UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Analizador de gadgets ROP para la arquitectura RISC-V

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Comes Sanchis, Josep

Tutor/a: Ripoll Ripoll, José Ismael

CURSO ACADÉMICO: 2021/2022

A mi familia, para que pueda sentirse orgullosa

Agradecimientos

Agradecer a Ismael Ripoll, tutor del trabajo por transmitirme su conocimiento sobre ciberseguridad. Gracias por siempre responder a mis dudas de una manera muy clara y apasionada, motivándome a profundizar más en el tema.

Resum

Arran del desenvolupament de la tècnica de protecció NX (Non eXecutable), l'ús de shellcodes per a l'execució de codi arbitrari es va veure seriosament afectat. Fruit d'aquesta limitació, els atacants van desenvolupar noves tècniques per poder explotar les vulnerabilitats presents en els sistemes informàtics. La tècnica més important és ROP (Return Oriented Programming), que permet reordenar el codi executable present en un procés per aconseguir executar el que un atacant desitja.

El present treball aborda la implementació d' una eina en el llenguatge C, que a partir d' un fitxer executable ELF de l' arquitectura de computadors RISC-V, trobi tots els gadgets que podrien ser utilitzats per a la construcció de payloads.

Paraules clau: RISC-V, ROP, Return Oriented Programming, ELF, Buffer overflow, Exploit

Resumen

A raíz del desarrollo de la técnica de protección NX (Non eXecutable), el uso de shellcodes para la ejecución de código arbitrario se vio seriamente afectado. Fruto de esta limitación, los atacantes desarrollaron nuevas técnicas para poder explotar las vulnerabilidades presentes en los sistemas informáticos. La técnica más importante es ROP (Return Oriented Programming), que permite reordenar el código ejecutable presente en un proceso para conseguir ejecutar lo que un atacante desee.

El presente trabajo aborda la implementación de una herramienta en el lenguaje C, que a partir de un fichero ejecutable ELF de la arquitectura de computadores RISC-V, halle todos los gadgets que podrían ser utilizados para la construcción de payloads.

Palabras clave: RISC-V, ROP, Return Oriented Programming, ELF, Buffer overflow, Exploit

Abstract

Following the development of the NX (Non eXecutable) protection technique, the use of shellcodes for arbitrary code execution was seen seriously affected. As a result of this limitation, the attackers developed new techniques to exploit the vulnerabilities present. in computer systems. The most important technique is ROP (Return Oriented Programming), which allows reordering the present executable code in a process to get whatever an attacker wants to execute.

This paper deals with the implementation of a tool in the C language, which from an ELF executable file of the RISC-V computers, find all the gadgets that could be used for the construction of payloads.

Key words: RISC-V, ROP, Return Oriented Programming, ELF, Buffer overflow, Exploit

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

En el presente trabajo se estudia el funcionamiento de una de las técnicas que se está empleando a fecha de la escritura del documento por los atacantes. La intención latente no es alentar al lector en el uso de la técnica para fines malignos, más bien conocer sus entresijos para ser conscientes de los peligros que existen y el alcance de estos, además de poder razonar o crear mecanismos que impidan a los atacantes emplear esta técnica para lograr su cometido.

1.1 Motivación

Desde que tengo uso de razón, la informática ha estado presente en mi vida. Gracias a aquel portátil repleto de juegos supe que me apasionaba. Poco a poco me fui formando por mi cuenta, llegué en una edad bastante temprana a saber como instalar dos sistemas operativos al mismo tiempo, qué componentes formaban un ordenador así como los distintos tipos de virus informáticos. Más adelante, alrededor de los once años, fue la primera vez que oí hablar de términos como *hacker* o *DDoS* a causa de la caída del sitio web MegaUpload [1] y empecé a interesarme más por el tema. Durante mis años de adolescencia indagué en el tema de videoconsolas, llegando a modificar mi vieja PlayStation 3 para poder permitir la carga de *homebrew*. Unos años más tarde, entré en la universidad en la carrera que siempre había querido. En ella aprendí cosas como la programación, qué es un sistema operativo o cómo funcionan las redes informáticas, que fueron perfilando mis gustos dentro de la informática. Finalmente, en un intento por explorar algo distinto, me decidí por estudiar explotación binaria donde descubrí un mundo desconocido y apasionante. Durante este estudio, aprendí qué era un *buffer overflow* y como se aprovechaba, qué es un *shellcode* y que mecanismo existe para poder neutralizar el uso de este código en los ataques, qué era el *fuzzing*, etc., hasta que me topé con la técnica ROP (*Return Oriented Programming*). Al profundizar, jugar y entender esta técnica comprendí su funcionamiento, lo que me hizo valorar la increíble capacidad que pueden llegar a tener los hackers. Una técnica que permite conseguir unos resultados que parecen más próximos a la magia que a la informática, ¿cómo era posible que alguien pensara en encadenar trozos de funciones para llegar a ejecutar las acciones que quisiera? A partir de ese momento encontré el lugar de la informática que más me apasionaba.

En cuanto a la técnica ROP, destacar que es una técnica bastante interesante debido a su funcionamiento, y es que permite 'recompilar' el código del programa en tiempo de ejecución. La característica particular de la técnica es que, a la hora de escribir un programa ROP, el set de instrucciones es desconocido, varía en función de las instrucciones contenidas en el archivo binario. A fecha de la escritura de este trabajo se puede concluir que esta técnica es la empleada en la mayoría de *exploits* basados en la vulnerabilidad

de buffer overflow, y es que tras la aparición de la medida de protección DEP (*Data Execution Prevention*), se hizo imposible realizar ataques de inyección de código. Todo esto y más se detalla en el capítulo 2.3. Un ejemplo de uso de la técnica ROP, sería el exploit construido para aprovechar la vulnerabilidad CVE-2021-30889 que afecta a la versión 4.03 del firmware de la PlayStation 5. [2].

Por último, pero no menos importante, la arquitectura de computadores RISC-V pretende desempeñar un papel relevante en los próximos años. Sin ir más lejos, en el momento de la escritura de este trabajo, la EPI (*European Processor Initiative*), ya ha conseguido crear y probar satisfactoriamente las primeras muestras de ingeniería de sus procesadores EPAC 1.0 RISC-V [3]. Este proyecto está financiado por el programa **Horizon 2020** de la UE, y cuenta con la participación de 27 socios de 10 países miembros de la Unión Europea, con el objetivo de lograr la independencia europea en materia de informática de alto rendimiento. El motivo principal por el que organismos como la UE apuesten por dicha arquitectura, se debe a que desde su creación se decidió publicar bajo licencia BSD, por lo que cualquiera es libre de modificar la arquitectura base; otra cosa que permite dicha licencia, es modificar la arquitectura y cerrarla, de manera que pueden existir empresas que ofrezcan productos basados en una arquitectura propia derivada de RISC-V.

Actualmente, todavía no pueden encontrarse procesadores RISC-V para el mercado general. Si existen placas similares a la *Raspberry Pi* [4], o dispositivos pensados para la computación de sistemas embebidos [5]. Otra posible aplicación que se está llevando a cabo, es el uso de coprocesadores RISC-V junto con procesadores ARM, como el empleado en el procesador ruso *Baikal-S* [6].

1.2 Objetivos

El objetivo del trabajo es desarrollar una herramienta que se encargue de proporcionar *gadgets* ROP para archivos ejecutables de la arquitectura de computadores RISC-V. Por ende, los objetivos son los siguientes:

- Estudiar la arquitectura RISC-V, el juego de instrucciones, el ABI de llamadas a funciones y el mecanismo de llamadas al sistema.
- Analizar el formato de ficheros ejecutables ELF.
- Estudiar la técnica ROP.
- Construir una herramienta para buscar gadgets ROP en un ejecutable.

1.3 Estado del arte

En el momento de la escritura de este documento, existen herramientas que ofrecen distintos grados de funcionalidad en el ámbito ROP (búsqueda de gadgets, generación de *payloads*, etc) no obstante, ninguna trabaja sobre la arquitectura RISC-V.

Algunas de las herramientas disponibles son las siguientes:

- **Ropshell**: Herramienta en la cual se basa este trabajo. Programa accesible a través de su web. Permite subir archivos ejecutables de las arquitecturas x86, x86_64, y ARM, para posteriormente buscar o generar gadgets, así como obtener gadgets de archivos que previamente se hayan subido.
  URL: *Ropshell*

- **Pwntools**: Se trata de un framework para la resolución de problemas propuestos en CTFs y una librería para la escritura de exploits. Escrito en Python, cuya intención es facilitar al máximo la escritura de exploits. También integra un módulo para la programación ROP.
  URL: *Pwntools*

- **ROPGadget**: Inspecciona binarios de diferentes arquitecturas (x86, x86_64, ARM, ARM64, PowerPC, SPARC y MIPS) para después generar un payload.
  URL: *ROPgadget*

- **Ropper**: Herramienta cuya principal característica es que puede producir un payload automatizado para las arquitecturas ARM, ARM64, x86, x86_64, MIPS, MIPS64, PowerPC y PowerPC 64. Además ofrece payloads para las llamadas al sistema de `execve` y `mprotect` y un API para integrar la herramienta en scripts propios. También ofrece otras utilidades centradas en los archivos binarios, como modificar sus cabeceras, mostrar información acerca de los segmentos que lo componen, etc.
  URL: *Ropper*

- **Ropc**: Esta herramienta trabaja sobre la arquitectura x86. Asegura ofrecer un lenguaje turing completo de alto nivel para implementar programas ROP. Da soporte a saltos condicionales, funciones recursivas, variables locales, punteros, etc.
  URL: *Ropc*

1.4 Estructura de la memoria

En el primer capítulo se explican todos aquellos aspectos teóricos necesarios para entender el trabajo. En la sección 2.1, se estudia la arquitectura RISC-V. A continuación, se repasa el formato de los archivos ejecutables ELF en la sección 2.2. Al final del capítulo, en la sección 2.3, se repasan las técnicas de ataque más importantes utilizadas a lo largo de la historia.

En el capítulo 3 se comenta la implementación del proyecto. En las secciones 3.1 y 3.2 se ha explicado cómo se ha creado el entorno de desarrollo. En las secciones 3.3 y 3.4, se detalla el funcionamiento del programa.

En el capítulo 4, se evalúa la herramienta, realizando pruebas contra un programa vulnerable, desarrollado a modo de prueba de concepto.

Por último, en el capítulo 5 se reflexiona sobre el trabajo realizado y concluye el mismo valorando el grado de cumplimiento de los objetivos planteados. También presenta el trabajo futuro a realizar.

1.5 Convenciones

Durante el trabajo se hará uso de las siguientes convenciones tipográficas:

Itálica
Indica nuevos términos, URLs o extranjerismos. También se utilizará en la expansión de acrónimos.

`Monoespaciada`
Usada en la representación de elementos relacionados con la programación, como pueden ser números en hexadecimal, registros del procesador, instrucciones, etc.

Negrita
Empleada para destacar términos o en las listas en las que se definan términos.

MAYÚSCULA
Utilizada para representar acrónimos y nombres de las arquitecturas de computadores.

'Comillas'
Para indicar que una palabra o expresión se utiliza con un sentido especial.

CAPÍTULO 2
Marco teórico

2.1 Arquitectura RISC-V

RISC-V nace en 2010 de la mano de Yunsup Lee, Krste Asanović, David Patterson y Andrew Waterman en los laboratorios de Computación Paralela de la Universidad de Berkeley. El objetivo era desarrollar una arquitectura para docencia e investigación; de hecho, RISC-V es el quinto proyecto colaborativo en la UC Berkeley para el diseño de un procesador RISC. A raíz de su popularidad, la meta a la que aspira RISC-V es convertirse en una ISA universal a partir de los siguientes puntos [7]:

- RISC-V debe acoplarse a todo tipo de procesadores, desde microcontroladores para sistemas embebidos hasta supercomputadoras.
- RISC-V debe funcionar bien en cualquier tipo de software y con cualquier lenguaje de programación.
- RISC-V debe poderse implementar bajo cualquier tipo de tecnología, ya sean FPGAs (*Field-Programmable Gate Arrays*), ASICs (*Application-Specific Integrated Circuits*) o cualquier chip personalizado.
- RISC-V debe permitir un alto grado de especialización.
- RISC-V debe ser estable, es decir, el juego de instrucciones base no debe cambiar y no debe ser descontinuado.

Desde el 2011, fecha de la publicación de la primera especificación hasta hoy, se ha ido dando forma al juego de instrucciones; en su primera versión, únicamente se detallaba un juego de instrucciones básico. A partir de la versión 2 de la especificación, ya se empezaron a diseñar las extensiones, las cuales se detallan en la siguiente sección. En 2015, se crea la organización **RISC-V International** que posee, mantiene y publica la propiedad intelectual relacionada con RISC-V. Esta organización cuenta con una extensa lista de miembros, como por ejemplo Google, IBM o Arduino. Otro de los objetivos de esta organización es mantener la estabilidad de la arquitectura, permitiendo su evolución basándose en razones técnicas de manera lenta y segura.

