|  |  |
| --- | --- |
|  | Lab 1 - Investigating Vulnerabilities in C/C++ Language |
|  |  |
|  | James Cassidy  Secure Software Development  2/11/22 |

**Table of Contents**

[**Executive Summary** 2](#_Toc95650219)

[**Proposal** 2](#_Toc95650220)

[**Risk** 2](#_Toc95650221)

[**Recommendation** 2](#_Toc95650222)

[**Testing Performed** 3](#_Toc95650223)

[**Vulnerabilities Detected** 3](#_Toc95650224)

[**Mitigation Techniques** 4](#_Toc95650225)

[**Filter Input** 4](#_Toc95650226)

[**Data Execution Prevention (DEP)** 4](#_Toc95650227)

[**Address Space Layout Randomisation (ASLR)** 4](#_Toc95650228)

[**Stack Canaries** 4](#_Toc95650229)

[**Preventative Measures** 4](#_Toc95650230)

[**Conclusion** 5](#_Toc95650231)

[**Appendices** 5](#_Toc95650232)

[**Bibliography** 8](#_Toc95650233)

# **Executive Summary**

Extensive penetration testing using a debugger and a reverse engineering tool demonstrates there is a buffer overflow vulnerability within this program that can be easily exploited. The business will suffer catastrophic ramifications should an attacker strike.

According to my research, reducing or eliminating this vulnerability will ensure the product will run as intended with no loss in performance and business data and assets will remain intact.

## **Proposal**

To address the problem, I propose the following:

* ***Filtering and limiting input*.**
* ***Code refactoring and rewrites****.* This includes the printf function and introduction of stack canaries.
* ***Data Execution Prevention (DEP)***
* ***Address Space Layout Randomisation (ASLR).***

These preventive measures and mitigations techniques are further explained under their respective headings.

## **Risk**

There is a risk involved with this:

* ***Delaying release****.* The product will need to be refactored partially, which will increase costs. However, failure to fix the vulnerability will result in far greater costs than the delay itself.

## **Recommendation**

To fix this vulnerability without disruption to the rest of the business’s current operations, an increase in the IT budget and deadline extension will need to be approved to support the time spent on fixing this issue.

# **Testing Performed**

Gdb-peda is a command line debugger that was utilised to penetration test this product.

Within gdb-peda, I am able to disassemble the program and see various functions including one called ‘Hacked’. Disassembling ‘Hacked’. It has the return address of 0x080491a2. This is the function I will attempt to gain access to.

To find out how many bytes it takes to overflow the buffer, I will utilise ‘pattern\_create 100’ within gdb-peda, which produces a cyclic string of 100 characters. This value will be passed into the program and give a segmentation fault, crashing the program.

Using ‘pattern\_search’ within gdb-peda after the segmentation fault, it states what the byte offsets are after the cyclic string has been. It states 20 bytes are written before overwriting the instruction pointer. Knowing this, a payload can be created that will overwrite the instruction pointer with the return address of the hacked function.

Exiting gdb-peda, creation of the payload can take place. The payload will first contain 20 bytes of character ‘A’ (the amount before the instruction pointer is overwritten), then write the address of the function using little endian, with ‘\x’ between each value.

Once the payload has been created and entered into the program, the main function runs, as well as the hacked function. I am asked to enter my name and student number. Completing this it shows I have successfully hacked into the hacked function.

# **Vulnerabilities Detected**

The vulnerability detected within this program was a stack-based buffer overflow. This is an approach that involves the attacker sending malicious code to the program, which is then stored in the stack buffer. This can overwrite the data of the stack including the return pointer, which can give the attacker control of aspects of the program or crash the program putting it into an infinite loop leading to a lack of availability.

An attacker can also exploit the buffer overflow further with a ‘return-to-libc’ attack. It causes the process to return one by one to the start of a chain of library functions. These library functions may perform the operation an attacker wants but are most commonly used to execute malicious shellcode which is generally outside the scope of the program’s security policy. Root access can be granted through malicious shellcode injection, giving attackers complete control of a system and in turn, the business.

