



TRABAJO FIN DE GRADO
INGENIERÍA EN INFORMÁTICA

Análisis del uso de desinfectantes y técnicas de fuzzing para la detección de vulnerabilidades en software

Autor

José Luis París Reyes

Director

Gustavo Romero López



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE
TELECOMUNICACIÓN

Granada, Julio de 2022

Análisis del uso de desinfectantes y técnicas de fuzzing para la detección de vulnerabilidades en software

José Luis París Reyes

Palabras clave: análisis estático de código, análisis dinámico de código, fuzzing, desinfectantes, afl++, automatización, Linux, análisis de vulnerabilidades, GCC, CLANG

Resumen

Este trabajo consiste en el uso de análisis estático y dinámico junto a técnicas de fuzzing para encontrar vulnerabilidades en binarios. Con la finalidad de demostrar lo eficaz que puede llegar a ser su uso. Se han realizado las siguientes actividades:

1. Encontrar el software objeto de estudio.
2. Realizar análisis estático, análisis dinámico y técnicas de fuzzing sobre este software.
3. Análisis de los informes generados en las pruebas.

Analysis of the use of sanitizers and fuzzing techniques for the detection of vulnerabilities in software

José Luis París Reyes

Keywords: static code analysis, dynamic code analysis, fuzzing, sanitizers, afl++, automation, Linux, vulnerability scan, GCC, CLANG

Abstract

This project covers the use of static code analysis, dynamic code analysis and fuzzing techniques with the purpose of finding vulnerabilities in binary programs. In order to demonstrate how effective its use can be. The following steps has been performed:

1. Find the software to be studied.
2. Perform static code analysis, dynamic code analysis and fuzzing over this software.
3. Analyze the generated reports.

Yo, **José Luis París Reyes**, alumno de la titulación Ingeniería Informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 77650489M, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: José Luis París Reyes

Granada a 7 de Julio de 2022

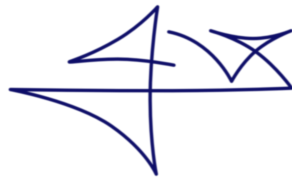
D. **Gustavo Romero López**, Profesor del Departamento de Arquitectura y Tecnología de Computadores de la Universidad de Granada.

Informa:

Que el presente trabajo, titulado *Análisis del uso de desinfectantes y técnicas de fuzzing para la detección de vulnerabilidades en software.*, ha sido realizado bajo su supervisión por **José Luis París Reyes**, y autoriza la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 7 de Julio de 2022 .

El director:

A handwritten signature in blue ink, consisting of stylized, overlapping loops and curves.

Gustavo Romero López

Agradecimientos

A mi familia, por apoyarme en todos los retos que he afrontado a día de hoy y, en especial, a mi madre María José, por su entereza ante las adversidades que han aparecido en su vida. Me ha enseñado a no rendirme y a dar lo mejor de mí mismo.

A mis abuelos, José y Ana, por haber ayudado a mi familia en todo lo que han podido y más. No podría haber llegado hasta aquí sin ellos.

A aquellos profesores que aman su trabajo y motivan a sus alumnos a seguir aprendiendo. Ojalá todo el personal docente fuese como ellos.

Gracias a todos.

Índice general

1. Introducción	9
1.1. Motivación	9
1.2. Contexto	10
1.3. Metodología	11
1.4. Estructura	11
2. Especificación de requisitos	13
2.1. Planificación temporal	13
2.1.1. Estado del arte del trabajo	13
2.1.2. Análisis	14
2.1.3. Diseño e implementación	14
2.1.4. Pruebas	14
2.1.5. Redacción del documento de memoria	15
2.2. Recursos	16
2.2.1. Recursos humanos	16
2.2.2. Recursos hardware	16
2.2.3. Recursos software	17
2.3. Presupuesto	17
3. Estado del arte	19
3.1. Análisis estático	19
3.1.1. GCC	20
3.1.2. CLANG	21
3.1.3. Ventajas y desventajas	23
3.2. Sanitizers	23
3.2.1. Address Sanitizer	25
3.2.2. Thread Sanitizer	28
3.2.3. Memory Sanitizer	29
3.2.4. Leak Sanitizer	30
3.2.5. Undefined Behavior Sanitizer	31
3.2.6. Ventajas y desventajas	31
3.3. Fuzzing	33
3.3.1. Categorías de testeo	34

3.3.2. Tipos de técnicas	34
3.3.3. AFL++	36
4. Análisis	41
4.1. Paquetes de la distribución Linux	41
4.2. Software a analizar	42
4.3. Configuración del entorno de compilación	42
4.4. Informes generados	43
4.5. Fuzzing	44
5. Diseño e implementación	45
5.1. Automatización	45
5.2. Elementos de automatización	46
5.2.1. Descarga y obtención de ficheros de compilación del paquete	46
5.2.2. Directorios generados por asp	47
5.2.3. Generación del archivo Makefile y compilación de bi- narios	47
5.2.4. Generación de informes de datos en base a los tiempos de compilación y ejecución	48
5.2.5. Configuración del software de fuzzing para inicio y fi- nalización de las pruebas	49
6. Pruebas	51
6.1. Análisis estático	51
6.1.1. Cronie	53
6.1.2. Sudo	57
6.1.3. Su	59
6.1.4. Chsh	61
6.1.5. Chfn	61
6.1.6. Mount	63
6.1.7. Umount	66
6.1.8. Paquete shadow	67
6.2. Análisis dinámico	68
6.2.1. AddressSanitizer	69
6.2.2. MemorySanitizer	71
6.2.3. LeakSanitizer	72
6.3. Fuzzing	73
6.3.1. Bash	74
6.3.2. Sudo	75
6.3.3. Tcpdump	81

ÍNDICE GENERAL	4
<hr/>	
7. Conclusiones	84
7.1. Trabajo realizado	84
7.2. Trabajo futuro	85
Bibliografía	89
Acrónimos	90

Índice de figuras

1.1. Distribución de la gravedad de las vulnerabilidades registradas entre 2001 y 2022	10
2.1. Diagrama de Gantt	16
3.1. Análisis estático de un programa C con doble liberación de puntero usando GCC	21
3.2. Análisis estático de un programa C con doble liberación de puntero usando CLANG	22
3.3. Tiempo de compilación normal y análisis estático en segundos	24
3.4. Desinfección de un programa C con puntero no liberado	26
3.5. Desinfección de un programa C con puntero liberado antes de su llamada	27
3.6. Desinfección de un programa C con condición de carrera	29
3.7. Desinfección de un programa C++ con memoria no inicializada	30
3.8. Desinfección de un programa C++ con fuga de memoria	30
3.9. Proporción del uso de memoria del programa instrumentado con Address Sanitizer y Memory Sanitizer frente al programa original	32
3.10. Proporción del uso de memoria del programa instrumentado con Thread Sanitizer frente al programa original	32
3.11. Ejecución de AFL++	37
3.12. Ejecución paralela de AFL++ con 4 procesos	38
3.13. Mensaje de error de AFL++	39
4.1. Ejemplo del reporte del analizador estático de CLANG	43
5.1. Directorios generados por la utilidad asp	47
6.1. Informe de allocator sizeof operand mismatch en el paquete cronie	53
6.2. Informe de argument with "nonnull" attribute passed null en el paquete cronie	54
6.3. Informe de dead assignment en el paquete cronie	55

6.4. Informe de dead increment en el paquete cronie	55
6.5. Informe de dead nested assignment en el paquete cronie . . .	56
6.6. Informe de memory error en el paquete cronie	57
6.7. Informe de argument with "nonnull" attribute passed null en el paquete sudo	58
6.8. Informe 1 de result of operation is garbage or undefined en programa su	60
6.9. Informe 2 de result of operation is garbage or undefined en programa su	61
6.10. Informe de memory leak en programa chfn	62
6.11. Informe 1 de use after-free en programa mount	64
6.12. Informe 2 de use after-free en programa mount	65
6.13. Informe 3 de use after-free en programa mount	65
6.14. Informe de dereference of null pointer en programa mount . .	66
6.15. Informe 1 de uninitialized argument value en programa gpasswd	67
6.16. Informe 2 de uninitialized argument value en programa gpasswd	68
6.17. Programa passwd con sanitizer AddressSanitizer	69
6.18. Programa gpasswd con sanitizer AddressSanitizer	70
6.19. Programa expiry con sanitizer AddressSanitizer	71
6.20. Programa su con sanitizer MemorySanitizer	72
6.21. Programa passwd con sanitizer LeakSanitizer	72
6.22. Programa gpasswd con sanitizer LeakSanitizer	73
6.23. Ejecución de AFL++ sobre el binario bash	74
6.24. Crash en el binario bash	75
6.25. Depuración con gdb del binario bash	75
6.26. Volcado hexadecimal con xxd de un informe del binario sudo	76
6.27. Resultado de la minimización de un volcado hexadecimal para el binario sudo	77
6.28. Depuración con gdb del binario sudo	78
6.29. Traza del programa gdb sobre el binario sudo	78
6.30. Buffer Overflow en el binario sudo	80
6.31. Ejecución de afl++ sobre el binario tcpdump	82
6.32. Resultado de la minimización de un volcado hexadecimal para el binario tcpdump	82
6.33. Depuración con gdb del binario tcpdump	83
6.34. Reporte de los sanitizers en el binario tcpdump	83

Índice de cuadros

2.1. Planificación temporal del proyecto	15
2.2. Presupuesto del proyecto	18
5.1. Binarios con bit SUID activado y creador root	46
6.1. Errores encontrados en los binarios analizados 1	52
6.2. Errores encontrados en los binarios analizados 2	52

Listings

3.1.	Programa C con doble liberación de puntero	21
3.2.	Programa C con puntero no liberado	25
3.3.	Programa C con puntero liberado antes de su llamada	26
3.4.	Programa C con puntero liberado antes de su llamada	28
3.5.	Programa C con hebras de escritura simultánea	29
3.6.	Programa C++ con lectura de memoria sin escritura	30
3.7.	Instrumentación de binario de línea de comandos con AFL++	40
4.1.	Gestor de paquetes pacman	41
4.2.	Uso de la herramienta asp	42
4.3.	Orden para recuperar los archivos de código fuente del paquete	42
4.4.	Búsqueda de archivos con el bit SUID activado	42
4.5.	Descarga de archivos de compilación de los paquetes	43
5.1.	Configuración de la variable de entorno para instrumentación	47
5.2.	Descarga de archivos con el código fuente de cada paquete	47
5.3.	Automatización de la orquestación y compilación de binarios	48
5.4.	Script para medir el tiempo de ejecución o compilación de un programa	49
5.5.	Script para realizar la configuración inicial de AFL++	49
5.6.	Script para restablecer la configuración inicial tras las pruebas	49
6.1.	Uso de afl-tmin para limpiar basura del caso de prueba	76
6.2.	Comprobación del error en el binario sudo	77
6.3.	Compilación del binario sudo con AddressSanitizer y UndefinedSanitizer	79
6.4.	Captura de paquetes con tcpdump y volcado en fichero de salida	81
6.5.	Ejecución de afl++ para el binario tcpdump	81
7.1.	Configuración de la variable de entorno para instrumentación	86

Capítulo 1

Introducción

1.1. Motivación

La evolución de las tecnologías ha propiciado un gran avance para el ser humano, desde los primeros circuitos electrónicos hasta el supercomputador más complejo, capaz de hacer cálculos matemáticos enormes en tiempo inapreciable, identificación de patrones en ADN o predicción de fenómenos meteorológicos, entre otras cosas.

Toda esta tecnología se compone a grandes rasgos de un hardware y un software que le permite funcionar correctamente y satisfacer las necesidades de los usuarios. Dicha evolución ha propiciado la sofisticación de estos dos componentes, provocando que se vuelvan lo suficientemente complejos como para que no se puedan tener en cuenta todas las causalísticas en las que pueden fallar.

Pese al continuo desarrollo y uso de herramientas para la detección prematura de fallos, siguen existiendo distintas vulnerabilidades que afectan a la mayoría del hardware y software usado a diario. Puesto que la mayoría de estas vulnerabilidades se producen sobre el componente del software, este trabajo abordará principalmente la seguridad del mismo desde dos puntos de vista diferentes, en esencia es una analogía al ataque y defensa que podemos encontrar en la ciberseguridad.

Por un lado, se demostrará cómo el uso de desinfectantes y análisis estático de código permite la detección de diferentes vulnerabilidades en nuestro software teniendo acceso al código del mismo, es decir, desde el punto de vista de un desarrollador que quiere proteger su código. Por otro lado, se usarán las técnicas de fuzzing más famosas para mostrar el punto de vista de una persona que busca encontrar una brecha de seguridad en una aplicación de la que desconoce su funcionamiento interno, aunque en este caso se usará

sobre aplicaciones de las que se disponga el código fuente, principalmente para conseguir mejores resultados.

1.2. Contexto

Solo en el año 2021 se reportaron más de 20.000 vulnerabilidades, frente a las más de 18.000 que se reportaron en el año anterior. Si se siguen comprobando los datos para años anteriores, se puede ver cómo se ha incrementado cada año el número de vulnerabilidades que se registran desde el principio de siglo. Debido a la continua expansión que sufre la tecnología y propiciado por la pandemia que sufrió el mundo durante los años 2019 y 2020, el número de vulnerabilidades registradas en los últimos 5 años ha crecido rápidamente.

En el análisis elaborado por el NIST (*National Institute of Standards and Technology*) de los Estados Unidos (ver Figura 1.1) [24], se puede comprobar el aumento del número de vulnerabilidades desde el año 2001 hasta el periodo actual del año 2022 en el CVSS (*Common Vulnerability Scoring System*). Estas vulnerabilidades están diferenciadas en 3 niveles: bajo, medio y alto, en función de su riesgo.

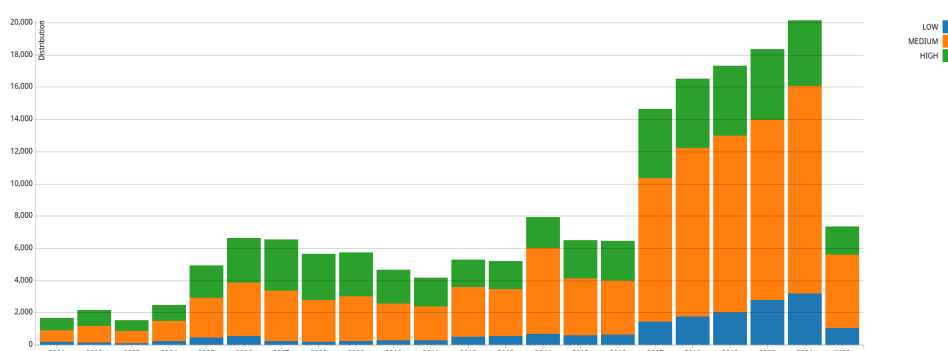


Figura 1.1: Distribución de la gravedad de las vulnerabilidades registradas entre 2001 y 2022

El principal problema reside en el hecho de que las empresas y desarrolladores no dedican el suficiente tiempo a comprobar si su código es seguro, por ello, en este proyecto se mostrarán tres de las técnicas más comunes para comprobar la seguridad de un producto. Se trata del uso de análisis estático y desinfectantes en el compilado del código además de técnicas de fuzzing que se realizan durante la ejecución de un programa.

1.3. Metodología

El primer paso es encontrar aquel software objetivo de estudio, principalmente se usarán binarios propios de la distribución de Linux. Después, se debe proceder a descargar el código fuente de estas utilidades.

Tras conseguir los ficheros que posteriormente se compilarán para conseguir los binarios requeridos, se debe configurar el compilador de forma que realice un análisis estático o dinámico del código, a petición del usuario. Esto permitirá la generación de informes que posteriormente serán analizados en busca de brechas de seguridad o falsos positivos.

Por último, se aplicarán técnicas de fuzzing sobre estos binarios generados en busca de más vulnerabilidades, para ello se usará principalmente el famoso software de fuzzing *AFL++* [30].

1.4. Estructura

A continuación, se mostrará la estructura que seguirá el proyecto, y por tanto, la documentación del mismo:

1. **Introducción**

Se realiza la presentación del trabajo que se va a elaborar así como la motivación que propició su desarrollo.

2. **Especificación de requisitos**

Se muestra la planificación temporal que seguirá el proyecto, detallando el tiempo invertido en cada sección. Además, se muestran los recursos necesarios para la realización del trabajo y el presupuesto aproximado del que se requiere.

3. **Estado del arte**

En esta sección se hace una revisión bibliográfica sobre todas las herramientas y técnicas que abordarán el proyecto. Por un lado el análisis estático y dinámico, compiladores y desinfectantes usados, así como ventajas y desventajas de su uso. Por otro lado, se muestran las técnicas de fuzzing, finalizando con la explicación del software de fuzzing que se usará principalmente, *AFL++*.

4. **Análisis**

Se analizan los pasos que se deben seguir para realizar el proyecto y los problemas que se puedan generar. A estos problemas se plantearán soluciones en la siguiente sección.

5. Diseño e implementación

Tras haber realizado el análisis de los problemas a resolver en el anterior capítulo, se plantean soluciones, se diseñan e implementan.

6. Pruebas

Se realiza el análisis estático y dinámico, así como las técnicas de fuzzing sobre los binarios que son objeto de estudio. Además, se recaban informes que también son analizados en esta sección.

7. Conclusiones

Se plantea un breve resumen del trabajo y se muestran las conclusiones a las que se han llegado tras haber realizado todas las pruebas y análisis de informes anteriores.

Capítulo 2

Especificación de requisitos

En este capítulo se detallará la planificación temporal seguida para la realización del proyecto, así como los recursos utilizados y presupuesto del mismo.

2.1. Planificación temporal

Se recoge aquí la planificación temporal llevada a cabo durante el desarrollo del proyecto. Esta planificación se ha dividido en secciones en las que se resumen las acciones realizadas. Más tarde se representará el tiempo empleado en cada sección en un diagrama de Gantt.

2.1.1. Estado del arte del trabajo

En esta primera sección se recoge toda la información que ha permitido el desarrollo del proyecto. Además, se encuentran la mayoría de referencias bibliográficas del mismo.

Toda esta información se divide principalmente en 3 subsecciones que se exponen a continuación.

Análisis estático

Expone la definición de este tipo de análisis, compiladores que lo tienen completamente integrado y las ventajas y desventajas de su utilización. Además, se añaden unos ejemplos simples que permiten ver su funcionamiento.

Análisis dinámico

Esta es la parte más extensa dentro de la sección del Estado del Arte. Al igual que en el análisis estático, se muestran las ventajas y desventajas

de este tipo de análisis y, también se explican diferentes desinfectantes y se añaden ejemplos en los que se expone su funcionamiento.

Fuzzing

La última parte del trabajo consiste en la aplicación de técnicas de *fuzzing* sobre binarios, por lo que este apartado recoge toda la información referente a las mismas, así como la documentación y motivos de uso del software de *fuzzing* elegido para las pruebas.

2.1.2. Análisis

Se analiza el problema a resolver y se divide en diferentes pasos que se deben completar para conseguirlo. Antes de empezar a trabajar en el problema se debe haber dividido correctamente en las secciones que se deben completar para resolverlo, de forma que se tenga una visión general del problema.

2.1.3. Diseño e implementación

Se diseñan los procesos que permiten completar de forma satisfactoria las partes del problema examinadas en el capítulo anterior. Principalmente, se han automatizado todos los procesos que eran susceptibles de ello para disminuir el tiempo invertido en tareas tediosas y se ha mostrado su implementación.

2.1.4. Pruebas

Esta sección ha sido la que más tiempo ha requerido del proyecto, pero afortunadamente parte de ella se podía realizar de forma paralela con otras tareas, por lo que se podía combinar, por ejemplo, la revisión del Estado del arte junto a la ejecución de pruebas de *fuzzing*.

El principal problema que radica en esta parte es que el software de *fuzzing* puede ejecutarse de forma indefinida, que suele traducirse en horas de ejecución hasta que se encuentran fallos en el programa o puede no arrojar ningún resultado válido.

La estrategia principal ha sido ejecutar estas técnicas de *fuzzing* desde una parte temprana del proyecto mientras se trabajaba en otras partes.

2.1.5. Redacción del documento de memoria

Este es el periodo empleado para redactar la memoria del trabajo con todas las secciones anteriores. Además, se añade el capítulo de la introducción y las conclusiones a las que se ha llegado tras haber realizado todo el proyecto.

La redacción del documento ha estado presente desde una fase temprana del desarrollo del proyecto, y ha sido lo último en completarse, por lo que ha sido la sección que más tiempo ha permanecido sin finalizar, aunque no es la que más horas efectivas ha requerido.

El siguiente cuadro permite visualizar con mayor detalle el tiempo invertido en cada sección del proyecto (ver Cuadro 2.1). Aunque la fase del Estado del arte fue la primera en empezarse, ha estado abierta durante un periodo grande de tiempo debido a que se ha ido añadiendo contenido a la vez que se trabajaba en otras secciones.

Cuadro 2.1: Planificación temporal del proyecto

Sección	Inicio	Fin	Duración (días)
S1	05/02/2022	02/06/2022	117
S1.1	05/02/2022	03/04/2022	57
S1.2	05/03/2022	10/05/2022	66
S1.3	20/04/2022	02/06/2022	43
S2	05/04/2022	25/04/2022	20
S3	25/04/2022	20/05/2022	25
S4	25/04/2022	25/06/2022	61
S5	12/02/2022	05/07/2022	143
Total	150		

A continuación, se muestra un Diagrama de Gantt (ver Figura 2.1) en el que se muestra la planificación temporal que tiene el proyecto, desde su inicio hasta la entrega de este documento.

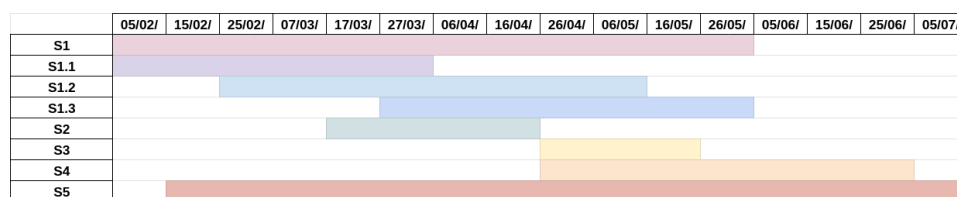


Figura 2.1: Diagrama de Gantt

2.2. Recursos

Se diferencian 3 tipos de recursos que han estado presentes durante todo el trabajo: hardware, software y humanos.