2.1.1. Juego de instrucciones

El juego de instrucciones de la arquitectura RISC-V fue diseñado basándose en juegos de instrucciones anteriores como MIPS, SPARC o ARM; es más, RISC-V nace a partir de los inconvenientes existentes en adoptar un juego de instrucciones propietario para la investigación académica. El primero de ellos reside en la propiedad intelectual de dichas arquitecturas, y es que por norma general no permiten una implementación libre, porque podría llegar a mermar sus beneficios. Otro inconveniente a tener en cuenta, es que aunque si es posible emplear alguna arquitectura propietaria para la investigación, resultaría difícil poder comercializar ideas exitosas nacidas en proyectos de investigación. Por último, debido a la complejidad que presentan hoy en día dichas arquitecturas, resultaría bastante costoso implementar completamente en el hardware una arquitectura completa; y aunque si pueda ser posible crear un subconjunto de ese juego de instrucciones, sin la arquitectura completa el software que no haya sido modificado para correr en la nueva sub-arquitectura, no podrá funcionar [8]. Es por estos motivos, que desde un principio se concibió un juego de instrucciones modular, de manera que pudieran crearse procesadores especializados en un área en concreto o simplemente, prescindir de circuitería innecesaria [9]. En el momento de la escritura de este trabajo, existen extensiones que todavía están en la fase de borrador, extensiones indefinidas, así como extensiones ratificadas. La nomenclatura que siguen los conjuntos de instrucciones de referencia se expresa de la forma `RVBBX`, donde `RV` es una parte fija, `BB` hace referencia a los bits de la arquitectura (32, 64, etc.) y `X` hace referencia al identificador de la extensión. A continuación se detallan las extensiones aprobadas:

- `I`: Set de instrucciones básico que cualquier procesador debe implementar.
- `M`: Set de instrucciones para la multiplicación y división de números enteros.
- `A`: Set de instrucciones atómicas para la programación concurrente.
- `F`: Set de instrucciones para números de coma flotante de simple precisión (IEEE-754). Añade además otros 32 registros para operar con los números de coma flotante.
- `D`: Set de instrucciones para números de coma flotante de doble precisión (IEEE-754). Amplía a 64 bits el ancho de los registros de coma flotante.
- `Q`: Set de instrucciones para números de coma flotante de cuádruple precisión (IEEE-754). Amplía a 128 bits el ancho de los registros de coma flotante.
- `C`: Set de instrucciones comprimido. Define en un formato de 16 bits para las instrucciones del conjunto `RV32I`.
- `Zicsr`: Set de instrucciones para los registros de control y estado. A los 32 registros base, es posible añadir registros de control y estado, para por ejemplo, contadores.
- `Zifencei`: Set de instrucciones para la sincronización entre escrituras en la memoria de instrucciones y búsquedas de instrucciones en la misma.

También existe la extensión `G`, pensado como ISA de propósito general. Esta extensión no es más que el conjunto de las extensiones `IMAFDZicsr_Zifencei`.

2.1.2. Interfaz Binaria de Aplicaciones

El ABI o *Application Binary Interface* es la interfaz entre dos módulos de programa, uno de los cuales es, a menudo, una librería o sistema operativo, a nivel de lenguaje de máquina. Un ABI define cómo se accede a las estructuras de datos o las rutinas computacionales en código de máquina, que es un formato de bajo nivel que depende del hardware. Por el contrario, un API define este acceso en el código fuente que es un formato de nivel relativamente alto, independiente del hardware y, a menudo legible por humanos.

El ABI cubre aspectos como:

- El set de instrucciones del procesador, detallando la organización de los registros, tipos de acceso a memoria.
- El tamaño, alineamiento y formato de los tipos de datos básicos a los cuales el procesador tiene acceso directo.
- El formato binario de los ficheros de código objeto, librerías, etc.
- La convención de llamada a funciones, que controla como se pasan los argumentos a las funciones y como se devuelve el resultado de la ejecución de dichas funciones.
- El procedimiento que debe seguir una aplicación para realizar llamadas al sistema operativo. Si el ABI especifica llamadas al sistema directas en vez de llamar a un procedimiento para la ejecución de las llamadas al sistema, se especificará el número de llamada al sistema a emplear.

La adhesión a las ABIs (las cuales pueden o no estar oficialmente estandarizadas) es normalmente trabajo del compilador, sistema operativo o de la librería; generalmente los programadores de aplicaciones no deben tratar con las ABIs directamente, aunque en ciertos escenarios si es posible.

2.1.3. Mecanismo de llamadas al sistema

El *kernel* del sistema operativo es realmente quién interactúa con el hardware instalado en un computador. El kernel provee un mecanismo de abstracción que puede ser usado por los programas de usuario para solicitar al núcleo del sistema operativo que realice ciertas operaciones como escribir en la pantalla, leer un fichero, crear un socket, etc. Este mecanismo se conoce como *system call* o *syscall* y está definido dentro del ABI.
Para que un programa pueda emplear este mecanismo, necesita elegir el número de syscall que va a utilizar, y proporcionar (si es requerido) los argumentos necesarios; tanto el número de syscall como los argumentos se suministran a través de los registros.

Registro	***Descripción***
a0	Primer argumento
a1	Segundo argumento
a2	Tercer argumento
a3	Cuarto argumento
a4	Quinto argumento
a5	Sexto argumento
a7	Nº de syscall

Tabla 2.1: Registros utilizados para realizar una syscall

El siguiente paso es ejecutar una instrucción (`ecall` en RISC-V, `svc 0` en ARM, `syscall` en AMD64, etc.) que transferirá el control al SO para que ejecute una rutina, que procesará la petición y devolverá el resultado al programa de usuario a través de los registros.

Por ejemplo, en la arquitectura RISC-V, para poder escribir en la pantalla el mensaje `Hola`, el contenido de los registros debe ser el siguiente:

Registro	***Valor***	***Descripción***
a0	0x1	Descriptor de fichero (STDOUT)
a1	0x0010460	Dirección de memoria del mensaje a imprimir
a2	0x4	Longitud del mensaje
a7	0x40	Syscall write

Tabla 2.2: Contenido de los registros al realizar una syscall

Cuando el kernel procese esta llamada al sistema, devolverá en el registro `a0` un código de estado que indicará si la operación se completó correctamente o hubo algún error.

2.1.4. Convención de llamada a funciones

Cuando un programa se encuentra en ejecución, es muy probable que realice llamadas a subrutinas. Para poder realizar estas llamadas, ha de existir un consenso para determinar la manera en que las subrutinas reciben parámetros de su invocante y devuelven un resultado. Entre las diferencias que existen entre las diferentes implementaciones se encuentran el lugar donde los parámetros, valores de retorno y direcciones de retorno son proporcionados, así como la distribución de procesos (a la hora de llamar a una función), entre la subrutina invocante y la invocada, y la posterior restauración del entorno tras la ejecución de la misma.

El mecanismo que subyace es el siguiente: el procesador guarda el valor del contador de programa correspondiente a la siguiente instrucción a ejecutar, de forma que cuando la función termine y ejecute la instrucción `ret`, el programa pueda seguir su flujo de ejecución por donde se había quedado anteriormente. En cuanto a los argumentos, pueden ser transferidos mediante los registros del procesador o mediante la memoria; esto depende de la convención de llamada a función.

En el caso de RISC-V se sigue una convención similar a la empleada en la arquitectura x86_64; una función (invocante) se encarga de proporcionar los argumentos a otra función (invocado), por medio de los registros `a0 - a7`. Si se necesitaran más argumentos se pueden pasar a través de la pila. Que papel desempeña cada registro se puede ver en la tabla 2.3. Finalmente, el invocado se encarga de devolver un código de finalización mediante los registros `a0 - a1` (según necesite más o menos bytes), así como también devolver el flujo de ejecución a la función invocadora. Este último paso se puede realizar de las siguientes formas:

- Si la dirección de retorno se encuentra en el registro `ra` (*return address)*:
  - `ret`
  - `jalr zero, 0(ra)` (Ambas son equivalentes)
- Si la dirección de retorno se encuentra en la pila:
  - Se restaura la dirección de retorno y luego se salta a ella:
    `lw ra, 4(sp)`; `ret`

Nombre	*ABI mnemónico*	*Descripción*
x0	zero	Cableado a cero
x1	ra	Dirección de retorno
x2	sp	Puntero de pila
x3	gp	Puntero global
x4	tp	Puntero a almacenamiento local del hilo
x5	t0	Temporal/registro de enlace alternativo
x6-7	t1-2	Temporales
x8	s0/fp	Registro guardado/puntero de marco de pila
x9	s1	Registro guardado
x10-11	a0-1	Argumentos de función/valores de retorno
x12-17	a2-7	Argumentos de función
x18-27	s2-11	Registros guardados
x28-31	t3-6	Temporales

Tabla 2.3: Convención de nombres de registros de propósito general

Siguiendo los estándares de esta arquitectura, el registro `ra` sirve para almacenar la dirección de retorno justo antes de invocar a una subrutina. En este instante se podría preguntar, ¿Qué ocurre si dentro de la subrutina se llama a otra subrutina? En este caso la primera debería guardar el valor del registro `ra` en la pila antes de invocar a la segunda.

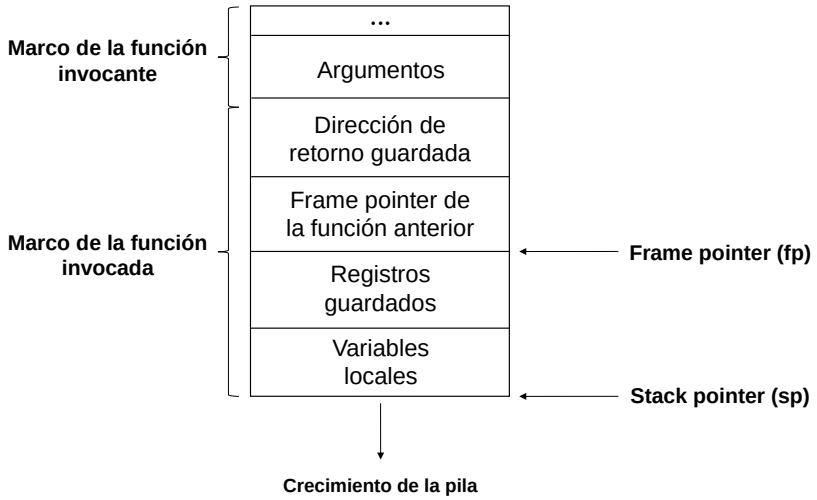

Figura 2.1: Estado de la pila tras invocar a una función

En la figura 2.1 se ilustra un ejemplo de este tipo de casos. En ella se representa el estado de la pila tras invocar una subrutina, quedando la dirección de retorno almacenada en la primera posición del marco de pila de la subrutina invocadora.

2.2 Ficheros ejecutables ELF

Los ficheros ELF (*Executable and Linkable Format*) son los empleados para archivos ejecutables, archivos de código objeto, bibliotecas compartidas, etc. Resulta importante analizar el formato que presentan dichos ficheros debido a que de estos ficheros será de dónde se leerán los *opcodes* para posteriormente descodificar las instrucciones. Para el estudio, se utiliza la especificación 1.2 [10]. En las siguientes subsecciones se explican las distintas partes que componen un archivo ELF.

2.2.1. Formato

El formato de este archivo puede verse desde dos puntos de vista diferentes: la vista de enlace (*Linking View*) y la vista de ejecución (*Execution View*).

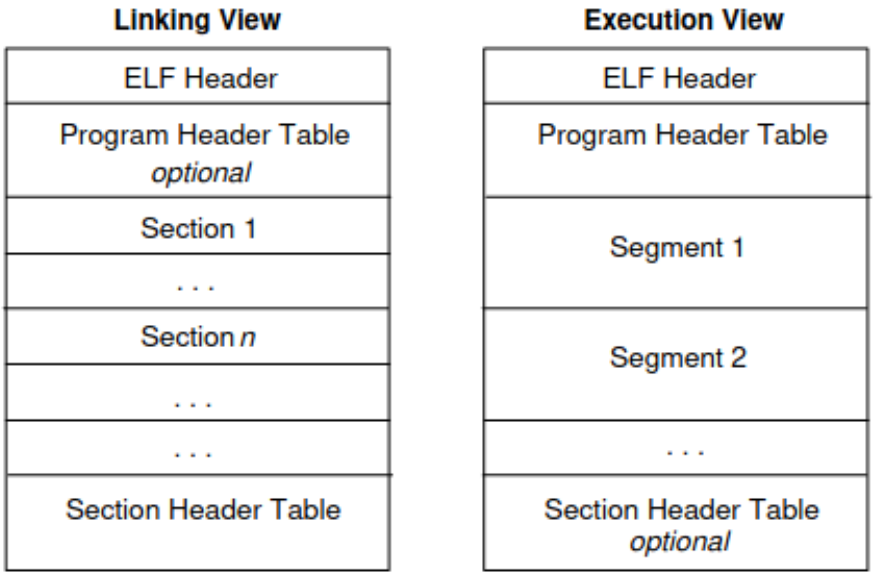

Figura 2.2: Formato de un archivo ejecutable ELF.
Extraído de: https://marcoramilli.com/2010/12/09/executable-and-linking-format/

Desde la perspectiva de enlace, el archivo se estructura en secciones (*Section Header*), indexadas desde la tabla *Section Header Table*. Las secciones contienen información acerca del enlazado y la relocalización que será empleada por el cargador del programa para cargar un ejecutable. Algunos ejemplos de secciones son los siguientes:

- **.text:** Instrucciones del programa.
- **.data:** Variables inicializadas del programa.
- **.dynamic:** Información de enlace dinámico.
- **.strtab:** Cadenas de caracteres utilizadas por los símbolos del programa.

