# Proyecto Final
# Curso de Sistemas Operativos y Laboratorio

## Título del proyecto

Buffer Overflow

## Miembros del equipo

Huber Steven Arroyave Rojas
Assandry Enrique Barón Rodríguez

## Resumen

En este proyecto se hablará acerca de una de las vulnerabilidades que ha estado presente durante varios años y ha sido una de las más peligrosas, dicha vulnerabilidad es llamada Buffer Overflow. En este caso se realizará una simulación práctica en la cual, se buscará realizar un ataque de desbordamiento de buffer en un entorno controlado como virtualbox para poder evaluar su impacto en la memoria del sistema y a su vez, explorar las diferentes medidas de mitigación. Con este proyecto, se podrá entender cómo pueden ser explotadas las fallas que se presentan en la gestión de la memoria. También por que es importante que se tenga en cuenta las prácticas de programación segura.

## Introducción

El desbordamiento de buffer pasa cuando un programa escribe más datos de los que puede almacenar en un área de la memoria (buffer), ocasionando que se sobreescriban otras partes de la memoria adyacente. Para explicar mejor esto, es como si se estuviera metiendo más ropa de la que cabe en una maleta y al final, algunas cosas van a terminar fuera de lugar. Eso es lo que sucede con el desbordamiento de buffer. Esto, ocasiona que se generen errores o incluso ataques maliciosos, los cuales pueden ocasionar algún fallo en el sistema o escalar los privilegios, otorgando al atacante control no autorizado sobre el sistema comprometido.

A pesar de que hoy en día existen diferentes medidas de protección incorporadas en los sistemas operativos modernos, todavía existen varios sistemas que siguen siendo vulnerables, debido al código heredado que no fue diseñado con las estrictas prácticas de seguridad actuales, o a implementaciones mal gestionadas de la memoria en aplicaciones nuevas. La naturaleza de lenguajes de bajo nivel como C y C++, que permiten un manejo manual y directo de memoria, si bien ofrecen un gran control y eficiencia, también introduce un mayor riesgo de errores si no se implementan validaciones de entrada y límites de buffer adecuados.

Por ende comprender esta problemática es fundamental para desarrollar software más robusto y para defender los sistemas informáticos en un entorno digital cada vez más complejo y amenazante.

# Antecedentes o marco teórico

El buffer overflow ha sido responsable de diferentes ataques informáticos, los cuales han sido muy perjudiciales para el tema de la informática. Un ejemplo de esto fue el gusano Morris en el año de 1998, el cual aprovechó el desbordamiento de buffer para poder propagarse y causar interrupciones masivas en la red. Este problema, lo podemos encontrar en diferentes lenguajes como C y C++, debido a que en dichos lenguajes el manejo de la memoria es manual y no existen los controles automáticos de límite. La figura 1 muestra un ejemplo de un buffer overflow en un arreglo (array)

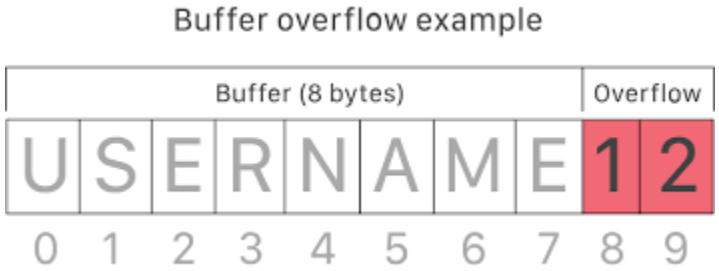


Figura 1: Ejemplo de un Buffer overflow

Para esta vulnerabilidad, existen varios tipos de ataques como el stack overflow. Esto sucede cuando se escribe más información en la pila de la que fue reservada, ocasionando que se sobreescriba la dirección de retorno almacenada por funciones. Permitiendo que el programa pueda ser engañado, para que se redirija hacia un código malicioso, como un shellcode inyectado en el mismo buffer.

El heap-based overflow, se produce cuando un programa escribe más datos en una región del heap, es decir, en la memoria dinámica, la cual es asignada por diferentes funciones como malloc(), lo cual puede corromper las estructuras de control, modificar los punteros y permitir la ejecución de código arbitrario.

