Estudio
Vulnerabilidades persistentes: Escritura fuera de límites (BO)

Financiado por la Unión Europea
NextGenerationEU

GOBIERNO DE ESPAÑA
MINISTERIO PARA LA TRANSFORMACIÓN DIGITAL Y DE LA FUNCIÓN PÚBLICA
SECRETARÍA DE ESTADO DE TELECOMUNICACIONES E INFRAESTRUCTURAS DIGITALES

Plan de Recuperación, Transformación y Resiliencia

incibe_
INSTITUTO NACIONAL DE CIBERSEGURIDAD

TLP:CLEAR

Financiado por la Unión Europea
NextGenerationEU

GOBIERNO DE ESPAÑA
MINISTERIO PARA LA TRANSFORMACIÓN DIGITAL Y DE LA FUNCIÓN PÚBLICA
SECRETARÍA DE ESTADO DE TELECOMUNICACIONES E INFRAESTRUCTURAS DIGITALES

Plan de Recuperación, Transformación y Resiliencia

incibe_
INSTITUTO NACIONAL DE CIBERSEGURIDAD

Mayo 2026

INCIBE-CERT_ESTUDIO_VULNERABILIDADES_PERSISTENTES_ESCRITURA FUERA DE LÍMITES_v1.0

Financiado por la Unión Europea NextGenerationEU
GOBIERNO DE ESPAÑA MINISTERIO PARA LA TRANSFORMACIÓN DIGITAL Y DE LA FUNCIÓN PÚBLICA SECRETARÍA DE ESTADO DE TELECOMUNICACIONES E INFRAESTRUCTURAS DIGITALES
Plan de Recuperación, Transformación y Resiliencia
incibe_ INSTITUTO NACIONAL DE CIBERSEGURIDAD

Índice

ÍNDICE DE FIGURAS Y TABLAS

1. Sobre este estudio

En el mundo de la seguridad informática, **las vulnerabilidades de escritura fuera de límites (*buffer overflow*) representan una amenaza crítica que ha persistido a lo largo de las décadas**. A menudo, estas vulnerabilidades se asocian con lenguajes de programación de bajo nivel, como C o C++, y sistemas antiguos. Sin embargo, la realidad es que este tipo de vulnerabilidad puede manifestarse en una amplia variedad de contextos y lenguajes, afectando tanto a *software* moderno como a aplicaciones heredadas.

Por ejemplo, errores comunes, como no controlar el tamaño de los vectores o usar librerías o componentes de terceros vulnerables, pueden abrir la puerta a los ciberataques. La falta de validación adecuada de los datos de los datos de entrada y el uso descuidado de funciones que manipulan la memoria, son otros factores habituales que contribuyen a que estas vulnerabilidades sigan siendo una amenaza relevante, incluso en tecnologías de vanguardia.

Resolver las vulnerabilidades de *buffer overflow* exige un conocimiento profundo del funcionamiento de las computadoras, ya que estas vulnerabilidades afectan directamente la interacción entre el procesador y la memoria principal. Son problemas intrínsecamente complejos que exigen conocimientos sobre cómo el *hardware* y el *software* manejan la memoria. Entender la arquitectura del sistema, cómo se asigna y se maneja la memoria, y cómo los datos pueden ser manipulados en niveles bajos es crucial para implementar medidas de seguridad efectivas.

Aunque existen muchas herramientas y técnicas para tratar de encontrar estas vulnerabilidades, ninguna es completamente efectiva. Los programas pueden ser muy complicados y probar todas las posibles formas en que podría ocurrir una escritura fuera de límites es prácticamente imposible. **En muchos casos, la manifestación de estas vulnerabilidades no es predecible, debido a que dependen de lo que está ocurriendo en la memoria del sistema en un momento dado**. Un programa podría funcionar correctamente la mayor parte del tiempo, pero fallar de manera impredecible en un momento concreto debido a un error de escritura fuera de límites, lo que hace que sea difícil de detectar y solucionar.

Aunque las vulnerabilidades de escritura fuera de límites son un desafío persistente, **la combinación de una educación continua y la aplicación de buenas prácticas puede reducir significativamente su impacto**. Incluso pequeños cambios pueden tener un gran efecto positivo, haciendo que nuestras aplicaciones sean más seguras y resistentes frente a los ataques.

Este estudio establece las bases para comprender la magnitud del problema y sus causas principales, mientras explora algunas **buenas prácticas que pueden representar soluciones rápidas** para mitigar las vulnerabilidades de escritura fuera de límites en las aplicaciones. Por tanto, puede ser de utilidad para desarrolladores de *software*, equipos de seguridad informática y gestores de proyectos tecnológicos que buscan mejorar la seguridad y robustez de sus sistemas, pero está abierto a cualquiera que quiera aprender, de forma general, sobre este tipo de vulnerabilidad.

Financiado por la Unión Europea NextGenerationEU
GOBIERNO DE ESPAÑA MINISTERIO PARA LA TRANSFORMACIÓN DIGITAL Y DE LA FUNCIÓN PÚBLICA SECRETARÍA DE ESTADO DE TELECOMUNICACIONES E INFRAESTRUCTURAS DIGITALES
Plan de Recuperación, Transformación y Resiliencia
incibe_ INSTITUTO NACIONAL DE CIBERSEGURIDAD

2. Organización del documento

El estudio se inicia con la sección **3.- Introducción**, donde se describe la vulnerabilidad de escritura fuera de límites y los fundamentos de la arquitectura de computadores que la originan.

La sección **4.- Tipologías de análisis y herramientas**, proporciona una visión detallada de las diversas metodologías que se pueden utilizar para estudiar este tipo de vulnerabilidades. Se abordan tanto las técnicas y herramientas de desarrollo y preproducción, como el análisis estático, dinámico y en tiempo de producción.

En la sección **5.- Preparación de un entorno de test**, se ofrecen directrices sobre cómo configurar un entorno seguro y controlado para llevar a cabo pruebas de análisis de vulnerabilidades de escritura fuera de límites.

La sección **6.- Ejemplo de análisis de vulnerabilidades de escritura fuera de límites**, plantea un caso práctico realizado en el entorno de pruebas previamente configurado, demostrando cómo identificar y explotar vulnerabilidades de escritura fuera de límites.

En la sección **7.- Prevención y buenas prácticas**, se discuten las estrategias clave para mitigar estas vulnerabilidades, como el uso de tecnologías preventivas o la mejora de las prácticas de programación segura.

Finalmente, la sección **8.- Conclusión**, resume los puntos más importantes del documento y se plantea el compromiso entre funcionalidad y seguridad.

3. Introducción

La vulnerabilidad de escritura fuera de límites es notable por su prevalencia, ocupando consistentemente los primeros puestos y situándose en el top 1 del CWE top 25 de MITRE en 2023[1]. Esta vulnerabilidad, común en lenguajes con gestión de memoria débil, puede ser aprovechada para ejecutar código arbitrario, corromper datos o causar denegación de servicio, amenazando la seguridad de las aplicaciones.

La escritura fuera de límites (CWE 787[2]), es un tipo de **restricción inadecuada de operaciones dentro de los límites de un búfer de memoria**[3] y supone un problema crítico en la programación que puede tener graves consecuencias para la seguridad y estabilidad de un *software*. Para comprender cómo funciona, es esencial conocer la arquitectura de los sistemas actuales.

3.1. Arquitectura y gestión de memoria

El diseño de las arquitecturas de CPU más utilizadas hoy en día se deriva de la arquitectura de Von Neumann, un modelo fundamental en la computación.

- **Componentes de la CPU:** esta arquitectura se compone de tres componentes principales: la unidad aritmético-lógica, la unidad de control y los registros.
  - **Unidad aritmético-lógica (ALU, en inglés)**: la ALU es responsable de ejecutar las instrucciones matemáticas y lógicas. Es el "cerebro" de la CPU, donde se realizan los cálculos y operaciones lógicas.
  - **Unidad de control**: la unidad de control gestiona el flujo de datos dentro del sistema. Obtiene las instrucciones desde la memoria principal y utiliza el puntero de instrucción (IP) para determinar qué instrucción ejecutar a continuación. Esta unidad coordina todas las actividades dentro de la CPU para asegurar que las instrucciones se ejecuten en el orden correcto.
  - **Registros**: los registros son pequeñas unidades de almacenamiento dentro de la CPU que contienen datos temporales y de control. A pesar de su tamaño reducido, son cruciales para la rápida ejecución de instrucciones. Existen varios tipos de registros, entre los que se destacan:
    - **Puntero de instrucción (IP)**: contiene la dirección de la próxima instrucción a ejecutar
    - **Registros de propósito general**: utilizados durante la ejecución general de instrucciones.
    - **Registros de estado**: indican el estado de la ejecución, como si el resultado de una operación es cero o uno, por ejemplo.
    - **Registros de segmento**: dividen el espacio de memoria en segmentos más manejables.