Desde la vista de ejecución, el archivo se estructura en segmentos (*Program Header*), indexados desde la tabla *Program Header Table*. Estos segmentos se cargan en la memoria del proceso en tiempo de ejecución. Un segmento puede contener una o más secciones.

Como se ha comentado anteriormente, estos archivos se usan para distintos fines, por lo que es posible que un fichero ELF contenga Section Header Table sin Program Header

Table, o viceversa; la única parte que siempre debe estar presente es la cabecera ELF (*ELF Header*), la cual se explica en la siguiente sección. Para nuestro cometido, obviaremos la vista de enlace por un motivo: queremos extraer instrucciones de ficheros binarios que contengan algún segmento ejecutable, por lo que si se proporciona un fichero que no tenga Program Header Table no nos interesa.

2.2.2. Cabecera

La cabecera ELF reside al principio del archivo y describe una especie de mapa de contenido. La estructura que representa los datos de la cabecera es `Elf32_Ehdr`. En esta se almacena información acerca de la arquitectura para la cual está compilado el binario (`e_machine`), la dirección de entrada (`e_entry`), y alguna otra información más. El vector `e_ident[EI_NIDENT]` contiene información independiente de la arquitectura para decodificar e interpretar el archivo. Entre estos datos se encuentra el *magic number*, para que tipo de arquitectura fue creado (32 o 64 *bits*), si es *big-endian* o *little-endian*, entre otros.

Dado que la herramienta debe leer la Program Header Table, se tendrá que desplazar por el archivo tantos bytes como indique el campo `e_phoff`. El número de entradas y el tamaño en bytes de cada entrada en la tabla, vienen indicados por los campos `e_phnum` y `e_phentsize` respectivamente.

2.2.3. Program Header Table

La Program Header Table de un ejecutable o de un objeto compartido es un vector de estructuras `Elf32_Phdr` para ejecutables de 32 bits y `Elf64_Phdr` para ejecutables de 64 bits, cada una de las cuales describe un segmento que el cargador del programa necesitará para crear el proceso y lanzarlo a ejecución. De los campos que conforman la estructura, interesan los siguientes:

- `p_flags`: Indica los permisos del segmento.
- `p_memsz`: Indica el número de bytes que ocupa el segmento en la imagen de memoria del programa.

Nos interesan estos dos campos en particular, porque inspeccionando su valor se pueden discriminar los segmentos y guardar aquellos que resulten de interés; en concreto, se van a guardar segmentos ejecutables y segmentos que tengan permisos de lectura y escritura, de los cuales se escogerán los de mayor longitud. Esto servirá para escribir cadenas de caracteres, tales como `/bin/bash`, pudiendo leerlas después.

2.3 Evolución de las técnicas de ataque

2.3.1. Debilidades software

Existen multitud de debilidades software (inyección SQL, *Cross Site Scripting*, desbordamiento de enteros, etc), pero ninguna de ellas es una técnica de ataque, sino que las técnicas de ataque se basan en aprovechar estas flaquezas para conseguir diferentes objetivos: elevación de privilegios, robo de información, suplantación de identidad, etc. El error de programación que subyace a todas las técnicas que se van a explicar en este documento es el desbordamiento de buffer o buffer overflow [11]. Todas las debilidades antes mencionadas y más, se definen y catalogan en el sistema CWE (*Common Weakness*

Enumeration); dicho proyecto está patrocinado por la compañía Mitre y mantenido por una comunidad de corporaciones y organizaciones gubernamentales como Apple, la NSA, u Oracle entre otras. Cuando dichos fallos son aprovechados en forma de vulnerabilidad, existe otra lista encargada de recogerlas. Dicha lista se conoce como CVE (*Common Vulnerabilities and Exposures*), también gestionada por la compañía Mitre. Cada referencia tiene un número de identificación `CVE-ID`, descripción de la vulnerabilidad, que versiones del software están afectadas, posible solución al fallo (si existe) o como configurar el software para mitigar la vulnerabilidad y referencias a publicaciones o entradas de foros o blog donde se ha hecho pública la vulnerabilidad o se demuestra su explotación. El `CVE-ID` antes nombrado, sigue la siguiente estructura: `CVE-YYYY-NNNN`, dónde `YYYY` indica el año en el que es descubierta la vulnerabilidad y `NNNN` el número de la vulnerabilidad. Desde el año 1999 se han registrado más de 10000 CVE qué aprovechan el fallo de buffer overflow, siendo el último de mayor importancia el **CVE-2021-3156** [12].

Con la cantidad de fallos que surgen anualmente, ¿Cómo se puede evaluar el impacto de una vulnerabilidad? Existe un indicador llamado CVSS (*Common Vulnerability Scoring System*) que se encarga de puntuar estas vulnerabilidades, basándose en tres indicadores:

- **Base**: Representa las cualidades intrínsecas de la vulnerabilidad que son constantes en el tiempo y a través del entorno de usuario.
- **Temporal**: Representa las características de la vulnerabilidad que cambian en el tiempo.
- **Environmental**: Representa las características de la vulnerabilidad que son específicas a un entorno de usuario.

A partir de la métrica **base**, se produce una puntuación entre los valores 0 y 10, que puede después ser alterada por las métricas *temporal* y *environmental*. Este índice permite a las empresas detrás de los productos afectados, determinar que vulnerabilidades parchear primero.

2.3.2. Buffer Overflow

Como se ha puntualizado anteriormente, el desbordamiento de buffer se trata de una debilidad que un atacante puede aprovechar. Dicho error se produce cuando un programa escribe más datos en un buffer de los que este puede almacenar, o cuando un programa intenta escribir datos en un área de la memoria fuera de los límites del buffer. Si el buffer de entrada es proporcionado por el usuario, puede llevar a fatídicas consecuencias como se explica a continuación [13].

Para poder entender las consecuencias que acarrea un desbordamiento de buffer, primero hay que conocer el funcionamiento de la convención de llamada a funciones explicado en la sección 2.1.1. En los últimos párrafos de dicha sección, se explica la estructura de la pila al realizar llamadas entre funciones. Esta explicación se ha realizado suponiendo que la dirección de retorno se almacena en la pila.

Pero, ¿cómo es posible que la dirección de retorno sea sobrescrita si esta se sitúa en direcciones más altas? Pues bien, esto se debe a la naturaleza de la propia pila y el buffer. La pila se sitúa en las direcciones más altas y crece hacia las direcciones más bajas. Cuándo se inserta una palabra en la pila, el valor del registro `sp` se decrementa en cuatro unidades; por contra, cuando se extrae una palabra, el valor de `sp` aumenta en cuatro unidades. En cambio, el primer elemento del buffer es el situado en la dirección más baja de este, y cada vez que se escribe en él, se escribe desde direcciones de memoria más bajas, hacia más altas.

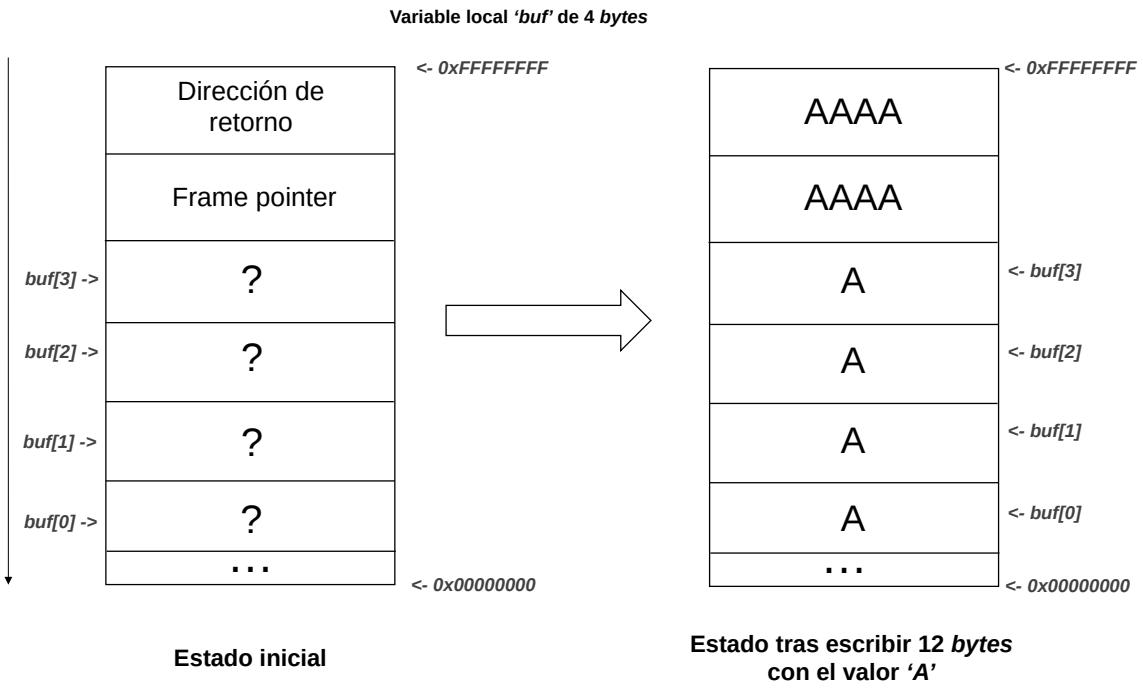

Figura 2.3: Ejemplo de buffer overflow

Como se observa en la figura 2.3, si se escriben más datos de los que el buffer puede almacenar, se sobreescriben todos los datos que estén en las direcciones más altas de la memoria del proceso. Una parte de estos datos se corresponden con la dirección de retorno, cuyo valor será cargado en el contador del programa tras ejecutar el epílogo de la función. Este detalle resulta crucial, pues si al sobreescribir dicho valor, se hace por un valor que represente una dirección válida del proceso, el flujo de ejecución continuará desde ese punto, interpretando como si fuera una instrucción los bytes situados en dicha dirección; en cambio, si esa dirección no es válida, el programa finalizará por **violación del segmento**. El peligro real de esta debilidad está en la posibilidad de escribir una dirección válida, que apunte a una instrucción real que un atacante controle.

En un buffer overflow 'clásico', se sobrescribe escrupulosamente la dirección de retorno, para que apunte a una zona de la pila (generalmente el inicio del buffer) donde reside un shellcode; dicho shellcode, se diseña previamente y se inserta en el buffer como si de datos se tratase. Para que la ejecución de este código se pueda llevar a cabo, la región de la memoria perteneciente a la pila debe contar con permiso de ejecución. Afortunadamente, los procesadores modernos cuentan con una medida de seguridad (DEP), que impide que en la memoria de un proceso existan zonas con los permisos de escritura y ejecución al mismo tiempo. Para aquellos procesadores en los que no esté disponible esta medida, existe el parche de Linux PaX que emula dicha protección.

2.3.3. Return-to-libc

En los casos en los que la medida de seguridad DEP esté presente, aunque sea posible insertar un shellcode y redirigir el flujo de ejecución hacia él, el programa finalizará impidiendo el ataque. El funcionamiento de la técnica *return-to-libc* o *ret-2-libc*, consiste en redireccionar el flujo de ejecución, de manera que se invoquen funciones de la librería del sistema como `execve` o `system` [14]. Dichas invocaciones se preparan insertando convenientemente los argumentos (según la convención de llamada), de manera que el atacante pueda ejecutar comandos arbitrarios.

Una medida que permite frustrar este tipo de ataques es la protección ASLR (*Address Space Layout Randomization*). ASLR se encarga de cargar las regiones que componen un ejecutable (pila, heap, librerías, etc.) en una posición aleatoria para cada ejecución; de este modo, cada vez que se ejecute un programa, una función determinada será situada en posiciones de memoria diferentes, dificultando la técnica de return-to-libc, ya que en principio no se conoce donde estará la función a ejecutar.