También encontramos el string format vulnerability, el cual, ocurre cuando una función de formateo como el printf(), utiliza de una manera directa una entrada del usuario como cadena

de formato. Ocasionando, que se afecte la integridad del programa, ya que esto, permite que se pueda leer o escribir en las direcciones de memorias arbitrarias.:
Para que estos ataques funcionen, es necesario entender cómo está organizada la memoria del sistema, principalmente en las arquitecturas como x86. Donde los registros EIP (Instruction Pointer) , el cual se encarga de controlar la ejecución de las instrucciones y ESP (Stack Pointer) que se encarga del acceso a la pila. Por lo cual, un atacante que sepa cómo funcionan dichos registros puede redirigir el flujo del programa mediante técnicas como Sleds de NOP, que son instrucciones que no hacen nada, usadas como alfombra para poder aterrizar de una forma segura en el shellcode. Offsets calculados para ubicar exactamente dónde escribir y la sobreescritura de direcciones de retorno que es cuando se escribe más allá del final de un buffer.
A pesar de que se han realizado grandes avances en la protección para poder evitar estos ataque, como la aleatorización del espacio de direcciones (ASLR), que es una técnica de seguridad en la cual, se aleatoriza las direcciones de la memoria, evitando que un atacante no sepa hacia dónde redirigir la ejecución, ya que las direcciones cambian cada vez que se ejecuta el programa. La prevención de la ejecución de los datos (DEP/NX bit), la cual, impide la ejecución de código áreas de la memoria, donde solo deberían contener solo datos. También está los canarios en pila (stack canaries), los cuales permiten detectar y prevenir los ataques de stack buffer overflow antes de que se pueda sobreescribir la dirección de retorno de una función. Todavía hay muchísimas aplicaciones modernas que aún contienen código vulnerable o que ejecutan bibliotecas inseguras.
En la tabla 1, se puede ver una comparativo entre los ataques mencionados anteriormente y las mitigaciones típicas:

| Tipo de ataque | Memoria afectada | Impacto común | Mitigaciones típica |
|---|---|---|---|
| Stack Overflow | Pila | Ejecución de shellcode | Canarios, DEP |
| Heap-Based Overflow | Memoria dinámica | Corrupción de estructuras | Safe unlinking, ASLR |
| String Format Vulnerability | Memoria arbitraria | Lectura/escritura de memoria | Validación de formatos |

Tabla 1: Comparación de los tipos de ataques

De acuerdo a lo visto en el curso de sistemas operativos, este proyecto se enfoca en comprender cómo las vulnerabilidades de memoria pueden ser aprovechadas para comprometer la seguridad de un sistema.
La gestión de procesos, la asignación de memoria, la ejecución de instrucciones y la protección de memoria son conceptos fundamentales, que son aplicables a la comprensión y explotación de los Buffer Overflow. Además, se analizaran las contramedidas implementadas por los

sistemas operativos, como la aleatorización de espacio de direcciones (ASLR) y la protección de no ejecución (NX/DEP) para mitigar los ataque de desbordamiento de buffer.

# Objetivos (principal y específicos)

- **Objetivo principal:** Realizar un ataque de desbordamiento de buffer en un entorno controlado, para poder entender su funcionamiento y cómo prevenirlo

- **Objetivos específicos:**
    - Investigar todo lo relacionado con los fundamentos teoricos y tecnicos del buffer overflow
    - Simular el ataque en un entorno controlado utilizando algunas herramientas como kali Linux e Immunity Debugger.
    - Identificar las direcciones de memoria y los offsets que puedan ser relevantes para el ataque
    - Crear un código malicioso para poder inyectar y ejecutar el ataque
    - Analizar y explorar diferentes soluciones para poder mitigar el riesgo de los desbordamientos de buffer, y evaluar su efectividad frente a los ataques simulados.