2.2.1. Recursos humanos

Se trata principalmente de aquellas personas involucradas en el desarrollo del proyecto. Se pueden diferenciar dos figuras:

- **José Luis París Reyes**, alumno del Grado en Ingeniería Informática por la Universidad de Granada, al que corresponde la autoría de este mismo documento.
- **D. Gustavo Romero López**, profesor perteneciente al departamento de Arquitectura y Tecnología de Computadores, como tutor del proyecto, cuya funcionalidad principal ha sido la de dirigir al estudiante para la investigación y estructura del mismo.

2.2.2. Recursos hardware

En esta sección se recoge información relevante sobre los dispositivos que se han usado para la investigación:

- **Ordenador de sobremesa personal**, en el que se han realizado todas las pruebas de análisis y ejecución de fuzzing, puesto que era el dispositivo con mayor potencia del que se disponía. Las características principales que influyen en la eficiencia de la ejecución de programas se muestran a continuación:
 - **Procesador:** Ryzen 5 5600X - 3700GHz - 6 Núcleos / 12 Hilos
 - **Tarjeta Gráfica:** Nvidia RTX 3070 Ti
 - **Memoria RAM:** 4x8GB DDR4 - 3200 MHz
- **Ordenador portátil personal**, usado con poca frecuencia y solo en caso de desplazamiento. La funcionalidad principal ha sido la conexión mediante SSH al ordenador principal para seguir trabajando en el

proyecto. Puesto que no se ha usado como dispositivo principal para la ejecución de pruebas, no se detallarán sus características.

- **Conexión de alta velocidad a internet**, en todo momento se ha dispuesto de una conexión de fibra óptica con velocidad de 1Gbps a internet, principalmente para la consulta bibliográfica, descarga y subida de ficheros desde la máquina.

2.2.3. Recursos software

Se trata de aquel software del que se ha hecho uso y que será ampliado con mayor detalle en el capítulo 3.

- **Sistema Operativo Linux**, ambos dispositivos (ordenador de sobremesa y ordenador portátil) tienen instalada una distribución de Linux basada en Arch Linux, en este caso la distribución Manjaro. Sobre este sistema operativo se han instalado todos los programas y se han ejecutado todas las pruebas.
- **Compiladores con integración de desinfectantes**, que se usarán para proteger nuestro código de diferentes vulnerabilidades, se ampliará más información en el capítulo 3.
- **Software de fuzzing**, con la finalidad de encontrar fallos en una aplicación, se ampliará más información en el capítulo 3.
- **Lenguajes de programación**, principalmente de scripting, para la automatización y ejecución de pruebas.

2.3. Presupuesto

Una vez detallados los recursos de los que se dispone, así como de la planificación temporal, se puede calcular el presupuesto del desarrollo de este proyecto (ver Cuadro 2.2).

El coste del hardware es aproximado y se estima en un alquiler por hora, aunque puede variar debido a las prestaciones que tenga el propio hardware. Tras haber completado las pruebas no sería necesaria la posterior utilización del equipo.

El autor ha trabajado una media de 2.5 horas al día durante 150 días para la realización de este proyecto, a un precio estimado de 10€ por hora. Puesto que no se tenía experiencia previa en el análisis de binarios y técnicas de *fuzzing*, la realización de las pruebas y análisis de informes ha requerido más tiempo del esperado.

Cuadro 2.2: Presupuesto del proyecto

Recurso	Coste	Tiempo	Coste total
Trabajo realizado por el autor	10€	375h	3.750€
Trabajo realizado por el tutor	30€	12h	360€
Alquiler de ordenador de sobremesa personal	5€	375h	1.875€
Alquiler de ordenador portátil personal	3€	50h	150€
Conexión de alta velocidad a internet	30€	6m	180€
Sistema Operativo Linux	0€	-	0€
Compiladores con integración de desinfectantes	0€	-	0€
Software de fuzzing	0€	-	0€
Lenguaje de programación	0€	-	0€
Total			6.315€

Por otro lado, se ha tenido en cuenta el trabajo del tutor, a un precio aproximado de 30€ por hora, durante unas 12 horas a lo largo del semestre, para tutorías y resolución de dudas que han surgido al estudiante durante la realización del proyecto.

Capítulo 3

Estado del arte

Una vez presentado el proyecto, y hecha una breve descripción acerca del software que se usará, se procede a detallar con un punto de vista más técnico los programas y técnicas que se llevarán a cabo en la realización del trabajo.

Este capítulo abordará principalmente tres secciones: el análisis estático de código, los desinfectantes (o comúnmente conocidos como sanitizers), y las técnicas de fuzzing. Además, se añadirán ejemplos y aplicaciones de las mismas, junto con sus ventajas y desventajas.

3.1. Análisis estático

Hoy en día, el desarrollo de la tecnología y software cada vez más y más complejo ha propiciado que los desarrolladores no puedan percatarse de todos los errores que cometen a lo largo de la codificación de un programa. Algunos de estos errores son fallos leves que el propio compilador puede reconocer, pero en otras ocasiones, el fallo pasará desapercibido hasta su aparición en tiempo de ejecución, lo que puede provocar una vulnerabilidad explotable por un usuario con conocimientos para ello.

Existen herramientas que permiten realizar un análisis estático del código que, a diferencia de los sanitizers, que son propios de un análisis dinámico porque se realizan en tiempo de ejecución, actúan inspeccionando el código sin tener que ejecutarlo, por ende no es necesario llegar a compilarlo, aunque pueden ser usados en dicho proceso. Principalmente se implementan de tres formas diferentes [10]:

- Usando una tecnología que permita filtrar el código creando un AST (*Abstract Syntax Tree*), es decir, se trata de crear un árbol sintáctico en base a la estructura del código analizado en el que cada nodo denota una construcción de una estructura en el código [32].

- Utilizando expresiones regulares para detectar ciertas cadenas en el código.
- Combinación de las dos opciones anteriores.

La diferencia entre estas opciones radica en que el uso de expresiones regulares es más simple y flexible, pero a su vez, concurre en la detección de bastantes falsos positivos e ignora el contexto que rodea al código que coincide con la regla. Por otro lado, la técnica que genera un AST permite tratar el código como un programa entero, y no como archivos rellenos de texto, lo que permite una mejor detección de ocurrencias [10].

A continuación, se va a mostrar una descripción con mayor detalle de los compiladores a los que se harán referencia durante todo el proyecto, pues son aquellos completamente compatibles tanto con análisis de código estático como dinámico.

3.1.1. GCC

Nace originalmente en 1987 de la mano de Richard Stallman, bajo el nombre de *GNU C Compiler* como el compilador del sistema operativo GNU y código C aunque, a día de hoy, su acrónimo simboliza *GNU Compiler Collection* ya que incluye la interfaz front end que permite la compilación de código Objective-C, Fortran, Ada, Go y D, así como la compilación de código C++ y las librerías pertenecientes a cada lenguaje [4].

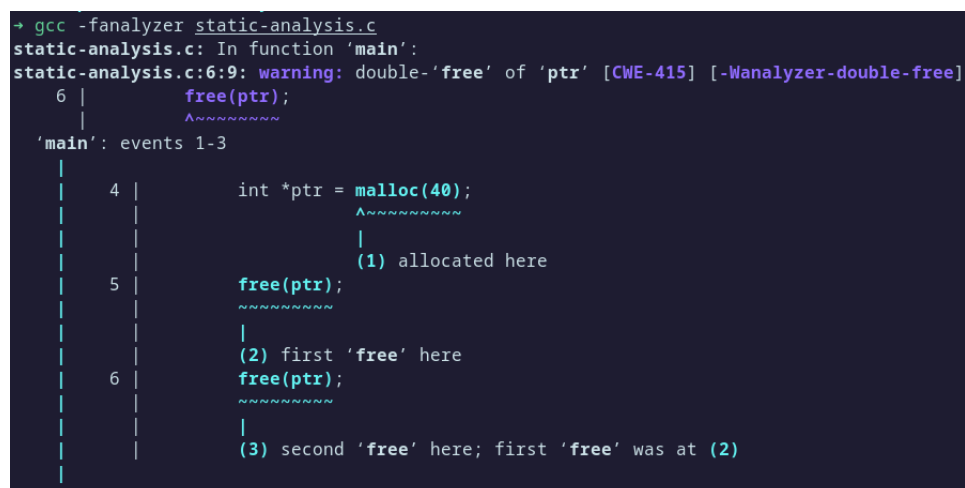
A pesar de su antigüedad, no fue hasta el 15 de noviembre de 2019 en un correo electrónico cuando se mostró la necesidad de añadir un mecanismo de análisis estático al proyecto, cuya finalidad en principio fuese la de encontrar, por ejemplo, advertencias sobre acceso a memoria que hubiese sido previamente liberada. Además, en dicho email también se muestran las ventajas de añadir un analizador de código a GCC [21].

A la fecha de creación de este trabajo, la versión más actual lanzada es GCC 12.1 y, al mismo tiempo que se da soporte para las versiones 12, 11, 10 y 9, se desarrolla la versión 13 [4]. Por último, para hacer uso del analizador estático que se encuentra en el proyecto, se ha de usar la bandera *-fanalyzer* con el fichero que contiene el código fuente que se desea comprobar.

Posteriormente, se muestra un código básico en lenguaje C que realiza una liberación de memoria dos veces sobre el mismo puntero (ver Listing 3.1) [22]. En este caso, el programa podría compilarse perfectamente y dar error en tiempo de ejecución, pero haciendo uso del analizador se puede detectar antes de ello (ver Figura 3.1).

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 int main(int argc, const char *argv[]) {
6     int *ptr = malloc(40);
7     free(ptr);
8     free(ptr);
9     return 0;
10 }
```

Listing 3.1: Programa C con doble liberación de puntero



```
+ gcc -fanalyzer static-analysis.c
static-analysis.c: In function 'main':
static-analysis.c:6:9: warning: double-'free' of 'ptr' [CWE-415] [-Wanalyzer-double-free]
   6 |         free(ptr);
     |         ^~~~~~
'main': events 1-3
|
| 4 |         int *ptr = malloc(40);
|           |
|           | (1) allocated here
| 5 |         free(ptr);
|           |
|           | (2) first 'free' here
| 6 |         free(ptr);
|           |
|           | (3) second 'free' here; first 'free' was at (2)
```

Figura 3.1: Análisis estático de un programa C con doble liberación de puntero usando GCC

3.1.2. CLANG

Otro de los compiladores que se usarán principalmente en este trabajo será el compilador CLANG, que permite la compilación de código C, C++ y Objective-C, basado en el proyecto LLVM que nace en la Universidad de Illinois en el año 2000 con el objetivo de producir una estrategia de compilación basada en SSA¹ [18] capaz de soportar tanto compilación estática como dinámica de ciertos lenguajes de programación. Se trata principalmente de una colección de herramientas junto a un compilador modular [7].

¹Hace referencia a *Static Single Assignment*, que en diseño de compiladores es una representación intermedia (*IR*) que requiere que cada variable sea asignada una sola vez y definida antes de su uso.

Entre las librerías que soporta el compilador CLANG se encuentra la librería que permite realizar un análisis estático de un código fuente pero, a diferencia de gcc, el analizador de código estático no está integrado directamente en el compilador, por lo que se ha de construir la herramienta que permite realizarlo de forma manual o instalarlo desde los repositorios de la distribución Linux que se esté usando, siempre que esté disponible [6].

El analizador está diseñado para ser simple, por lo que se puede invocar desde una terminal con la orden *scan-build*. Haciendo uso del mismo código C que se usó para el analizador de GCC (ver Listing 3.1), se ejecutará la utilidad de análisis estático de CLANG, lo que genera por defecto un directorio con ficheros HTML temporal que podrá visualizarse con mayor detalle con la orden *scan-view* perteneciente a la misma herramienta (ver Figura 3.2).

[Summary](#) > Report 4f8501

Bug Summary

File: static-analysis.c
Warning: [line 6, column 2](#)
Attempt to free released memory

[Report Bug](#)

Annotated Source Code

Press ['?'](#) to see keyboard shortcuts

[Show analyzer invocation](#)

☐ Show only relevant lines

```
1  #include <stdlib.h>
2
3  int main(int argc, char* argv[]) {
4      int *ptr = malloc(40);
5
6      free(ptr);
7
8      free(ptr);
9  }
```

1 Memory is allocated →

2 ← Memory is released →

3 ← Attempt to free released memory

Figura 3.2: Análisis estático de un programa C con doble liberación de puntero usando CLANG

3.1.3. Ventajas y desventajas

Entre las ventajas del análisis de código estático se pueden destacar aquellas que principalmente facilitan y agilizan comprobaciones de fallos. Por un lado la escalabilidad, puesto que la realización a mano de estas comprobaciones en proyectos enormes es inviable, o el tiempo invertido en ello sería desmesurado. Además, la automatización de las tareas tediosas juega un papel fundamental en estas herramientas, ya que permiten realizar las mismas comprobaciones fácilmente sobre el software cada vez que se modifica el código. Por último, se consigue una estandarización de la salida y descripción de los problemas permitiendo, en primer lugar, una mayor facilidad para el reconocimiento de los mismos y, en segundo lugar, el analizador puede mostrar información de posibles correcciones al usuario.

Se debe añadir que el análisis estático permite a los desarrolladores tener un mayor control sobre su código, pero su uso también tiene una serie de desventajas que, normalmente, implican que el desarrollador dedique parte de su tiempo a revisar. Una de las desventajas ya comentadas es el hecho de que se puedan generar falsos positivos que tenga que revisar el programador a mano. Esto a su vez puede estar relacionado directamente con el contexto en el que se analice el programa, ya que estas herramientas que analizan el código fuente pueden no detectar fallos derivados de la configuración u otros motivos.

Sin duda, los principales problemas de este tipo de análisis es que se debe tener acceso directo al código fuente del programa y, que la ejecución de los analizadores suele ser mayor al tiempo de compilación del software. Por otro lado, se debe recalcar que la no detección de fallos no implica que el programa esté libre de ellos.

Tras haber comentado diversas ventajas y desventajas del análisis estático, se llega a la conclusión de que el mayor problema de que el propio compilador haga comprobaciones por el usuario es el hecho de que el tiempo de compilación puede ser entre 2 y 5 veces mayor que el tiempo normal de compilación. Se puede comprobar fácilmente compilando varios programas, todos ellos haciendo uso de la optimización *-O2* del compilador *gcc* (ver Figura 3.3).

3.2. Sanitizers

Como se mencionó en la anterior sección, los desinfectantes pertenecen a la categoría de análisis dinámico de código y son herramientas que pueden

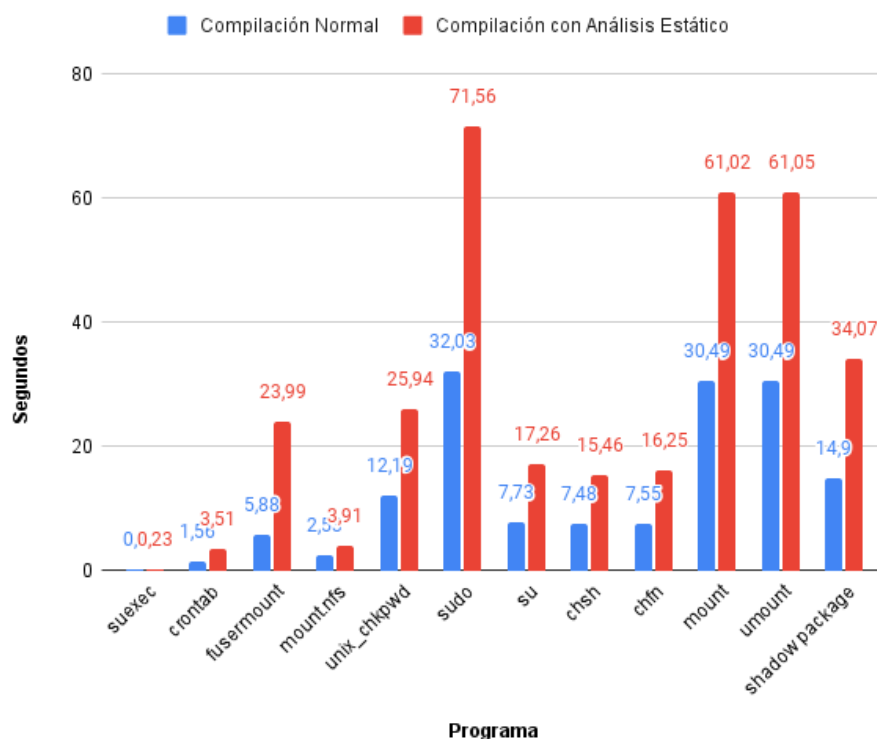


Figura 3.3: Tiempo de compilación normal y análisis estático en segundos

detectar bugs en dicho código, como *buffer overflows*² [1], *dangling pointer*³ [3] o comportamientos no esperados del programa.

Principalmente existen dos compiladores que tienen soporte para estas herramientas: *gcc* y *clang* [2], descritos con mayor detalle en la sección anterior. Las opciones para activarlas se pasan al compilador mediante el uso de banderas y, en función del compilador que se use, se dispondrá de diferentes sanitizers.

En esta sección se mostrarán los sanitizers más útiles a la hora de compilar un código: Address Sanitizer, Thread Sanitizer, Memory Sanitizer, Leak Sanitizer y Undefined Behavior Sanitizer.

²Vulnerabilidad en el código que ocurre cuando el volumen de datos excede la capacidad de almacenamiento de un buffer, por lo que el programa accede a direcciones de memoria que no debería.

³Puntero que direcciona a memoria no reservada tras haber eliminado los datos que se almacenaban en ese lugar.

3.2.1. Address Sanitizer

Es una herramienta desarrollada por Google para la detección de errores de acceso a memoria como *use-after-free*⁴ y *fugas de memoria*⁵ que pueden provocar degradación del rendimiento en el software [5].

Este desinfectante se puede usar tanto para código C como código C++ y utiliza instrumentación en tiempo de ejecución para realizar un seguimiento de las asignaciones de memoria. Si la herramienta se usa sobre un programa que no tiene fugas de memoria o problemas que pueda detectar, no mostrará nada, simplemente devolverá el ejecutable. Sin embargo, en caso de un fallo detectado, se mostrará información de un tipo u otra en función del error.

La opción para activar el sanitizer se hace en tiempo de compilación mediante la bandera *-fsanitize=address*. A continuación, se mostrarán algunos de los errores más comunes y el uso del desinfectante en cuestión.

Puntero no liberado

Para este caso, se hace uso de un programa que termina la ejecución del mismo sin liberar el puntero del que hace uso (ver Listing 3.2) [5].

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 int main(int argc, const char *argv[]) {
6     char *s = malloc(100);
7     strcpy(s, "Hello world!");
8     printf("string is: %s\n", s);
9     return 0;
10 }
```

Listing 3.2: Programa C con puntero no liberado

Tras la desinfección y posterior ejecución del binario generado, se comprueba el error que se devuelve (ver Figura 3.4), en el que se muestra que hay una fuga de memoria de 100 Bytes (el tamaño del puntero) en 1 asignación.

⁴Error que surge cuando un programa continua usando un puntero después de que se haya liberado.

⁵Situación en la que existen objetos en el heap que no se usarán más, pero el recolector de basura es incapaz de eliminarlos de la memoria.

```
→ ./leak
string is: Hello world!

=====
==17859==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 100 byte(s) in 1 object(s) allocated from:
    #0 0x7f8bb5d28dd9 in __interceptor_malloc /usr/src/debug/gcc/libsanitizer/asan/asan_malloc_linux.cpp:145
    #1 0x55b4aae351b1 in main (/home/xfear/Desktop/TFG/address-sanitizer/leak+0x11b1)
    #2 0x7f8bb5a9430f in __libc_start_call_main (/usr/lib/libc.so.6+0x2d30f)

SUMMARY: AddressSanitizer: 100 byte(s) leaked in 1 allocation(s).
```

Figura 3.4: Desinfección de un programa C con puntero no liberado

Puntero usado tras su liberación

Análogo a la anterior demostración, en este caso el puntero se libera antes de su llamada, por lo que hay un error de *use-after-free* (ver Listing 3.3) [5].

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 int main(int argc, const char *argv[]) {
6     char *s = malloc(100);
7     free(s);
8     strcpy(s, "Hello world!");
9     printf("string is: %s\n", s);
10    return 0;
11 }
```

Listing 3.3: Programa C con puntero liberado antes de su llamada

En este problema el error que da el sanitizer de salida es bastante más largo y detallado que en el anterior caso (ver Figura 3.5). Se detecta una escritura de 13 Bytes (procedente de la operación de escritura que hacemos en el código con la función *strcpy*). Tras esto, se muestra una traza de la pila donde la operación ocurre, indicando que se trata de la línea 8 en la función *main*. Además, el desinfectante muestra el lugar en el que la memoria fue asignada.

Por último, el sanitizer señala la línea en la que la memoria es asignada (línea 6) y posteriormente liberada (línea 7). Asimismo, el resto de la salida muestra una traza de como la memoria se ha movido y las direcciones exactas a las que se accede.