2.3.4. Return Oriented Programming

Finalmente llegamos a la técnica ROP, también conocida como *borrowed code chunks*, en la cual se basa este trabajo. Esta técnica es una generalización de ret-2-libc; en lugar de ejecutar funciones enteras se van a ejecutar trozos de estas (normalmente los epílogos). Por norma general, la arquitectura empleada para el estudio de esta técnica es x86; como se expone en la sección 1.3, todas las herramientas trabajan sobre esta arquitectura.

x86 es una arquitectura CISC, además presenta una 'geometría asimétrica': no todas las instrucciones se representan con el mismo número de bytes. Esto permite que una secuencia aleatoria de bytes pueda asociarse a una instrucción concreta del procesador. Veamos un ejemplo [15]:

Dado este programa en C:

```
1 void main() {
2     int i = 58623;
3 }
```

Al compilarlo y desensamblar la instrucción que asigna el valor a la variable se obtiene:

```
1 c7 45 fc ff e4 00 00   mov   DWORD PTR [ebp-0x4], 0xe4ff
```

Se puede notar que el opcode de la instrucción, contiene los bytes `0xff` y `0xe4`. Dichos bytes se corresponden con el opcode de otra instrucción:

```
1 $ nasmshell
2 nasm> disas
3 disas mode
4 ndisasm> FFE4
5 FFE4  jmp esp
```

Asumiremos que todas las instrucciones ocupan 4 bytes. Sin embargo, esto no supone ninguna limitación, ya que se pueden aprovechar instrucciones existentes. Además, dada la temprana edad de la arquitectura RISC-V, ya se ha demostrado que este tipo de ataques es posible [16].

La dificultad presente en esta arquitectura, es que a diferencia de x86 la instrucción `ret` no restaura la dirección de retorno de la pila al contador de programa, sino que salta a la dirección a la que apunta el registro `ra`. Esto provoca que los gadgets deban tener una instrucción que restaure dicha dirección en este registro, incrementando inevitablemente la longitud de los gadgets.

Volviendo a la técnica ROP, el objetivo de un atacante consiste en ir encadenando secuencias de instrucciones que acaben con una instrucción de retorno. Estas secuencias son los llamados gadgets. Lo interesante de esta técnica es la posibilidad de ir encadenando gadgets, pudiendo controlar la pila. Ilustremos esto con un ejemplo.

Supongamos que tenemos un programa compilado sin opciones especiales, y en él se encuentran las siguientes instrucciones:

- En la dirección `0x08050000`, `li a0, 0x0`

- En la dirección `0x08050004`, `ecall`
- En la dirección `0x08050008`, `lw a7, 4(sp)`
- En la dirección `0x0805000C`, `lw ra, 0(sp)`
- En la dirección `0x08050010`, `ret`

Para ilustrar como sería el proceso del encadenamiento de gadgets, se va a intentar escribir el valor `0x0000005D` en el registro `a7` y finalizar de manera ordenada la ejecución del programa, invocando la llamada del sistema `exit`. En la figura 2.4 se muestra el estado de la pila, justo antes de restaurar la dirección de retorno guardada por la función que invoca a la subrutina vulnerable; esta dirección será restaurada por la función vulnerable. En este caso, **`sp`** apunta al valor de la dirección de retorno, que ha sido sobrescrita, con el fin de retomar el flujo de ejecución en otro punto distinto. Con el fin de representar de una manera más clara el ejemplo, se ha decidido representar cada gadget de un color (excepto las partes de color morado, que representan relleno).

El proceso que seguirá es el siguiente:

1. Se desapilará el valor `0x8050008` al contador del programa. Tras esto el programa pasará a ejecutar la instrucción presente en dicha dirección.
2. Ahora se ejecutará el primer gadget, correspondiente con las instrucciones `lw a7, 4(sp)`; `lw ra, 0(sp)`; `ret`.
   - 2.1 `lw a7, 4(sp)`: Esta instrucción cargará el valor `0x5D` en el registro `a7`.
   - 2.2 `lw ra, 0(sp)`: Esta instrucción cargará el valor de la dirección del siguiente gadget en el registro `ra`.
   - 2.3 `ret`: Por último esta instrucción retomará el flujo de ejecución por la dirección que la instrucción anterior cargó en el registro `ra`.
3. Tras acabar la ejecución del primer gadget, el flujo de ejecución continúa por las instrucciones presentes la dirección `0x08050000`, ejecutando el siguiente gadget, cuyas instrucciones son
   `li a0, 0x0`; `ecall`.
   - 3.1 `li a0, 0x0`: Esta instrucción escribe el valor cero en el registro `a0`.
   - 3.2 `ecall`: Solicita al SO la terminación del proceso. Como previamente se ha escrito el valor `0x5D` en el registro `a7` y su argumento en el registro `a0`, se procederá con la llamada al sistema terminando ordenadamente la ejecución del programa.

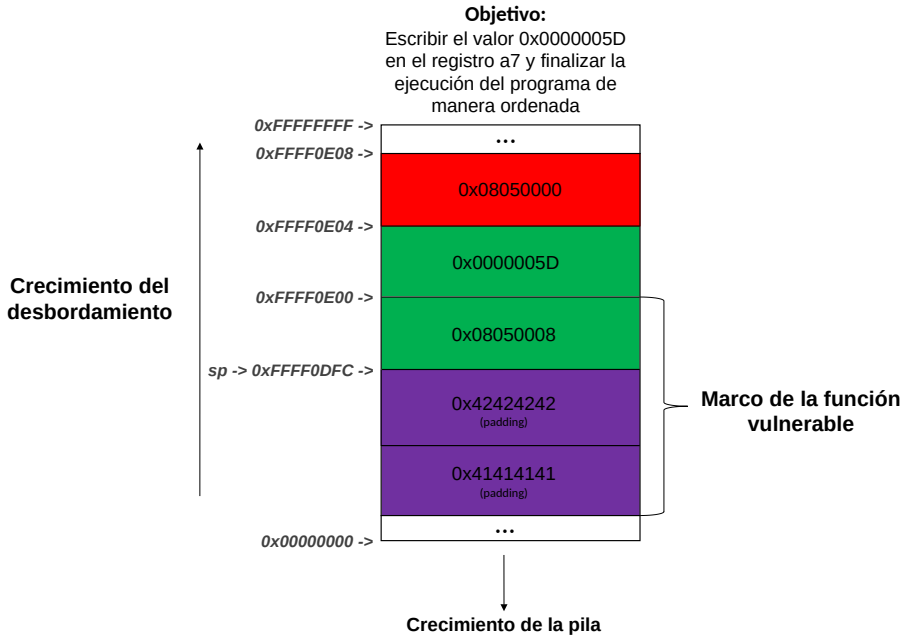

Figura 2.4: Ejemplo de encadenamiento de gadgets

2.3.5. Variaciones de ROP

El mundo de la seguridad informática podría resumirse con la expresión 'la pescadilla que se muerde la cola'; cada vez que se inventa una técnica de explotación de vulnerabilidades, aparecen técnicas de prevención que posteriormente serán sorteadas con nuevas técnicas. En esta sección, se describen algunas técnicas derivadas de ROP surgidas a causa de introducir medidas como *Control Flow Integrity*, *StackShield*, limitar el número de instrucciones return, etc.

La primera técnica a comentar, es conocida como **JOP** (*Jump Oriented Programming*) [17]. Este método surge a raíz de una de las medidas de prevención que consiste en limitar el número de instrucciones return (en caso de RISC-V `ret`). La diferencia respecto a ROP, radica en que en vez de desapilar desde la pila al contador de programa, se utilizan instrucciones de salto indirecto para controlar un registro de propósito general. Por ejemplo, si se dispusiera de la instrucción `jr t1` se podría saltar a la dirección guardada en el registro `t1`. La figura 2.5 muestra de una manera gráfica las diferencias entre la técnica ROP y JOP.

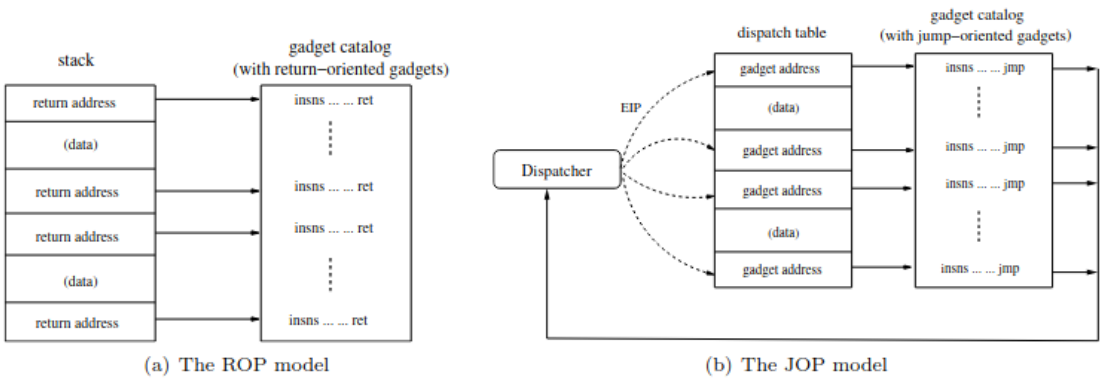

Figura 2.5: Diferencias entre las técnicas ROP y JOP.
Extraído de: *https://marcoramilli.com/2011/12/14/from-rop-to-jop/*

La siguiente técnica resulta de lo más sorprendente, ya que es efectiva contra aquellos software propietarios de los cuales no se tiene su binario o no se conoce su código fuente. Esta técnica es conocida como **BROP** (*Blind Return Oriented Programming*) [18]. Para poder llevar a cabo este ataque, es necesario que el servicio remoto sea vulnerable a un desbordamiento de la pila, que este se reinicie tras el desbordamiento (*forking server*) y que contenga un gadget para poder escribir en un descriptor de fichero. De esta manera, el atacante será capaz de obtener suficiente información a través de la conexión y proceder así con el ataque ROP.

Por último, nombrar alguna técnica más como puede ser **SROP** (*SigReturn Oriented Programming*), que aprovecha las señales por parte del sistema operativo para tener el control sobre todo el estado del procesador, en un único gadget [19].

CAPÍTULO 3
Implementación

En este capítulo se explica el proceso de diseño e implementación de la herramienta, a la que hemos nombrado **ropv**.

3.1 Creación del entorno de desarrollo

Como se menciona en la sección 2.1 todavía no existen procesadores para el mercado general, por lo que para la implementación, desarrollo y *testing* de la herramienta existen dos posibilidades:

- Adquirir alguna placa como la HiFive1 Rev B [20].
- Construir un entorno de desarrollo cruzado.

De entre las dos opciones, se va a emplear la segunda. Esta únicamente requiere en descargar e instalar el entorno de virtualización y emulación **QEMU** y configurar la *toolchain* cruzada.

3.2 Qemu

Qemu es un emulador y virtualizador genérico de código abierto. Presenta dos modos de emulación y un modo de virtualización; en este trabajo no se considerará el modo de virtualización. A continuación se especifican las características de cada modo de emulación:

- *Full-system emulation*: Emula un sistema completo, esto es, provee un modelo virtual para una máquina completa (CPU, memoria, tarjetas de red, dispositivos de almacenamiento, etc), que correrá en un sistema operativo anfitrión. En este modo, se emula el espacio de usuario, el kernel y el hardware.
- *User-mode emulation*: Permite lanzar procesos compilados para una CPU (p. ej. RISC-V) en otra CPU (p. ej. ARM), traduciendo syscalls 'al vuelo'. A diferencia del modo anterior, en este modo únicamente se emula el espacio de usuario.

Entrando más en detalle en cómo se realizan estas traducciones de llamadas al sistema, QEMU provee un AEE (*Application Execution Environment*), actuando así como si estuviera en modo supervisor. Este entorno se encarga de capturar las llamadas al sistema, traducirlas según la arquitectura de la máquina anfitrión y retornar el resultado al programa de usuario. También provee un manejador de señales, de manera que pueda redirigir

cualquier señal generada por el kernel del sistema anfitrión directamente al programa, por ejemplo, al realizar una división entre cero. Finalmente, también soporta distintos hilos de ejecución, por lo que, si un programa necesita crear un hilo, QEMU emulará la syscall `clone` y proporcionará una nueva CPU virtual para el hilo creado.

De las opciones de emulación descritas anteriormente, resulta mejor emplear la segunda. El motivo reside en que empleando la opción User-mode emulation, únicamente es necesaria la toolchain cruzada. Esta toolchain cuenta con un repositorio disponible en *https://github.com/riscv-collab/riscv-gnu-toolchain* que se puede descargar y compilar para multitud de propósitos: ya sea para generar ejecutables de 32 bits, desensamblar binarios para su posterior análisis, depurar ejecutables de 64 bits con formato de instrucciones comprimido, etc.