[1] https://cwe.mitre.org/top25/archive/2023/2023_stubborn_weaknesses.html

[2] https://cwe.mitre.org/data/definitions/787.html

[3] https://cwe.mitre.org/data/definitions/119.html

Ilustración 1: Diagrama básico de una CPU

- **Interacción con la memoria**: la CPU interactúa con la memoria principal, conocida como RAM. Cuando un programa se ejecuta en una computadora, este se convierte en un proceso. Gracias a la arquitectura de los sistemas informáticos modernos, es posible ejecutar varios procesos al mismo tiempo en una sola computadora, pero, aunque parece que estos procesos se ejecutan simultáneamente, en realidad, la computadora cambia rápidamente entre ellos. Este rápido cambio da la apariencia de que todos se ejecutan a la vez, pero se llama cambio de contexto. Cada proceso necesita diferentes datos para funcionar correctamente, como la instrucción específica que debe ejecutar en un momento dado. Por lo tanto, el sistema operativo debe mantener un registro detallado de toda la información relevante para cada proceso. La memoria que utiliza cada proceso está organizada de forma secuencial y sigue una estructura específica.
  - La **pila** de usuario contiene la información necesaria para ejecutar el programa. Esta información incluiría el contador del programa actual, registros guardados y más información. La sección después de la pila de usuario es memoria no utilizada y se usa en caso de que la pila crezca (hacia abajo).
  - Las **regiones de biblioteca** compartida se utilizan para vincular estática o dinámicamente las bibliotecas que utiliza el programa.
  - El ***heap*** aumenta y disminuye dinámicamente dependiendo de si un programa asigna memoria dinámicamente.
  - El **código y los datos del programa** almacenan el programa ejecutable y las variables inicializadas.
- **Representación de datos en memoria**: el concepto conocido como *Endiannes* ayuda a entender cómo se representan los datos en memoria. Por ejemplo, el número hexadecimal `0x12345678` tiene su byte más significativo (`12`) a la izquierda y el menos significativo (`78`) a la derecha. En arquitecturas *Big Endian*, este número se almacena comenzando con el byte más significativo en la dirección de memoria

más baja (12 34 56 78). En *Little Endian*, se almacena comenzando con el byte menos significativo en la dirección más baja (78 56 34 12). Los procesadores Intel y AMD utilizan el formato *Little Endian*. En contraste, los procesadores *PowerPC* y SPARC utilizan el formato *Big Endian*. La arquitectura ARM es versátil, pudiendo operar en ambos modos, aunque suele funcionar en modo *Little Endian* en muchos dispositivos móviles y embebidos.

El ancho de palabra de la arquitectura del sistema (32 bits *vs*. 64 bits) es crucial. En procesadores de 32 bits, los registros, incluidas las direcciones de memoria y el IP, son de 32 bits de longitud, lo que significa que las direcciones de memoria son de 4 bytes. En sistemas de 64 bits, estos registros son de 64 bits, lo que implica direcciones de memoria de 8 bytes. Esta diferencia afecta directamente cómo se calculan y manipulan los *offsets* y las direcciones durante la explotación. Además, el espacio de direcciones en sistemas de 32 bits está limitado a 4 GB (2^{32}), haciendo que las direcciones sean más predecibles y, en algunos casos, más fáciles de explotar. En contraste, el espacio de direcciones en sistemas de 64 bits es mucho mayor (2^{64}), complicando la predicción de direcciones.

3.2. Desbordamiento causado por la escritura fuera de límites

Se da cuando un programa escribe datos más allá del final o antes de su espacio reservado en memoria o búfer. Esta vulnerabilidad suele originarse cuando un programa realiza operaciones incorrectas con punteros o índices. Por ejemplo, al calcular una posición en memoria, el programa puede referenciar una ubicación fuera de los límites establecidos para el búfer. Posteriormente, una operación de escritura en esta memoria puede producir resultados impredecibles y dañinos. La escritura fuera de límites no se limita únicamente a la copia secuencial de datos excesivos desde una ubicación inicial fija. También, puede surgir de problemas como aritmética de punteros incorrecta, acceso a punteros inválidos debido a una inicialización incompleta o la liberación incorrecta de memoria. Cuando esto ocurre, pueden surgir varios problemas, como la corrupción de datos, fallos en el sistema o incluso la ejecución de código malicioso. Existen varios tipos de vulnerabilidades de escritura fuera de límites que dependen de dónde ocurren o en qué condiciones se dan: el desbordamiento de búfer basado en pila (*Stack-based Buffer Overflow*), el desbordamiento de búfer basado en memoria dinámica (*Heap-based Buffer Overflow*), la condición de escritura arbitraria (*Write-what-where Condition*) y la escritura fuera de los límites del búfer (*Buffer Underwrite o Buffer Underflow*). Sin ánimo de ser exhaustivos, en esta guía hablaremos del desbordamiento de búfer de forma general.

Ningún lenguaje es por definición completamente inmune a los desbordamientos de búfer, ya que esta vulnerabilidad puede surgir debido a errores en la lógica de programación y a la manera en que los desarrolladores gestionan la memoria. Sin embargo, es cierto que es más probable que ocurran en programas desarrollados con lenguajes que permiten al programador gestionar directamente el espacio de memoria, como C o C++.

Reescritura de variables

Para entender cómo ocurren los desbordamientos de memoria, es fundamental examinar cómo se asignan y organizan las variables en la memoria. En el contexto del código en C

a continuación, se observa que la variable entera y el búfer de caracteres se asignan adyacentes en la memoria debido a la asignación en bytes contiguos[4].

```
#include <stdio.h>
int main(int argc, char **argv)
{
    int variable = 0;
    char buffer[13];
    gets(buffer); // Cuidado, no verifica la longitud de la entrada
    if (variable != 0) {
        printf("El valor de la variable ha cambiado\n");
    } else {
        printf El valor de la variable NO ha cambiado\n");
    }
    return 0;
}
```

Aunque la pila crece hacia abajo, los datos se copian o escriben en el búfer desde direcciones inferiores a superiores. Esto significa que, dependiendo de cómo se introduzcan los datos en el búfer, es posible sobrescribir la variable entera adyacente. En el código en C, la función `gets` se utiliza para introducir datos en el búfer desde la entrada estándar, pero es peligrosa porque no verifica adecuadamente la longitud de los datos ingresados, lo que permite ingresar más de 13 bytes de datos y sobrescribir la variable entera.

Dado que la variable entera y el búfer de caracteres están uno al lado del otro y considerando que el búfer tiene una longitud de 13 bytes, el número mínimo de caracteres necesarios para sobrescribir la variable entera es de 14 caracteres. A esta distancia relativa en memoria se la conoce como *offset*. Los primeros 13 caracteres llenan el búfer, y el 14º carácter sobrescribe el primer *byte* de la variable entera, lo que puede alterar su valor y causar comportamientos inesperados o vulnerabilidades de seguridad en el programa. Por ejemplo, imaginemos que se introduce por entrada estándar una cadena de más de 13 caracteres, como "AAAAAAAAAAAAA" (14 caracteres en total). Los primeros 13 caracteres llenarían el búfer, mientras que el 14º carácter sobrescribiría la memoria donde se almacena `variable`. En ASCII, 'A' tiene un valor de 65, por lo que al sobrescribir `variable` con 'A', su valor cambia de 0 a 65. Esto hace que la condición `if (variable != 0)` se evalúe como verdadera, mostrando el mensaje "`El valor de la variable ha cambiado `".

[4] Esta disposición no siempre es constante. La configuración del compilador y la pila puede requerir que las variables se alineen en límites de tamaño específicos (por ejemplo, 8 bytes, 16 bytes) para optimizar el acceso a la memoria y el rendimiento del programa. Por ejemplo, si se asigna una matriz de 12 bytes en una pila alineada para 16 bytes, el compilador agregaría automáticamente 4 bytes adicionales para asegurar la alineación adecuada.

Reescritura de punteros

En el contexto del código en C, a continuación, se observa que un puntero a función y un búfer de caracteres se asignan adyacentes en la memoria debido a la asignación en bytes contiguos.

```
#include <stdio.h>
void funcion1()
{
    printf("Esta es la función 1\n");
    printf("¡Flujo de ejecución desviado!\n");
}
void funcion2()
{
    printf("Esta es la función 2\n");
}
void funcion3()
{
    printf("Esta es la función 3\n");
}
int main(int argc, char **argv)
{
    int (*new_ptr)() = funcion2;
    char buffer[13];
    gets(buffer); // Cuidado, no verifica la longitud de la entrada
    new_ptr();
    return 0;
}
```

En este ejemplo, los datos se leen en un búfer utilizando la función `gets`, que como hemos visto no verifica adecuadamente la longitud de los datos ingresados, permitiendo que se introduzcan más de 13 bytes de datos y sobrescriban las ubicaciones de memoria adyacentes. La variable `new_ptr` es un puntero a función que inicialmente apunta a la función `funcion2`. Aunque la pila crece hacia abajo, los datos se copian o escriben en el búfer desde direcciones inferiores a superiores. Esto significa que, dependiendo de cómo se introduzcan los datos en el búfer, es posible sobrescribir el puntero a función `new_ptr`. El objetivo aquí es sobrescribir `new_ptr` para que apunte a la función `funcion1` en lugar de `funcion2`. Supongamos que la dirección de la función `funcion1` es 0x080484b6. Si asumimos que la arquitectura de la máquina es *Little Endian*, esta dirección se debe escribir en el búfer en orden inverso de bytes: \xb6\x84\x04\x08. Dado que el búfer tiene una longitud de 13 bytes, el número mínimo de caracteres necesarios para sobrescribir `new_ptr` (*offset*) es de 17 caracteres: 13 para llenar el búfer y 4 para sobrescribir `new_ptr`. Imaginemos que se introduce por entrada estándar una cadena como