# **Mitigation Techniques**

There are a number of techniques that can be utilised to mitigate a buffer overflow.

## **Filter Input**

Filtering a program could check the input size and strip out any bad characters within the input that are deemed ‘unsafe’. Blacklisting can be utilised to validate the input against a list of known attack patterns whilst whitelisting can be implemented to accept input from a predefined list of good patterns, helping reduce the threat of common buffer overflow attacks.

## **Data Execution Prevention (DEP)**

DEP forces structures such as the stack to be marked as non-executable, preventing buffer overflow attacks from being executed by an attacker. This is compounded with modern CPUs supporting No-Execute (NX), allowing the CPU to enforce execution rights at a hardware level.

## **Address Space Layout Randomisation (ASLR)**

ASLR randomises the offsets it uses in memory layouts. This helps to ensure memory addresses are not predictable. This does not resolve the vulnerabilities such as the buffer overflow but makes them harder to find and exploit.

## **Stack Canaries**

These are values placed in memory before the stack return pointer. This is monitored to make sure it has not been changed. Buffer overflow has to work through the memory to reach a return address and therefore through the canary. While it won’t prevent the attack, the business will be made aware when an attack takes place.

# **Preventative Measures**

The easiest way top prevents a buffer overflow is to use a language that isn’t susceptible to them. The program penetration tested was written in C which allows direct access to the memory, making these exploits more common. Languages such as Java or Python do not allow this and are immune to these attacks

Completely changing a language during development is not viable in this case and so secure practises are required to handle this exploit.

There was no access to the source code when viewing this program. Ghidra is a reverse engineering tool that was utilised to view specific functions of this program.

Analysing the main function, ‘printf’ has been used. This is unsafe. A safer function to use would be ‘snprintf()’. This prevents a buffer overflow from occurring, therefore reducing the threat level almost entirely.

# **Conclusion**

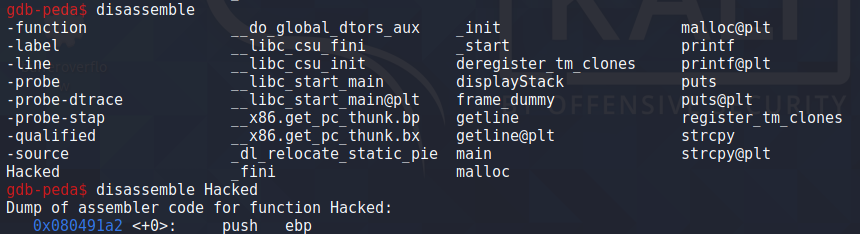
While use of a debugger and reverse engineering were needed to gain access to hidden functions, something the majority of customers do not have, attackers will use this and far more to accomplish their task.