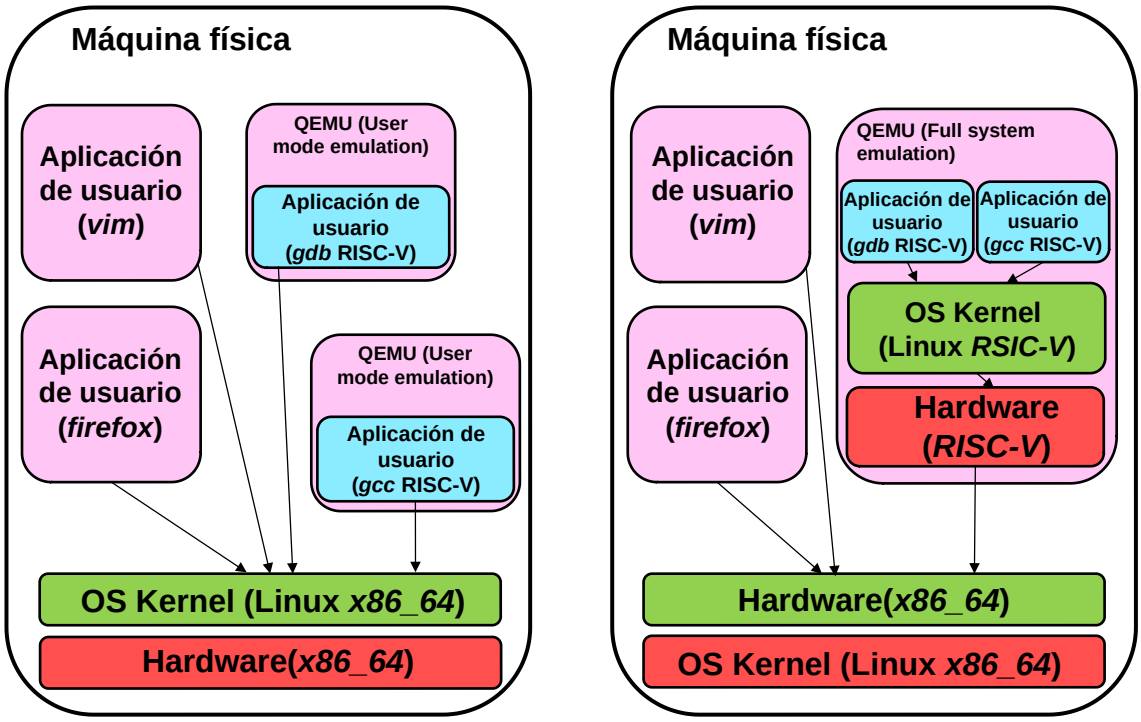

Figura 3.1: Diferencias entre los modos de emulación

3.2.1. Toolchain cruzada

Para poder llevar a cabo el desarrollo de la herramienta, se necesita la toolchain cruzada antes mencionada. Indicar que después de descargar todo el material disponible en el repositorio, es posible configurar aspectos como: el directorio donde se instalará, soporte para 32 bits, 64 bits o ambos, las extensiones de la arquitectura RISC-V, etc. En este caso se ha decidido que el lugar de instalación sería `/opt/rv32` y se ha configurado un entorno de 32 bits junto con el set de instrucciones básico y operaciones atómicas, empleando el ABI `ilp32`.

Dentro de la toolchain cruzada, se incluyen todas las herramientas necesarias para poder trabajar adecuadamente, esto es: compiladores, depurador, desensamblador, etc. De todas estas herramientas, se va a presentar una breve introducción al compilador cruzado. Para que un programa pueda ejecutarse, es necesario que previamente sea traducido a un lenguaje que un procesador pueda entender; este proceso se conoce como compilación. El proceso de compilación consta de diferentes fases, que se describen a continuación:

1. **Preprocesado**: En esta primera fase, se produce el código de entrada para el compilador; se eliminan los comentarios, se sustituyen las macros predefinidas y se incluye el código proveniente de otros archivos.

2. **Análisis**: En esta fase se analiza el código en busca de errores. Estos errores pueden ser variables no inicializadas, variables que se están usando sin haber sido declaradas previamente, la falta de cerrar algún par de llaves, etc. Finalmente, se genera el código intermedio que será utilizado por la siguiente fase.

3. **Compilación**: En esta fase se toma el código intermedio generado y se optimiza, produciendo así el código objeto. Este código es el código final que irá al ejecutable, aunque antes de generar el ejecutable ha de atravesar una fase más.

4. **Enlazado**: Finalmente, tras obtener el código objeto se enlazan todas las librerías empleadas en el programa al archivo ejecutable. De esta manera se genera el archivo ejecutable, que contendrá el código fuente ahora convertido en código objeto, y referencias a librerías que se hayan podido utilizar (como por ejemplo la librería matemática). En caso de que se decidiera compilar un binario con la opción `-static`, se añadirían al ejecutable las librerías empleadas.

En el caso del compilador cruzado, las diferencias aparecen en las dos últimas fases; en la fase de compilación se generará código objeto para una arquitectura diferente a la del ordenador en el que se ejecuta el compilador, y la fase de enlazado enlazará con las librerías de la arquitectura destino, obteniendo así un ejecutable para una arquitectura distinta.

3.3 Descodificación de instrucciones

Tal y como se estudia en la sección 2.2, para poder extraer instrucciones es necesario procesar aquellos segmentos con permisos de ejecución, a partir de estos se extraen las instrucciones del ejecutable. Estas instrucciones vendrán codificadas en palabras de 32 bits, por lo que deberán ser descodificadas para poder extraer gadgets. Este proceso se ha simplificado al máximo, pues se ha usado la herramienta *objdump* que viene incluida en la propia toolchain. Esta herramienta permite extraer multitud de información de ficheros objeto como por ejemplo: obtener la información relativa a la cabecera del fichero, obtener las section headers, mostrar su código desensamblado, etc. Gracias al uso de esta herramienta, el trabajo se ha centrado en dos objetivos: procesar el archivo desensamblado para poder descodificar las instrucciones; y crear un método de selección de gadgets.

Para lograr el primer objetivo se ha utilizado objdump para desensamblar el binario y volcar su contenido en un fichero temporal; este fichero será procesado línea por línea y una vez procesado será eliminado. De entre todas las líneas, se procesarán aquellas que pertenezcan a las instrucciones de una función, las demás serán ignoradas.

A la par que se procesan dichas líneas, se almacenará una instrucción por cada línea procesada mediante un tipo de datos creado para la ocasión; dicho tipo de datos representa a una instrucción de 32 bits.

```
typedef struct ins32_t
{
    unsigned int address; // Instruction address
    short immediate;      // Inmediate value
    bool useImmediate;    // Check if it uses an imm. value
    bool isCompressed;    // Check if the instruction is in compressed mode
    op_t operation;       // Operation with registers
    char *disassembled;   // Disassembled instruction
    char regDest[3];      // Register where the result is placed
} ins32_t;
```

Listing 3.1: Tipo de datos para representar una instrucción

Los campos más importantes que forman el tipo de datos son: por un lado `address` y `disassembled`, que serán usados para presentar las instrucciones que forman un gadget, y por otro lado el campo `operation`, que se usará principalmente para la selección de gadgets apoyándose en el resto de campos.

Cada vez que se rellenen los datos de una instrucción, se insertará en una lista de objetos `ins32_t`. El motivo es poder recorrer esta lista en orden descendente, para así obtener los gadgets.

3.4 Selección de gadgets

El criterio para la selección de gadgets viene indicado por el usuario. En el momento de ejecutar el programa se puede especificar que tipo de gadgets se desea obtener: gadgets **ROP**, gadgets **JOP** o gadgets **SYSCALL**. Es posible indicar que se desea obtener más de un tipo de gadgets al mismo tiempo. Independientemente del tipo de gadget escogido, se aplica un filtrado general. En caso de seleccionar gadgets JOP, se aplicará un segundo filtro sobre el resultado producido por el filtrado general.

El algoritmo de filtrado general se aplica una vez introducida la primera instrucción en el gadget (la última de la lista), y filtra para eliminar aquellas instrucciones que no sean útiles en el contexto de ROP [1], como por ejemplo instrucciones atómicas, operaciones de resta sobre el registro `sp`, etc. Adicionalmente, no se tendrán en cuenta como válidas las instrucciones de llamada al sistema, de salto o de retorno.

Sí existe una diferencia en el modo de componer un gadget. En el caso de gadgets RET, tras lo expuesto en la sección 2.3.4, al tener que contener alguna instrucción que restaure la dirección de retorno al registro `ra`, tras varias pruebas y análisis se ha establecido la longitud máxima de un gadget en 30 instrucciones; en los otros casos, este número se reduce hasta 6 instrucciones. Para agrupar las instrucciones según el efecto que produzcan, se ha creado el siguiente enumerable:

```
typedef enum
{
    LOAD, STORE, CMP, JMP,
    ADD, OR, AND, SHIFT,
    SUB, SET, NOP, MOV,
    CALL, BRK, NOT, NEG,
    RET, ATOMIC, IO, MUL,
    DIV
} op_t;
```

Listing 3.2: Enumerable para representar el tipo de instrucción

[1] Aunque las instrucciones *call* son útiles no serán tratadas como válidas.

En él se han detallado todos los efectos posibles que una operación puede realizar, agrupando aquellos que son similares, como por ejemplo todos los tipos de salto condicional (`CMP`).

Antes de analizar los filtros implementados, destacar que en RISC-V es posible que en un binario coexistan instrucciones codificadas en formato comprimido (16 bits) y otras en formato sin comprimir (32 bits) [21]. Ante esta posibilidad, se ha estudiado interpretar las instrucciones de una manera similar al ejemplo extraído de [15]. Finalmente, este tipo de análisis no ha conducido a resultados fructíferos, por lo que se ha decidido procesar las instrucciones leyendo los 32 bits que la forman, o si hubiera alguna instrucción en formato comprimido, leyendo 16 bits. En la estructura creada para representar a las instrucciones, el campo `isCompressed` hace esta distinción.

Para llevar a cabo el filtrado general, se sigue el siguiente algoritmo:

```
1: instrucciones_invalidas = {CMP, JMP, BRK, RET, CALL, ATOMIC, IO,
2: UNSUPORTED, SYSCALL}
3: if (instrucción_actual ∉ instrucciones_invalidas) and
4: ((instrucción_actual decrementa sp) or (instrucción_actual modifica pc)) then
5:     Descartar gadget
6: else
7:     Añadir instrucción al gadget
8:     longitud_gadget += 1 # La longitud máxima de un gadget es de 5 instrucciones
9: end if
```

Algoritmo 1: Algoritmo de filtrado general

Para obtener gadgets JOP, se ha implementado el siguiente filtro:

```
1: registro_referencia = instrucción[0].regDest
2: i ← longitud_gadget − 1
3: coincidencias ← 0
4: while i ≥ 1 do
5:     if instrucción[i].regDest = registro_referencia then
6:         coincidencias += 1
7:     end if
8:     i = i - 1
9: end while
10: if coincidencias ≥ (longitud_gadget / 2) then
11:     Descartar gadget
12: end if
```

Algoritmo 2: Algoritmo de filtrado JOP

A la par que se van procesando las instrucciones para obtener gadgets según la configuración especificada al lanzar el programa, se irán insertando los gadgets que se vayan obteniendo en un diccionario, evitando así mostrar gadgets repetidos.

CAPÍTULO 4
Evaluación

En esta sección se explica el modo de uso del programa, realizando una prueba de uso contra una aplicación desarrollada para la ocasión.