"AAAAAAAAAAAAAAA\xb6\x84\x04\x08" (17 caracteres en total, donde "\xb6\x84\x04\x08" es la representación en *little endian* de 0x080484b6). Los primeros 13 caracteres llenarían el búfer, mientras que los siguientes 4 caracteres sobrescribirían la memoria donde se almacena `new_ptr` con la dirección de `funcion1`. Esto hace que, cuando se llame a `new_ptr()`, en lugar de ejecutar `funcion2`, se ejecute `funcion1`, mostrando el mensaje:

```
Esta es la función 1
¡Flujo de ejecución desviado!
```

3.3. Posexplotación de la escritura fuera de límites

Las consecuencias posexplotación de un ataque de escritura fuera de límites (*buffer overflow*) pueden agruparse en varias categorías:

- **Escalada de privilegios**: la vulnerabilidad de escritura fuera de límites podría permitir a un atacante obtener permisos o privilegios más altos en el sistema, lo que le permitiría realizar acciones que normalmente estarían restringidas. Esto incluye modificar configuraciones del sistema, acceder a datos protegidos y ejecutar comandos con privilegios de administrador. Esta escalada de privilegios es particularmente peligrosa porque permite evadir restricciones de seguridad, tomar control completo del sistema y desactivar mecanismos de seguridad adicionales, dejando el sistema aún más vulnerable a futuros ataques.
- **Compromiso de información y sesiones de usuario**: mediante la explotación exitosa sería posible acceder y manipular datos sensibles almacenados en la memoria, como información personal y credenciales de usuario. Esta información puede ser utilizada para secuestrar sesiones activas, obteniendo acceso no autorizado a cuentas y datos confidenciales. Este tipo de compromiso compromete la privacidad y la seguridad de los usuarios, permitiendo realizar actividades fraudulentas en nombre de las víctimas y potencialmente exponiendo datos sensibles a otros actores maliciosos.
- **Ejecución de código arbitrario**: una de las consecuencias más graves es la posibilidad de inyectar y ejecutar código malicioso en el sistema afectado. Esto puede permitir tomar control completo del sistema, instalar *malware* o crear *backdoors* para accesos futuros. La capacidad de ejecutar código arbitrario permite manipular el sistema de manera significativa, ejecutar comandos dañinos, robar información confidencial y realizar actividades destructivas o persistentes en el sistema comprometido.
- **Corrupción de datos y sabotaje**: estos ataques pueden resultar en la corrupción de datos críticos en la memoria, llevando a comportamientos incorrectos del sistema y no deterministas. La corrupción de datos puede ser utilizada como una forma de sabotaje, dañando la operación de la organización y causando impactos reputacionales y financieros significativos.
- **Cierre inesperado del sistema**: el desbordamiento de la memoria puede también finalizar inesperadamente los programas (ejemplos típicos son los mensajes de *segmentation fault*, *bus error* o *access violation*), llevando a interrupciones en las operaciones, derivando en la denegación de servicio y afectando la disponibilidad de los recursos y servicios para los usuarios legítimos.

4. Tipologías de análisis y herramientas

Existen diferentes métodos de detección y análisis de vulnerabilidades de *buffer overflow*, pudiendo emplear diferentes herramientas, dependiendo del contexto y el tipo de aplicación (binarios de escritorio o aplicaciones web) que se considere.

En el **análisis estático** (SAST), se identifican patrones en el código que podrían sugerir la presencia de vulnerabilidades de desbordamiento de búfer. Ejemplos notables de estas herramientas son **SonarQube**[5] **o Brakeman**[6], que revisa el código para detectar posibles desbordamientos de búfer al analizar el manejo de memoria y la correcta asignación de búferes.

El **análisis dinámico** (DAST) implica interactuar con aplicaciones web mientras está en funcionamiento para identificar posibles vulnerabilidades de desbordamiento de búfer. Las herramientas utilizadas para aplicaciones web incluyen **OWASP ZAP**[7] **o *Burp Suite***[8] **(*Community Edition*),** que permiten interceptar y modificar el tráfico HTTP/HTTPS, inyectando payloads maliciosos para probar la robustez de la aplicación frente a la manipulación de búferes. Para el análisis dinámico de binarios, se utilizan herramientas como **Radare2**[9], un *framework* de ingeniería inversa que permite analizar binarios y detectar desbordamientos de búfer observando cómo se manipula la memoria en tiempo real, o depuradores, como ***Immunity Debugger***[10] y **GDB**[11] **(GNU *Debugger*),** que proporcionan capacidades para depurar aplicaciones en tiempo real, monitorizar el uso de la memoria y detectar condiciones de desbordamiento de búfer. Facilitan la inserción de puntos de interrupción y el seguimiento detallado de la ejecución del programa, ayudando a identificar con precisión los puntos donde ocurren los desbordamientos.

La detección de actividad maliciosa **en tiempo real** también resulta un análisis muy interesante para tener en cuenta. Para aplicaciones web y procesos de servidor, se utilizan aplicaciones como **OWASP AppSensor**[12], que se puede instrumentar puntos de detección de patrones anómalos que pueden incluir por ejemplo la verificación constante del tamaño de los datos en los búferes, el análisis de patrones de escritura en memoria, la detección de *payloads* anómalos, la monitorización del comportamiento del proceso para detectar fallos de segmentación, la validación y sanitización de entradas, y la inspección de direcciones de retorno y punteros; o **Web Application Firewalls (WAF), como modsecurity**[13] **y OWASP Core Rule Set**[14] **(CRS)** que monitorizan y bloquean tráfico HTTP con indicios de contener *payloads*.

[5] https://www.sonarsource.com/products/sonarqube/downloads/

[6] https://github.com/presidentbeef/brakeman

[7] https://www.zaproxy.org/download

[8] https://portswigger.net/burp/communitydownload

[9] https://github.com/radareorg/radare2

[10] https://www.immunityinc.com/products/debugger/index.html

[11] https://www.sourceware.org/gdb/download/

[12] https://github.com/jtmelton/appsensor

[13] https://github.com/owasp-modsecurity/ModSecurity

[14] https://github.com/coreruleset/coreruleset

5. Preparación de un entorno de test

A continuación, configuraremos un entorno de laboratorio de seguridad aislado para realizar pruebas y análisis de vulnerabilidades de escritura fuera de límites. Este entorno controlado minimiza los riesgos para los sistemas de producción y permite simular diversos escenarios de ataque con flexibilidad. Utilizaremos máquinas virtuales (VMs) con VirtualBox, una herramienta de virtualización gratuita y fácil de usar, para crear entornos aislados y replicables. Este laboratorio incluirá varios elementos clave:

- Existen recursos gratuitos en Internet que se pueden utilizar para probar vulnerabilidades de escritura fuera de límites. Uno de ellos es la máquina virtual ***Brainpan***[15] alojada en VulnHub[16], un servidor desde el que se **descarga un fichero binario vulnerable**. Para el propósito de preparación del entorno bastará con importar la OVA y acceder a la IP asignada por el adaptador de red del hipervisor, que puede descubrirse usando nmap, concretamente a la URL: <IP>:10000/bin para descargar el fichero brainpan.exe.

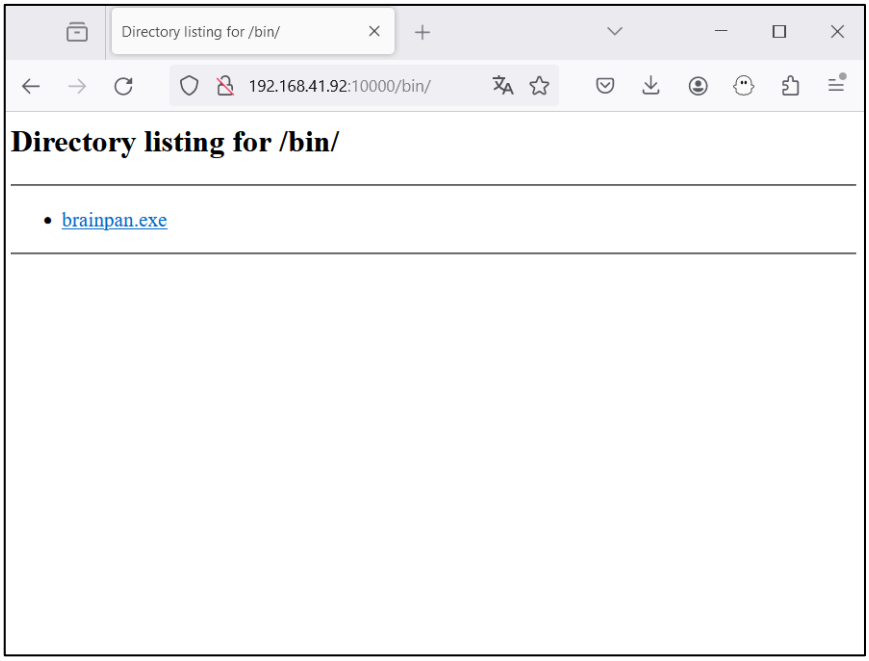


Ilustración 2: Máquina Brainpan

- Se puede descargar una máquina virtual Windows[17] para el **análisis del fichero objetivo** donde instalaremos la herramienta *Immunity Debugger*. Esta máquina virtual nos proporciona un entorno controlado y seguro para realizar análisis de software, depuración y pruebas.