```

+ ./uaf
=====
==19303==ERROR: AddressSanitizer: heap-use-after-free on address 0x60b000000f0 at pc 0x7efc61836d6f bp 0x7ffdd2028
990 sp 0x7ffdd2028138
WRITE of size 13 at 0x60b000000f0 thread T0
#0 0x7efc61836d6e in __interceptor_memcpy /usr/src/debug/gcc/libsanitizer/sanitizer_common/sanitizer_common_int
erceptors.inc:827
#1 0x5560219e11ec in main (/home/xfear/Desktop/TFG/address-sanitizer/uaf+0x11ec)
#2 0x7efc6161f30f in __libc_start_call_main (/usr/lib/libc.so.6+0x2d30f)
#3 0x7efc6161f3c0 in __libc_start_main@GLIBC_2.2.5 (/usr/lib/libc.so.6+0x2d3c0)
#4 0x5560219e10d4 in _start (/home/xfear/Desktop/TFG/address-sanitizer/uaf+0x10d4)

0x60b000000f0 is located 0 bytes inside of 100-byte region [0x60b000000f0,0x60b00000154)
freed by thread T0 here:
#0 0x7efc618b3a79 in __interceptor_free /usr/src/debug/gcc/libsanitizer/asan/asan_malloc_linux.cpp:127
#1 0x5560219e11d1 in main (/home/xfear/Desktop/TFG/address-sanitizer/uaf+0x11d1)
#2 0x7efc6161f30f in __libc_start_call_main (/usr/lib/libc.so.6+0x2d30f)

previously allocated by thread T0 here:
#0 0x7efc618b3d49 in __interceptor_malloc /usr/src/debug/gcc/libsanitizer/asan/asan_malloc_linux.cpp:145
#1 0x5560219e11c1 in main (/home/xfear/Desktop/TFG/address-sanitizer/uaf+0x11c1)
#2 0x7efc6161f30f in __libc_start_call_main (/usr/lib/libc.so.6+0x2d30f)

SUMMARY: AddressSanitizer: heap-use-after-free /usr/src/debug/gcc/libsanitizer/sanitizer_common/sanitizer_common_in
terceptors.inc:827 in __interceptor_memcpy
Shadow bytes around the buggy address:
 0x0c167fff7fc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x0c167fff7fd0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x0c167fff7fe0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x0c167fff7ff0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x0c167fff8000: fa fa fa fa fa fa fa fa fd fd fd fd fd fd fd fd
=>0x0c167fff8010: fd fd fd fd fd fa fa fa fa fa fa fa fa fa[fd]fd
 0x0c167fff8020: fd fd fd fd fd fd fd fd fd fd fa fa fa fa fa fa
 0x0c167fff8030: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
 0x0c167fff8040: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
 0x0c167fff8050: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
 0x0c167fff8060: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
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
Shadow gap: cc
==19303==ABORTING

```

Figura 3.5: Desinfección de un programa C con puntero liberado antes de su llamada

Overflow del heap

Para demostrar este error, se usará el mismo programa, con la diferencia de que el tamaño del puntero ha sido reducido hasta que no quepa el mensaje que se quiere almacenar (ver Listing 3.4) [5].

La salida del desinfectante tras la compilación y ejecución es muy similar a la anterior, con la diferencia de que se indica que se ha producido un overflow del heap en lugar de un error de *use-after-free*. El resto de información es la misma que en el resultado previo (ver Figura 3.5).

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 int main(int argc, const char *argv[]) {
6     char *s = malloc(12);
7     free(s)
8     strcpy(s, "Hello world!");
9     printf("string is: %s\n", s);
10    return 0;
11 }
```

Listing 3.4: Programa C con puntero liberado antes de su llamada

3.2.2. Thread Sanitizer

Consiste en un módulo de instrumentación del compilador y una librería de ejecución en tiempo real para código C y C++ que permite detectar condiciones de carrera en un software. Estas condiciones de carrera son uno de los fallos más comunes y difíciles de detectar. Ocurren cuando dos hebras acceden a la misma variable de forma simultánea y al menos una de las acciones es de escritura.

La instrumentación del desinfectante se hace en tiempo de compilación a través de la bandera *-fsanitize=thread*. A continuación, se ha creado un programa que permite que se produzca una condición de carrera, para ello, se hace uso de una variable global en la que escriben dos hebras (ver Listing 3.5) [15].

```
1 #include <pthread.h>
2 #include <stdio.h>
3
4 int Global;
5
6 void *Thread1(void *x) {
7     Global++;
8     return NULL;
9 }
10
11 void *Thread2(void *x) {
12     Global--;
13     return NULL;
14 }
15
16 int main() {
17     pthread_t t[2];
```

```

18 pthread_create(&t[0], NULL, Thread1, NULL);
19 pthread_create(&t[1], NULL, Thread2, NULL);
20 pthread_join(t[0], NULL);
21 pthread_join(t[1], NULL);
22 }

```

Listing 3.5: Programa C con hebras de escritura simultánea

Tras la compilación y ejecución del binario, el desinfectante proporciona un aviso de que una condición de carrera se está produciendo en el software (ver Figura 3.6).

```

→ ./simple_race
=====
WARNING: ThreadSanitizer: data race (pid=24180)
  Write of size 4 at 0x559764906100 by thread T2:
    #0 Thread2 /home/xfear/Desktop/TFG/thread-sanitizer/simple_race.c:12:9 (simple_race+0xdeaf4)

  Previous write of size 4 at 0x559764906100 by thread T1:
    #0 Thread1 /home/xfear/Desktop/TFG/thread-sanitizer/simple_race.c:7:9 (simple_race+0xdea4)

  Location is global 'Global' of size 4 at 0x559764906100 (simple_race+0x0000033c2100)

  Thread T2 (tid=24183, running) created by main thread at:
    #0 pthread_create <null> (simple_race+0x9288e)
    #1 main /home/xfear/Desktop/TFG/thread-sanitizer/simple_race.c:19:3 (simple_race+0xdeb5f)

  Thread T1 (tid=24182, finished) created by main thread at:
    #0 pthread_create <null> (simple_race+0x9288e)
    #1 main /home/xfear/Desktop/TFG/thread-sanitizer/simple_race.c:18:3 (simple_race+0xdeb48)

SUMMARY: ThreadSanitizer: data race /home/xfear/Desktop/TFG/thread-sanitizer/simple_race.c:12:9 in
Thread2
=====
ThreadSanitizer: reported 1 warnings

```

Figura 3.6: Desinfección de un programa C con condición de carrera

3.2.3. Memory Sanitizer

Permite la detección de lectura de memoria no inicializada en programas de código C y C++. Esta problema ocurre cuando la memoria asignada en el heap o la pila se lee antes de ser escrita. Para el uso de este desinfectante se requiere usar la bandera *-fsanitize=memory* en *clang* en la compilación del programa.

A continuación, se muestra un programa en el que se produce una lectura de memoria en un vector sin haber inicializado esa posición (ver Listing 3.6) [15].

```

1 #include <stdio.h>
2
3 int main(int argc, char** argv) {
4     int* a = new int[10];

```



```
5  a[5] = 0;
6  if (a[argc])
7      printf("xx\n");
8  return 0;
9 }
```

Listing 3.6: Programa C++ con lectura de memoria sin escritura

El problema radica en que con una compilación normal, no siempre se detectaría el fallo en una ejecución, mientras que si usamos el Memory Sanitizer, se mostrará un aviso de una fuga de memoria (ver Figura 3.7).

```
→ ./memory_unallocated
==31139==WARNING: MemorySanitizer: use-of-uninitialized-value
#0 0x55f42d83c405 in main /home/xfear/Desktop/TFG/memory_sanitizer/memory_unallocated.cc:6:7
#1 0x7f7a2387930f in __libc_start_call_main libc-start.c
#2 0x7f7a238793c0 in __libc_start_main@GLIBC_2.2.5 (/usr/lib/libc.so.6+0x2d3c0)
#3 0x55f42d7b6124 in _start (/home/xfear/Desktop/TFG/memory_sanitizer/memory_unallocated+0x21124)
SUMMARY: MemorySanitizer: use-of-uninitialized-value /home/xfear/Desktop/TFG/memory_sanitizer/memory_u
nallocated.cc:6:7 in main
Exiting
```

Figura 3.7: Desinfección de un programa C++ con memoria no inicializada

3.2.4. Leak Sanitizer

Uno de los desinfectantes más simples, se basa en la detección de la no liberación de memoria dinámica previamente asignada para el caso del lenguaje C (con la función *malloc*) y el lenguaje C++ (con la función *new*). Está integrado en el sanitizer AddressSanitizer [15] pero también se puede invocar por separado para hacer uso de esta funcionalidad únicamente.

En el código de la sección anterior (ver Listing 3.5) se produce una asignación de memoria sin una posterior liberación y será compilado con la bandera *-fsanitize=leak* para su detección. Tras la ejecución del binario, el desinfectante mostrará el aviso de la fuga de memoria que se está produciendo (ver Figura 3.8).

```
→ ./memory_unallocated
=====
==32058==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 40 byte(s) in 1 object(s) allocated from:
#0 0x7fd4445440e2 in operator new[](unsigned long) /usr/src/debug/gcc/libsanitizer/lsan/lsan_interceptors.cpp:240
#1 0x556a2c72f161 in main /home/xfear/Desktop/TFG/leak_sanitizer/memory_unallocated.cc:4
#2 0x7fd44402d30f in __libc_start_call_main (/usr/lib/libc.so.6+0x2d30f)

SUMMARY: LeakSanitizer: 40 byte(s) leaked in 1 allocation(s).
```

Figura 3.8: Desinfección de un programa C++ con fuga de memoria

3.2.5. Undefined Behavior Sanitizer

No todos los errores que se pueden producir en la ejecución de un software incurren en un problema grave, algunos de ellos simplemente producen un comportamiento inesperado del programa. El concepto de comportamiento indefinido hace referencia a que la semántica de cierta operación está en un estado indefinido y el compilador asume que esa operación no ha ocurrido, por lo que es libre de realizar cualquier operación inesperada que, normalmente radica en producir resultados erróneos o terminar la ejecución del programa [27].

Es aquí donde encaja el último sanitizer que se verá en esta sección y que puede ser activado haciendo uso de la bandera `-fsanitize=undefined`. El resultado es mostrar un error en función del tipo de fallo, entre los que destacan:

- Divisiones entre 0.
- Comprobación de referencia a un puntero nulo.
- Acceso fuera de límites a un vector.
- Comprobación de que las operaciones de desplazar bits a la izquierda y derecha en una variable no quedan en un estado indefinido.
- Overflow de variables de tipo entero con signo.

3.2.6. Ventajas y desventajas

Una vez que se han detallado los desinfectantes de uso más común, se procede a detallar las ventajas y desventajas de su uso.

Por un lado, las ventajas de hacer uso de estas herramientas son las mismas que para el análisis estático, con la diferencia de que estas comprobaciones se realizan durante la ejecución del programa, es decir, se produce un análisis dinámico.

Por otro lado, como desventajas se deben destacar que el tiempo de ejecución de los programas así como el uso de memoria se ven incrementados. Este problema se agrava en aquellos programas cuyo uso de memoria sea alto y en función del desinfectante que se use. Por ejemplo, se ha comparado el uso de memoria de binarios compilados con *CLANG* y la opción de optimización `-O2` haciendo uso de los desinfectantes *Address Sanitizer* y *Memory Sanitizer* (ver Figura 3.9), el resultado al que se ha llegado es que el uso de memoria se ve incrementado en 2.68 y 2.13 veces respectivamente

Proporción del uso de memoria del programa instrumentado frente al programa original

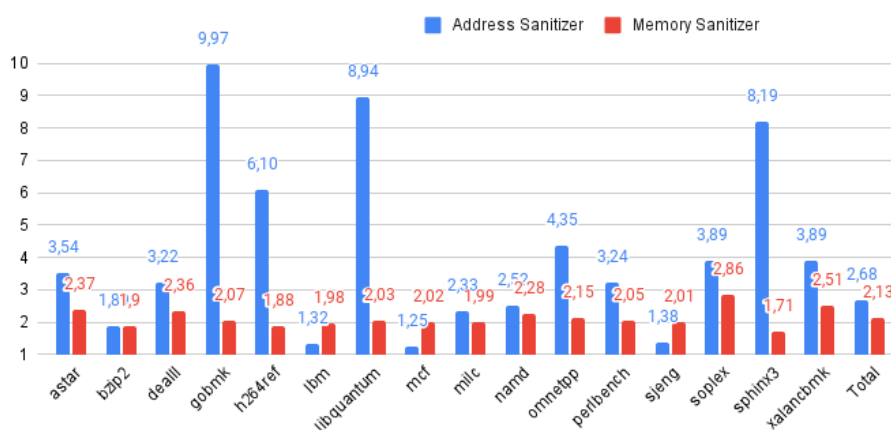


Figura 3.9: Proporción del uso de memoria del programa instrumentado con Address Sanitizer y Memory Sanitizer frente al programa original

de media [28][31].

Otro ejemplo de degradación de rendimiento e incremento de uso de memoria es en el caso del uso de *Thread Sanitizer* con el que se realizó un benchmark sobre las distintas partes del software Chromium y donde se llegó a la conclusión de que este desinfectante puede llegar a incrementar entre 5 y 10 veces el tiempo de ejecución de un programa y entre 2 y 5 veces el uso de memoria del mismo (ver Figura 3.10) [29].

Proporción del tiempo de ejecución y uso de memoria del programa instrumentado con Thread Sanitizer frente al programa original

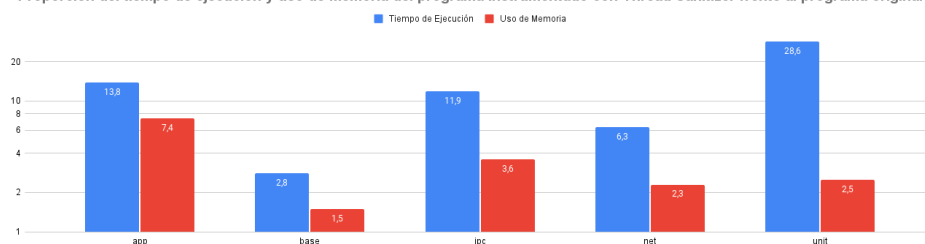


Figura 3.10: Proporción del uso de memoria del programa instrumentado con Thread Sanitizer frente al programa original

La propia documentación del compilador CLANG asegura que normalmente el incremento de tiempo de ejecución se ve afectado entre 5 y 15 veces mientras que el uso de memoria se ve incrementado entre 5 y 10 veces del

programa original [8].

3.3. Fuzzing

A grandes rasgos, se conoce como *fuzzing* a la técnica de enviar datos anómalos a un sistema con la intención de conseguir un comportamiento que no es el esperado, normalmente la interrupción de la ejecución del programa por un fallo. Esta técnica se utiliza principalmente con 3 objetivos diferentes:

- Control de calidad de un software desde un punto de vista interno.
- Administración de sistemas para asegurar el correcto funcionamiento del entorno en el que se trabaja.
- Descubrimiento de vulnerabilidades desde la vista de un usuario externo.

Una definición más precisa del término *fuzzing* sería

Una técnica de prueba altamente automatizada que cubre numerosos casos límite usando datos inválidos (de archivos, protocolos de red, llamadas API y otras fuentes) como entrada de una aplicación para garantizar mejor la ausencia de vulnerabilidades explotables. El nombre proviene de la tendencia de las aplicaciones de módem a fallar debido a entradas aleatorias causadas por ruido de línea en líneas telefónicas borrosas. [25]

Una herramienta que hace uso de este procedimiento se basa en la generación de datos *semiválidos*⁶ que envía a una aplicación para observar su comportamiento. Si la aplicación falla en el procesamiento de estos datos puesto que no está realizando el comportamiento esperado, la herramienta guarda los datos para un posterior análisis y prosigue con la siguiente iteración. Si por el contrario, los datos son aceptados por la aplicación perfectamente, se pueden descartar y seguir generando nuevos datos. La ventaja principal de estas herramientas es la velocidad con la que se pueden generar las permutaciones en las cadenas de datos inválidos, así como procesarlas, reduciendo la ejecución de pruebas a periodos de tiempo manejables para el usuario.

Normalmente estas técnicas se combinan con el uso de desinfectantes anteriormente descritos, incluso el compilador *CLANG* tiene su propia herramienta de *fuzzing*, lo que implica una mayor eficiencia a la hora de detectar fallos. La combinación de ambas produce degradación en el proceso *fuzzing*, pero pueden llegar a encontrar errores que por sí solas no encontrarían.

⁶Los datos son lo suficientemente correctos para que el programa los acepte, pero se añaden datos anómalos que puedan provocar un comportamiento inesperado

3.3.1. Categorías de testeo

La diferencia más remarcable entre las técnicas de *fuzzing* es la división entre testeo estructural y funcional o, *White-Box testing* y *Black-Box testing* respectivamente. Mientras que el primero requiere de un acceso al código fuente, el segundo se basa en probar el software haciendo uso de interfaces externas, sin tener en cuenta el código que compone a la aplicación. Por último, existe un tercer tipo, conocido como *Grey-Box testing* que no es más que una combinación de los dos anteriores, basándose en la idea de que la estructura interna del programa ayuda a definir el diseño de los casos de prueba que se ejecutan por interfaces externas.

White-Box testing

Gracias al beneficio que supone el acceso al código fuente, este es el único método que permite conseguir un 100 % de cobertura a la hora de asegurar el correcto funcionamiento de un programa. Este tipo de técnica puede estar basada en el análisis estático o dinámico, o puede ser una combinación de ambos, haciendo referencia al uso de sanitizers y análisis que comentamos en secciones anteriores. Un ejemplo de esta técnica son los *test unitarios*⁷ o *test de integración*⁸ [11].

Black-Box testing

Mientras que la naturaleza del *White-Box testing* puede ser estática o dinámica, el *Black-Box testing* se caracteriza por tener una naturaleza solo dinámica, pues no tiene en cuenta el funcionamiento interno de un programa. Esta diferencia permite que estas técnicas consuman menos tiempo a la hora de ejecutarse pero, por ende, el número de resultados se ve reducido. Normalmente se usa con la intención de comprobar que se cumplen unos criterios de aceptación en un software o la búsqueda de vulnerabilidades. Realmente los resultados que arroja este tipo de testeo es pobre, por lo que se añadirá normalmente cualquier información que se conozca del funcionamiento interno, convirtiéndose en el *Grey-Box testing* mencionado anteriormente [11].

3.3.2. Tipos de técnicas

Tras revisar cómo el conocimiento del funcionamiento interno de un programa puede afectar al diseño de los casos de prueba que se usarán en las

⁷Tipo de testeo de software que se basa en comprobar si pequeñas porciones de código funcionan como deberían.

⁸Tipo de testeo de software que verifica que dos módulos se integran y funcionan correctamente.

herramientas y técnicas de fuzzing, se puede hacer una división en cuatro tipos de los principales mecanismos.

Fuzzing aleatorio

Es la técnica más sencilla, se basa en la generación completamente aleatoria de datos que se usarán en la entrada del software. En el uso no se aplica ninguna métrica, semilla o gramática para definir las cadenas generadas, por lo que es poco útil, ya que es ineficiente.

Fuzzing basado en mutación

Se caracteriza por ser un mecanismo fácil de implementar puesto que su función principal es la de coger valores válidos y realizar mutaciones en ellos para servirlos como entrada al software. Un ejemplo es el intercambio de bits menos significativos en estos valores válidos y servirlos a la aplicación.

Fuzzing basado en generación

Esta técnica, al contrario que las anteriores, no genera cadenas aleatorias ni hace uso de cadenas válidas a las que aplica mutación. En su lugar, genera entradas basándose en un regla, lo que permite que se creen siempre valores de entrada con una estructura previamente definida que, por regla general, es correcta para el programa.

Fuzzing evolutivo

Se basa en el uso de *algoritmos genéticos* para crear conjuntos de casos de prueba. Esta generación está basada principalmente en el framework de *fuzzing* diseñado por el usuario y las respuestas recibidas del objetivo al que se somete a las pruebas. El primer conjunto de casos de prueba se crea por el método de generación mencionado anteriormente y los demás conjuntos se crean siguiendo los siguientes pasos:

1. A cada elemento del conjunto de casos de prueba se le asigna una puntuación, que es una combinación de múltiples métricas definidas por el usuario y monitorizadas durante la ejecución.
2. Los elementos con menor puntuación se eliminan del conjunto.
3. Se aplica mutación a los elementos restantes.
4. Se combinan casos de prueba con gran puntuación para generar otros más óptimos. Este proceso se usa principalmente para reemplazar los elementos que se eliminaron en el paso 2.

3.3.3. AFL++

Tras haberse explicado con detalle las diferentes técnicas de fuzzing que existen, a continuación se expondrá la información de la herramienta que se usará en los siguientes capítulos para la ejecución de pruebas de fuzzing. Se trata de *AFL++* [30], basado en el famoso fuzzer AFL (*American Fuzzy Loop*) [16].

En primer lugar, AFL es un fuzzer que emplea instrumentación en tiempo de compilación junto al uso de fuzzing evolutivo para descubrir de forma automática casos de prueba interesantes para un binario que permiten alcanzar estados no esperados del mismo. Comparado con otros fuzzers de este estilo, American Fuzzy Loop está diseñado para ser práctico, ya que tiene un rendimiento aceptable haciendo uso de un gran número de estrategias de fuzzing. Principalmente se ha elegido para este proyecto porque no necesita apenas configuración, lo que se traduce en un uso menos complejo. A grandes rasgos, el algoritmo que emplea se basa en los siguientes pasos

1. Carga casos de prueba iniciales que haya suministrado el usuario en la cola.
2. Coge el siguiente archivo de entrada de la cola.
3. Intenta recortar el caso de prueba al tamaño más pequeño que no altere el comportamiento medido del programa.
4. Muta repetidamente el archivo usando técnicas de fuzzing.
5. Si cualquier mutación generada resulta en un nuevo estado recogido en la instrumentación del programa, se añade la salida mutada como nueva entrada en la cola.
6. Vuelve al paso 2.

Estas herramientas se pueden usar de dos formas diferentes, aunque una de ellas retornará mejores resultados que la otra. En primer lugar, se puede lanzar contra un programa binario sin instrumentación, es decir, un programa del que no se conoce el código fuente, resultando en el ya mencionado Black-box testing. Por otro lado, se puede instrumentar la compilación del binario si se dispone de su código fuente, esto es, haciendo uso de la utilidad *afl-gcc-fast* para código C o *afl-g++-fast* para código C++. En este último caso, los resultados del fuzzer serán mucho mejores.

Lamentablemente el desarrollo de AFL lleva discontinuado desde hace varios años, aunque hoy en día sigue siendo igual de útil. Sin embargo, gracias a la comunidad de desarrollo de código abierto, se ha lanzado una

nueva herramienta conocida como AFL++, con procedimientos mucho más complejos y una gran variedad de mejoras y funcionalidades añadidas que podemos encontrar en el repositorio de la herramienta [30], donde se encuentra la guía para su instalación y ejecución.