4.1 Utilización de la herramienta

A continuación se explican las opciones que proporciona el programa. La herramienta distingue entre cuatro modos de funcionamiento: un modo genérico por defecto, en el cual se procesan las instrucciones para obtener todos los gadgets útiles presentes en el programa (figura 4.2); un segundo modo que mostrará solo los gadgets ROP (figura 4.3); un tercer modo que mostrará solo los gadgets JOP (figura 4.4); y un cuarto modo que mostrará solo los gadgets SYSCALL (figura 4.5). El primer modo de estos cuatro, muestra de manera simultánea todos los tipos de gadgets, mientras que cada uno de los tres modos restantes, muestra de manera individual aquellos gadgets a los que hace referencia. Es posible indicar que se quiere mostrar más de un mismo tipo de gadget, ya sea con la primera opción o encadenando distintas opciones específicas.

```
josep@josep-lp:ropv$ ./ropv --help
Usage: ropv [OPTION...] file
Tool for ROP explotation (ELF binaries & RISC-V architecture)

  -a, --all                  Show all gadgets. Option selected by default
  -r, --ret                  Show only RET gadgets
  -j, --jop                  Show only JOP gadgets
  -s, --sys                  Show only SYSCALL gadgets
  -?, --help                 Give this help list
      --usage                Give a short usage message
  -V, --version              Print program version

Report bugs to comes.josep2@gmail.com.
josep@josep-lp:ropv$
```

Figura 4.1: Banner de ayuda del programa

```
josep@josep-lp:ropv$ ./ropv -a ./libc.so.6
0x00015c14: lw ra, 12(sp); sw s0, 0(a5); lw s0, 8(sp); sw a0, 4(a5); lw s1, 4(sp
); addi sp, sp, 16; ret;
0x00015cf8: addi a0, a0, 1060; li a1, 129; li a2, 1; li a3, 0; ecall;
0x00015fd8: li a7, 93; li a0, 0; ecall;
0x00016090: lw t1, -1000(t1); add t1, t1, tp; sw a0, 0(t1); li a0, -1; jr t0;
0x00016120: lw ra, 140(sp); lw s0, 136(sp); lw s1, 132(sp); lw s2, 128(sp); addi
 sp, sp, 144; ret;
0x00016240: lw s0, 40(sp); lw ra, 44(sp); addi sp, sp, 48; jr a6;
0x000162e0: lw ra, 28(sp); lw s0, 24(sp); lw a0, 0(sp); lw s1, 20(sp); addi sp,
sp, 32; ret;
0x000163d0: addi a4, a4, -1420; slli a0, a0, 0x2; add a0, a0, a4; lw a5, 0(a0);
add a5, a5, a4; jr a5;
0x00016540: lw ra, 12(sp); snez a0, a0; neg a0, a0; addi sp, sp, 16; ret;
0x00016628: lw ra, 92(sp); lw s0, 88(sp); lw s1, 84(sp); lw s2, 80(sp); lw s3, 7
6(sp); lw s4, 72(sp); lw s5, 68(sp); lw s6, 64(sp); lw s7, 60(sp); lw s8, 56(sp)
; lw s9, 52(sp); lw s10, 48(sp); lw s11, 44(sp); addi sp, sp, 96; ret;
0x00016b3c: lw ra, 12(sp); lw s0, 8(sp); lw s1, 4(sp); addi sp, sp, 16; ret;
0x000171fc: lw ra, 108(sp); lw s0, 104(sp); lw s1, 100(sp); lw s2, 96(sp); lw s3
, 92(sp); lw s4, 88(sp); lw s5, 84(sp); lw s6, 80(sp); lw s7, 76(sp); lw s8, 72(
sp); lw s9, 68(sp); lw s10, 64(sp); lw s11, 60(sp); mv a0, a5; addi sp, sp, 112;
 ret;
0x000174f8: lw ra, 44(sp); lw s0, 40(sp); lw a0, 4(sp); lw s1, 36(sp); lw s2, 32
(sp); lw s3, 28(sp); addi sp, sp, 48; ret;
```

Figura 4.2: Salida de la opción “mostrar todos los gadgets”

```
josep@josep-lp:ropv$ ./ropv -r ./libc.so.6
0x00015c14: lw ra, 12(sp); sw s0, 0(a5); lw s0, 8(sp); sw a0, 4(a5); lw s1, 4(sp
); addi sp, sp, 16; ret;
0x00016120: lw ra, 140(sp); lw s0, 136(sp); lw s1, 132(sp); lw s2, 128(sp); addi
 sp, sp, 144; ret;
0x000162e0: lw ra, 28(sp); lw s0, 24(sp); lw a0, 0(sp); lw s1, 20(sp); addi sp,
sp, 32; ret;
0x000163ec: lw ra, 44(sp); lw s0, 40(sp); lw s1, 36(sp); lw s2, 32(sp); lw s3, 2
8(sp); lw s4, 24(sp); lw s5, 20(sp); addi sp, sp, 48; ret;
0x00016540: lw ra, 12(sp); snez a0, a0; neg a0, a0; addi sp, sp, 16; ret;
0x00016628: lw ra, 92(sp); lw s0, 88(sp); lw s1, 84(sp); lw s2, 80(sp); lw s3, 7
6(sp); lw s4, 72(sp); lw s5, 68(sp); lw s6, 64(sp); lw s7, 60(sp); lw s8, 56(sp)
; lw s9, 52(sp); lw s10, 48(sp); lw s11, 44(sp); addi sp, sp, 96; ret;
0x0001699c: lw ra, 44(sp); lw s0, 40(sp); lw s1, 36(sp); lw s2, 32(sp); lw s3, 2
8(sp); lw s4, 24(sp); lw s5, 20(sp); lw s6, 16(sp); lw s7, 12(sp); lw s8, 8(sp);
 lw s9, 4(sp); addi sp, sp, 48; ret;
0x00016b3c: lw ra, 12(sp); lw s0, 8(sp); lw s1, 4(sp); addi sp, sp, 16; ret;
0x000171fc: lw ra, 108(sp); lw s0, 104(sp); lw s1, 100(sp); lw s2, 96(sp); lw s3
, 92(sp); lw s4, 88(sp); lw s5, 84(sp); lw s6, 80(sp); lw s7, 76(sp); lw s8, 72(
sp); lw s9, 68(sp); lw s10, 64(sp); lw s11, 60(sp); mv a0, a5; addi sp, sp, 112;
 ret;
0x000174f8: lw ra, 44(sp); lw s0, 40(sp); lw a0, 4(sp); lw s1, 36(sp); lw s2, 32
(sp); lw s3, 28(sp); addi sp, sp, 48; ret;
0x000176e0: lw ra, 60(sp); lw s0, 56(sp); lw s1, 52(sp); lw s2, 48(sp); lw s4, 4
```

Figura 4.3: Salida de la opción “mostrar gadgets ROP”

```
josep@josep-lp:ropv$ ./ropv -j ./libc.so.6
0x00016090: lw t1, -1000(t1); add t1, t1, tp; sw a0, 0(t1); li a0, -1; jr t0;
0x00016240: lw s0, 40(sp); lw ra, 44(sp); addi sp, sp, 48; jr a6;
0x00018d84: lw s9, 68(sp); li a4, 0; li a3, 0; li a2, 0; addi sp, sp, 112; jr t1
;
0x00019530: lw s7, 92(sp); li a4, 0; li a3, 0; li a2, 0; addi sp, sp, 128; jr t1
;
0x00019bf0: lw s5, 84(sp); li a4, 0; li a3, 0; li a2, 0; addi sp, sp, 112; jr t1
;
0x0001b948: lw s6, 96(sp); li a4, 0; li a3, 0; li a2, 0; addi sp, sp, 128; jr t1
;
0x0001c520: lw s4, 88(sp); li a4, 0; li a3, 0; li a2, 0; addi sp, sp, 112; jr t1
;
0x0001f1cc: addi sp, sp, 128; jr t1;
0x000242b8: addi a0, a0, -736; addi sp, sp, 48; jr a5;
0x0003d740: lw s7, 252(t0); lw s8, 256(t0); lw s9, 260(t0); lw s10, 264(t0); lw
s11, 268(t0); jr t1;
0x000554b0: li s8, -1; li a1, 90; li t2, 42; addi a6, sp, 132; addi t5, sp, 180;
 jr a4;
0x0005a5b8: li s8, -1; li a4, 90; li t1, 42; addi a1, sp, 132; addi t3, sp, 180;
 jr a5;
0x0005e1f0: lw a0, 160(a0); jr a5;
0x00060090: lw a5, 36(s0); lw s0, 40(sp); lw ra, 44(sp); addi sp, sp, 48; jr a5;
0x00060374: lw s2, 16(sp); mv a2, s1; lw s1, 20(sp); li a3, 0; addi sp, sp, 32;
```

Figura 4.4: Salida de la opción “mostrar gadgets JOP”

```
josep@josep-lp:ropv$ ./ropv -s ./libc.so.6
0x00015cf8: addi a0, a0, 1060; li a1, 129; li a2, 1; li a3, 0; ecall;
0x00015fd8: li a7, 93; li a0, 0; ecall;
0x00017774: addi a0, a0, 1772; li a1, 129; li a2, 1; li a3, 0; ecall;
0x0001793c: addi a0, a0, 1316; li a1, 129; li a2, 1; li a3, 0; ecall;
0x00024f8c: addi a0, a0, 64; li a1, 129; li a2, 1; li a3, 0; ecall;
0x000260e0: addi a0, a0, -112; li a1, 129; li a2, 1; li a3, 0; ecall;
0x000277b8: addi a0, a0, -1832; li a1, 129; li a2, 1; li a3, 0; ecall;
0x0002ae30: sw a5, 132(sp); li a7, 135; li a0, 0; mv a2, a4; li a3, 8; ecall;
0x0002af38: addi a2, sp, 136; li a7, 134; mv a0, s2; li a3, 8; ecall;
0x0002b058: li a7, 129; ecall;
0x0002b074: li a7, 136; li a1, 8; ecall;
0x0002b0bc: li a7, 133; li a1, 8; ecall;
0x0002b3fc: li a7, 132; ecall;
0x0002b85c: li a7, 421; li a3, 8; ecall;
0x0002b994: sw s2, 20(sp); li a7, 138; mv a0, s1; mv a1, s0; mv a2, sp; ecall;
0x0002cc4c: addi a0, a0, 1260; li a1, 129; li a2, 1; li a3, 0; ecall;
0x0002ceec: addi a0, a0, 588; li a1, 129; li a2, 1; li a3, 0; ecall;
0x0002cfe4: addi a0, a0, 340; li a1, 129; li a2, 1; li a3, 0; ecall;
0x0002d1e0: li a7, 422; mv a0, s6; li a1, 129; li a2, 1; li a3, 0; ecall;
0x0002d354: lw a0, -1692(a0); li a1, 129; li a2, 1; li a3, 0; ecall;
0x0002d59c: lw a0, 1820(a0); li a1, 129; li a2, 1; li a3, 0; ecall;
0x0002de10: addi a0, a0, 1904; li a1, 129; li a2, 1; li a3, 0; ecall;
0x0002def0: li a1, 129; li a2, 1; li a3, 0; ecall;
```

Figura 4.5: Salida de la opción “mostrar gadgets SYSCALL”

4.2 Prueba de funcionamiento

Tal y como se explica en la sección 3.2 todas las pruebas se han realizado sobre un procesador de la arquitectura x86_64, empleando **Qemu** para poder ejecutar código ensamblador de la arquitectura RISC-V. Esto ha permitido una gran agilidad a la hora de realizar las pruebas durante el desarrollo, porque para ejecutar binarios compilados para RISC-V se hace exactamente igual que para un binario compilado para x86_64 (ver figuras 4.6, 4.7 y 4.8)