[15] https://www.vulnhub.com/entry/brainpan-1,51/

[16] https://www.vulnhub.com/

[17] https://developer.microsoft.com/en-us/windows/downloads/virtual-machines/

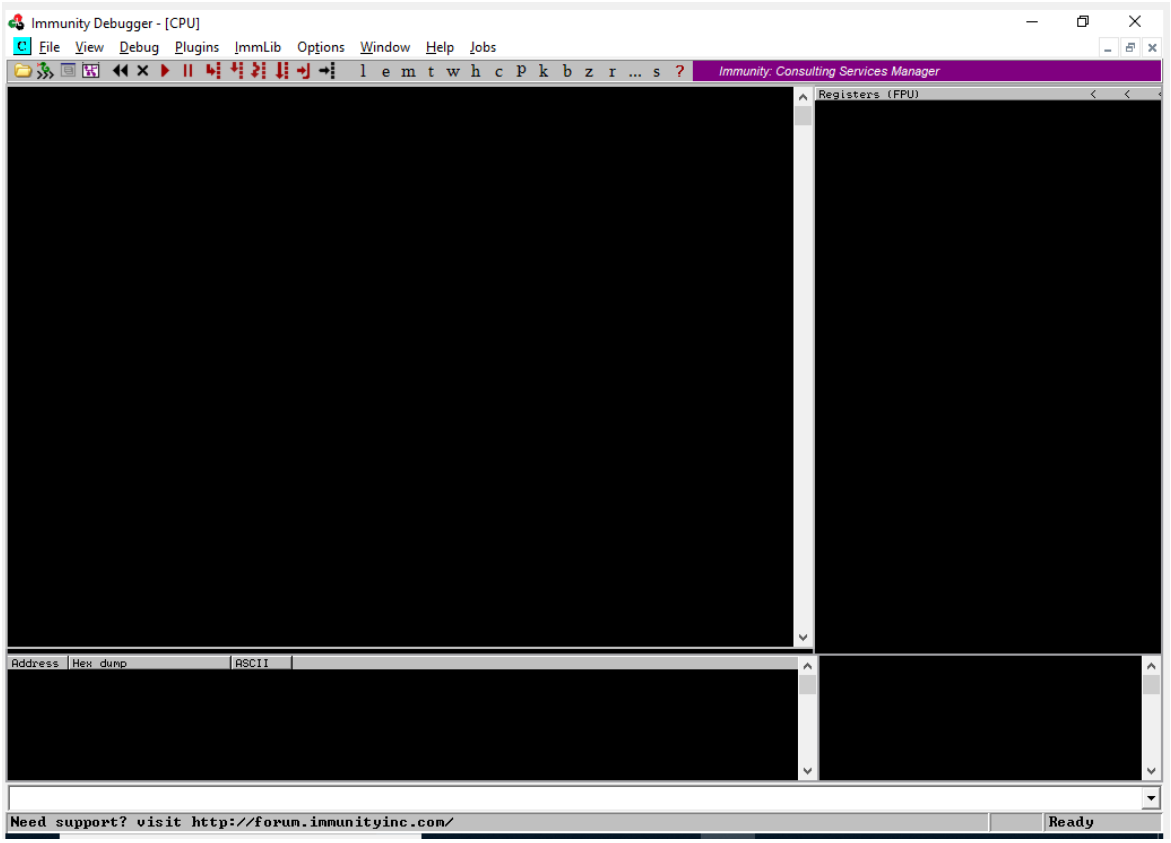
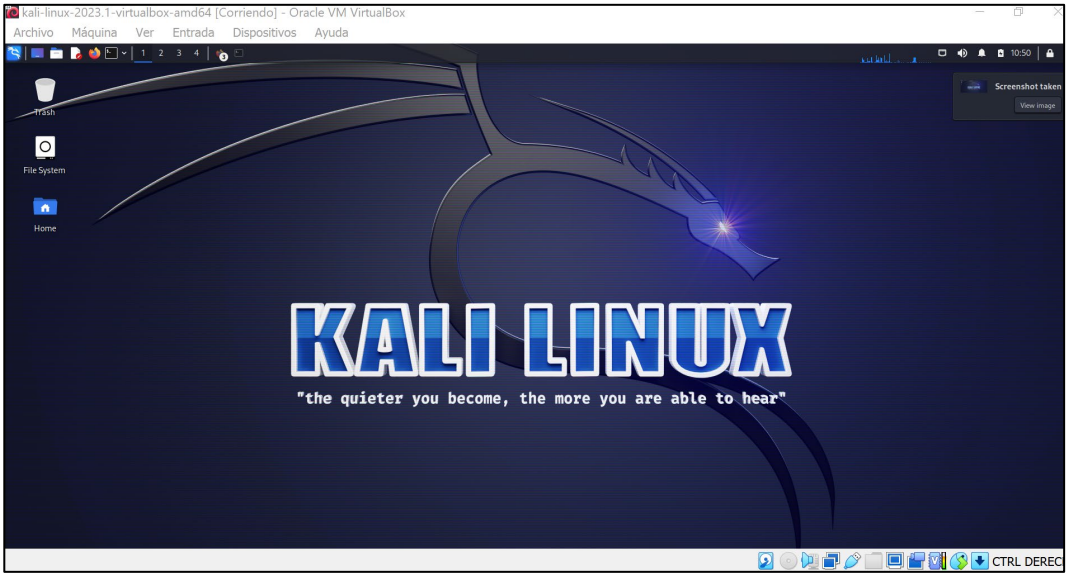


Ilustración 3: Máquina Windows con Immunity Debugger

- Usaremos una máquina virtual Kali para la **generación de *payloads***. Kali Linux[18] es una distribución basada en Debian que incluye una amplia variedad de herramientas de seguridad y de análisis forense, incluyendo Metasploit, que se utiliza para crear y gestionar *payloads*.



[18] https://www.kali.org/

*Ilustración 4: **Máquina Kali***

Es esencial desplegar máquinas virtuales en un segmento de red aislado utilizando el modo NAT en VirtualBox, lo que asigna una dirección IP privada accesible solo desde el dispositivo anfitrión y permite agregar más máquinas virtuales a la misma red según sea necesario.

*Ilustración 5: **Configuración del adaptador de red***

Financiado por la Unión Europea NextGenerationEU
GOBIERNO DE ESPAÑA MINISTERIO PARA LA TRANSFORMACIÓN DIGITAL Y DE LA FUNCIÓN PÚBLICA SECRETARÍA DE ESTADO DE TELECOMUNICACIONES E INFRAESTRUCTURAS DIGITALES
Plan de Recuperación, Transformación y Resiliencia
incibe_ INSTITUTO NACIONAL DE CIBERSEGURIDAD

6. Ejemplo de análisis de vulnerabilidades de escritura fuera de límites

En esta sección repasamos los conceptos importantes para entender la escritura fuera de límites en la práctica y su funcionamiento básico, a través de un ejemplo.

6.1. Entorno de test e identificación de la muestra

Desde el entorno Windows realizaremos las siguientes operaciones. Primero, descargamos el fichero brainpam.exe. Al ejecutarlo, observamos que se lanza un servidor a la escucha en el puerto 9999.

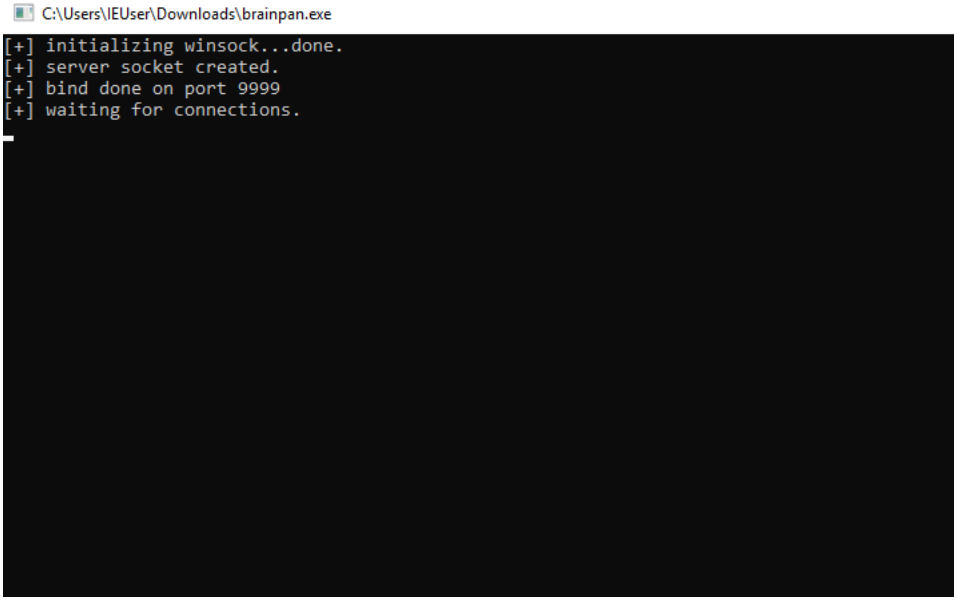


Ilustración 6: Ejecución de brainpam.exe a la escucha de peticiones

Podemos utilizar el comando `netstat` para confirmar que el servidor está activo en dicho puerto.

```
netstat -an | find "9999"
```

Vamos a verificar el archivo utilizando el comando "*strings*" de Sysinternals[19] sobre brainpan.exe. Dado que la salida nos devuelve el uso de funciones como strcpy[20] es muy posible que esta sea la causa de la vulnerabilidad de escritura fuera de límites.