Para hacer funcionar el software *AFL++* se debe instrumentar el programa que se desee con la utilidad *afl-gcc-fast* y ejecutar el binario generado con la orden *afl-fuzz* añadiendo diferentes argumentos:

- Un directorio en el que se encuentran los ficheros que se usarán para los casos de prueba.
- Un directorio en el que se guardarán los informes generados e información de utilidad tras la realización de las pruebas.
- El binario instrumentado.

Una vez que el programa se esté ejecutando se mostrará por pantalla distinta información, como el tiempo de ejecución, la velocidad, los ciclos realizados y los datos sobre fallos, entre otras cosas (ver Figura 3.11).

american fuzzy lop ++4.00c {default} (./afl-example) [fast]			
process timing		overall results	
run time : 0 days, 0 hrs, 0 min, 13 sec		cycles done : 5	
last new find : 0 days, 0 hrs, 0 min, 0 sec		corpus count : 4	
last saved crash : none seen yet		saved crashes : 0	
last saved hang : none seen yet		saved hangs : 0	
cycle progress		map coverage	
now processing : 2.9 (50.0%)		map density : 0.00% / 0.00%	
runs timed out : 0 (0.00%)		count coverage : 1.00 bits/tuple	
stage progress		findings in depth	
now trying : havoc		favored items : 3 (75.00%)	
stage execs : 1034/1766 (58.55%)		new edges on : 4 (100.00%)	
total execs : 15.0k		total crashes : 0 (0 saved)	
exec speed : 1097/sec		total tmouts : 0 (0 saved)	
fuzzing strategy yields		item geometry	
bit flips : disabled (default, enable with -D)		levels : 4	
byte flips : disabled (default, enable with -D)		pending : 1	
arithmetics : disabled (default, enable with -D)		pend fav : 0	
known ints : disabled (default, enable with -D)		own finds : 3	
dictionary : n/a		imported : 0	
havoc/splice : 2/13.9k, 0/0		stability : 100.00%	
py/custom/rq : unused, unused, unused, unused			
trim/eff : 33.33%/1, disabled		[cpu000: 16%]	

Figura 3.11: Ejecución de AFL++

Este software también ha sido elegido por su capacidad de escalabilidad horizontal⁹. Su ejecución se asigna a un nodo de la CPU, por lo que éste debe compartirse con otras herramientas del sistema. Para mitigar dicho problema se pueden realizar ejecuciones maestro-esclavo en diferentes nodos

⁹Capacidad de un sistema para adaptarse a la carga de trabajo de forma que se modularice o se añadan más nodos para soportar mejor dicha carga.

de la CPU, en el que se define una ejecución maestra que recibe los datos de todas las esclavas (ver Figura 3.12).

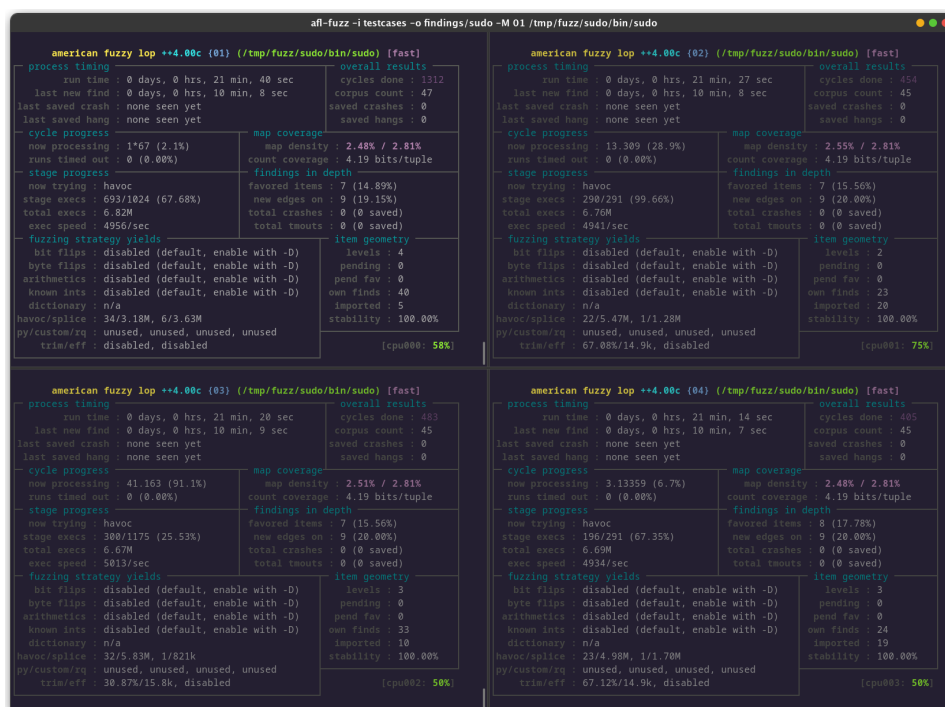


Figura 3.12: Ejecución paralela de AFL++ con 4 procesos

Por último, puede ocurrir que cuando se intente ejecutar se presente un error referente a las notificaciones del núcleo (ver Figura 3.13). En este caso la solución al error viene en el mismo mensaje, que se traduce en redirigir las notificaciones del núcleo para que la herramienta pueda analizar correctamente los fallos del software analizado.

Instrumentación de programas especiales

Como se mencionó anteriormente, la instrumentación de los binarios consiste en compilar los archivos fuente con la utilidad *afl-gcc-fast* o *afl-clang-fast* en función de si se quiere usar el proyecto GNU o LLVM. Este paso es común a todos los binarios, pero algunos requieren de un paso extra para poder realizar las técnicas de *fuzzing* de forma efectiva.

La mayoría de binarios sometidos a análisis en este proyecto son herramientas que reciben valores por argumento, es decir, al mismo tiempo que son invocadas. Este es el caso de utilidades como *sudo*, *cat*, *unzip*, *cp*, etc. Esto produce un problema para *AFL++*, puesto que dicho software envía las cadenas generadas al binario por entrada estándar una vez que se ha

```
+ afl-fuzz -i testcases -o findings ./afl-example
afl-fuzz++4.00c based on afl by Michal Zalewski and a large online community
[+] afl++ is maintained by Marc "van Hauser" Heuse, Heiko "hexcoder" Eißfeldt, Andrea Fioraldi and Dominik Maier
[+] afl++ is open source, get it at https://github.com/AFLplusplus/AFLplusplus
[+] NOTE: This is v3.x which changes defaults and behaviours - see README.md
[+] No -M/-S set, autoconfiguring for "-S default"
[*] Getting to work...
[+] Using exponential power schedule (FAST)
[+] Enabled testcache with 50 MB
[*] Checking core_pattern...

[-] Hmm, your system is configured to send core dump notifications to an
external utility. This will cause issues: there will be an extended delay
between stumbling upon a crash and having this information relayed to the
fuzzer via the standard waitpid() API.
If you're just testing, set 'AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES=1'.

To avoid having crashes misinterpreted as timeouts, please log in as root
and temporarily modify /proc/sys/kernel/core_pattern, like so:

echo core >/proc/sys/kernel/core_pattern

[-] PROGRAM ABORT : Pipe at the beginning of 'core_pattern'
    Location : check_crash_handling(), src/afl-fuzz-init.c:2198
```

Figura 3.13: Mensaje de error de AFL++

invocado el programa que se va a analizar.

La solución a este problema es sencilla y consiste en utilizar una cabecera que proporciona la implementación de *AFL++* llamada *argv-fuzz-inl.h* y hacer una llamada a la función *AFL_INIT_ARGV*, que transforma los valores de entrada estándar o *stdin* en valores de argumento o *argv*. Para realizar este cambio se deben seguir los siguientes pasos:

1. Descargar y añadir el fichero *argv-fuzz-inl.h* al directorio en el que se encuentre el fichero fuente del programa objetivo.
2. Editar el fichero fuente principal del programa objetivo con los siguientes cambios (ver Listing 3.7):
 - Incluir la cabecera *argv-fuzz-inl.h*.
 - Realizar una llamada a la función *AFL_INIT_ARGV* en la función *main* del programa antes que se ejecute cualquier código.

```
1 #include <stdio.h>
2
3 ...
4
5 #include "argv-fuzz-inl.h"
6
7 int main(int argc, char** argv) {
8     AFL_INIT_ARGV();
9
10    ...
11
```

```
12  return 0;  
13  }
```

Listing 3.7: Instrumentación de binario de línea de comandos con AFL++

Capítulo 4

Análisis

En este capítulo se realizará un análisis del problema a resolver, es decir, se desglosa el problema en los pasos que se deben completar para plantear una solución al mismo, entre ellos se encuentra la obtención del código fuente de los binarios que se desee analizar, la instrumentación y ejecución de pruebas, así como el análisis de los informes que generen estas pruebas.

4.1. Paquetes de la distribución Linux

Principalmente se hará un análisis de los paquetes de la propia distribución Linux que se está usando para el desarrollo de este proyecto, concretamente Manjaro, una distribución basada en Arch Linux.

El gestor de paquetes propio de esta distribución y el que se usará en este caso es *Pacman* [20], heredado de la distribución Arch Linux. Para conseguir el código fuente de un paquete se deben seguir los siguientes pasos:

1. Encontrar a qué paquete pertenece la utilidad de la que se quiere conseguir el código. Se realiza mediante la siguiente orden (ver Listing 4.1) donde la herramienta *which* devuelve la ruta absoluta de la utilidad mientras que el gestor de paquetes haciendo uso de los argumentos *-Q* y *-o* permite buscar los paquetes a los que pertenece dicha utilidad y sus ficheros de configuración.

```
pacman -Qo $(which <utilidad>)
```

Listing 4.1: Gestor de paquetes pacman

2. Se usa la herramienta *asp* [19] para recuperar los archivos fuente de compilación utilizados para crear el paquete. Con la siguiente orden (ver Listing 4.2). Estos son los ficheros que se usarán en el siguiente paso para descargar el código fuente que se compilará para obtener los binarios.

```
asp export <paquete>
```

Listing 4.2: Uso de la herramienta asp

3. Recuperar los archivos que contienen el código fuente de los programas con la siguiente orden (ver Listing 4.3). Haciendo uso de los parámetros contemplados en la misma se evita la verificación de la firma PGP ¹ [14] y se solicitan los archivos fuentes al servidor.

```
makepkg -do --skipgpcheck
```

Listing 4.3: Orden para recuperar los archivos de código fuente del paquete

4.2. Software a analizar

Una vez que se tiene la capacidad para conseguir el código fuente de los paquetes de la distribución, se debe elegir cuáles son los más interesantes para su análisis porque principalmente no interesa cualquier herramienta, solo aquellas en las que una vulnerabilidad podría resultar en un problema grave para el sistema.

En las distribuciones Linux los programas más importantes en los que no debe haber vulnerabilidades son aquellos que se ejecutan con privilegios de administrador. El objetivo aquí es buscar aquellas herramientas con el bit SUID ² [12] activado cuyo creador sea el usuario *root*.

A continuación se muestra la orden necesaria para llevar a cabo este proceso (ver Listing 4.4) haciendo uso del programa *find* con los argumentos que permiten definir el tipo de permiso y el creador del archivo.

```
find / -perm -u=s -user root 2>/dev/null
```

Listing 4.4: Búsqueda de archivos con el bit SUID activado

4.3. Configuración del entorno de compilación

Tras tener acceso al código fuente de cada paquete, se debe compilar el mismo para conseguir los binarios que se van a analizar. En función de si lo que se busca es una instrumentación para *fuzzing* o simplemente compilar el código con un análisis estático o dinámico, se debe configurar la variable

¹Pretty Good Privary hace referencia a un programa de cifrado que proporciona autenticación para la transferencia de datos.

²Permiso que permite a cualquier usuario ejecutar una herramienta como si fuese el creador de la misma.

de entorno que lo defina, en este caso para el compilador *GCC* (ver Listing 4.5), la cual es *CFLAGS* para código C y *CPPFLAGS* para código C++.

```
CFLAGS = gcc <argumentos para instrumentar>
CPPFLAGS = g++ <argumentos para instrumentar>
```

Listing 4.5: Descarga de archivos de compilación de los paquetes

Esto es, cuando accedemos al directorio que contiene los archivos fuente, normalmente se hace uso del script *configure* para generar el archivo *Makefile* que posteriormente será ejecutado con la utilidad *make* para generar todos los binarios del paquete.

4.4. Informes generados

En el caso de que se aplique instrumentación al código, se generarán informes si se usa el compilador *CLANG*, mientras que estos informes se verán en pantalla si se usa *GCC*. En cualquier caso, se debe analizar dicha información para conocer las vulnerabilidades o fallos que puede tener el código.

Un ejemplo de un informe generado por el analizador estático del compilador *CLANG* (ver Figura 4.1) muestra un resumen del tipo y cantidad de las vulnerabilidades encontradas. También permite navegar a través de ellas para llegar directamente a la parte de código que se debe analizar.

Bug Summary

Bug Type	Quantity	Display?
All Bugs	177	<input checked="" type="checkbox"/>
API		
Argument with 'nonnull' attribute passed null	1	<input checked="" type="checkbox"/>
Logic error		
Assigned value is garbage or undefined	48	<input checked="" type="checkbox"/>
Dereference of undefined pointer value	22	<input checked="" type="checkbox"/>
Result of operation is garbage or undefined	3	<input checked="" type="checkbox"/>
Uninitialized argument value	98	<input checked="" type="checkbox"/>
Unused code		
Dead assignment	5	<input checked="" type="checkbox"/>

Figura 4.1: Ejemplo del reporte del analizador estático de CLANG

4.5. Fuzzing

La última parte del problema es aplicar técnicas de *fuzzing* sobre los binarios generados a partir de instrumentación. Para la realización de esta parte se usará el software de *fuzzing* que se ha detallado en el capítulo 3, *AFL++*.

Por último, se deben revisar los informes que se hayan generado para conocer con exactitud qué problemas se han generado en la ejecución del binario y las cadenas que han propiciado ese resultado.

Capítulo 5

Diseño e implementación

En este capítulo se abordará el diseño e implementación al análisis realizado en el capítulo anterior. Se describirá la solución empleada así como los elementos que la constituyen.

5.1. Automatización

La mayoría de tareas requeridas en el análisis son susceptibles de ser automatizadas, para ello se han creado diferentes scripts que agilizan distintos procesos.

Los scripts que se han creado para el desarrollo de este proyecto resuelven las siguientes funciones:

- Listar los paquetes a los que pertenecen los binarios que se van a analizar y descargar los ficheros de compilación del paquete.
- Conseguir los archivos con el código fuente de los programas.
- Generar el archivo Makefile y compilar los binarios.
- Calcular el tiempo de ejecución y compilación de los binarios para generar informes de datos.
- Configuración del software de fuzzing para el inicio y finalización de las pruebas.

Todos los scripts se han exportado al directorio */usr/local/bin* y este se ha añadido a la variable de entorno *PATH*, para que se puedan utilizar en cualquier lugar.

5.2. Elementos de automatización

Para cada una de las funciones mencionadas anteriormente se ha desarrollado un programa o script que ha permitido automatizar y agilizar el proceso de obtención de los binarios que se necesitasen.

5.2.1. Descarga y obtención de ficheros de compilación del paquete

La realización de esta tarea se resume a crear un script en el lenguaje de scripting *Bash*, en este caso *download_source.sh* (ver Listing 5.1) cuya funcionalidad es la de usar el gestor de paquetes *pacman* y la utilidad *asp* como se detalla en el anterior capítulo sobre un fichero con los nombres de paquetes que contengan los binarios que se desean analizar. Este fichero se ha rellenado previamente con la salida del comando *find* expuesto en el capítulo 4 que, para una mejor visualización, se ha resumido en una tabla (ver Cuadro 5.1).

Cuadro 5.1: Binarios con bit SUID activado y creador root

Índice	Nombre	Paquete	Índice	Nombre	Paquete
1	su	shadow	15	expiry	shadow
2	fusermount	fuse2	16	sudo	sudo
3	fusermount3	fuse3	17	ksu	krb5
4	chsh	shadow	18	sg	shadow
5	cdrecord	cdrtools	19	mount	util-linux
6	mount.nfs	nfs-utils	20	crontab	cronie
7	chfn	shadow	21	readcd	cdrtools
8	rscsi	cdrtools	22	gpasswd	shadow
9	nvidia-modprobe	nvidia-utils	23	chage	shadow
10	passwd	shadow	24	unix_chkpwd	pam
11	suexec	apache	25	umount	util-linux
12	pkexec	polkit	26	newgrp	util-linux
13	mount.cifs	cifs-utils	27	mount.ecryptfs_private	ecryptfs-utils
14	cdda2wav	cdrtools			

```
#!/bin/bash
```

```
file=$1
```

```
while read line, do
```

```
    package = $(pacman -Qo $(which $line) | cut -d " " -f 5 | sort  
                | uniq)
```

```
asp export $package
done < $file
```

Listing 5.1: Configuración de la variable de entorno para instrumentación

El script realiza un trabajo simple, llama al gestor de paquetes y encauza su salida con diferentes utilidades para formatearla:

- El comando *cut -d " " -f 5* permite escoger simplemente el nombre del paquete de la salida, puesto que esta tiene el formato “ruta-del-binario is owned by nombre-paquete”.
- La utilidad *sort* ordena la salida.
- La herramienta *uniq* elimina duplicados.

Por último, usa *asp* para descargar los ficheros de compilación del paquete generando un directorio por cada paquete (ver Figura 5.1).

```
+ ls -d */
apache/  cifs-utils/  curl/          fuse2/  krb5/      nvidia-utils/  polkit/  sudo/
cdrtools/  cronie/      ecryptfs-utils/ fuse3/  nfs-utils/  pam/           shadow/  util-linux/
```

Figura 5.1: Directorios generados por la utilidad *asp*

5.2.2. Directorios generados por *asp*

Para esta tarea también se ha creado un script en lenguaje *Bash*, aunque más simple que el anterior que, realmente se resume a ejecutar un comando (ver Listing 5.2).

```
find . -maxdepth 1 -type d \( ! -name . \) -exec bash -c "cd '{}'"
    && makepkg -do --skipgpcheck" \;
```

Listing 5.2: Descarga de archivos con el código fuente de cada paquete

El script busca todos los directorios, a excepción de “.” y ejecuta las órdenes *cd* para cambiar de directorio y *makepkg -do --skipgpcheck* para la obtención de ficheros con el código fuente. Una vez terminada la ejecución, en cada directorio se encontrará otro directorio con archivos, entre los que destacará el script *configure* que se usará en la siguiente tarea.

5.2.3. Generación del archivo Makefile y compilación de binarios

Antes de ejecutar el script creado para esta tarea se debe haber configurado la variable de entorno necesaria, en este caso *CFLAGS* para código C con la instrumentación que se desee, en función de si se va a aplicar un

análisis estático o dinámico o se realizarán pruebas de fuzzing.

El script creado muestra por pantalla los valores de las variables de entorno importantes en el momento de la ejecución, navega dentro del directorio que se pasa por argumento borrando la carpeta `src` si está disponible, ya que puede contener vestigios de una ejecución anterior con otra orquestación. Después, descarga los archivos con el código fuente en el directorio `src`, accede al único directorio que existe dentro de este mismo y usa el script *configure* para generar el archivo Makefile, que ejecuta posteriormente para compilar los archivos y generar los binarios (ver Listing 5.3).

```
#!/bin/bash

echo "CC value: $CC"
echo "CFLAGS value: $CFLAGS"
echo "CXX value: $CXX"
echo "CPPFLAGS value: $CPPFLAGS"
echo "-----"

cd $1

if [[ -d src ]]
then
    rm -rf src
fi

makepkg -do --skipgpcheck
cd src
cd $(ls -d */ )

./configure
make
```

Listing 5.3: Automatización de la orquestación y compilación de binarios

5.2.4. Generación de informes de datos en base a los tiempos de compilación y ejecución

Una de las tareas más importantes del proyecto es conocer la degradación del rendimiento de un programa tras haber sido compilado con instrumentación, para ello se ha creado el script *exec.time.py* (ver Listing 5.4) en el lenguaje de programación Python, que permite medir el tiempo de ejecución de un programa o su tiempo de compilación en función de los argumentos que se usen.

```
1  #!/usr/bin/env python3
2
3  import subprocess
4  import time
5  import sys
6
```

```
7 def main():
8     start = time.perf_counter()
9     result = subprocess.run([sys.argv[1], sys.argv[2]])
10    end = time.perf_counter()
11
12    print("Time: {:.2f}".format(end-start))
13
14    if __name__ == '__main__':
15        main()
```

Listing 5.4: Script para medir el tiempo de ejecución o compilación de un programa

Para calcular el tiempo de compilación se deben usar los argumentos *make* y el nombre del binario que se desee compilar ejecutando el script desde el directorio en el que se encuentre el fichero Makefile.

5.2.5. Configuración del software de fuzzing para inicio y finalización de las pruebas

Cada vez que se han realizado pruebas relacionadas con el proyecto se han reproducido en la propia distribución Linux nativa del estudiante. Es por esta razón por la que se han creado dos scripts en *Bash*, *start_afl_work.sh* (ver Listing 5.5) y *finish_afl_work.sh* (ver Listing 5.6) que permiten configurar el software de fuzzing y restablecer dicha configuración para alterar lo mínimo posible la máquina.

```
#!/bin/bash

echo core > /proc/sys/kernel/core_pattern
echo performance | tee /sys/devices/system/cpu/cpu*/cpufreq/
scaling_governor
```

Listing 5.5: Script para realizar la configuración inicial de AFL++

```
#!/bin/bash

cp /home/xfear/Desktop/TFG/core_pattern.old /proc/sys/kernel/
core_pattern
echo ondemand | tee /sys/devices/system/cpu/cpu*/cpufreq/
scaling_governor
```

Listing 5.6: Script para restablecer la configuración inicial tras las pruebas

La configuración inicial se resguarda en el fichero *core_pattern.old* y se copia cada vez que se finalizan las pruebas. Por otro lado, el software de fuzzing pide al usuario que active el modo *performance* del procesador al tener este una frecuencia variable, puesto que de lo contrario, se podrían

arrojar valores erróneos y el rendimiento del software de pruebas se vería mermado.