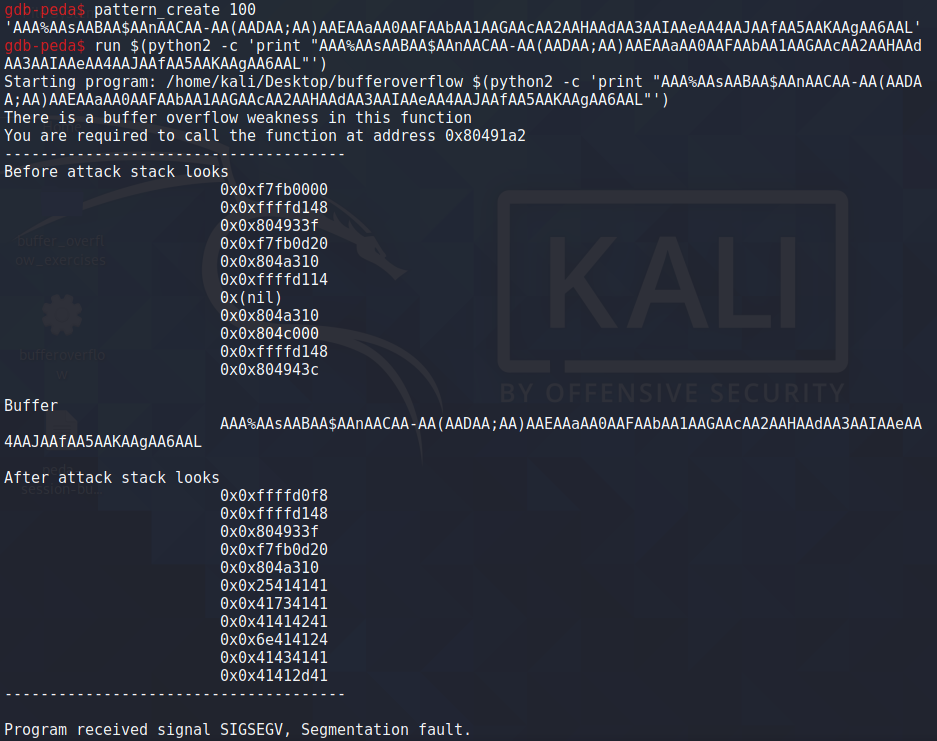
From what has been discussed, code refactoring and rewrites including snprintf() and stack canaries respectively are the best step forward in resolving this exploit. The re-written function should make it much harder for an attacker to exploit the product. This in conjunction with ASLR and DEP implementation should make buffer overflow exploitation almost impossible. Filtering input into the product will reduce this further.

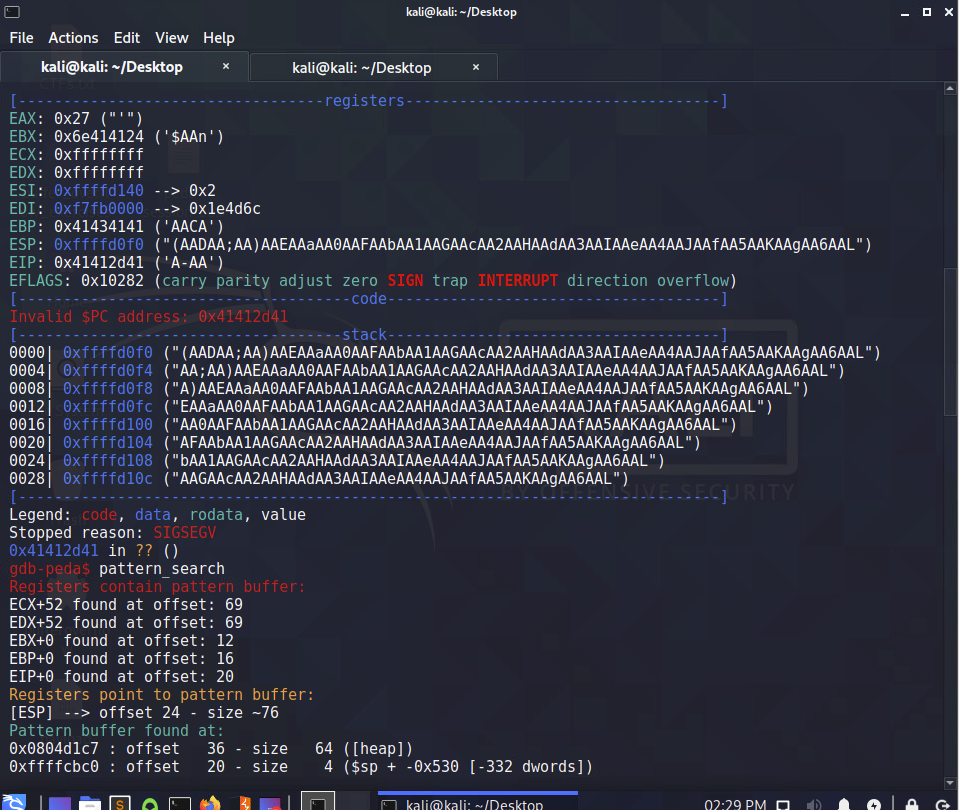
However, if this attack was to occur after previous implementation, implementation of stack canaries will alert the business if such an attack has happened.