Ahora lo analizaremos desde la aplicación Immunity Debugger. Para ello, lo abrimos, cargamos `brainpan.exe` y lo ejecutamos con `F9`. Esto nos permite monitorizar el comportamiento del programa en tiempo real. Desde Immunity Debugger podemos ver varias ventanas, entre ellas el código ensamblador del programa y la información de los registros de la CPU (ventana superior derecha). Por ejemplo, observando el valor del

[19] https://learn.microsoft.com/en-us/sysinternals/downloads/sysinternals-suite

[20] https://cwe.mitre.org/data/definitions/676

registro EIP, representado como un valor hexadecimal de 8 caracteres, 0x004017D0, concluimos que cada registro es de 32 bits (4 *bytes*), ya que 8 caracteres hexadecimales corresponden a 32 bits (cada carácter hexadecimal representa 4 bits). Dado el conjunto de instrucciones y el tamaño de los registros, podemos confirmar que el procesador en cuestión pertenece a la arquitectura x86 de 32 bits.

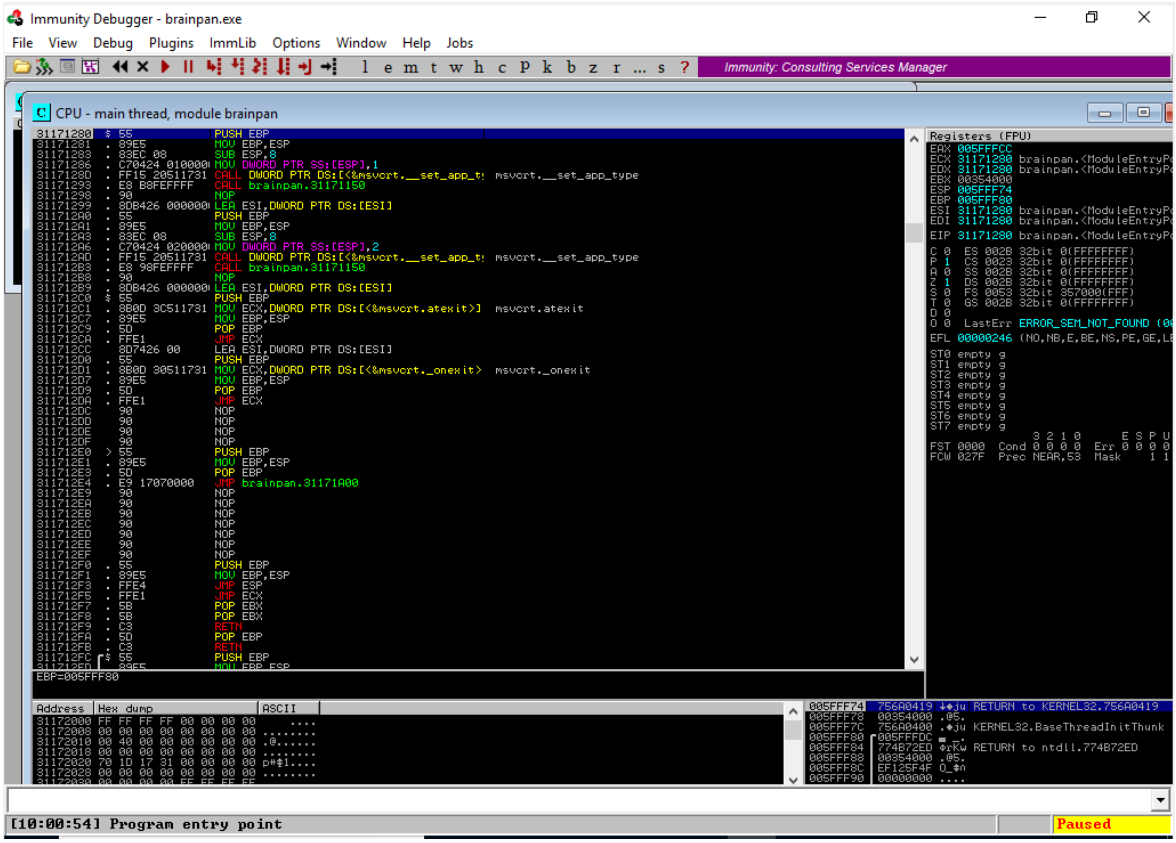


*Ilustración 7: **Entorno de Immunity Debugger***

6.2. Objetivo del análisis

El objetivo es determinar si se produce un desbordamiento de búfer de entrada de datos del servidor y se sobrescriben los registros. Si los datos enviados superan la capacidad del búfer, estos datos podrían sobrescribir la memoria adyacente, lo que puede incluir registros importantes de la CPU. El desbordamiento de búfer al que nos referimos generalmente ocurre en la pila (*stack*), en este contexto, el búfer es una variable local de una función que se almacena en la pila. Cuando hay un desbordamiento de este búfer, los datos adicionales sobrescriben los registros importantes como EBP y EIP. En la arquitectura x86, los registros EBP (*Extended Base Pointer*) y EIP (*Extended Instruction Pointer*) juegan roles cruciales en el control del flujo del programa y la gestión de la pila (stack). El registro EBP es utilizado principalmente para apuntar al inicio del marco de pila (*stack frame*) de la función actual y sirve como referencia para acceder a los parámetros de la función y las variables locales de manera más sencilla. El registro EIP es el puntero de instrucción extendido que siempre apunta a la siguiente instrucción que la CPU debe ejecutar y controla el flujo de ejecución del programa.

La sobreescritura con datos arbitrarios de relleno (*padding)* de estos registros suele ocasionar fallos en la ejecución del programa que se manifiestan como violaciones de

segmento con la interrupción del proceso. Sin embargo, si un atacante logra sobrescribir estos registros con datos coherentes y precisos, puede controlar el flujo de ejecución del programa y ejecutar código arbitrario. Esto se hace inyectando *shellcode* y redirigiendo EIP para que apunte a dicho *shellcode*, explotando así la vulnerabilidad de desbordamiento de búfer para obtener el control del sistema.

```
Dirección Alta
+------------------------------+
|             Stack            |
|   (Variables locales y       |
|   marcos de pila)            |
+------------------------------+
|        Datos del Búfer       | <- Búfer y otras variables locales
| AAAAAAAAAAAAAAAAAAAAAAAAAAAA |
| AAAAAAAAAAAAAAAAAAAAAAAAAAAA |
| AAAAAAAAAAAAAAAAAAAAAAAAAAAA |
| AAAAAAAAAAAAAAAAAAAAAAAAAAAA |
+------------------------------+
|             EBP              | <- EBP sobrescrito con "AAAA"
+------------------------------+
|             EIP              | <- EIP sobrescrito con "AAAA"
+------------------------------+
Dirección Baja
```

Ilustración 8: Representación de buffer overflow en la pila de memoria

6.3. Confirmación de la vulnerabilidad

Para confirmar la vulnerabilidad, se puede inyectar un flujo de datos muy grande en el búfer y observar el comportamiento del programa. Dado que no conocemos el tamaño exacto del búfer, inyectaremos 1000 bytes y observaremos el estado de los registros. Para ello, en la máquina Kali preparamos un *script* en Python que genere un *payload* de prueba con una cadena de 1000 caracteres (por ejemplo, con el carácter 'A'). Luego, lo enviamos a la aplicación objetivo y observamos el comportamiento del programa, así como el estado de los registros. Tras la ejecución del script, el contenido de los registros EBP y EIP han sido sobrescritos por el carácter ASCII 'A' (0x41 en hexadecimal), lo que **confirma la existencia de la vulnerabilidad**.

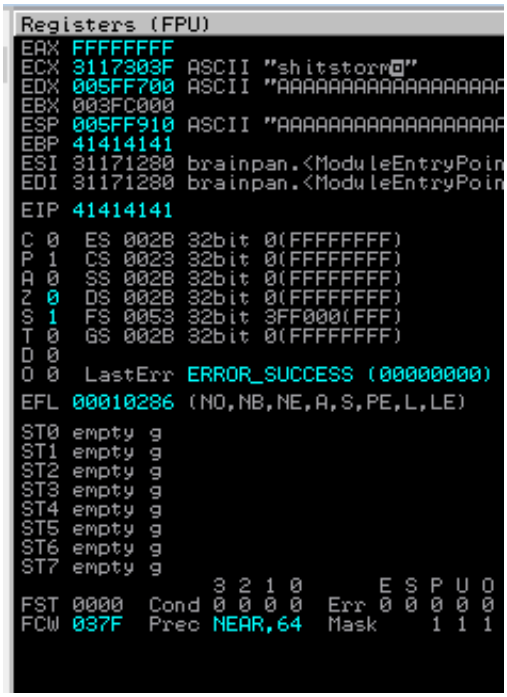


Ilustración 9: Valor de los registros con una entrada de 1000 'A'

6.4. Determinar el *offset* exacto

Determinar manualmente cuál es el byte que provoca el primer desbordamiento puede ser una tarea tediosa y propensa a errores. Por ello, los atacantes suelen utilizar patrones únicos de bytes para simplificar y agilizar este proceso.