Capítulo 6

Pruebas

Tras realizar un análisis del problema, un posterior diseño y, por último, una implementación del mismo, se llega a la última parte práctica del proyecto, en la cual se recogen los datos de las pruebas que se realicen para dar paso a las conclusiones.

Este capítulo estará dividido en 3 secciones, primero, un análisis estático de los programas, segundo, un análisis dinámico y, por último, técnicas de *fuzzing* aplicadas a los binarios generados. En este proyecto no se analizarán todos los programas encontrados anteriormente, debido a la cantidad de binarios que se hallaron con el bit SUID activado y a dificultades de compilación, ya que al no ser todos de tipo Open Source no se puede acceder a su código fuente para una posterior compilación.

Cabe destacar que las pruebas se realizan principalmente sobre utilidades propias de la distribución Linux del estudiante, Manjaro. Esta distribución es bastante famosa en la comunidad, por lo que la mayoría de los binarios estarán correctamente revisados y no se detectarán errores graves que se puedan resaltar.

Todos los ficheros de código usados en este proyecto y los análisis relevantes estarán en el repositorio de GitHub del estudiante, accesible en la bibliografía de este mismo trabajo [17].

6.1. Análisis estático

Debido a la facilidad que presenta el compilador *CLANG* para generar informes, será usado su analizador estático durante todas las pruebas de análisis estático, ya que permitirá comprobar con mayor agilidad la cantidad de errores y sus tipos.

A continuación, se presentan dos cuadros en los que se recogen todos los errores y su tipo que se han encontrado tras recurrir al análisis estático de los binarios (ver Cuadro 6.1) (ver Cuadro 6.2). Los dos cuadros podrían estar fusionados en uno, pero para una mejor visualización se han dividido en dos.

Se debe resaltar que no todos son programas unitarios, pues también se ha realizado análisis estático sobre un paquete entero, debido a que el fichero *Makefile* encargado de la compilación de cada utilidad en ese paquete no estaba configurado de forma que se pudiese generar cada binario por separado. Este es el caso de los paquetes *cronie*, *pam*, *fuse2*, *nfs-utils* y *shadow*.

Cuadro 6.1: Errores encontrados en los binarios analizados 1

Error	crontab	sudo	su	chsh	chfn	mount
Allocator sizeof operand mismatch	1	-	-	-	-	-
Argument with "nonnull" attribute passed null	1	1	-	-	-	-
Dead assignment	1	5	2	2	2	2
Dead increment	1	-	-	-	-	-
Dead nested assignment	1	-	8	8	8	8
Use of zero allocated	1	-	-	-	-	-
Assigned value is garbage or undefined	-	48	-	-	-	-
Dereference of undefined pointer value	-	22	-	-	-	-
Result of operation is garbage or undefined	-	3	2	2	2	2
Unitialized argument value	-	98	-	-	-	-
Memory leak	-	-	-	10	-	-

Cuadro 6.2: Errores encontrados en los binarios analizados 2

Error	umount	shadow	unix_chkpwd	fusemount	mount.nfs
Argument with "nonnull" attribute passed null	1	1	-	-	-
Dead assignment	1	5	2	2	2
Dead increment	1	-	-	-	-
Dead nested assignment	1	-	8	8	8
Use of zero allocated	1	-	-	-	-
Assigned value is garbage or undefined	-	48	-	-	-
Result of operation is garbage or undefined	-	3	2	2	2
Unitialized argument value	-	98	-	-	-
Memory leak	-	-	-	10	-

Tras tener todos los informes generados, se deben revisar y analizar para detectar posibles falsos positivos o reafirmar que se trata de un fallo que

pueda provocar una vulnerabilidad en el programa que pueda ser explotada.

Se analizarán en diferentes secciones en función del programa los errores encontrados. Por razones de tiempo no se analizarán todos los informes de los binarios, puesto que llevaría más tiempo del que se dispone para la elaboración de este proyecto. Además, no todos los errores son igual de importantes.

6.1.1. Cronie

La utilidad *Cron* permite ejecutar acciones de forma periódica, puede ser tras cada reinicio de la máquina, semanalmente, diariamente etc. En este caso, *crontab* permite administrar la tabla que resume la planificación de las entradas del demonio *cron*, es decir, recoge las acciones que se realizarán y su periodicidad [26].

En este paquete se han detectado un total de seis errores, de diferentes categorías. A continuación, se analizarán con más detalle.

Allocator sizeof operand mismatch

Este error ocurre cuando se realiza un *casting*¹ entre dos tipos de variables sin comprobar si ambos tipos son compatibles o no para que no se pierda información al usarla o al realizar llamadas de una función sobre ella.

En la imagen (ver Figura 6.1) se muestra el proceso, en el cual se realiza una llamada a *calloc* para reservar memoria en la variable *e*. Se reserva *sizeof(entry)* número de objetos del tamaño *sizeof(char)* y, por último, se realiza un *casting* al tipo *entry*. En este caso la variable *e* almacena una entrada del demonio *cron* con la finalidad de crear una tarea como se detalló anteriormente. Este código se encuentra en el fichero *entry.c* dentro de la función *load_entry*, que se usa para leer entradas desde la herramienta *crontab*.

```
e = (entry *) calloc(sizeof (entry), sizeof (char));  
  
if (e == NULL) {  
    ecode = e_memory;  
    goto eof;  
}
```

Result of 'calloc' is converted to a pointer of type 'entry', which is incompatible with sizeof operand type 'char'

Figura 6.1: Informe de allocator sizeof operand mismatch en el paquete cronie

¹Método que permite convertir una variable de un tipo en otro.

Realmente se está produciendo un falso positivo, pues se están guardando tantos bytes como tenga de tamaño la estructura *entry*, y se hace un *casting* a un puntero a dicha estructura. El analizador lo detecta como error puesto que se están almacenando menos bytes de los que corresponde, pero lo que se está haciendo es reservar el tamaño íntegro de la estructura *entry* y guardar su dirección en la variable *e*, para acceder a la información más tarde.

Argument with “nonnull” attribute passed null

Este error se produce cuando uno de los argumentos que se pasa como parámetro a una función no puede contener el valor *NULL* y se detecta que se está enviando ese argumento con dicho valor.

En la imagen del informe generado (ver Figura 6.2) se puede ver la función a la que se le está pasando un argumento con valor *NULL*, en este caso *execvpe*. Los argumentos *argv[0]* y *argv* no cambian durante toda la ejecución del programa y se inicializan al empezar la ejecución. En el caso de la última variable, *jobenv*, se pasa como argumento en la función en la que se ejecuta este fragmento de código. No se puede conocer de donde viene este valor, pero de ser *NULL*, el programa incurriría en un comportamiento indefinido, por lo que no se puede saber el comportamiento que podría tener el programa tras ocurrir esto. Este error ocurre en el fichero *popen.c* en la función *cron_popen*.

```
if (execvpe(argv[0], argv, jobenv) < 0) {  
25 ← Null pointer passed to 1st parameter expecting 'nonnull'  
    int save_errno = errno;  
    log_it("CRON", getpid(), "EXEC FAILED", program, save_errno);  
    if (*type != 'r') {  
        while (0 != (out = read(STDIN, buf, PIPE_BUF))) {  
            if ((out == -1) && (errno != EINTR))  
                break;  
        }  
    }  
    _exit(1);  
}
```

Figura 6.2: Informe de argument with “nonnull” attribute passed null en el paquete cronie

Dead Assignment

Un *Dead Assignment* se produce cuando se asigna un valor a una variable y nunca más es leído, lo que se considera como innecesario por parte del

compilador.

En el fragmento de código del informe generado (ver Figura 6.3) se puede apreciar un ejemplo de ello, en el que se aplica a la variable, que contiene un descriptor de fichero, *fd* el valor -1 con la intención de limpiar su contenido. No presenta un error real, puesto que la variable no se vuelve a leer en el resto de ejecución de la función. Este error se encuentra en el fichero *cron.c* en la función principal del programa.

```
/* obtain a random scaling factor for RANDOM_DELAY */
if (gettimeofday(&tv, &tz) != 0)
    tv.tv_usec = 0;
srandom((unsigned int)(pid + tv.tv_usec));
RandomScale = (double)random() / (double)(1lu << 31);
snprintf(buf, sizeof(buf), "RANDOM_DELAY will be scaled with factor %d%% if used.", (int)(RandomScale*100));
log_it("CRON", pid, "INFO", buf, 0);

acquire_daemonlock(0);

fd = -1;
```

Value stored to 'fd' is never read

Figura 6.3: Informe de dead assignment en el paquete cronie

Dead increment

Es similar al anterior error. Se produce cuando se incrementa una variable en repetidas ocasiones pero nunca es leída, por lo que el incremento es innecesario.

Este error ocurre en el fichero *pw_dup.c* dentro de la función *pw_dup* (ver Figura 6.4).

```
if (pw->pw_shell) {
    (void)memcpy(cp, pw->pw_shell, ssize);
    newpw->pw_shell = cp;
    cp += ssize;
}
```

Value stored to 'cp' is never read

Figura 6.4: Informe de dead increment en el paquete cronie

Dead nested assignment

Análogo a los dos anteriores, pero en este error se hace referencia a una variable anidada, es decir, una variable que se inicializa con la salida de una función pero nunca es usada.

La variable *num* en el fragmento de código generado no es accedida posteriormente, por lo que se presenta este error (ver Figura 6.5). Se produce en el fichero *misc.c* en la función *acquire_daemonlock*.

```
if (trylock_file(fd) < OK) {
    int save_errno = errno;

    memset(buf, 0, sizeof (buf));
    if ((num = read(fd, buf, sizeof (buf) - 1)) > 0 &&
        (otherpid = strtol(buf, &ep, 10)) > 0 &&
        ep != buf && *ep == '\n' && otherpid != LONG_MAX) {
        snprintf(buf, sizeof (buf),
            "can't lock %s, otherpid may be %ld", pidfile, otherpid);
    }
    else {
        snprintf(buf, sizeof (buf),
            "can't lock %s, otherpid unknown", pidfile);
    }
    fprintf(stderr, "%s: %s: %s\n", ProgramName, buf,
        strerror(save_errno));
    log_it("CRON", pid, "DEATH", buf, save_errno);
    exit(ERROR_EXIT);
}
```

Although the value stored to 'num' is used in the enclosing expression, the value is never actually read from 'num'

Figura 6.5: Informe de dead nested assignment en el paquete cronie

Memory error

En este caso se produce un problema de *Use of zero-allocated-memory* que quiere decir que se está haciendo uso de una variable a la que se le ha asignado 0 bytes de memoria² [23].

La función *malloc* podría reservar 0 bytes de memoria a la variable *envvar_name*, puesto que el valor de *envvar_name_size* viene dado por la función *find_envvar* y en el que se podría devolver dicha variable con un valor igual a 2, provocando el caso que se describe en el problema (ver Figura 6.6). De ocurrir esto, el programa seguramente forzaría su detención ya que se intenta escribir en una zona de memoria sin haber reservado ningún byte de la misma para dicha escritura. Se produce en el fichero *cronie_common.c* en la función *expand_envvar*.

²Las funciones *calloc*, *malloc* y *realloc* aceptan el valor 0 como argumento, no se reserva memoria pero un puntero válido es devuelto y el bloque de memoria puede ser más tarde modificado por *realloc*.

```

while (find_envvar(source, &envvar_p, &envvar_name_size)) {
  1 Loop condition is true. Entering loop body →

  char *envvar_name, *envvar_value;
  size_t prefix_size;

  /* Copy content before env var name */
  prefix_size = envvar_p - source;

  if (prefix_size > 0) {
    2 ← Assuming 'prefix_size' is <= 0 →

    3 ← Taking false branch →

    if ((strlen(result) + prefix_size + 1) > max_size) {
      goto too_big;
    }
    strncat(result, source, prefix_size);

    /* skip envvar name */
    source = envvar_p + envvar_name_size;

    /* copy envvar name, ignoring $, { and } chars */
    envvar_p++;
    envvar_name_size--;

    if (*envvar_p == '{') {
      4 ← Assuming the condition is true →

      5 ← Taking true branch →

      envvar_p++;
      envvar_name_size = envvar_name_size - 2;
    }

    envvar_name = malloc(envvar_name_size + 1);
    6 ← Memory is allocated →

    strncpy(envvar_name, envvar_p, envvar_name_size);
    envvar_name[envvar_name_size] = '\0';

    7 ← Use of zero-allocated memory
  }
}

```

Figura 6.6: Informe de memory error en el paquete cronie

6.1.2. Sudo

El programa *sudo* se usa principalmente para ejecutar un comando como si fuese otro usuario, normalmente el super usuario. Se han detectado 177 errores en el paquete que contiene esta utilidad, pero no serán analizados todos por los motivos que se han expuesto anteriormente. Casi todos los errores se presentan en el fichero *gram.c* en la función *sudoers*. Realmente

no son errores importantes porque el fichero se compone de una estructura de control *switch-case* que contiene 151 casos y se pueden dar fallos tras realizar un proceso compuesto por más de 20 tomas de decisiones en función de los valores de las variables y su ejecución.

En la mayoría de casos es difícil dar un veredicto final sobre si el fallo podría llegar a ser una vulnerabilidad importante en el programa, puesto que se produce en casos extraños y tras diferentes operaciones con variables inicializadas por funciones de otros ficheros, por lo que se requeriría un análisis más profundo de este paquete para determinar si se trata o no de un fallo grave de seguridad.

Argument with “nonnull” attribute passed null

Se puede dar una situación en la que el puntero *data* que se pasa como argumento a la función *memcpy* valga *NULL* (ver Figura 6.7). Puesto que este puntero señala a la dirección de memoria de la que se lee, y ésta no es válida, se daría un error de *segmentation fault (core dumped)*, provocando que el programa fuerce su detención. Tras realizar un análisis más exhaustivo del informe, esta condición se dará en una situación muy inusual, puesto que se debe haber reproducido el mismo proceso que señala el analizador, que contiene hasta 28 tomas de decisiones en función de los valores de las variables.

El error se ha producido en el fichero *protobuf-c.c* en la función *protobuf_c_buffer_simple_append*.

```
memcpy(simp->data + simp->len, data, len);
```

29 ← Null pointer passed to 2nd parameter expecting 'nonnull'

Figura 6.7: Informe de argument with “nonnull” attribute passed null en el paquete sudo

Assigned value is garbage or undefined

Se trata de un error lógico que hace referencia que el valor asignado en una variable contiene basura o está indefinido, es decir, no se ha inicializado correctamente antes de su lectura. Provocaría un comportamiento indefinido en el programa y se ha reproducido en hasta 48 ocasiones, Algunos de ellos pueden presentar este fallo tras haber ocurrido una secuencia de decisiones de hasta 20 casos.

Dereference of undefined pointer value

Se produce cuando se accede a un valor indefinido en un puntero, al igual que el anterior, provoca un comportamiento indefinido en el programa y se presenta hasta en 22 ocasiones tras ocurrir una secuencia de decisiones de hasta 20 casos, por lo que el error no sería muy común.

Result of operation is garbage or undefined

Al igual que los anteriores errores de valores indefinidos, en algunas operaciones de comparación dentro de la estructura *switch-case* se producen situaciones en las que el programa podría llegar a un estado indefinido tras una toma de decisiones de hasta 20 casos puesto que el valor de una variable o el campo de un puntero contiene un valor indefinido.

Unitialized argument value

Otro error que provocaría un comportamiento indefinido del programa tras una larga toma de decisiones debido al uso de argumentos no inicializados en funciones.

Dead assignment

No presenta ningún error grave puesto que tanto en la función *yydestruct* como en *sudopersparse* del fichero *gram.c* se inicializan variables cuyo valor nunca más es leído.

6.1.3. Su

Principalmente esta herramienta sirve para cambiar de usuario en una distribución Linux, normalmente se invoca sin argumentos para acceder al super usuario. Ha presentado hasta 12 errores, de los cuales no merece la pena analizar 10, puesto que son código que más tarde no se ha utilizado, como *Dead assignment* y *Dead nested assignment* en la función *ul_SHA1Transform* del fichero *sha1.c*.

Result of operation is garbage or undefined

Este error, que aparece duplicado en los informes y se encuentra en el fichero *strutils.c*, viene dado porque una variable de tipo *uint64_t* contiene un valor indefinido a la hora de realizar una comprobación. A continuación, se demostrará que este error es un falso positivo, puesto que la variable, aunque puede tomar un valor indefinido en una situación particular, nunca llegaría a realizarse dicha comprobación, por lo que la variable nunca es leída.

En la siguiente imagen (ver Figura 6.8), se muestra la función en cuestión, en la que se declara una variable con nombre *tmp* y se pasa junto con otros parámetros a la función *ul_strtou64*. Esta última devuelve un valor que será almacenado en *rc* y sobrescribe la variable *tmp* con un valor si se cumplen ciertas condiciones. Por último, se comprueba si el valor que se devuelve de esta función es 0 y si la variable sobrescrita es mayor que un límite. Este es el caso en el que el analizador detecta que hay un problema.

```
int ul_strtou32(const char *str, uint32_t *num, int base)
{
    uint64_t tmp;
    1 'tmp' declared without an initial value →

    int rc;

    rc = ul_strtou64(str, &tmp, base);
    2 ← Calling 'ul_strtou64' →

    12 ← Returning from 'ul_strtou64' →

    if (rc == 0 && tmp > UINT32_MAX)
        13 ← The left operand of '>' is a garbage value

        rc = -(errno = ERANGE);
    if (rc == 0)
        *num = (uint32_t) tmp;
    return rc;
}
```

Figura 6.8: Informe 1 de result of operation is garbage or undefined en programa su

Si se analiza la función a la que se llama (ver Figura 6.9), el valor de *tmp* o, en su definición, *num*, sólo se sobrescribe en el caso en el que la salida de la función *strtoimax* devuelve un valor igual o superior a 0. En caso contrario, la variable *errno* se sobrescribe con un valor definido previamente (distinto a 0) y se da la condición verdadera en la última unidad de control *if-else*, por lo que se devuelve un valor distinto de 0.

Al ser el valor de la variable *rc* distinto a 0, el valor de *tmp* no se tiene en cuenta y no se llega a leer, por lo que no importa que esté en un estado indefinido.

```

int ul_strtou64(const char *str, uint64_t *num, int base)
{
    char *end = NULL;
    int64_t tmp;

    errno = 0;
    if (str == NULL || *str == '\0')
        3 ← Assuming 'str' is not equal to NULL →

        4 ← Assuming the condition is false →

        5 ← Taking false branch →

        return -EINVAL;

    /* we need to ignore negative numbers, note that for invalid negative
     * number strtoumax() returns negative number too, so we do not
     * need to check errno here */
    tmp = (int64_t) strtoumax(str, &end, base);
    if (tmp < 0)
        6 ← Assuming 'tmp' is < 0 →

        7 ← Taking true branch →

        errno = ERANGE;
    else {
        errno = 0;
        *num = strtoumax(str, &end, base);
    }

    if (errno || str == end || (end && *end))
        8 ← Assuming the condition is false →

        9 ← Assuming 'str' is not equal to 'end' →

        10 ← Assuming 'end' is null →

        return -EINVAL;
    return 0;
    11 ← Returning without writing to 'num' →
}

```

Figura 6.9: Informe 2 de result of operation is garbage or undefined en programa su

6.1.4. Chsh

Este programa permite cambiar el *shell* con el que el usuario inicia sesión. Presenta exactamente los mismos problemas que el binario *su*, puesto que comparten los ficheros *sha1.c* y *strutils.c*. Es decir, al igual que la herramienta anterior no se encuentran fallos de seguridad, pues se reporta principalmente un falso positivo y código sin utilizar.

6.1.5. Chfn

La utilidad *chfn* permite cambiar la información que muestra el programa *finger*, que se encarga de mostrar cuatro piezas de información que

pueden ser editadas mediante *chfn*. Estos campos son: el nombre real, el lugar de trabajo, su teléfono personal y el teléfono de casa.

Al igual que *su* y *chsh*, presenta 10 fallos de código sin utilizar y el falso positivo analizado anteriormente. Además, presenta 10 fallos referentes a fuga de memoria, todos en la función principal del fichero *chfn.c*.

Memory leak

Se conoce como *fuga de memoria* a un fallo que ocurre en un programa cuando se reserva una zona de memoria y tras diferentes operaciones con punteros se queda dicha zona "huérfana", es decir, ningún puntero apunta a dicha zona, por lo que no se puede recuperar y estará reservada pero sin uso o lo que se conoce comúnmente como fuga o *leak*.

En el caso del programa *chfn* se encuentra un falso positivo en los errores, puesto que esta fuga de memoria se produce justo antes de finalizar la ejecución del programa, ya que en todos los lugares que aparece se devuelve acto seguido un valor de *EXIT_FAILURE* o *EXIT_SUCCESS* por lo que no debe preocupar esta fuga (ver Figura 6.10).

```

if (ctl.interactive)
    9 ← Taking true branch →
    ask_info(&ctl);
    add_missing(&ctl);
    10 ← Calling 'add_missing' →
    23 ← Returned allocated memory →

if (!ctl.changed) {
    24 ← Assuming field 'changed' is 0 →
    25 ← Taking true branch →
    printf(_("Finger information not changed.\n"));
    26 ← Potential leak of memory pointed to by 'ctl.newf.office_phone'
    return EXIT_SUCCESS;
}

return save_new_data(&ctl) == 0 ? EXIT_SUCCESS : EXIT_FAILURE;

```

Figura 6.10: Informe de memory leak en programa *chfn*

6.1.6. Mount

Aunque esta utilidad tiene una gran lista de argumentos disponibles, su finalidad es básicamente la de montar un sistema de ficheros, en una ruta del sistema para que se pueda acceder a él.

Al igual que las utilidades de este paquete, comparte errores de código innecesario y el falso positivo de que el resultado de una operación contiene basura o está indefinido. Además, presenta otros errores que deben ser analizados para detectar una posible vulnerabilidad en el software.