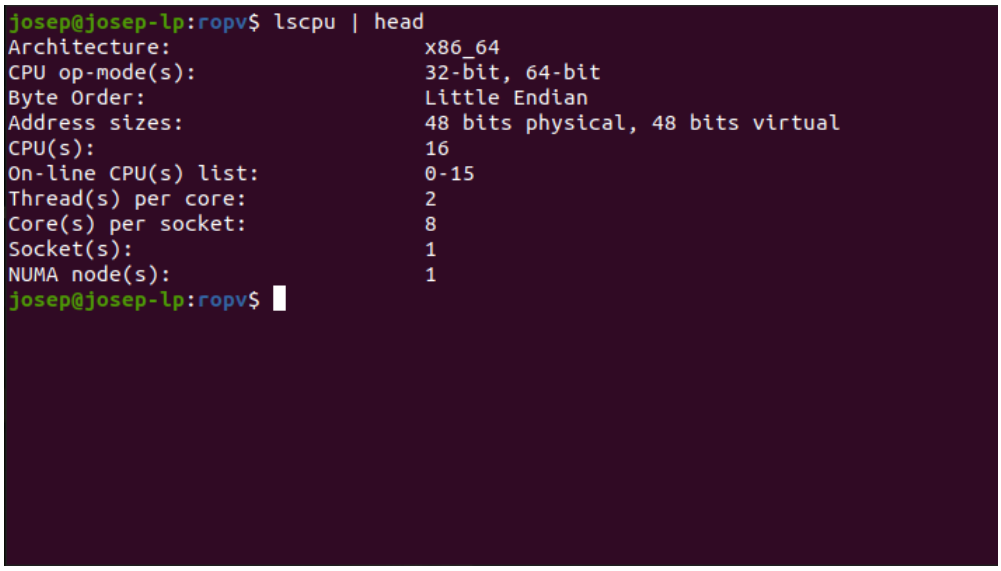

Figura 4.6: Información del equipo de pruebas

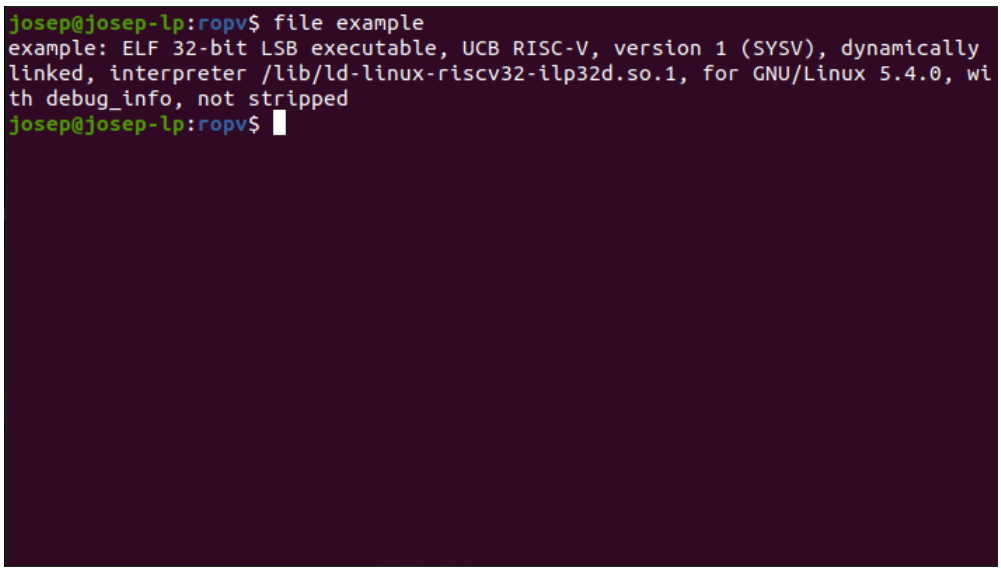

Figura 4.7: Programa compilado para RISC-V

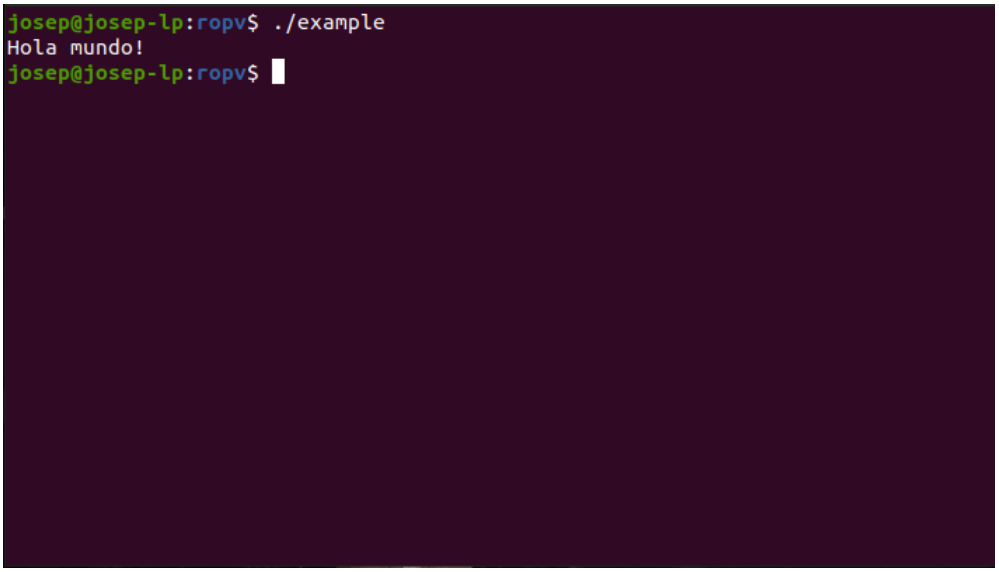

Figura 4.8: Ejecución de un programa compilado para RISC-V

La aplicación desarrollada para probar la herramienta simula la típica utilidad de cambio de contraseña. En esta utilidad se permite al usuario introducir su contraseña una primera vez, que a continuación será comparada con una segunda entrada de texto; si ambas coinciden se permite el cambio de contraseña. La función que recoge los datos introducidos por el usuario es `gets`, que no realiza ningún tipo de comprobación sobre el tamaño del texto introducido, pudiendo desencadenar en un buffer overflow.

Esta aplicación también estará con el bit `setuid` activo, ya que para realizar operaciones críticas como esta, solo el usuario *root* está autorizado.

```
1  #include <stdio.h>
2  #include <string.h>
3
4  #define MAXBUF 32
5
6  char copia[MAXBUF];
7
8  int main(int argc, char *argv[])
9  {
10   char buf[MAXBUF];
11   puts("Introduzca su nueva contraseña:");
12   gets(buf);
13   puts("Introduzca de nuevo su contraseña:");
14   gets(copia);
15   ...
16   return 0;
17 }
```

Listing 4.1: Programa de prueba vulnerable a buffer overflow

El objetivo de la prueba será abrir el editor **vi** con permisos de root. Para ello, el contenido de los registros deberá ser el siguiente:

Registro	*Valor*	*Descripción*
a0	0xffffe100	Dirección de la cadena /bin/vi
a1	0xffffe0e8	Puntero al array de los argumentos
a2	0x0	Indica que no se van a proporcionar variables de entorno
a7	0xDD	Número de syscall

Como esta prueba se realiza sobre un entorno controlado, el programa se compilará con la opción `-static` para que así todas las funciones de las bibliotecas empleadas sean añadidas al código del programa. Adicionalmente, se usará el depurador **gdb** para realizar la prueba de una manera más ágil, pudiendo desensamblar funciones en tiempo real, visualizar el estado de los registros, establecer puntos de ruptura estratégicos, etc.

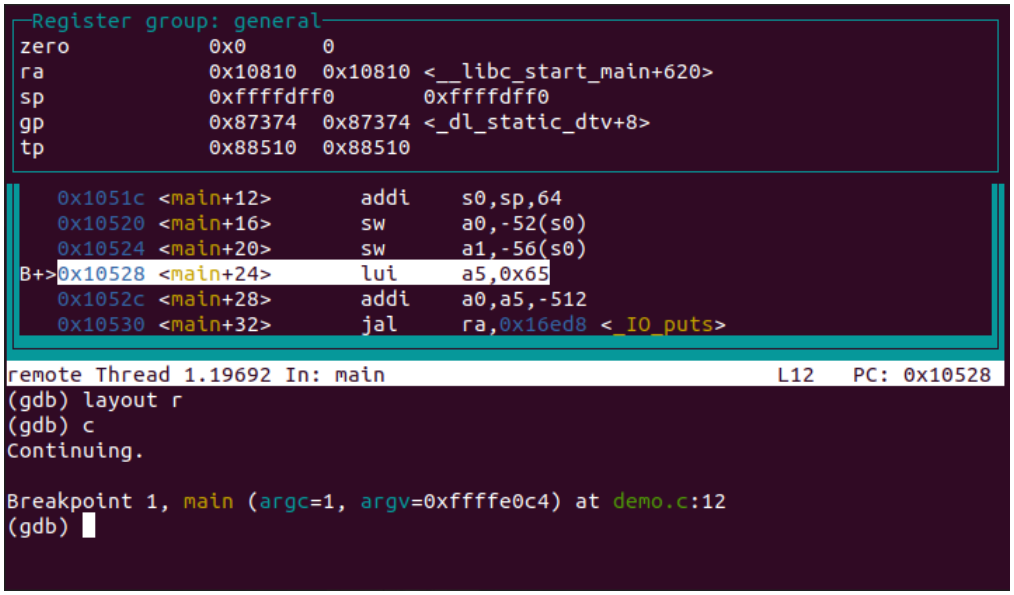

Figura 4.9: Depurador gdb

Para conseguir abrir el editor, se envía a la aplicación un payload generado a partir de los gadgets proporcionados por la herramienta, junto con un ajuste implementado manualmente para poder ir encadenando los gadgets. Esta inyección se consigue con un script escrito en Python, que se puede leer en el anexo A.

La información suministrada a la aplicación es el programa ROP. Tras desbordar el buffer se encadenan los gadgets, secuestrando el flujo de ejecución del programa para que, en lugar de continuar con su comportamiento natural, se modifique el estado de los registros hasta realizar la llamada al sistema que cambiará la imagen de memoria del proceso, por la del editor vi.

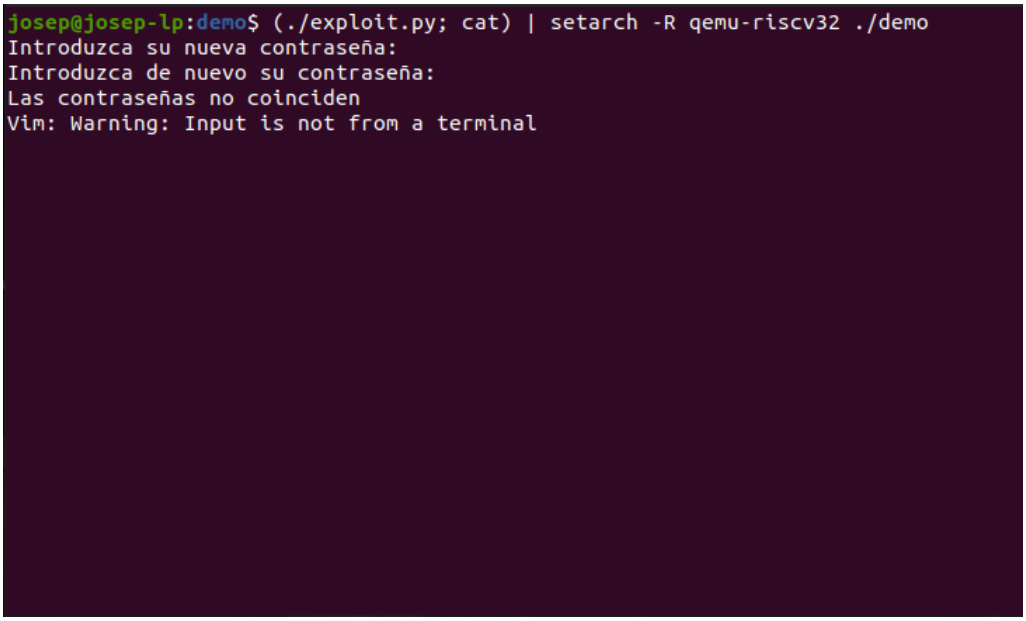

Figura 4.10: Resultado de la inyección del payload

Una vez obtenido el editor vi, se va a intentar modificar el archivo /etc/shadow, que solo el usuario root puede editarlo.

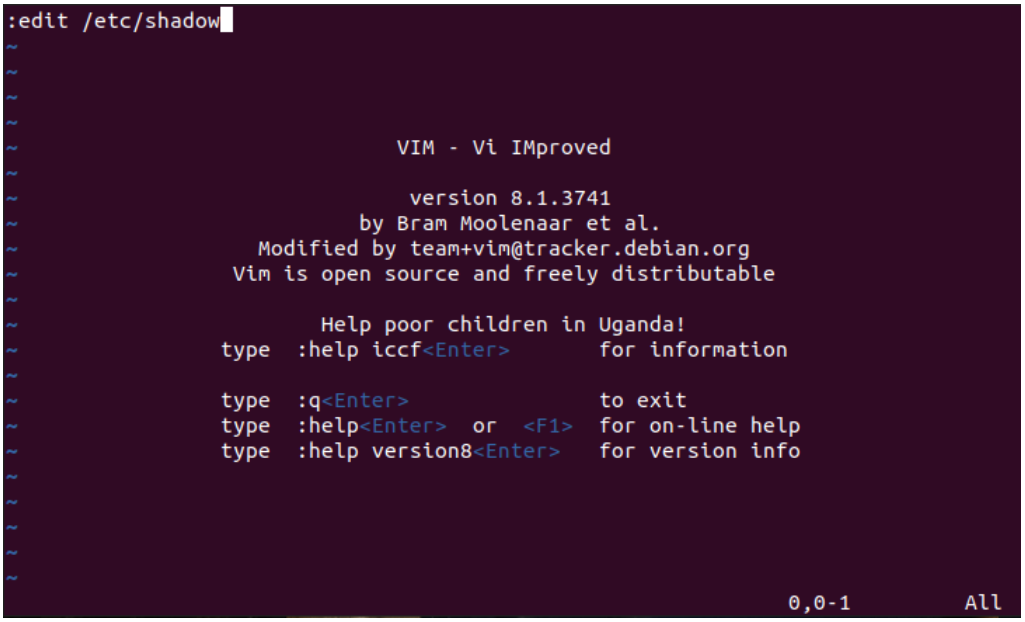

Figura 4.11: Selección del archivo /etc/shadow

Figura 4.12: Edición del archivo seleccionado

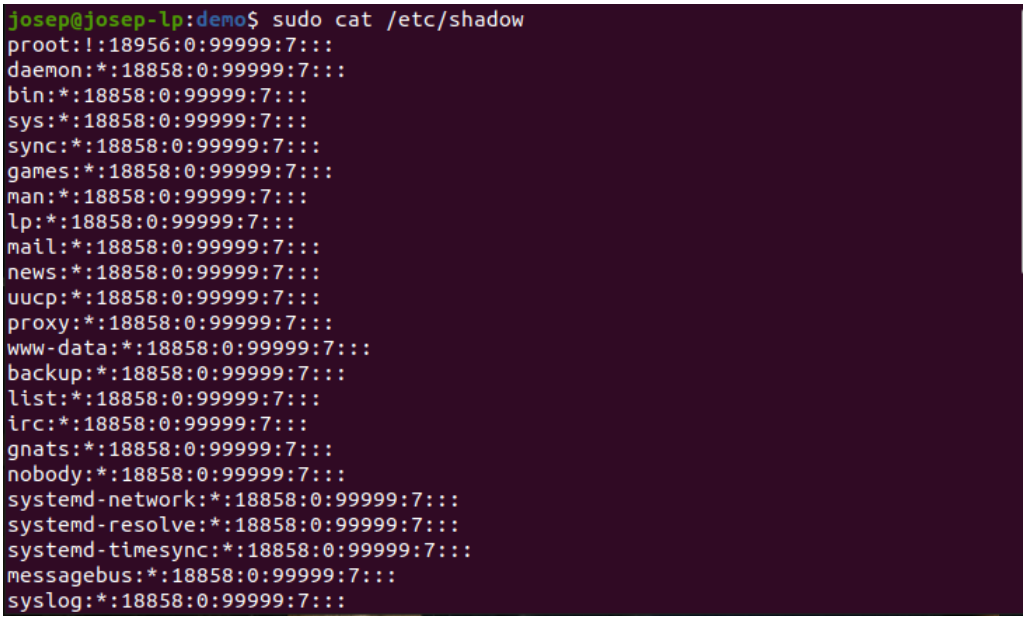

Figura 4.13: Resultado de la edición

Finalmente recalcar que esta prueba se ha realizado bajo un entorno controlado, únicamente con el fin de mostrar que a partir de los gadgets obtenidos con la herramienta ha sido posible obtener un programa arbitrario. En un entorno real, esta tarea resulta más compleja debido a que puedan existir dificultades como que el fichero se compilara sin la opción `-static`, que el ejecutable cuente con la medida de protección ASLR, también es posible que la aplicación se trate de un servicio remoto por lo que en principio no se dispondría del ejecutable para poder inspeccionar su contenido, etc.

