20Prevention%20(DEP)%20is,from%20those%20areas%20of%20memory.)

AOL AIM chat software vulnerability

<https://www.computerworld.com/article/2586310/aol-instant-messenger-vulnerable-to-hackers.html>

<https://www.exploit-db.com/exploits/21196>

<https://www.kb.cert.org/vuls/id/41301>

lseek

<https://www.geeksforgeeks.org/lseek-in-c-to-read-the-alternate-nth-byte-and-write-it-in-another-file/>

argc & argv

<https://stackoverflow.com/questions/18681078/in-which-memory-segment-command-line-arguments-get-stored>

Windows 2008 R2

<https://www.microsoft.com/en-in/download/details.aspx?id=2227>

Exploiting the HTTP.sys Remote Code Execution Vulnerability

<https://nvd.nist.gov/vuln/detail/CVE-2015-1635>

<https://isc.sans.edu/diary/MS15034+HTTPsys+IIS+DoS+And+Possible+Remote+Code+Execution+PATCH+NOW/19583>

https://anantshri.github.io/manual_verification/templates/ms15-034

<https://www.youtube.com/watch?v=vw4JFKZ3IS0>

<https://www.softwire.com/insights/explaining-a-security-vulnerability-the-iis-range-header-attack-cve-2015-1635/>

<https://medium.com/@hackthebox/shutting-down-windows-os-webservers-with-ms15-034-vulnerability-db9c29c0510a>

Microsoft patch for CVE-2015-1635

<https://learn.microsoft.com/en-us/security-updates/securitybulletins/2015/ms15-034>