Use after-free

Este problema encontrado en el fichero *list.h* en la función *list_del*, aunque hace referencia principalmente al fichero *probe.c*, donde se encuentran las funciones *free_monitor_entry* y *mnt_unref_monitor* que son las verdaderas implicadas en este error. Este error también incurre en un falso positivo por parte del analizador.

El proceso comienza en la función se puede ver el código de la función *mnt_unref_monitor* (ver Figura 6.11). La funcionalidad principal es la de eliminar y liberar uno a uno todos los elementos del monitor *mn* haciendo uso de la función *free_monitor_entry*.

Mientras tanto, se puede comprobar que la función *free_monitor_entry* (ver Figura 6.12) comprueba que la entrada del monitor *me* no sea nula y la elimina de la lista total para después proceder a liberar la memoria.

```

void mnt_unref_monitor(struct libmnt_monitor *mn)
{
    if (!mn)
        1 Assuming 'mn' is non-null →

        2 ← Taking false branch →

        return;

    mn->refcount--;
    if (mn->refcount <= 0) {
        3 ← Assuming field 'refcount' is <= 0 →

        4 ← Taking true branch →

        mnt_monitor_close_fd(mn);          /* destroys all file descriptors */

        while (!list_empty(&mn->ents)) {
            5 ← Loop condition is true. Entering loop body →

            13 ← Loop condition is true. Entering loop body →

            struct monitor_entry *me = list_entry(mn->ents.next,
                                                    struct monitor_entry, ents);
            free_monitor_entry(me);

            6 ← Calling 'free_monitor_entry' →

            12 ← Returning; memory was released →

            14 ← Calling 'free_monitor_entry' →

        }

        free(mn);
    }
}

```

Figura 6.11: Informe 1 de use after-free en programa mount

```

static void free_monitor_entry(struct monitor_entry *me)
{
    if (!me)
        7 ← Assuming 'me' is non-null →
        8 ← Taking false branch →
        15 ← Taking false branch →
        return;
    list_del(&me->ents);
    16 ← Calling 'list_del' →
    if (me->fd >= 0)
        9 ← Assuming field 'fd' is < 0 →
        10 ← Taking false branch →
        close(me->fd);
        free(me->path);
        free(me);
    11 ← Memory is released →
}

```

Figura 6.12: Informe 2 de use after-free en programa mount

El problema aquí viene porque el analizador ha detectado que se estaba ejecutando la función de *list_del* (ver Figura 6.13) sobre una estructura de datos ya liberada, porque aparentemente esta no cambia. Sin embargo, esta función hace exactamente esto, ajustar la lista de punteros a las estructuras originales, por lo que puede parecer que aparentemente no esté cambiando el elemento que se desea liberar, pero la realidad es que sí lo está haciendo.

```

_INLINE_ void __list_del(struct list_head * prev,
                        struct list_head * next)
{
    next->prev = prev;
    prev->next = next;
}

```

Figura 6.13: Informe 3 de use after-free en programa mount

Dereference of null pointer

Se produce cuando se está haciendo referencia al campo de un puntero nulo. El analizador lo detecta en el fichero *dev.c* dentro de la función *blkid_debug_dump_dev*. Tras revisar el análisis generado (ver Figura 6.14), se detecta que se ha producido un falso positivo, puesto que en el código, la función *list_for_each* declarada en *list.h* recorre la estructura *dev*, que previamente se ha comprobado que no tiene el valor *NULL*, asignando valores a *p* en función del siguiente valor de la estructura *dev*. Este bucle que se ejecuta, comprueba si la variable que asigna (*p*) llega al final de la estructura, en ese momento no se ejecuta el interior del bucle. Por lo que no se llega en ningún momento a que el campo *next* del puntero *p* sea nulo.

```
void blkid_debug_dump_dev(blkid_dev dev)
{
    struct list_head *p;

    if (!dev) {
        1 Assuming 'dev' is non-null →

        2 ← Taking false branch →

        printf(" dev: NULL\n");
        return;
    }

    fprintf(stderr, " dev: name = %s\n", dev->bid_name);
    fprintf(stderr, " dev: DEVNO=\"%0x%01x\n", (unsigned long)dev->bid_devno);
    fprintf(stderr, " dev: TIME=\"%lld.%lld\n", (long long)dev->bid_time, (long long)dev->bid_otime);
    fprintf(stderr, " dev: PRI=\"%d\n", dev->bid_pri);
    fprintf(stderr, " dev: flags = 0x%08X\n", dev->bid_flags);

    list_for_each(p, &dev->bid_tags) {
        3 ← Value assigned to 'p' →

        4 ← Loop condition is true. Entering loop body →

        7 ← Access to field 'next' results in a dereference of a null pointer (loaded from variable 'p')

        blkid_tag tag = list_entry(p, struct blkid_struct_tag, bit_tags);
        if (tag)
            5 ← Assuming 'tag' is null →

            6 ← Taking false branch →

            fprintf(stderr, " tag: %s=\"%s\n", tag->bit_name,
                tag->bit_val);
        else
            fprintf(stderr, " tag: NULL\n");
    }
}
```

Figura 6.14: Informe de dereference of null pointer en programa mount

6.1.7. Umount

La utilidad complementaria a montar un sistema de ficheros ha presentado los mismos errores que la anterior herramienta, exceptuando que no se

presentan los errores referentes a que la condición de una rama se evalúa como un valor basura.

6.1.8. Paquete shadow

Este paquete contiene varias utilidades que en caso de que tuviesen una vulnerabilidad podría desencadenar en un escalado de privilegios al ser explotada por un atacante. Puesto que no se ha podido compilar binario por binario para conocer con exactitud los problemas de cada uno, se analizarán algunos reportes de los 74 encontrados que pertenezcan solamente a dichas utilidades, que son las siguientes:

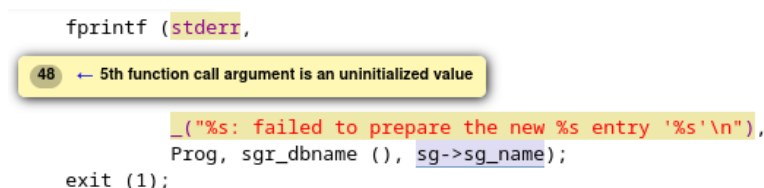
- *Passwd*: cambiar la contraseña de una cuenta de usuario.
- *Gpasswd*: sirve para administrar los grupos de la distribución Linux editando los ficheros */etc/group* y */etc/shadow*.
- *Expiry*: comprueba la fecha de expiración de la contraseña de un usuario y fuerza cambios en la misma cuando son necesarios.
- *Sg*: ejecuta un comando con un id de grupo diferente.
- *Chage*: cambia la información de expiración de la contraseña de un usuario.

Además, se debe añadir, que utilidades como *chsh*, *su* o *chfn*, analizadas anteriormente, también se encuentran en este paquete, por lo que varios errores producidos son los errores que generan estas utilidades en su código.

Unitialized argument value

Se produce en varios ficheros, pero en este caso solamente es objetivo de análisis el que ocurre en el fichero *gpaswd.c* en la función *update_group*.

El analizador estático ha detectado que el campo *sg_name* de la estructura de datos *sg* no está inicializada antes de pasarse a la función correspondiente (ver Figura 6.15).



```
fprintf (stderr,  
48 ← 5th function call argument is an uninitialized value  
_("%s: failed to prepare the new %s entry '%s'\n"),  
Prog, sgr_dbname (), sg->sg_name);  
exit (1);
```

Figura 6.15: Informe 1 de unitialized argument value en programa gpaswd

Tras realizar un análisis exhaustivo el analizador está dando por falsa la condición *is_shadowgrp* en todas las estructuras de datos *if-else* que iniciarían esta estructura de datos, por lo que, como menciona la documentación y como afirma el código (ver Figura 6.16), se ha comprobado que la estructura de datos *sg* no se va a inicializar en ningún momento por el hecho de que no se va a utilizar. En cambio, en el último paso, el analizador da por verdadero el contenido del booleano *is_shadowgrp*, cuando este no ha cambiado ningún momento, pues realmente se inicializa al principio de la ejecución debido a que representa la existencia o no del fichero */etc/gshadow*.

```
/*
 * get_group - get the current information for the group
 *
 * The information are copied in group structure(s) so that they can be
 * modified later.
 *
 * Note: If !is_shadowgrp, *sg will not be initialized.
 */
#ifdef SHADOWGRP
static void get_group (struct group *gr, struct sgrp *sg)
#else
static void get_group (struct group *gr)
#endif
```

Figura 6.16: Informe 2 de uninitialized argument value en programa *gpasswd*

Se concluye que este informe es un falso positivo, debido a que el analizador no está teniendo en cuenta cuándo se inicializa esta variable y está calculando el árbol de decisiones creyendo que cambia en algún momento.

Branch condition evaluates to a garbage value

Este error aparece en el fichero *gpasswd.c* en la función principal del programa por el mismo problema que sucedía anteriormente. El analizador está tomando diferentes valores de la variable *is_shadowgrp* durante la ejecución del análisis, la cual no cambia en ningún momento, por lo que se produce otro falso positivo.

6.2. Análisis dinámico

En esta sección, se ha realizado análisis dinámico a aquellos programas que han podido ser instrumentados, es decir, aquellos de los que se disponía del código fuente y a los que se ha podido aplicar instrumentación.

Los sanitizers usados para estas pruebas han sido todos los detallados en el capítulo 3, aunque no todos han permitido detectar errores, por lo que esta fase de pruebas se dividirá en 3 partes, una por cada desinfectante que

ha encontrado fallos en los binarios: *AddressSanitizer*, *MemorySanitizer* y *LeakSanitizer*.

Para la instrumentación de los desinfectantes se ha usado el compilador *gcc*, exceptuando al sanitizer *MemorySanitizer*, que es propio del compilador *CLANG*.

6.2.1. AddressSanitizer

Este desinfectante ha encontrado errores en utilidades del paquete *shadow*: *passwd*, *gpasswd* y *expiry*. Los errores están directamente relacionados con el *LeakSanitizer* debido a que está integrado en este sanitizer.

En primer lugar, se ha analizado la utilidad *passwd* (ver Figura 6.17). El reporte de errores señala que hay fuga de memoria hasta en 5 ocasiones, con un total de 77 bytes en declaraciones realizadas en el fichero *pwmem.c*.

```
==585329==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 48 byte(s) in 1 object(s) allocated from:
#0 0x7fd94b357411 in __interceptor_calloc /usr/src/debug/gcc/libsanitizer/asan/asan_malloc_linux.cpp:77
#1 0x5564df6f6115 in __pw_dup /home/xfear/Desktop/TFG/binaries_source_code/shadow/src/shadow-4.11.1/lib/pwmem.c:24

Indirect leak of 12 byte(s) in 1 object(s) allocated from:
#0 0x7fd94b30afaa in __interceptor_strdup /usr/src/debug/gcc/libsanitizer/asan/asan_interceptors.cpp:439
#1 0x5564df6f62a4 in __pw_dup /home/xfear/Desktop/TFG/binaries_source_code/shadow/src/shadow-4.11.1/lib/pwmem.c:54

Indirect leak of 9 byte(s) in 1 object(s) allocated from:
#0 0x7fd94b30afaa in __interceptor_strdup /usr/src/debug/gcc/libsanitizer/asan/asan_interceptors.cpp:439
#1 0x5564df6f62e2 in __pw_dup /home/xfear/Desktop/TFG/binaries_source_code/shadow/src/shadow-4.11.1/lib/pwmem.c:61

Indirect leak of 6 byte(s) in 1 object(s) allocated from:
#0 0x7fd94b30afaa in __interceptor_strdup /usr/src/debug/gcc/libsanitizer/asan/asan_interceptors.cpp:439
#1 0x5564df6f625e in __pw_dup /home/xfear/Desktop/TFG/binaries_source_code/shadow/src/shadow-4.11.1/lib/pwmem.c:47

Indirect leak of 2 byte(s) in 1 object(s) allocated from:
#0 0x7fd94b30afaa in __interceptor_strdup /usr/src/debug/gcc/libsanitizer/asan/asan_interceptors.cpp:439
#1 0x5564df6f6218 in __pw_dup /home/xfear/Desktop/TFG/binaries_source_code/shadow/src/shadow-4.11.1/lib/pwmem.c:40

SUMMARY: AddressSanitizer: 77 byte(s) leaked in 5 allocation(s).
```

Figura 6.17: Programa *passwd* con sanitizer *AddressSanitizer*

En segundo lugar, la utilidad *gpasswd* (ver Figura 6.18), que también ha presentado fugas de memoria. En este caso se ha producido hasta en 6 ocasiones con un total de 83 bytes en declaraciones realizadas en el fichero *pwmem.c*.


```

==586140==ERROR: LeakSanitizer: detected memory leaks

    pwck      userdel.o
Direct leak of 48 byte(s) in 1 object(s) allocated from:
    #0 0x7fb98827d411 in __interceptor_calloc /usr/src/debug/gcc/libsanitizer/asan/asan_malloc_linux.cpp:7
    7wd      gpasswd.c      grpconv.o      newgidmap.c
    #1 0x55d120c03af5 in __pw_dup /home/xfear/Desktop/TFG/binaries_source_code/shadow/src/shadow-4.11.1/lib/pwmem.c:24
    pwconv    usermod.o
Indirect leak of 12 byte(s) in 1 object(s) allocated from:
    #0 0x7fb988230faa in __interceptor_strdup /usr/src/debug/gcc/libsanitizer/asan/asan_interceptors.cpp:4
    39      #1 0x55d120c03c84 in __pw_dup /home/xfear/Desktop/TFG/binaries_source_code/shadow/src/shadow-4.11.1/lib/pwmem.c:54
    pwunconv  vipw.o
Indirect leak of 9 byte(s) in 1 object(s) allocated from:
    #0 0x7fb988230faa in __interceptor_strdup /usr/src/debug/gcc/libsanitizer/asan/asan_interceptors.cpp:
    groupdel.c      id.o      new_subid_ra
    439      #1 0x55d120c03cc2 in __pw_dup /home/xfear/Desktop/TFG/binaries_source_code/shadow/src/shadow-4.11.1/lib/pwmem.c:61
Indirect leak of 6 byte(s) in 1 object(s) allocated from:
    #0 0x7fb988230faa in __interceptor_strdup /usr/src/debug/gcc/libsanitizer/asan/asan_interceptors.cpp:4
    39      #1 0x55d120c03c3e in __pw_dup /home/xfear/Desktop/TFG/binaries_source_code/shadow/src/shadow-4.11.1/lib/pwmem.c:47
Indirect leak of 6 byte(s) in 1 object(s) allocated from:
    #0 0x7fb988230faa in __interceptor_strdup /usr/src/debug/gcc/libsanitizer/asan/asan_interceptors.cpp:4
    39      #1 0x55d120c03bb7 in __pw_dup /home/xfear/Desktop/TFG/binaries_source_code/shadow/src/shadow-4.11.1/lib/pwmem.c:33
Indirect leak of 2 byte(s) in 1 object(s) allocated from:
    #0 0x7fb988230faa in __interceptor_strdup /usr/src/debug/gcc/libsanitizer/asan/asan_interceptors.cpp:4
    39      #1 0x55d120c03bf8 in __pw_dup /home/xfear/Desktop/TFG/binaries_source_code/shadow/src/shadow-4.11.1/lib/pwmem.c:40
SUMMARY: AddressSanitizer: 83 byte(s) leaked in 6 allocation(s).

```

Figura 6.18: Programa gpasswd con sanitizer AddressSanitizer

Por último, otra de las herramientas de este paquete que se ha analizado es *expiry* (ver Figura 6.19). Al igual que la herramientas anteriores, presenta fuga de memoria de hasta 83 bytes en 6 declaraciones en el fichero *pwmem.c*.

```

==585829==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 48 byte(s) in 1 object(s) allocated from:
#0 0x7f8c982c0411 in __interceptor_calloc /usr/src/debug/gcc/libsanitizer/asan/asan_malloc_linux.cpp:
77
#1 0x55a342bc3b85 in __pw_dup /home/xfear/Desktop/TFG/binaries_source_code/shadow/src/shadow-4.11.1/l
ib/pwmem.c:24

Indirect leak of 12 byte(s) in 1 object(s) allocated from:
#0 0x7f8c98273faa in __interceptor_strdup /usr/src/debug/gcc/libsanitizer/asan/asan_interceptors.cpp:
439
#1 0x55a342bc3d14 in __pw_dup /home/xfear/Desktop/TFG/binaries_source_code/shadow/src/shadow-4.11.1/l
ib/pwmem.c:54

Indirect leak of 9 byte(s) in 1 object(s) allocated from:
#0 0x7f8c98273faa in __interceptor_strdup /usr/src/debug/gcc/libsanitizer/asan/asan_interceptors.cpp:
439
#1 0x55a342bc3d52 in __pw_dup /home/xfear/Desktop/TFG/binaries_source_code/shadow/src/shadow-4.11.1/l
ib/pwmem.c:61

Indirect leak of 6 byte(s) in 1 object(s) allocated from:
#0 0x7f8c98273faa in __interceptor_strdup /usr/src/debug/gcc/libsanitizer/asan/asan_interceptors.cpp:
439
#1 0x55a342bc3cce in __pw_dup /home/xfear/Desktop/TFG/binaries_source_code/shadow/src/shadow-4.11.1/l
ib/pwmem.c:47

Indirect leak of 6 byte(s) in 1 object(s) allocated from:
#0 0x7f8c98273faa in __interceptor_strdup /usr/src/debug/gcc/libsanitizer/asan/asan_interceptors.cpp:
439
#1 0x55a342bc3c47 in __pw_dup /home/xfear/Desktop/TFG/binaries_source_code/shadow/src/shadow-4.11.1/l
ib/pwmem.c:33

Indirect leak of 2 byte(s) in 1 object(s) allocated from:
#0 0x7f8c98273faa in __interceptor_strdup /usr/src/debug/gcc/libsanitizer/asan/asan_interceptors.cpp:
439
#1 0x55a342bc3c88 in __pw_dup /home/xfear/Desktop/TFG/binaries_source_code/shadow/src/shadow-4.11.1/l
ib/pwmem.c:40

SUMMARY: AddressSanitizer: 83 byte(s) leaked in 6 allocation(s).

```

Figura 6.19: Programa expiry con sanitizer AddressSanitizer

Todos ellos son falsos positivos, pues se producen siempre y cuando se llame al programa sin darle valores como argumento, por lo que en este caso los binarios lo que hacen es utilizar otro módulo para inicializar las variables que el analizador detecta como fugas de memoria, ya que se usan funciones para asignar y liberar memoria. El analizador estático lo que ha hecho es inicializar sus variables y detectar la memoria antes de la llamada a este módulo, por lo que se detecta una fuga de memoria, pero ya se ha comprobado que es un falso positivo tras revisar el código de *pwmem.c*.

6.2.2. MemorySanitizer

El desinfectante para la detección de lecturas de memoria no inicializadas ha detectado fallos en los programas *su*, *chsh* y *chfn* del paquete *util-linux* y el programa *passwd* del paquete *shadow*.

Todos los programas analizados presentan una detección de lectura de memoria no inicializada en la función *main* del fichero fuente de cada programa. Relacionada directamente con la librería *pam*. Como ejemplo se muestra la salida de la ejecución del programa *su* (ver Figura 6.20) tras haber sido instrumentado con este desinfectante.

```

Uninitialized bytes in __interceptor_fopen at offset 0 inside [0x701000000140, 14)
==226143==WARNING: MemorySanitizer: use-of-uninitialized-value
#0 0x7f61e0dd4825 (/usr/lib/libpam.so.0+0x5825)
#1 0x7f61e0dd5c7f (/usr/lib/libpam.so.0+0x6c7f)
#2 0x7f61e0dd7642 (/usr/lib/libpam.so.0+0x8642)
#3 0x55b8f2d21006 in supam_authenticate su-common.c
#4 0x55b8f2d1b6f3 in su_main (/home/xfear/Desktop/TFG/binaries_source_code/util-linux/src/util-linux-2.38/su+0xa96f3)
#5 0x55b8f2d187cb in main (/home/xfear/Desktop/TFG/binaries_source_code/util-linux/src/util-linux-2.38/su+0xa67cb)
#6 0x7f61e0adf28f (/usr/lib/libc.so.6+0x2928f)
#7 0x7f61e0adf349 in __libc_start_main (/usr/lib/libc.so.6+0x29349)
#8 0x55b8f2c94594 in _start /build/glibc/src/glibc/csu/../sysdeps/x86_64/start.S:115

SUMMARY: MemorySanitizer: use-of-uninitialized-value (/usr/lib/libpam.so.0+0x5825)
Exiting

```

Figura 6.20: Programa su con sanitizer MemorySanitizer

No se sabe a ciencia cierta si puede ser un falso positivo o no, pues depende de la librería *pam* que se tenga instalada en ese momento en la compilación del programa.

6.2.3. LeakSanitizer

En este último caso, se han detectado errores en dos programas del paquete *shadow*; *passwd* y *gpasswd*.