En nuestro caso, este patrón debe ser lo suficientemente largo como para asegurarse de que sobrescriba el registro EIP. Herramientas como *pattern_create* de Metasploit facilitan para esta tarea, ya que generan una cadena de caracteres no repetitiva que facilita la identificación del offset exacto. El valor con el que se sobrescriba el registro será una subsección del patrón original y ayudará a identificar el offset exacto del desbordamiento.

```
┌──(kali㉿kali)-[~]
└─$ /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 1000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac
5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0A
f1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6
Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak
2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7A
m8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3
Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar
9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4A
u5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0
Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az
6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1B
c2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7
Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2B
```

Ilustración 10: Creación de un patrón de 1000 bytes con pattern_create

A continuación, se muestra el valor modificado de los registros tras sobrescribir sus valores legítimos. En el caso de EIP, toma el valor 0x35724134.

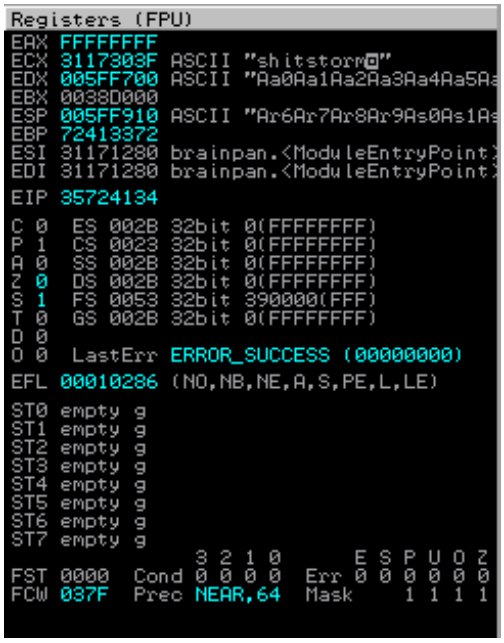


Ilustración 11: Valor de los registros con una entrada de 1000 bytes basada en patrón

Con el valor de EIP sobrescrito, podemos herramientas como *pattern_offset* de Metasploit para determinar la posición exacta del patrón que causó la sobrescritura. Esto permite calcular el número de bytes necesarios para alcanzar y sobrescribir el registro EIP con precisión.

```
┌──(kali㉿kali)-[~]
└─$ /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q 35724134
[*] Exact match at offset 524
```

Ilustración 12: Determinación del offset

Ahora validamos que podemos sobrescribir EIP a partir del *offset*, por ejemplo, con la cadena BBBB, que equivale a 0x42424242.

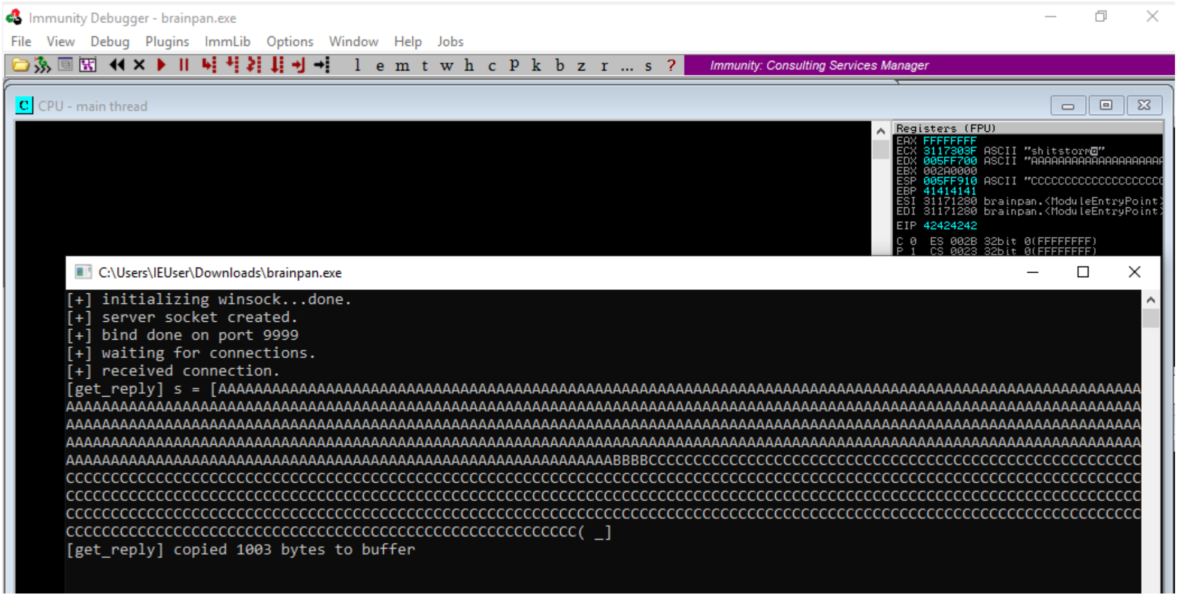


Ilustración 13: Rescritura selectiva del registro EIP con la cadena 'BBBB'

6.5. Inyección de *shellcode*

Cuando hablamos de inyectar un *shellcode*, nos referimos a insertar un pequeño fragmento de código que realiza una acción específica, como abrir una *shell* inversa o ejecutar un comando. Para que este *shellcode* se ejecute correctamente, necesitamos redirigir el flujo de ejecución del programa hacia él. Aquí es donde entra en juego el registro EIP. Para redirigir el EIP al *shellcode*, construimos un *payload* más complejo que incluye dos componentes esenciales:

- **NOP *sled***: aunque parezca que conocer el offset exacto es suficiente, la realidad de la memoria dinámica y la posibilidad de pequeñas variaciones en la ejecución del programa hacen que un NOP *sled* sea una práctica valiosa para asegurar el éxito del *exploit*. En muchos sistemas modernos, mecanismos de seguridad como ASLR cambian la disposición de la memoria con cada ejecución del programa, lo que hace difícil apuntar directamente al inicio del *shellcode*. Incluso sin mecanismos de seguridad, pueden ocurrir variaciones menores en la asignación de memoria debido a otras funciones o variables en la pila. El NOP *sled* proporciona un margen de error que incrementa significativamente las probabilidades de que el *shellcode* se ejecute correctamente, haciendo que la explotación sea más fiable y efectiva. Son una serie de instrucciones NOP (*No Operation*) que no hacen nada y simplemente permiten que el flujo de ejecución llegue hasta el *shellcode*. En ensamblador x86, la instrucción NOP se representa como 0x90.
- ***Shellcode***: es el código malicioso que se desea ejecutar. El *shellcode* debe estar bien diseñado para realizar la tarea deseada sin causar fallos en el programa. Este código se coloca después del NOP *sled*.

Para ganar control sobre dónde ubicamos el *shellcode*, primero sobrescribimos el registro EIP con la dirección de una instrucción `JMP ESP`. Esta instrucción le dice al procesador que salte a la dirección que actualmente apunta ESP, que es la primera dirección disponible en la pila. De esta manera, nos aseguramos de que el salto no interfiere con otras partes críticas del programa. Para maximizar las probabilidades de éxito y asegurar que la

ejecución del *shellcode* no falle debido a pequeñas variaciones en la memoria, introducimos el NOP *sled*. A continuación, y de forma adyacente, escribimos el *shellcode*.

```
Dirección Alta
+------------------------------+
|             Stack            |
|   (Variables locales y       |
|   marcos de pila)            |
+------------------------------+
|        Datos del Búfer       | <- Búfer y otras variables locales
| AAAAAAAAAAAAAAAAAAAAAAAAAAAA |
| AAAAAAAAAAAAAAAAAAAAAAAAAAAA |
| AAAAAAAAAAAAAAAAAAAAAAAAAAAA |
| AAAAAAAAAAAAAAAAAAAAAAAAAAAA |
|            EBP               | <- EBP sobrescrito con "AAAA"
+------------------------------+
|            EIP               | <- EIP sobrescrito direcc. JMP ESP
+------------------------------+
|          NOP Sled            | <- Instrucciones NOP (\x90)
| \x90\x90\x90\x90\x90\x90\x90 |
| \x90\x90\x90\x90\x90\x90\x90 |
+------------------------------+
|         Shellcode            | <- Código malicioso
| \xdb...                      |
| ...                          |
+------------------------------+
Dirección Baja
```

Ilustración 14: Representación de la explotación del buffer overflow con un shellcode

Para encontrar la instrucción JMP ESP en el programa, abrimos el depurador y hacemos clic derecho en la ventana del depurador, luego seleccionamos "*Search*" (Buscar) y después "*Command*" (Comando). Ingresamos JMP ESP como el comando de búsqueda y hacemos clic en "*Find*" (Buscar). Esto nos mostrará todas las ubicaciones en la memoria donde aparece la instrucción JMP ESP. Por ejemplo, podríamos encontrarla en la dirección 0x311712F3.

Una vez que tenemos la dirección de JMP ESP, cambiamos el registro EIP en nuestro script para que apunte a esta dirección. Es importante notar que cuando enviamos esta dirección al *stack*, debe ser introducida en *Little Endian* (orden inverso). Por lo tanto, la dirección se enviará como xf3\x12\x17\x31. Después del JMP ESP, nos aseguramos de que el espacio apuntado por ESP contenga nuestro NOP *sled* seguido del *shellcode*. El *shellcode* propiamente dicho puede generarse con Metasploit dentro de Kali. Sin embargo, la generación y los detalles específicos del mismo van más allá del propósito de esta guía, que se centra en el aprovechamiento de la vulnerabilidad de desbordamiento de búfer.