# Metodología

- Investigación exhaustiva para comprender en profundidad la teoría sobre buffer overflow, sus diversas manifestaciones (stack, heap, format string), las causas fundamentales relacionadas con la gestión de memoria insegura, y las implicaciones de seguridad a través del análisis de casos de ataque.
- Preparación del entorno de las pruebas en el entorno de virtualización virtualbox utilizando los sistemas operativos de Kali Linux, windows y brainpan (Es una máquina Linux que cuenta con un servicio Windows vulnerable a Buffer Overflow)
- Compilación del código vulnerable con flags de seguridad deshabilitadas
- Realización de análisis dinámico detallado del programa vulnerable utilizando depuradores como GDB, Immunity Debugger, mona.py y scripts de Metasploit para la identificación de patrones y la búsqueda de módulos sin protección
- Identificación de las direcciones más relevantes, offsets y badchars.
- Desarrollo del payload que podría incluir un shellcode para la ejecución de código arbitrario o para la manipulación del flujo de ejecución; y se construirá un exploit para enviar este payload al programa vulnerable, observando si se logra la redirección del flujo y la ejecucion del codigo deseado.
- Documentación de los resultados, hallazgos, conclusiones y el análisis de las mitigaciones, evaluando la efectividad teórica, frente al ataque simulado.

# Cronograma

| ID | Tarea | Duración | Fecha de Inicio | Fecha de Fin |
|---|---|---|---|---|
| 1 | Fase 1: Investigación y Configuración | | | |
| 1.1 | Revisión Bibliográfica y Marco Teórico | 7 | 20/05/2025 | 26/05/2025 |
| 1.2 | Preparación del Entorno Virtual (VirtualBox, OS) | 4 | 27/05/2025 | 30/05/2025 |
| 1.3 | Instalación y Configuración de Herramientas de Ataque | 5 | 31/05/2025 | 04/06/2025 |
| 2 | Fase 2: Análisis y Explotación del Binario Vulnerable | | | |
| 2.1 | Fuzzing y Detección de Crash | 6 | 05/06/2025 | 10/06/2025 |
| 2.2 | Depuración con Immunity Debugger/GDB | 7 | 11/06/2025 | 17/06/2025 |
| 2.3 | Identificación de Offsets y Badchars | 4 | 18/06/2025 | 21/06/2025 |
| 2.4 | Generación de Shellcode (msfvenom) | 3 | 22/06/2025 | 24/06/2025 |
| 2.5 | Desarrollo del Exploit Script (Python) | 5 | 25/06/2025 | 29/06/2025 |
| 2.6 | Ejecución y Validación del Exploit | 4 | 30/06/2025 | 03/07/2025 |
| 3 | Fase 3: Análisis de Mitigaciones y Documentación | | | |
| 3.1 | Investigación y Análisis de Técnicas de Mitigación | 2 | 04/07/2025 | 05/07/2025 |
| 3.2 | Pruebas con Mitigaciones Activas (si aplica) | 1 | 06/07/2025 | 06/07/2025 |
| 3.3 | Redacción de Resultados y Conclusiones | 2 | 07/07/2025 | 08/07/2025 |
| 3.4 | Revisión y Edición Final del Documento | 2 | 09/07/2025 | 10/07/2025 |

Tabla 2: Diagrama de Gantt

Para ver en mejor detalle Diagrama de Gantt

# Referencias


- Cloudflare. (s.f.). *Buffer overflow*. Cloudflare. https://www.cloudflare.com/es-es/learning/security/threats/buffer-overflow/
- Foster, J. , Osipov, V., Bhalla, N., Heinen, N., & Liu, Y. (2005). *Buffer overflow attacks: Detect, exploit, prevent*. Syngress Publishing. https://repo.zenk-security.com/Techniques%20d.attaques%20%20.%20%20Failles/Buffer%20Overflow%20Attacks%20-%20Detect%20Exploit%20Prevent.pdf
- Rapid7. (s.f.). *Metasploit Framework*. Rapid7. https://docs.rapid7.com/metasploit/
- Immunity Inc. (s.f.). *Immunity Debugger*. https://www.immunityinc.com/products/debugger/
- GNU Project. (s.f.). *GDB: The GNU Project Debugger*. Sourceware. https://www.gnu.org/software/gdb/