With an increase to the IT budget and deadline extension, this product will be free from this exploit and keep the business’s assets and data secure.

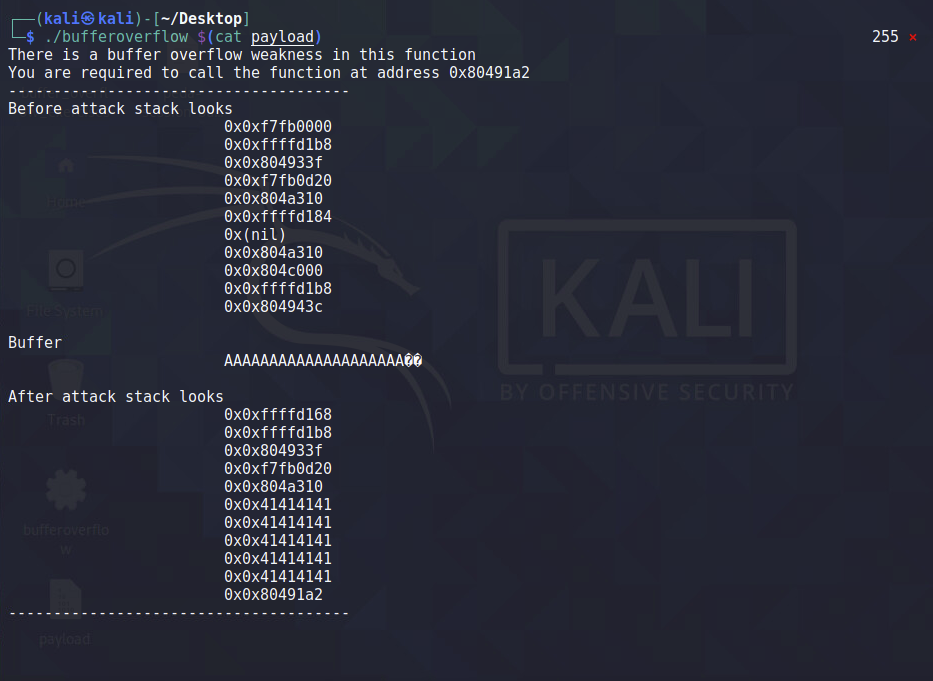
# **Appendices**

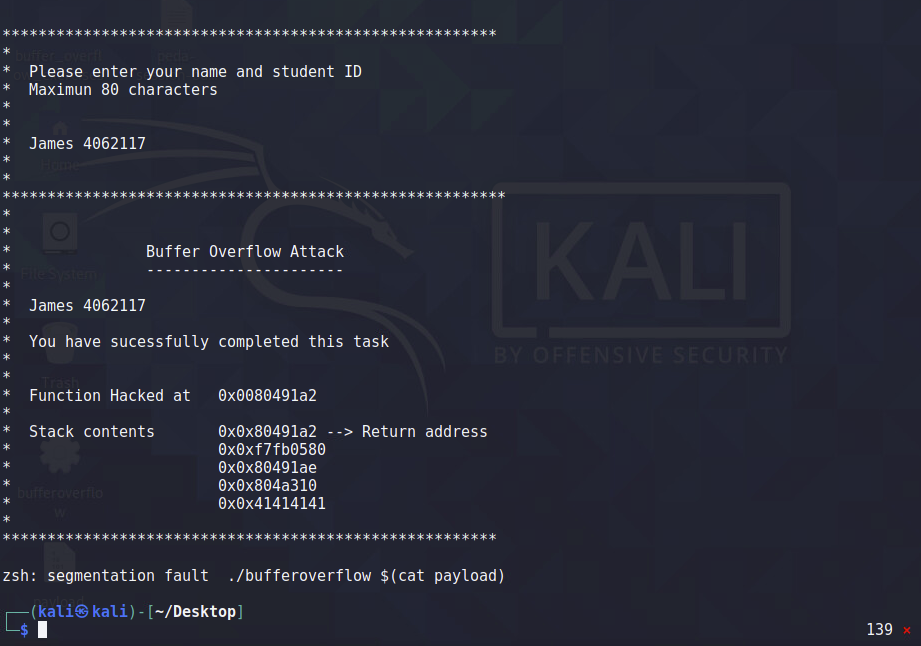


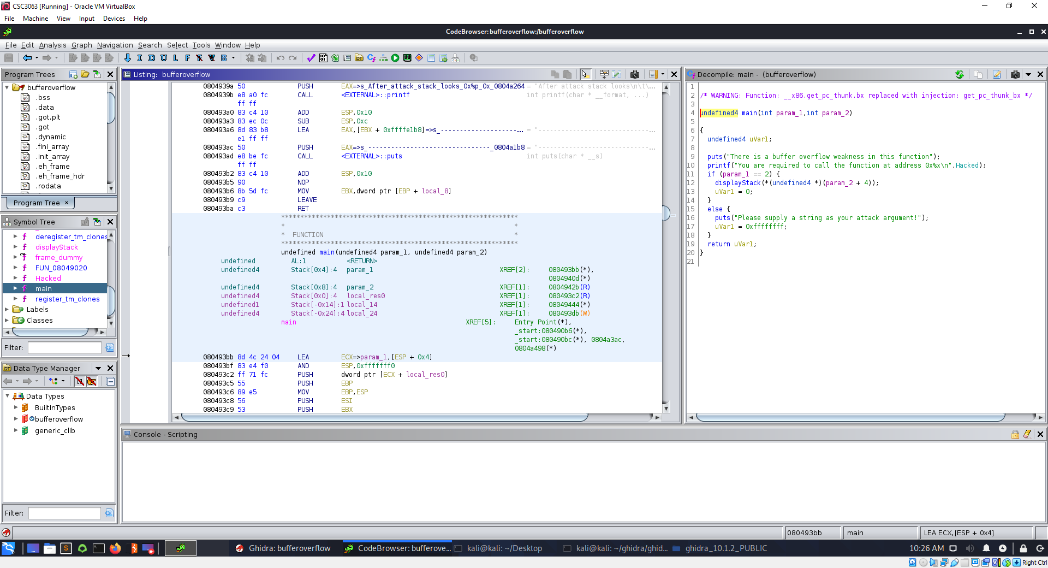




Payload used to gain access to the hidden function within the program.







# **Bibliography**

dgookin. (2020, January 4). *The snprintf() function*. Retrieved from C for Dummies: https://c-for-dummies.com/blog/?p=3921

Fortneit. (n.d.). *Buffer Overflow*. Retrieved from Forneit: https://www.fortinet.com/resources/cyberglossary/buffer-overflow

Henry-Stroker, S. (2019, January 8). *https://www.networkworld.com/article/3331199/what-does-aslr-do-for-linux.html*. Retrieved from Network World: https://www.networkworld.com/article/3331199/what-does-aslr-do-for-linux.html

Ninja, S. (2015, August 6). *Infosec*. Retrieved from InfosecI nstitute: https://resources.infosecinstitute.com/topic/buffer-overflow-format-string-attacks-basics-part-2/

O'Kane, P. (n.d.). *Buffer Overflow.* Belfast: Queen's University Belfast.

Owasp. (n.d.). *Buffer Overflow*. Retrieved from Owasp: https://owasp.org/www-community/vulnerabilities/Buffer\_Overflow

Polynomial. (2012, Ausgust 12). *How do ASLR and DEP Work*. Retrieved from Stack Exchange: https://security.stackexchange.com/questions/18556/how-do-aslr-and-dep-work

Team, S. E. (2017, Febuary 7). *How to detect, prevent, and mitigate buffer overflow attacks*. Retrieved from Synopsys: https://www.synopsys.com/blogs/software-security/detect-prevent-and-mitigate-buffer-overflow-attacks/

**Words used in Report (Excluding Title Page and Table of Contents) = 1098**