Financiado por la Unión Europea NextGenerationEU
GOBIERNO DE ESPAÑA MINISTERIO PARA LA TRANSFORMACIÓN DIGITAL Y DE LA FUNCIÓN PÚBLICA SECRETARÍA DE ESTADO DE TELECOMUNICACIONES E INFRAESTRUCTURAS DIGITALES
Plan de Recuperación, Transformación y Resiliencia
incibe_ INSTITUTO NACIONAL DE CIBERSEGURIDAD

7. Prevención y buenas prácticas

Prevenir y solucionar las vulnerabilidades de escritura fuera de límites implica conocer y considerar la aplicación de una serie de medidas. A continuación, repasamos algunas de ellas.

7.1. Configuración y desarrollo seguros

- **Mecanismos de protección**: para prevenir la escritura fuera de límites, es crucial implementar mecanismos de seguridad:
  - ASLR (*Address Space Layout Randomization*): ASLR es una característica del sistema operativo, habilitada de forma predeterminada en la mayoría de los sistemas modernos como Windows o Linux. Esta técnica, aleatoriza la ubicación de las áreas clave de la memoria del proceso, como el *stack*, *heap* y las librerías, dificultando la predicción de direcciones de memoria, haciendo más complicado explotar las vulnerabilidades de escritura fuera de límites.
  - *Stack Canaries*: se trata de valores conocidos que se colocan en la pila antes del retorno de la función. Si un desbordamiento de buffer modifica este valor, el programa puede detectar el cambio y abortar la ejecución, previniendo así ataques de escritura fuera de límites. En compiladores como GCC puede habilitarse con la opción *-fstack-protector* (que protege las funciones que declaran buffers locales mayores a 8 bytes) o *-fstack-protector-all* (que protege todas las funciones independientemente del tamaño del buffer.
  - Fortificación de fuente: hay compiladores que añaden la fortificación de fuente (como GCC con la opción D_FORTIFY_SOURCE=2 que realiza comprobaciones adicionales en tiempo de ejecución para algunas funciones estándar de C por ejemplo `strcpy`, `sprintf`, etc.).
  - *Control Flow Integrity* (CFI): es una técnica que asegura que el flujo de control de un programa sigue un camino predefinido. CFI puede prevenir que un atacante redirija la ejecución del programa a ubicaciones inesperadas. Puede ser implementado utilizando compiladores avanzados como Clang, que soportan esta característica.
  - *Shadow Stack*: es una copia separada de la pila de llamadas del programa que se utiliza para verificar la integridad del flujo de control. Cualquier discrepancia entre la pila principal y el *shadow stack* puede indicar un intento de explotación. Esta técnica suele requerir soporte del *hardware* o modificaciones en el sistema operativo y, actualmente, es más común en sistemas integrados y plataformas de seguridad avanzada.
  - DEP/NX (Data Execution Prevention/No-eXecute): es una característica de *hardware* y *software* que marca ciertas áreas de la memoria como no ejecutables. Esto previene que el código inyectado en estas áreas sea ejecutado. En Windows, DEP se puede habilitar desde las opciones de

configuración del sistema. En Linux, la mayoría de los sistemas modernos tienen NX habilitado.

- **Prácticas de programación segura**: la implementación de las mejores prácticas es fundamental para reducir al máximo la ocurrencia de estas vulnerabilidades. Algunas prácticas recomendadas incluyen:
  - Evitar funciones peligrosas: es fundamental evitar el uso de funciones peligrosas que no realizan comprobaciones de límites. En el caso de C/C++, funciones como *gets(), strcpy(), scanf(), y sprintf()* son propensas a vulnerabilidades de escritura fuera de límites. En su lugar, se deben utilizar funciones seguras como *fgets(), strncpy(), snprintf(), y scanf_s()* que permiten especificar el tamaño del buffer. El uso de una función segura, como *strncpy*, en lugar de *strcpy(),* hubiera sido suficiente en nuestro ejemplo, para desactivar la vulnerabilidad. Veamos un ejemplo de ello.

```
#include <stdio.h>
#include <string.h>

int main() {
    // Declarar un búfer de entrada y un búfer de destino
    char inputBuffer[100]; // Búfer de entrada
    char destino[50];  // Búfer de destino

    // Solicitar al usuario que ingrese una cadena
    printf("Por favor, ingrese una cadena: ");
    fgets(inputBuffer, sizeof(inputBuffer), stdin);

    // Copiar la cadena de entrada al búfer de destino
    strncpy(destino, inputBuffer, sizeof(destino) - 1);

    // Asegurarse que destino esté terminada en null
    destino[sizeof(destino) - 1] = '\0';

    // Imprimir la cadena de destino
    printf("Destino: %s\n", destino);

    return 0;
}
```

  - Validación y sanitización de los datos de entrada: todas las entradas de datos suministradas por el usuario deben ser validadas y sanitizadas rigurosamente. Veamos el siguiente control de tamaño de un buffer en javascript en la entrada de datos.

```
function safeCopy(inputData, bufferSize = 10) {
    if (inputData.length > bufferSize) {
        throw new Error("buffersize exceeded");
    }
    return inputData.slice(0, bufferSize);
}

try {
    let data = "This is a test string";
    console.log(safeCopy(data, 10));
} catch (e) {
    console.error(e.message);
}
```

- Programación segura de puntos críticos: es crucial identificar y asegurar los puntos críticos en el código donde se manipulan buffers y otros recursos de memoria. Implementar controles de límites y verificaciones antes de cualquier operación de escritura puede prevenir vulnerabilidades de escritura fuera de límites. Veamos el siguiente manejo de excepciones con un ejemplo en javascript.

```
let data = [1, 2, 3, 4, 5];

function safeAccess(index) {
    if (index >= 0 && index < data.length) {
        return data[index];
    } else {
        throw new Error("Index out of bounds");
    }
}

try {
    console.log(safeAccess(5));
} catch (e) {
    console.error(e.message);
}
```

7.2. Prácticas adicionales para la prevención

Además, pueden implementarse algunas técnicas adicionales, como:

- **Análisis exhaustivo en el SSDLC**: como vimos en la sección 4, el uso de herramientas y sondas de análisis es muy importante para detectar problemas de seguridad. Incluso durante su puesta en producción, el análisis en tiempo real para la detección de *payloads* ayuda identificar datos entrantes de carácter malicioso en el sistema y bloquear ataques antes de que puedan producirse. Utilizando una combinación de análisis basado en patrones y comportamientos, y respuestas automáticas, los sistemas pueden mantenerse más protegidos contra amenazas de desbordamientos causados por entradas de datos arbitrarios.
- **Pruebas de penetración**: pueden ayudar a identificar y mitigar vulnerabilidades de escritura fuera de límites que podrían haber pasado desapercibidas durante el desarrollo y las pruebas iniciales. Estas pruebas simulan ataques reales durante la vida en producción del sistema y proporcionan una visión práctica de las debilidades de seguridad de la aplicación.
- **Arquitectura de sistemas**: la renovación de CPU antiguas de 32 bits por las de 64 bits pueden tener un impacto significativo en la seguridad general del sistema además de en sus prestaciones. Los procesadores de 64 bits ofrecen un espacio de direcciones mucho más amplio, lo que dificulta que los atacantes predigan y exploten vulnerabilidades de desbordamiento de memoria. Esto mejora la efectividad de técnicas de mitigación como ASLR y *stack canaries*, que se benefician enormemente del mayor espacio de direcciones disponible.
- **Ejecutar con menores privilegios:** si un atacante explota una vulnerabilidad de desbordamiento de búfer en un programa que se ejecuta con privilegios mínimos, el daño potencial se limita a lo que esos privilegios permiten. Por ejemplo, si una aplicación comprometida solo tiene acceso de lectura a ciertos archivos, el atacante no podrá modificar ni eliminar esos archivos, incluso si logra ejecutar código

malicioso. Ejecutar programas con los menores privilegios necesarios puede limitar el impacto de una explotación exitosa.

- **Formación y capacitación en seguridad**: concienciar y formar a los empleados es esencial para reducir los riesgos de seguridad. A diferencia de otros tipos de vulnerabilidades, la escritura fuera de los límites exige comprender los principios de las arquitecturas de computadores. La formación continua en este campo y la adopción de una cultura de seguridad en el desarrollo de *software* tendrá un impacto positivo en la robustez de los sistemas y en la mitigación de vulnerabilidades.

8. Conclusión

Mejorar la seguridad de las aplicaciones implica centrarse en la prevención, detección y corrección de **una de las vulnerabilidades más importantes que existen: la escritura fuera de límites**.