Por un lado, el binario *passwd* (ver Figura 6.21) presenta una fuga de memoria en una operación de lectura en la función *main* del archivo *passwd.c*.

```

LeakSanitizer:DEADLYSIGNAL
==560208==ERROR: LeakSanitizer: SEGV on unknown address 0xffffffffffff020 (pc 0x7f9945e36797 bp 0x7ffccf53e310 sp 0x7ffccf53e280 T0)
==560208==The signal is caused by a READ memory access.
#0 0x7f9945e36797 in __lsan::GetMallocUsableSize(void const*) /usr/src/debug/gcc/libsanitizer/lsan/lsan_allocator.cpp:148
#1 0x7f994515923b (/usr/lib/libnss_systemd.so.2+0x3b23b)
#2 0x7f9945141c89 (/usr/lib/libnss_systemd.so.2+0x23c89)
#3 0x7f99451513c0 (/usr/lib/libnss_systemd.so.2+0x333c0)
#4 0x7f99451457e3 (/usr/lib/libnss_systemd.so.2+0x277e3)
#5 0x7f994514a1e1 (/usr/lib/libnss_systemd.so.2+0x2c1e1)
#6 0x7f994512d3ce in _nss_systemd_getspnam_r (/usr/lib/libnss_systemd.so.2+0xf3ce)
#7 0x7f9945cf3912 in getspnam_r (/usr/lib/libc.so.6+0x117912)
#8 0x7f9945e213c1 in pam_modutil_getspnam (/usr/lib/libpam.so.0+0xa3c1)
#9 0x7f99457468ab (/usr/lib/security/pam_unix.so+0x58ab)
#10 0x7f994574690e (/usr/lib/security/pam_unix.so+0x590e)
#11 0x7f9945746211 (/usr/lib/security/pam_unix.so+0x5211)
#12 0x7f9945744590 in pam_sm_chauthtok (/usr/lib/security/pam_unix.so+0x3590)
#13 0x7f9945e1a919 (/usr/lib/libpam.so.0+0x3919)
#14 0x7f9945e1f30e in pam_chauthtok (/usr/lib/libpam.so.0+0x830e)
#15 0x56248a26754a in do_pam_passwd /home/xfear/Desktop/TFG/binaries_source_code/shadow/src/shadow-4.11.1/libmisc/pam_pass.c:46
#16 0x56248a265c21 in main /home/xfear/Desktop/TFG/binaries_source_code/shadow/src/shadow-4.11.1/src/passwd.c:1048
#17 0x7f9945c0528f (/usr/lib/libc.so.6+0x2928f)
#18 0x7f9945c05349 in __libc_start_main (/usr/lib/libc.so.6+0x29349)
#19 0x56248a266314 in _start (/home/xfear/Desktop/TFG/binaries_source_code/shadow/src/shadow-4.11.1/src/passwd+0x4314)

LeakSanitizer can not provide additional info.
SUMMARY: LeakSanitizer: SEGV /usr/src/debug/gcc/libsanitizer/lsan/lsan_allocator.cpp:148 in __lsan::GetMallocUsableSize(void const*)
==560208==ABORTING

```

Figura 6.21: Programa passwd con sanitizer LeakSanitizer

Por otro lado, el programa *gpasswd* presenta fugas de memoria de hasta 83 bytes en 6 declaraciones (ver Figura 6.22). La primera, de 48 bytes en el archivo *pwmem.c* fue detectada por el *AddressSanitizer* mientras que, la segunda, se produce de forma indirecta en un archivo de la librería *libc* de la que se hace uso.

```
==560425==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 48 byte(s) in 1 object(s) allocated from:
    #0 0x7f7896b0cf68 in __interceptor_calloc /usr/src/debug/gcc/libsanitizer/lsan/lsan_interceptors.cpp:90
    #1 0x55fa25a7d7b5 in __pw_dup /home/xfear/Desktop/TFG/binaries_source_code/shadow/src/shadow-4.11.1/lib/pwmem.c:24

Indirect leak of 35 byte(s) in 5 object(s) allocated from:
    #0 0x7f7896b0d453 in __interceptor_malloc /usr/src/debug/gcc/libsanitizer/lsan/lsan_interceptors.cpp:75
    #1 0x7f789692fd5e in __strdup (/usr/lib/libc.so.6+0x9ed5e)

SUMMARY: LeakSanitizer: 83 byte(s) leaked in 6 allocation(s).
```

Figura 6.22: Programa gpasswd con sanitizer LeakSanitizer

Al igual que se concluyó con el desinfectante *AddressSanitizer* se trata de un falso positivo al inicializarse las funciones del desinfectante antes de que se inicialicen las variables que se van a asignar y liberar más tarde.

6.3. Fuzzing

Tras haber realizado un análisis estático y dinámico de las utilidades, como paso final se han realizado pruebas de fuzzing sobre binarios instrumentados. Al igual que en anteriores secciones, sólo se ha tenido en cuenta aquellos programas de los que se disponía el código fuente, ya que esta técnica pierde efectividad al ejecutarla contra programas no instrumentados para ello y, puesto que la finalidad de este trabajo es didáctica, se ha optado por usar solamente binarios cuyo código fuente estuviese disponible.

En esta sección no se han examinado los mismos binarios que en el análisis estático o dinámico, la razón principal es que la distribución Linux que se ha usado para este proyecto contiene la versión más reciente de estos binarios, es decir, esos programas están en continuo testeo por parte de los desarrolladores, por lo que se ha recurrido a software de una distribución más antigua, como lo es Ubuntu 14.04, cuyos paquetes son aproximadamente del año 2014.

Por otro lado, tampoco se han analizado los mismos ejecutables, puesto que no todos podían ser ejecutados mediante *AFL++* o devolvían tiempos de respuesta lo suficientemente altos como para que esta técnica no fuese efectiva.

En primer lugar, se muestra una imagen (ver Figura 6.23) de la ejecución de *AFL++* con el binario *bash* como ejemplo de la ejecución de este software. En ella se demuestra la velocidad con la que este programa puede encontrar fallos en un binario. En solo 10 minutos esta técnica ha provocado fallos hasta en 350 ocasiones el binario *bash*, de los que se han almacenado hasta 45 informes. Todo ello haciendo uso de casi 350 mil ejecuciones.


```

→ ./bash
[xfear@xfear bash-4.3]$ ~lo

malloc: unknown:0: assertion botched
free: start and end chunk sizes differ
last command: ~lo
Aborting...[1] 1322249 IOT instruction (core dumped) ./bash

```

Figura 6.24: Crash en el binario bash

Al abrir el binario en el depurador de gdb se puede recoger más información de lo que está pasando, donde se hace referencia a un problema con la función *malloc* (ver Figura 6.25).

```

(gdb) bt
#0 0x00007ffff7da636c in ?? () from /usr/lib/libc.so.6
#1 0x00007ffff7d56838 in raise () from /usr/lib/libc.so.6
#2 0x00007ffff7d40535 in abort () from /usr/lib/libc.so.6
#3 0x00005555555d1dcd in programming_error (format=<optimized out>) at error.c:176
#4 0x000055555556dc984 in xbotch (s=0x830c5 <error: Cannot access memory at address 0x830c5>, file=0x0, line=0, mem=<optimized out>, e=<optimized out>) at malloc.c:319
#5 internal_free (mem=0x555555960488, file=<optimized out>, line=<optimized out>, flags=<optimized out>) at malloc.c:906
#6 0x00007ffff77958ee in ?? () from /usr/lib/libnss_systemd.so.2
#7 0x00007ffff779adc0 in ?? () from /usr/lib/libnss_systemd.so.2
#8 0x00007ffff7783baa in ?? () from /usr/lib/libnss_systemd.so.2
#9 0x00007ffff77935fd in ?? () from /usr/lib/libnss_systemd.so.2
#10 0x00007ffff77874cd in ?? () from /usr/lib/libnss_systemd.so.2
#11 0x00007ffff778b996 in ?? () from /usr/lib/libnss_systemd.so.2
#12 0x00007ffff776f2b4 in _nss_systemd_getpwnam_r () from /usr/lib/libnss_systemd.so.2
#13 0x00007ffff7df4013 in getpwnam_r () from /usr/lib/libc.so.6
#14 0x00007ffff7df3a79 in getpwnam () from /usr/lib/libc.so.6
#15 0x000055555556d8398 in tilde_expand_word (filename=filename@entry=0x55555595ff28 "~/elihlo") at ./tilde.c:390
#16 0x000055555556d7fef in tilde_expand (string=<optimized out>, string@entry=0x55555595ff08 "~/elihlo") at ./tilde.c:238
#17 0x000055555555a19dd in bash_tilde_expand (s=s@entry=0x55555595ff08 "~/elihlo", assign_p=assign_p@entry=0) at general.c:1002
#18 0x00005555555f29bf in expand_word_internal (word=<optimized out>, quoted=<optimized out>, quoted@entry=0, isexp=<optimized out>, contains_dollar_at=contains_dollar_at@entry=0x7ffffffffffd570, expanded_something=<optimized out>, expanded_something@entry=0x7ffffffffffd574) at subst.c:8345
#19 0x00005555555fb499 in shell_expand_word_list (tlist=0x55555594e188, eflags=<optimized out>) at subst.c:9535
#20 expand_word_list_internal (list=<optimized out>, eflags=eflags@entry=31) at subst.c:9655
#21 0x000055555555fac37 in expand_words (list=0x830c5) at subst.c:9234
#22 0x000055555555b4202 in execute_simple_command (simple_command=0x55555595fd08, pipe_in=-1, pipe_out=-1, async=0, fds_to_close=0x55555594e0a8) at execute_cmd.c:3991
#23 execute_command_internal (command=<optimized out>, command@entry=0x55555595fc48, asynchronous=asynchronous@entry=0, pipe_in=pipe_in@entry=-1, pipe_out=<optimized out>, pipe_out@entry=-1, fds_to_close=<optimized out>, fds_to_close@entry=0x55555594e0a8) at execute_cmd.c:788
#24 0x000055555555acc98 in execute_command (command=0x830c5) at execute_cmd.c:390
#25 0x00005555555589c27 in reader_loop () at eval.c:160
#26 0x00005555555588713 in main (argc=<optimized out>, argv=0x7ffffffffffdcf8, env=<optimized out>) at shell.c:755

```

Figura 6.25: Depuración con gdb del binario bash

Lamentablemente al intentar compilar el binario con los desinfectantes para obtener más información del error provoca un *segmentation fault* cada vez que se intenta ejecutar y no permite un análisis más profundo.

6.3.2. Sudo

Este proceso se ha realizado en un contenedor docker usando la versión sudo-1.8.31p2 puesto que tras la instrumentación del binario se presentaban

diferentes problemas durante su ejecución al hacerlo en la máquina anfitriona.

La instrumentación de este programa ha sido más complicada, puesto que al ser una herramienta que se ejecuta por línea de comandos y no tiene valores de entrada más allá de los propios argumentos que se le proporcionan al invocarse, se debe instrumentar como se explicó en el capítulo 3.

Tras una hora de ejecución del software de *fuzzing* con 6 procesos paralelos se han reportado miles de errores, de los que se han guardado aproximadamente 30 entre todos los procesos, puesto que la mayoría son mutaciones simples en los que varía un bit de estas cadenas iniciales.

A continuación, se procede a analizar las cadenas resultantes que, en este caso, se trata de ficheros con valores hexadecimales que se deben decodificar usando la herramienta *xxd* propia de los sistemas Linux (ver Figura 6.26).

```
root:~# xxd /tmp/out/f01/crashes/id:000006*
00000000: 73df 3131 7375 6480 7375 6480 5e5e 3d5f  s..11sud.sud.^.=
00000010: 5e5e 555e 5e5e 7375 646f 0192 8564 6f65  ^U^U^U^sdo...doe
00000020: 6469 7400 2d41 002d 7300 8065 002d 6700  dit.-A.-s..e.-g.
00000030: 2d5c 002d 6800 2d42 5e5e 555e 5e8c 6d64  -.\. -h.-B^U^U^U.md
00000040: 6f01 9275 646f 6564 6974 002d 4100 2d62  o..udoedit.-A.-b
00000050: 002d 6500 2d00 2d31 7375 6480 7375 6480  .-e.-.-1sud.sud.
00000060: 5e46 5e5f 5e5e 555e 5e5e 7375 646f 0192  ^F^_^^U^U^U^sdo..
00000070: 8564 6f65 6469 7400 2d41 002d 7300 8065  .doedit.-A.-s..e
00000080: 002d 6700 2d48 002d 6800 2d42 5e5e 555e  .-g.-H.-h.-B^U^U^U
00000090: 5e5e 8c75 646f 0192 7564 6f65 6469 73f5  ^^..udo..udoedis.
000000a0: 2d41 002d 6200 2d65 005e 555e 5e5e 8c75  -A.-b.-e.^U^U^U^u
000000b0: 646f 0192 7564 6f65 6469 7400 2d41 002d  do..udoedit.-A.-
000000c0: 6200 2d65 ee2d 002d 3173 7564 8073 7564  b.-e.-.-1sud.sud
000000d0: 805e 465e 5f5e 5e55 5e5e 5e00 8065 002d  .^F^_^^U^U^U^..e.-
000000e0: 6700 2d48 002d 6800 2d42 5e5e 555e 5e00  g.-H.-h.-B^U^U^U^
000000f0: 8000 006f 0192 7564 6f65 6469 7400 2d41  ...o..udoedit.-A
00000100: 002d 6200 2d65 002d 002d 4300 2d45 2d00  .-b.-e.-.-C.-E.-
00000110: 2d43 002d 4500 2d65 002d 6700 2d48 002d  -C.-E.-e.-g.-H.-
00000120: 2d69 ef2d 4b00 4300 2d45 002d 0100 2d67  -i.-K.C.-E.-.-g
00000130: 052d 4800 2d2d 6900 2d4b 002d          .-H.-.-i.-K.-
```

Figura 6.26: Volcado hexadecimal con *xxd* de un informe del binario *sudo*

El problema que presentan estas cadenas es que contienen muchos valores basura, por lo que antes de empezar a trabajar con ellas se deben minimizar con la utilidad *afl-tmin* (ver Listing 6.1).

```
afl-tmin -i <id-crash> -o <fichero-donde-se-almacena-el-caso-
minimizado> -- <ruta al binario>
```

Listing 6.1: Uso de *afl-tmin* para limpiar basura del caso de prueba

La minimización produce una salida más sencilla de entender para el usuario (ver Figura 6.27).

```
root:~# xxd minimized.testcase
00000000: 3065 6469 7400 2d73 0030 3030 3030 5c00  0edit.-s.00000\
00000010: 3030 3030 3030 3030 3030 3030 3030 3030  0000000000000000
00000020: 3030 3030 3030 3030 3030 3030 3030 3030  0000000000000000
00000030: 3030 3030 3030 3030 3030 3030 3030 3030  0000000000000000
00000040: 30                                     0
```

Figura 6.27: Resultado de la minimización de un volcado hexadecimal para el binario `sudo`

Una vez que la cadena está minimizada, se comprueba el fallo en el binario *sudo* instrumentado (ver Listing 6.2).

```
cat minimized.testcase | /pwd/sudo/src/sudo
```

Listing 6.2: Comprobación del error en el binario `sudo`

Esta ejecución resulta en un error de “`malloc() invalid size (unsorted)`” e interrumpe la ejecución del programa. Para comprobar el error se reproducirá en el depurador *gdb* (ver Figura 6.28). La función *malloc* detecta una inconsistencia con el tamaño, por lo que decide abortar la ejecución del programa. Sin embargo, este error ocurre porque se ha producido una corrupción de memoria en el heap.

Si se sigue indagando en el error se puede comprobar que es la función *reallocarray* del fichero *getgrouplist.c* de la línea 101 la que provoca la inconsistencia (ver Figura 6.29).


```

malloc(): invalid size (unsorted)

Program received signal SIGABRT, Aborted.
__GI_raise (sig=sig@entry=6) at ../sysdeps/unix/sysv/linux/raise.c:50
50  ../sysdeps/unix/sysv/linux/raise.c: No such file or directory.
(gdb) bt
#0  __GI_raise (sig=sig@entry=6) at ../sysdeps/unix/sysv/linux/raise.c:50
#1  0x00007ffff7bf6859 in __GI_abort () at abort.c:79
#2  0x00007ffff7c6126e in __libc_message (action=action@entry=do_abort,
fmt=fmt@entry=0x7ffff7d8b298 "%s\n") at ../sysdeps/posix/libc_fatal.c:155
#3  0x00007ffff7c692fc in malloc_printerr (
str=str@entry=0x7ffff7d8da50 "malloc(): invalid size (unsorted)") at malloc.c:5347
#4  0x00007ffff7c6c0b4 in _int_malloc (av=av@entry=0x7ffff7dc0b80 <main_arena>,
bytes=bytes@entry=262148) at malloc.c:3736
#5  0x00007ffff7c6e154 in __GI___libc_malloc (bytes=262148) at malloc.c:3058
#6  0x000055555555a08c7 in sudo_getgrouplist2_v1 (name=0x5555560eb9e8 "root", basegid=0,
groupsp=0x7ffffffffffcd38, ngroupsp=0x7ffffffffffcd34) at ./getgrouplist.c:101
#7  0x000055555555d168e in sudo_make_gidlist_item (pw=0x5555560eb9b8, unused1=<optimized out>, type=1)
at ./pwutil_impl.c:272
#8  0x000055555555cfeec in sudo_get_gidlist (pw=0x5555560eb9b8, type=1) at ./pwutil.c:932
#9  0x000055555555a08c7 in runas_getgroups () at ./match.c:145
#10 0x000055555555b7f1e in runas_setgroups () at ./set_perms.c:1714
#11 set_perms (perm=<optimized out>) at ./set_perms.c:281
#12 0x000055555555e56af in sudoers_lookup (snl=0x55555630310 <sudo_read_nss.snl>, pw=0x5555560eb818,
validated=96, pwflag=0) at ./parse.c:299
#13 0x000055555555b9d5 in sudoers_policy_main (argc=<optimized out>, argv=<optimized out>, pwflag=0,
env_add=<optimized out>, verbose=<optimized out>, closure=0x7ffffffffffd408) at ./sudoers.c:324
#14 0x000055555555b688e in sudoers_policy_check (argc=3, argv=0x555555ee3258 <afl_init_argv.ret+8>,
env_add=0x0, command_info=0x7ffffffffffd4b0, argv_out=0x7ffffffffffd4a8, user_env_out=0x7ffffffffffd4a0)
at ./policy.c:872
#15 0x00005555555594deb in policy_check (plugin=<optimized out>, argc=3,
argv=0x555555ee3258 <afl_init_argv.ret+8>, env_add=0x0, command_info=0x7ffffffffffd4b0,
argv_out=0x7ffffffffffd4a8, user_env_out=0x7ffffffffffd4a0) at ./sudo.c:1140
#16 main (argc=4, argv=<optimized out>, envp=0x7ffffffffffe698) at ./sudo.c:255

```

Figura 6.28: Depuración con gdb del binario sudo

```

(gdb) up
#1  0x00007ffff7bf6859 in __GI_abort () at abort.c:79
79  abort.c: No such file or directory.
(gdb) up
#2  0x00007ffff7c6126e in __libc_message (action=action@entry=do_abort,
fmt=fmt@entry=0x7ffff7d8b298 "%s\n") at ../sysdeps/posix/libc_fatal.c:155
155  ../sysdeps/posix/libc_fatal.c: No such file or directory.
(gdb) up
#3  0x00007ffff7c692fc in malloc_printerr (
str=str@entry=0x7ffff7d8da50 "malloc(): invalid size (unsorted)") at malloc.c:5347
5347  malloc.c: No such file or directory.
(gdb) up
#4  0x00007ffff7c6c0b4 in _int_malloc (av=av@entry=0x7ffff7dc0b80 <main_arena>,
bytes=bytes@entry=262148) at malloc.c:3736
3736  in malloc.c
(gdb) up
#5  0x00007ffff7c6e154 in __GI___libc_malloc (bytes=262148) at malloc.c:3058
3058  in malloc.c
(gdb) up
#6  0x000055555555a08c7 in sudo_getgrouplist2_v1 (name=0x5555560eb9e8 "root", basegid=0,
groupsp=0x7ffffffffffcd38, ngroupsp=0x7ffffffffffcd34) at ./getgrouplist.c:101
101  groups = reallocarray(NULL, grpsize, sizeof(*groups));

```

Figura 6.29: Traza del programa gdb sobre el binario sudo

Llegados a este punto se reconoce que existe un problema de memoria,

por lo que es aquí donde entran en juego los desinfectantes para ayudar a encontrar el fallo. En este caso se va a volver a compilar el programa haciendo uso del *AddressSanitizer* y *UndefinedSanitizer* para intentar encontrar el error (ver Listing 6.3). Es importante el uso del argumento “-g” para habilitar al desinfectante mayor precisión a la hora de arrojar resultados y, si no se añade el argumento “--disable-shared”, el programa *sudo* resulta en un script de Bash que realiza diferentes llamadas en lugar de un binario, por lo que también es importante añadir dicho valor.

```
make clean
./configure CFLAGS='-fsanitize=address,undefined -g' LDFLAGS='-fsanitize=address,undefined' CC=clang --disable-shared
make
```

Listing 6.3: Compilación del binario sudo con AddressSanitizer y UndefinedSanitizer

Tras la compilación, se ejecuta el binario nuevo con el caso minimizado y se comprueba que el sanitizer devuelve un error de *heap buffer overflow* (ver Figura 6.31), por lo que se concluye que existe una vulnerabilidad en esta versión del programa *sudo*.

```

==109209==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x606000000959 at pc 0x0000006569ee bp 0x7fff2e178750
sp 0x7fff2e178748
WRITE of size 1 at 0x606000000959 thread T0
#0 0x6569ed in set_cmd /pwd/sudo-ASAN/plugins/sudoers/./sudoers.c:868:10
#1 0x64a844 in sudoers_policy_main /pwd/sudo-ASAN/plugins/sudoers/./sudoers.c:306:19
#2 0x61d0e3 in sudoers_policy_check /pwd/sudo-ASAN/plugins/sudoers/./policy.c:872:11
#3 0x5506b9 in policy_check /pwd/sudo-ASAN/src/./sudo.c:1140:11
#4 0x5433dc in main /pwd/sudo-ASAN/src/./sudo.c:255:11
#5 0x7f6ce6afd082 in __libc_start_main /build/glibc-SzIz7B/glibc-2.31/csu/../csu/libc-start.c:308:16
#6 0x41da2d in _start (/pwd/sudo-ASAN/src/sudo+0x41da2d)

0x606000000959 is located 0 bytes to the right of 57-byte region [0x606000000920,0x606000000959)
allocated by thread T0 here:
#0 0x49616d in malloc (/pwd/sudo-ASAN/src/sudo+0x49616d)
#1 0x655f41 in set_cmd /pwd/sudo-ASAN/plugins/sudoers/./sudoers.c:854:36
#2 0x64a844 in sudoers_policy_main /pwd/sudo-ASAN/plugins/sudoers/./sudoers.c:306:19
#3 0x61d0e3 in sudoers_policy_check /pwd/sudo-ASAN/plugins/sudoers/./policy.c:872:11
#4 0x5506b9 in policy_check /pwd/sudo-ASAN/src/./sudo.c:1140:11
#5 0x5433dc in main /pwd/sudo-ASAN/src/./sudo.c:255:11
#6 0x7f6ce6afd082 in __libc_start_main /build/glibc-SzIz7B/glibc-2.31/csu/../csu/libc-start.c:308:16

SUMMARY: AddressSanitizer: heap-buffer-overflow /pwd/sudo-ASAN/plugins/sudoers/./sudoers.c:868:10 in set_cmd
Shadow bytes around the buggy address:
0x0c0c7fff80d0: 00 00 00 00 00 00 06 fa fa fa fa 00 00 00 00
0x0c0c7fff80e0: 00 00 04 fa fa fa fa 00 00 00 00 00 00 00 fa
0x0c0c7fff80f0: fa fa fa fa fd fd fd fd fd fd fa fa fa fa
0x0c0c7fff8100: fd fd fd fd fd fd fa fa fa fa 00 00 00 00
0x0c0c7fff8110: 00 00 00 00 fa fa fa 00 00 00 00 00 00 00 00
=>0x0c0c7fff8120: fa fa fa 00 00 00 00 00 00 00[01]fa fa fa fa
0x0c0c7fff8130: fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c0c7fff8140: fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c0c7fff8150: fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c0c7fff8160: fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c0c7fff8170: fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
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
Shadow gap: cc
==109209==ABORTING

```

Figura 6.30: Buffer Overflow en el binario sudo

Este error fue reportado en el CVE-2021-3156 [9] y describe que las versiones anteriores a 1.9.5p2 contienen un error de *off-by-one* que puede resultar en un *heap buffer overflow* permitiendo un escalado de privilegios via “`sudoedit -s`” y un argumento de línea de comandos que termine con el caracter “\”.