CAPÍTULO 5
Conclusiones y trabajo futuro

El resultado de este proyecto es una herramienta totalmente funcional, diseñada con la finalidad de alcanzar los objetivos propuestos al inicio de este trabajo. A pesar de ser un campo con una complejidad técnica elevada, se ha conseguido abordar con éxito todos los objetivos planteados. Los resultados alcanzados se pueden resumir en los siguientes puntos:

- Se ha analizado la arquitectura de computadores RISC-V, entendiendo los motivos de su existencia.
- Se ha estudiado la interfaz binaria de aplicaciones, el mecanismo de llamadas al sistema y la convención de llamadas a funciones. Materia indispensable para poder utilizar la herramienta.
- Se ha examinado la estructura de los ficheros ELF para conocer de donde se pueden extraer instrucciones y datos.
- Se ha repasado la evolución de las diferentes técnicas de ataque, comprendiendo las causas por las que nace la técnica ROP.
- Se ha desarrollado una herramienta que analiza las instrucciones de un ejecutable para encontrar gadgets.
- Se ha preparado un programa vulnerable sobre el que probar la herramienta.
- Finalmente, se ha comprobado el correcto funcionamiento de la herramienta, atacando el programa vulnerable, construyendo un payload a partir de los gadgets encontrados por la herramienta.

En cuanto a líneas de trabajo futuras, la herramienta se puede modificar para trabajar sin depender de la toolchain, implementando un desensamblador de instrucciones autónomo que permita una mayor portabilidad del programa. También se puede extender la funcionalidad del programa, ofreciendo características de generación automática de payloads, algoritmos de filtrado más inteligentes, gracias al uso de la inteligencia artificial, la posibilidad de descartar instrucciones según los bytes de su opcode.

Bibliografía

[1] D. G. Aparicio, "Claves sobre el cierre de megaupload." https://www.20minutos.es/noticia/1282749/0/claves/cierre/megaupload, 20/01/2012.

[2] ChendoChap, "Exploit en el webkit de la ps5." https://github.com/ChendoChap/PS5-Webkit-Execution, 27/01/2022.

[3] AleksandarK, "La cpu europea epac 1.0 risc-v." https://www.techpowerup.com/287033/european-processor-initiative-epac-1-0-risc-v-test-chip-samples-delivered, 23/09/2021.

[4] L. Pounder, "Placa beaglev." https://www.tomshardware.com/news/beaglev-riscv-announced, 13/01/2021.

[5] A. Shilov, "Microcontrolador risc-v." https://www.tomshardware.com/news/russian-risc-v-microcontroller, 15/09/2021.

[6] I. Ros, "Procesador baikal-s." https://www.muycomputerpro.com/2021/12/20/baikal-s-un-procesador-basado-en-arm-que-integra-un-coprocesador-risc-v, 20/12/2021.

[7] A. Waterman and D. Patterson, "Why risc-v?," in *The RISC-V Reader: An Open Architecture Atlas*, ch. 1, pp. 2–13, Morgan Kaufmann, 2017.

[8] A. Waterman, *Design of the RISC-V Instruction Set Architecture*. PhD thesis, EECS Department, University of California, Berkeley, Jan 2016. pp. 3-14.

[9] A. Waterman and K. Asanović, "The risc-v instruction set manual volume i: Unprivileged isa," *Tech report*, vol. I, 2019.

[10] T. Committee, "Tool interface standard (tis) executable and linking format (elf) specification (version 1.2)," *TIS Committee*, 1995.

[11] T. M. Corporation, "Cwe version 3.1," *Cwe*, 2018.

[12] G. González, "Vulnerabilidad en el programa sudo." https://www.genbeta.com/linux/vulnerabilidad-sudo-permite-ganar-acceso-root-casi-cualquier-distro-linux, 27/01/2021.

[13] A. One, "Smashing the stack for fun and profit," *Phrack magazine*, vol. 7, 1996.

[14] C0ntex, "Bypassing non-executable-stack during exploitation using return-to-libc," *Phrack*, 2011.

[15] D. P. Castro, *Linux Exploiting*. Móstoles: 0xWORD, 2013.

[16] G. A. Jaloyan, K. Markantonakis, R. N. Akram, D. Robin, K. Mayes, and D. Naccache, "Return-oriented programming on risc-v," 2020.

[17] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," 2011.

[18] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," 2014.

[19] E. Bosman and H. Bos, "Framing signals - a return to portable shellcode," 2014.

[20] J. Horsey, "Placa hifive1 rev b." https://www.geeky-gadgets.com/hifive1-rev-b-wireless-open-source-15-03-2019/, 15/03/2019.

[21] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The risc-v compressed instruction set manual, version 1.7," pp. 2–4, 2015.

APÉNDICE A
Script empleado para la prueba

```python
#!/usr/bin/python3

from struct import pack
import sys

g1 = pack("<I", 0x00010b34)
g2 = pack("<I", 0x0002e0d8)
g3 = pack("<I", 0x00035280)
g4 = pack("<I", 0x0001cee4)
g5 = pack("<I", 0x00016ffc)
g6 = pack("<I", 0x0001e560)
g7 = pack("<I", 0x000515e8)
g8 = pack("<I", 0x0002837c)
vi_address = pack("<I", 0xffffe0c0)
vi_string = pack("<I",0x6e69622f)
vi_string += pack("<I",0x0069762f)
array_address = pack("<I", 0xffffe0a8)

def party():
    payload = b'a' * 44
    payload += g1
    payload += b'a' * 36
    payload += g3
    payload += b'a' * 36
    payload += g2
    payload += b'a' * 20
    payload += pack("<I", 0xdd) # Syscall execve
    payload += pack("<I", 0xffffe040) # Dummy value
    payload += g4
    payload += b'a' * 12
    payload += g5
    payload += b'a' * 8
    payload += array_address # Pointer to the array that holds the program name
    payload += b'a' * 16
    payload += g6
    payload += b'a' * 24
    payload += vi_address
    payload += b'a' * 8
    payload += vi_address # Address of /bin/vi
    payload += b'a' * 4
    payload += g7
    payload += vi_string
    payload += b'a' * (92 - len(vi_string))
    payload += g8
    payload += b'\x0a'
    sys.stdout.buffer.write(payload)
    payload = b'\x0a'
```

```
48      sys.stdout.buffer.write(payload)
49
50  if __name__ == "__main__":
51      party()
```

APÉNDICE B
Glosario de términos y acrónimos

ABI: Application Binary Interface.

ARM: Advanced Risc Machines.

CISC: Complex Instruction Set Computer.

Exploit: Programa o fragmento de código, diseñado para encontrar y obtener ventaja de una falla de seguridad o vulnerabilidad en una aplicación o computadora.

ELF: Executable and Linkable Format.

Gadget: Secuencia de instrucciones presentes en el programa o en las librerías compartidas que generalmente acaban con una instrucción que permiten al atacante seguir controlando el flujo de ejecución.

Hacker: Persona extremadamente curiosa con grandes habilidades en el manejo de computadoras que investiga un sistema informático para conocer las debilidades y desarrollar técnicas de mejora.

Homebrew: Aplicaciones y juegos no oficiales creados por programadores aficionados y expertos para cualquier plataforma, generalmente videoconsolas oficiales.

ISA: Instruction Set Architecture.

Opcode: Código de operación. Especifica la instrucción en código máquina que puede ejecutar el procesador.

PoC: Proof of Concept.

RISC: Reduced Instruction Set Computer.

Shellcode: Cadena de códigos de operación hexadecimales extraídos a partir de instrucciones típicas de lenguaje ensamblador, inyectado por un atacante en el espacio de direcciones de un proceso.

APÉNDICE C
Objetivos de Desarrollo Sostenible

El 25 de septiembre de 2015, fue aprobada por la Asamblea General de Naciones Unidas la Agenda 2030 de Desarrollo Sostenible. Esta agenda está compuesta por un conjunto de objetivos globales para erradicar la pobreza, proteger el planeta y asegurar la prosperidad para todos como parte de una nueva agenda de desarrollo sostenible. Cada objetivo tiene metas específicas que deben alcanzarse en los próximos 15 años.
Los Objetivos de desarrollo sostenibles (ODS), también conocidos como Objetivos Mundiales, quedarán fijados para la posteridad como el mayor esfuerzo organizado que la humanidad haya hecho para elevar el nivel y la calidad de vida de las personas en el mundo. Este esfuerzo ha proporcionado una estrategia única, una agenda colectiva que sociedades y gobiernos del planeta pueden seguir simultáneamente, sirviendo como una lista de prioridades que favorezcan el desarrollo de las naciones. Ahora, cuando ya se ha recorrido casi la mitad del trayecto propuesto para alcanzar los ODS, será fundamental el papel de la educación, de los gobiernos y de las empresas para poder lograrlos.
Los 17 ODS están integrados, ya que reconocen que las intervenciones en un área afectarán los resultados de otras y que el desarrollo debe equilibrar la sostenibilidad medioambiental, económica y social.
A continuación, se presenta una tabla con los Objetivos de Desarrollo Sostenibles, indicando el grado de relación de cada uno con el trabajo realizado.

Objetivos de Desarrollo Sostenibles	Alto	Medio	Bajo	No procede
ODS 1. **Fin de la pobreza**				**X**
ODS 2. **Hambre cero**				**X**
ODS 3. **Salud y bienestar**				**X**
ODS 4. **Educación de calidad**		**X**		
ODS 5. **Igualdad de género**				**X**
ODS 6. **Agua limpia y saneamiento**				**X**
ODS 7. **Energía asequible y no contaminante**				**X**
ODS 8. **Trabajo decente y crecimiento económico**			**X**	
ODS 9. **Industria, innovación e infraestructura**	**X**			
ODS 10. **Reducción de las desigualdades**				**X**
ODS 11. **Ciudades y comunidades sostenibles**				**X**
ODS 12. **Producción y consumos responsables**				**X**
ODS 13. **Acción por el clima**				**X**
ODS 14. **Vida submarina**				**X**
ODS 15. **Vida de ecosistemas terrestres**				**X**
ODS 16. **Paz, justicia e instituciones**				**X**
ODS 17. **Alianzas para lograr objetivos**				**X**

Tabla C.1: Grado de relación del trabajo con los ODS.

De los anteriores objetivos de desarrollo sostenibles mencionados, el presente trabajo está relacionado con:

- **Educación de calidad**, personalmente creo que la idea presentada promueve el desarrollo de contenidos por los que merece la pena pagar, esto es contenidos que tienen calidad y por los que una persona estaría dispuesta a pagar una cantidad de dinero porque le aporta valor. Además, la idea en la que se basa este trabajo ha sido ampliamente discutida por la comunidad hacker internacional; esto supone que haya multitud de documentación técnica de calidad, la cual hemos aprovechado en este trabajo.
- **Trabajo decente y crecimiento económico**, este ODS puede estar relacionado con el trabajo si lo tratamos desde el punto de vista de la ciberseguridad. Pienso que puede ayudar a estudiar las limitaciones de los nuevos procesadores RISC-V, valorando la dificultad en poder explotar los fallos presentes. Esto puede servir para poder mejorar el producto de cara a futuras revisiones.
- **Industria, innovación e infraestructura**, a su vez también creo que este ODS está relacionado. Durante el trabajo, estudiamos todos aquellos puntos que posteriormente son necesarios en la creación de nuestra herramienta, por lo que este proceso también sirve para poder crear contramedidas que afecten al propósito del programa creado.