La protección de los sistemas de software es responsabilidad de todos los profesionales involucrados en su desarrollo y mantenimiento, desde el diseño inicial hasta la implementación y evaluación final. Los arquitectos de *software* y *hardware* deben diseñar sistemas con medidas de seguridad robustas desde el principio, eligiendo lenguajes de programación seguros y configurando el *hardware* adecuadamente. Los desarrolladores deben aplicar prácticas de desarrollo seguro, utilizando herramientas de análisis de código para identificar y corregir vulnerabilidades. Los implementadores tienen la responsabilidad de asegurar que las configuraciones de seguridad se apliquen correctamente durante la integración y el despliegue del *software*. Finalmente, los evaluadores y auditores deben revisar y evaluar el *software* y la infraestructura mediante pruebas de penetración y auditorías de seguridad para detectar posibles vulnerabilidades que incluyan pruebas de escritura fuera de los límites permitidos. Cada uno de estos roles es esencial para asegurar que las medidas de seguridad sean efectivas y robustas, y su colaboración es fundamental para crear sistemas seguros y protegidos contra amenazas.

Implementar soluciones que sean seguras y eficientes es fundamental para el éxito y la sostenibilidad continuos de las aplicaciones y sistemas en el entorno digital moderno. Un desafío común en la implementación de soluciones seguras es el impacto potencial en el rendimiento del sistema. Sin embargo, es fundamental que las medidas de seguridad no se vean como un compromiso, sino como una parte integral del diseño del sistema. Aunque ciertas medidas pueden afectar al rendimiento, los productos deben ser seguros para proteger la integridad, confidencialidad y disponibilidad de los datos y servicios.

La nueva Reglamentación Europea de Ciberresiliencia (CRA)[21] exigirá a los productos *software* y *hardware* un compromiso con los estándares de seguridad establecidos, proporcionando **confianza tanto a los usuarios finales como a los operadores de los mismos**. A término, el objetivo es, junto con otras directivas como NIS2[22], fomentar una cultura de seguridad más amplia y robusta en toda la Unión Europea.

Está por ver como la nueva reglamentación afecte a la exigencia en los estándares de seguridad en los requisitos de desarrollo y mantenimiento de productos, pero seguro que servirá para mejorar la protección de vulnerabilidades. La clave, como casi siempre, estará en encontrar un equilibrio óptimo que permita mantener altos estándares de seguridad sin sacrificar la eficiencia operativa. Esto puede lograrse mediante la adopción de tecnologías avanzadas, la mejora continua de las habilidades del equipo y **una postura proactiva ante las ciberamenazas**.

[21] https://digital-strategy.ec.europa.eu/es/policies/cyber-resilience-act

[22] https://digital-strategy.ec.europa.eu/es/policies/nis2-directive

9. Acrónimos

- **ALU**: Arithmetic Logic Unit
- **ASLR**: Address Space Layout Randomization
- **CFI**: Control Flow Integrity
- **CPU**: Central Processing Unit
- **CRS**: Core Rule Set
- **CWE**: Common Weakness Enumeration
- **DAST**: Dynamic Application Security Testing
- **DEP**: Data Execution Prevention
- **EBP**: Extended Base Pointer
- **EIP**: Extended Instruction Pointer
- **ESP**: Extended Stack Pointer
- **GCC**: GNU Compiler Collection
- **GDB**: GNU Debugger
- **IP**: Instruction Pointer
- **NX**: No-eXecute
- **OWASP**: Open Web Application Security Project
- **RAM**: Random Access Memory
- **SAST**: Static Application Security Testing
- **SSDLC**: Secure Software Development Life Cycle
- **TCP**: Transmission Control Protocol
- **WAF**: Web Application Firewall

10. Bibliografía

Referencia - Título, autor, fecha y enlace web

[Ref.- 1] *Testing for Buffer Overflow*, OWASP, https://owasp.org/www-project-web-security-testing-guide/v41/4-Web_Application_Security_Testing/07-Input_Validation_Testing/13-Testing_for_Buffer_Overflow

[Ref.- 2] *CAPEC-100: Overflow Buffers*, MITRE, https://capec.mitre.org/data/definitions/100

[Ref.- 3] *Buffer Overflow Attack*, OWASP, https://owasp.org/www-community/attacks/Buffer_overflow_attack

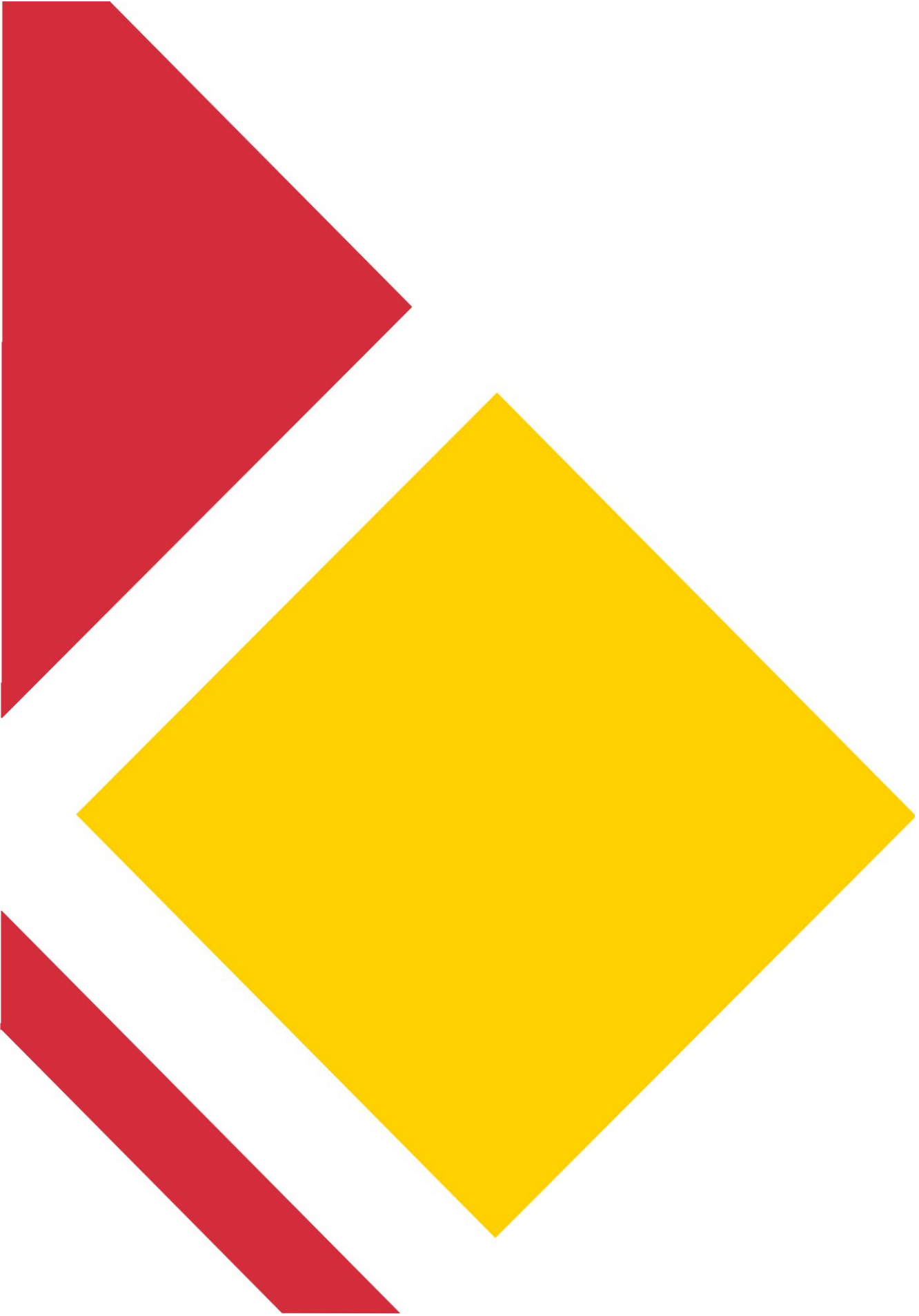
[Ref.- 4] *Computer Organization and Design: The Hardware/Software Interface, David A. Patterson, Morgan Kaufmann*, https://books.google.es/books/about/Computer_Organization_and_Design.html?id=3b63x-0P3_UC&redir_esc=y

[Ref.- 5] *CWE-787: Out-of-bounds Write, MITRE,* https://cwe.mitre.org/data/definitions/787.html

[Ref.- 6] *Bypassing Browser Memory Protections: Setting back browser security by 10 years*, Alexander Sotirov and Mark Dowd, Black Hat 2008, https://www.blackhat.com/presentations/bh-usa-08/Sotirov_Dowd/bh08-sotirov-dowd.pdf

[Ref.- 7] *Writing Secure Code*, Michael Howard y David LeBlanc, https://www.microsoftpressstore.com/store/writing-secure-code-9780735617223

[Ref.- 8] *On the effectiveness of nx, ssp, renewssp, and aslr against stack buffer overflows, Hector Marco y otros,* 2014 IEEE 13th International Symposium on Network Computing and Applications http://hmarco.org/renewssp/data/OnTheEffectivness-NX-SSP-RenewSSP-and-ASLR-Slides.pdf



Financiado por la Unión Europea
NextGenerationEU

GOBIERNO DE ESPAÑA
MINISTERIO PARA LA TRANSFORMACIÓN DIGITAL Y DE LA FUNCIÓN PÚBLICA
SECRETARÍA DE ESTADO DE DIGITALIZACIÓN E INTELIGENCIA ARTIFICIAL

Plan de Recuperación, Transformación y Resiliencia

incibe_
INSTITUTO NACIONAL DE CIBERSEGURIDAD