Pese a que este error fue encontrado en 2021, hacía más de 10 años que se encontraba en el binario *sudo* y aplicando técnicas de *fuzzing* y desinfectantes se pudo descubrir en pocas horas.

6.3.3. Tcpcdump

La herramienta *tcpdump* permite hacer captura de paquetes de red y análisis de protocolos. Se basa en la interfaz *libcap*, que es una interfaz independiente para capturar datagramas de red a nivel de usuario. A pesar de su nombre, esta utilidad también puede usarse para capturar tráfico UDP e ICMP, además del tráfico TCP. [13].

Esta utilidad también ha sido sometida a técnicas de *fuzzing* en una versión antigua para aumentar la probabilidad de encontrar fallos, en este caso ha sido la versión de 2016 *tcpdump-4.8.0*. En este caso la preparación del fichero que *AFL++* mutará ha diferido de los anteriores casos. Para este proceso, el fichero ha sido uno propio generado por *tcpdump* tras la captura de 4 paquetes, para ello se ha usado la propia herramienta para volcar los resultados en un fichero que se usará posteriormente como prueba (ver Listing 6.4).

```
sudo ./tcpdump -i <interfaz-de-red> -s0 -w <fichero-de-salida>
```

Listing 6.4: Captura de paquetes con *tcpdump* y volcado en fichero de salida

La forma de lanzar *AFL++* contra el binario *tcpdump* también varía de los anteriores casos, puesto que en esta prueba se busca explotar la utilidad del binario para visualizar ficheros de captura, es decir, el fichero generado por la propia aplicación pero mutado, con el objetivo de encontrar fallos. Para visualizar los ficheros se usan diferentes argumentos de *tcpdump*, pero el más importante son los caracteres “@@”, que permiten al software de *fuzzing* transformar las cadenas generadas en ficheros que se pasen como entrada al binario (ver Listing 6.5)

```
afl-fuzz -i <casos de prueba> -o <directorio de salida> tcpdump -vv -ee -nnr @@
```

Listing 6.5: Ejecución de *afl++* para el binario *tcpdump*

Una vez que se tiene el fichero de prueba que se va a mutar y tras la instrumentación del binario con *afl-clang-fast*, se procede a lanzar la herramienta de *fuzzing* con 6 procesos paralelos que, tras aproximadamente 1 hora y millones de iteraciones, encuentra una cadena que permite interrumpir la ejecución de un programa (ver Figura 6.31).

```

afl-fuzz -i testcases -o findings-ram/tcpdump -S f03 ../old-sour...

american fuzzy lop ++4.00c {f03} (...urce-code/tcpdump-4.8.1/tcpdump) [fast]
┌───────────┴───────────┐
┌ process timing ────────────┐ overall results ────────────┐
│ run time : 0 days, 0 hrs, 42 min, 33 sec │ cycles done : 0 │
│ last new find : 0 days, 0 hrs, 0 min, 7 sec │ corpus count : 6927 │
│ last saved crash : 0 days, 0 hrs, 7 min, 31 sec │ saved crashes : 1 │
│ last saved hang : none seen yet │ saved hangs : 0 │
└───────────┴───────────┘
┌───────────┴───────────┐
┌ cycle progress ────────────┐ map coverage ────────────┐
│ now processing : 2574.1 (37.2%) │ map density : 0.46% / 34.11% │
│ runs timed out : 0 (0.00%) │ count coverage : 2.20 bits/tuple │
└───────────┴───────────┘
┌───────────┴───────────┐
┌ stage progress ────────────┐ findings in depth ────────────┐
│ now trying : havoc │ favored items : 2773 (40.03%) │
│ stage execs : 576/883 (65.23%) │ new edges on : 3642 (52.58%) │
│ total execs : 12.2M │ total crashes : 1 (1 saved) │
│ exec speed : 4811/sec │ total tmouts : 0 (0 saved) │
└───────────┴───────────┘
┌───────────┴───────────┐
┌ fuzzing strategy yields ────────────┐ item geometry ────────────┐
│ bit flips : disabled (default, enable with -D) │ levels : 25 │
│ byte flips : disabled (default, enable with -D) │ pending : 3576 │
│ arithmetics : disabled (default, enable with -D) │ pend fav : 13 │
│ known ints : disabled (default, enable with -D) │ own finds : 4076 │
│ dictionary : n/a │ imported : 2850 │
│ havoc/splice : 3528/7.84M, 549/3.90M │ stability : 100.00% │
│ py/custom/rq : unused, unused, unused, unused │ │
│ trim/eff : 28.89%/359k, disabled │ │
└───────────┴───────────┘
[cpu001: 66%]

```

Figura 6.31: Ejecución de afl++ sobre el binario tcpdump

Una vez que se tiene el fichero del error se procede a minimizarlo igual que se hizo con el del binario *sudo*. En la siguiente imagen se puede ver el volcado del fichero resultante tras la operación de minimizado (ver Figura 6.32).

```

→ xxd min.testcase
00000000: d4c3 b2a1 0200 0400 3030 3030 3030 3030  ....00000000
00000010: 3030 3030 0800 0030 3030 3030 3030 3030  0000...00000000
00000020: 8600 0000 3030 3030 3030 3030 3030 3030  ....000000000000
00000030: 3030 3030 3030 3030 4d30 3030 3030 0000  00000000M00000..
00000040: 3011 3030 3030 3030 3030 3030 3030 3030  0.00000000000000
00000050: 3030 3030 3030 3030 3030 3030 3030 3030  0000000000000000
00000060: 3030 3030 3030 3030 3030 3030 01f4 3030  000000000000..00
00000070: 3030 3030 3030 3030 3030 3030 3030 3030  0000000000000000
00000080: 3030 3030 0223 3030 3030 3030 3030 3030  0000.#0000000000
00000090: 3030 0008 3030 0230 3030 2e30 0006 3030  00..00.000.0..00
000000a0: 3030 3030 000a 3030 3030 3030 3030 3030  0000..00000000

```

Figura 6.32: Resultado de la minimización de un volcado hexadecimal para el binario tcpdump

Tras analizar el binario con el depurador *gdb* se arroja un error en la línea 2502 del fichero *print-isakmp.c* (ver Figura 6.33).

```
Program received signal SIGSEGV, Segmentation fault.
ikev2_e_print (ndo=0x7fffffff910, base=0x0, tpay=<optimized out>, ext=<optimized out>, item_len=<optimized out>
, ep=<optimized out>, phase=2, doi=0, proto=<optimized out>, depth=2) at ./print-isakmp.c:2502
2502                                     base->flags & ISAKMP_FLAG_I,
```

Figura 6.33: Depuración con *gdb* del binario *tcpdump*

Por último, se compila este programa con *AddressSanitizer* y *UndefinedSanitizer* y se vuelve a ejecutar con el fichero que produce el error. Ambos sanitizers reportan fallos en la ejecución del binario, por un lado un comportamiento inesperado en el fichero *print-isakmp.c* y por otro un error de *segmentation fault* debido a una operación de lectura de memoria (ver Figura 6.34). Este problema podría incurrir en una vulnerabilidad en este software.

```
print-isakmp.c:2502:17: runtime error: member access within null pointer of type 'struct isakmp'
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior print-isakmp.c:2502:17 in
AddressSanitizer: DEADLYSIGNAL
=====
==11151==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000013 (pc 0x55c24c3c23db bp 0x7fffc70f7210 sp 0x7fffc70f
6e20 T0)
==11151==The signal is caused by a READ memory access.
==11151==Hint: address points to the zero page.
#0 0x55c24c3c23db in ikev2_e_print /home/xfear/Desktop/TFG/old-source-code/tcpdump-4.8-ASAN/./print-isakmp.c:2502:11
#1 0x55c24c3bd087 in ikev2_sub0_print /home/xfear/Desktop/TFG/old-source-code/tcpdump-4.8-ASAN/./print-isakmp.c:2739:8
#2 0x55c24c3bc56e in ikev2_sub_print /home/xfear/Desktop/TFG/old-source-code/tcpdump-4.8-ASAN/./print-isakmp.c:2785:8
#3 0x55c24c3bebbf in ikev2_p_print /home/xfear/Desktop/TFG/old-source-code/tcpdump-4.8-ASAN/./print-isakmp.c:1957:7
#4 0x55c24c3bcf9a in ikev2_sub0_print /home/xfear/Desktop/TFG/old-source-code/tcpdump-4.8-ASAN/./print-isakmp.c:2733:8
#5 0x55c24c3bc56e in ikev2_sub_print /home/xfear/Desktop/TFG/old-source-code/tcpdump-4.8-ASAN/./print-isakmp.c:2785:8
#6 0x55c24c394e7b in ikev2_print /home/xfear/Desktop/TFG/old-source-code/tcpdump-4.8-ASAN/./print-isakmp.c:2850:3
#7 0x55c24c3907f9 in isakmp_print /home/xfear/Desktop/TFG/old-source-code/tcpdump-4.8-ASAN/./print-isakmp.c:2919:3
#8 0x55c24c64a18e in udp_print /home/xfear/Desktop/TFG/old-source-code/tcpdump-4.8-ASAN/./print-udp.c:556:5
#9 0x55c24c37619f in ip_print_demux /home/xfear/Desktop/TFG/old-source-code/tcpdump-4.8-ASAN/./print-ip.c:381:3
#10 0x55c24c37db7b in ip_print /home/xfear/Desktop/TFG/old-source-code/tcpdump-4.8-ASAN/./print-ip.c:646:3
#11 0x55c24c5f926c in sl_if_print /home/xfear/Desktop/TFG/old-source-code/tcpdump-4.8-ASAN/./print-sl.c:78:10
#12 0x55c24c1e0e8c in pretty_print_packet /home/xfear/Desktop/TFG/old-source-code/tcpdump-4.8-ASAN/./print.c:339:18
#13 0x55c24c1b12c5 in print_packet /home/xfear/Desktop/TFG/old-source-code/tcpdump-4.8-ASAN/./tcpdump.c:2262:2
#14 0x7eff0ba58e03 (/usr/lib/libpcap.so.1+0x2be03)
#15 0x7eff0ba3ad67 in pcap_loop (/usr/lib/libpcap.so.1+0x2dd67)
#16 0x55c24c1acd8a in main /home/xfear/Desktop/TFG/old-source-code/tcpdump-4.8-ASAN/./tcpdump.c:1766:12
#17 0x7eff0b74328f (/usr/lib/libc.so.6+0x2928f)
#18 0x7eff0b743349 in __libc_start_main (/usr/lib/libc.so.6+0x29349)
#19 0x55c24c0c0764 in _start /build/glibc/src/glibc/csu/../sysdeps/x86_64/start.S:115

AddressSanitizer can not provide additional info.
SUMMARY: AddressSanitizer: SEGV /home/xfear/Desktop/TFG/old-source-code/tcpdump-4.8-ASAN/./print-isakmp.c:2502:11 in ikev2_
e_print
```

Figura 6.34: Reporte de los sanitizers en el binario *tcpdump*

Capítulo 7

Conclusiones

En este último capítulo del trabajo se expondrán las conclusiones a las que se ha llegado tras haber realizado todas las pruebas y haber planteado las soluciones al problema inicial. Por otro lado, también se añadirán posibles mejoras de cara a un trabajo futuro.

7.1. Trabajo realizado

Se han planteado diversas soluciones para ayudar a los desarrolladores a mejorar la seguridad de su código y prevenir fallos en el mismo. Estas soluciones o, más bien, herramientas, no deben considerarse inútiles o ineficaces, pues ya se ha demostrado que en la mayoría de ocasiones reflejan errores que un desarrollador ha podido pasar por alto durante el desarrollo de su código. Lamentablemente, estos errores podrían incurrir en graves fallos de seguridad con el peligro que puede suponer en manos de un usuario experimentado con intenciones maliciosas.

No cabe duda de que el uso de análisis estático o dinámico produce degradación en el rendimiento del software y provoca un incremento sustancial en el tiempo de compilación pero, realiza diversas comprobaciones que en proyectos grandes requerirían de un trabajo manual demasiado costoso. Principalmente hay que considerar que este análisis se realiza en fase de desarrollo del producto, y esta degradación de rendimiento no se vería reflejada en el software final que llega al usuario. Por último, es cierto que el desarrollador debe revisar los análisis generados para comprobar si son errores verídicos o, por el contrario, falsos positivos del analizador. Esto último puede requerir de un tiempo excesivo por parte del desarrollador en proyectos grandes, debido a la cantidad de casuísticas que tiene en cuenta el analizador.

Otra de las soluciones que se han planteado es el uso de técnicas de *fuzzing*, cuya ventaja frente a las anteriores es que se pueden ejecutar contra

binarios cuyo código fuente no está disponible, aunque con efectividad reducida. El problema principal que se presenta aquí es que un software de *fuzzing* puede estar generando cadenas aleatorias infinitamente, por lo que se podría ejecutar contra el binario durante horas o días y no arrojar ningún resultado válido. Aún así, es uno de los métodos más útiles y sencillos para detectar errores en software y, por ello, se debe tener en cuenta a la hora de desarrollar un programa.

Aunque se han realizado diversas pruebas en este trabajo, la mayoría de ellas han sido contra software bastante actualizado, lo que incurre en una gran cantidad de falsos positivos y una reducción de posibles errores encontrados. Esto es debido a que los desarrolladores desde hace años tienen en cuenta estas técnicas y herramientas a la hora de desarrollar su código, pero si usamos software que no esté actualizado desde antes del uso de estas técnicas (aproximadamente antes de 2015 o 2016), la cantidad de errores se verá incrementada, como se ha mostrado en el capítulo anterior. Esta reducción de fallos a lo largo de los años demuestra una vez más la efectividad de estas herramientas a la hora de encontrar errores durante el desarrollo del software.

Por último, tal y como se comentó en el capítulo 6, se reportó una vulnerabilidad en el binario *sudo* que llevaba más de 10 años oculta hasta que una persona lo reportó. Es importante resaltar que este error podría haberse encontrado aplicando *fuzzing* y desinfectantes en horas, por lo que no se debe subestimar la efectividad de estas herramientas.

7.2. Trabajo futuro

Todas las pruebas realizadas se han hecho sobre un grupo pequeño de binarios de una distribución Linux concreta. Este trabajo podría extenderse a todos los programas que residen en los repositorios de todas las distribuciones Linux para asegurar en gran medida la seguridad de los estos sistemas.

Además, en la sección de análisis estático, se han analizado solamente un análisis de cada tipo de vulnerabilidad encontrada, pero en ocasiones, se han encontrado decenas de ellos diferentes que pueden señalar vulnerabilidades que se hayan pasado por alto en este trabajo.

Por otro lado, respecto a las técnicas de *fuzzing*, se ha usado un único software, debido a la cantidad de opciones que presenta y la popularidad que tiene entre la comunidad. Sin embargo, existen decenas de programas diferentes en la red para este propósito, que podrían arrojar diferentes resultados incluso ejecutándose contra los mismos binarios de este proyecto.


```
#!/bin/bash

file=$1
while read line, do
    package = $(pacman -Qo $(which $line) | cut -d " " -f 5 | sort
               | uniq)
    asp export $package
done < $file
```

Listing 7.1: Configuración de la variable de entorno para instrumentación

Bibliografía

- [1] Buffer Overflow. <https://www.fortinet.com/resources/cyberglossary/buffer-overflow>.
- [2] Compiler sanitizers. <https://docs.conan.io/en/latest/howtos/sanitizers.html>.
- [3] Dangling Pointers in C. <https://www.javatpoint.com/dangling-pointers-in-c>.
- [4] GCC, the GNU Compiler Collection. <https://gcc.gnu.org/>.
- [5] HOWTO: Use Address Sanitizer. https://www.osc.edu/resources/getting_started/howto/howto_use_address_sanitizer.
- [6] Obtaining the Static Analyzer - CLANG. <https://clang-analyzer.llvm.org/installation#OtherPlatforms>.
- [7] The LLVM Compiler Infrastructure. <https://llvm.org/>.
- [8] Thread Sanitizer - CLANG. <https://clang.llvm.org/docs/ThreadSanitizer.html>.
- [9] CVE-2021-3156. MITRE, Enero 2021. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3156>.
- [10] A. RICHARDSON. What is static analysis? <https://www.securecodewarrior.com/blog/what-is-static-analysis>.
- [11] A. TAKANEN AND J. DEMOTT AND C. MILLER AND A. KETTUNEN. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 2018.
- [12] CARRIGAN, T. Linux permissions: SUID, SGID, and sticky bit. <https://www.redhat.com/sysadmin/suid-sgid-sticky-bit>.
- [13] E. CASEY. *Handbook of Digital Forensics and Investigation*. Elsevier Academic Press, 2010.

- [14] GARFINKEL, S. *PGP: pretty good privacy*. O' Reilly Media, Inc., 1995. https://books.google.es/books?hl=en&lr=&id=cSe_00nZqjAC&oi=fnd&pg=PR19&dq=pretty+good+privacy+pgp&ots=cYvlpXuwXI&sig=bVz3DThoIIPrVSfk0nHbgthlstY&redir_esc=y#v=onepage&q=pretty%20good%20privacy%20pgp&f=false.
- [15] GOOGLE. Address Sanitizer, Thread Sanitizer and Memory Sanitizer. <https://github.com/google/sanitizers/wiki/>.
- [16] GOOGLE. American Fuzzy Lop. <https://github.com/google/AFL>.
- [17] J. L. PARÍS. Repositorio para el Trabajo de Fin de Grado sobre análisis estático, dinámico y fuzzing. <https://github.com/XileonXL/TFG-sanitizers-and-fuzzing>.
- [18] JAIN, V. Static Single Assignment (with relevant examples). <https://www.geeksforgeeks.org/static-single-assignment-with-relevant-examples/>.
- [19] LINUX, A. ASP. <https://github.com/archlinux/asp>.
- [20] LINUX, A. Pacman, 2022. <https://wiki.archlinux.org/title/pacman>.
- [21] MALCOM, D. Add a static analysis framework to GCC. RFC, Noviembre 2019. <https://gcc.gnu.org/legacy-ml/gcc-patches/2019-11/msg01543.html>.
- [22] MALCOM, D. Static analysis in GCC 10. <https://developers.redhat.com/blog/2020/03/26/static-analysis-in-gcc-10>.
- [23] MICROSOFT. Allocating Zero Memory, 2021. <https://docs.microsoft.com/en-us/cpp/c-language/allocating-zero-memory?view=msvc-170>.
- [24] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. CVSS Severity Distribution Over Time. <https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cvss-severity-distribution-over-time#CVSSSeverityOverTime>.
- [25] OEHLERT, P. Violating Assumptions with Fuzzing. *IEEE Security & Privacy* 3, 2 (Abril 2005), 58–62. <https://ieeexplore.ieee.org/document/1423963>.
- [26] P. RON. *Expert Shell Scripting*. Springer, 2009. https://link.springer.com/chapter/10.1007/978-1-4302-1842-5_12.

- [27] POLACEK, M. GCC Undefined Behavior Sanitizer - ubsan. <https://developers.redhat.com/blog/2014/10/16/gcc-undefined-behavior-sanitizer-ubsan>.
- [28] POTAPENKO, A., BRUENING, D., VYUKOV, D., AND SEREBRYANY, K. AddressSanitizer: A Fast Address Sanity Checker. 6. <https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/43308.pdf>.
- [29] SEREBRYANY, K., AND ISKHODZHANOV, T. ThreadSanitizer - data race detection in practice. 9. <https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/35604.pdf>.
- [30] SOURCE, C. O. American Fuzzy Lop plus plus (AFL++). <https://github.com/AFLplusplus/AFLplusplus>.
- [31] STEPANOV, E., AND SEREBRYANY, K. MemorySanitizer: fast detector of uninitialized memory use in C++. 6. <https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/43308.pdf>.
- [32] W. WANG AND G. LI AND X. XIA AND Z. JIN. Detecting code clones with graph neural network and flow-augmented abstract syntax tree, Febrero 2020. <https://ieeexplore.ieee.org/abstract/document/9054857>.

Acrónimos

AFL++ American Fuzzy Lop. 3, 11, 36–39, 44, 73, 81

AST Abstract Syntax Tree. 19, 20

CVSS Common Vulnerability Scoring System. 10

GCC GNU Compiler Collection. 2, 20, 22

NIST National Institute of Standards and Technology. 10

PGP Pretty Good Privacy. 42

SSA Static Single Assignment. 21

