Trabajo Final de Master

Fuzzing: Descubriendo comportamientos inesperados

Master Interuniversitario en Seguridad de las Tecnologías de la Información y de las Comunicaciones

Autor: Sergio Lázaro Magdalena

Empresa: Innotec System
Mentor: Guillermo González González
Tutor del proyecto: Carlos Hernández Gañán
Tutor: Juan José Murgui García

Diciembre de 2018

RESUMEN

A lo largo de los años, multitud de vulnerabilidades han causado problemas en los sistemas informáticos de todo el mundo. Estas vulnerabilidades han sido explotadas debido a una insuficiente fase de pruebas durante el desarrollo de un determinado producto software. El *fuzzing* surge como respuesta a la falta de automatización de las pruebas durante el desarrollo de software adoptando una posición relevante y estableciéndose como un paso inicial en la detección de vulnerabilidades. Este proceso es posible debido a la creación por parte de los investigadores de seguridad de herramientas como los *fuzzers*. Aunque el objetivo de estas herramientas se mantiene fijo, la evolución que han experimentado ha convertido a los *fuzzers* en herramientas sofisticadas responsables de la detección de un gran número de vulnerabilidades críticas en los últimos años.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

La empresa Innotec System del grupo Entelgy es una empresa especializada en ciberseguridad, inteligencia y gestión de riesgos apostando por el I+D en todas sus áreas. Por ello, el departamento de hacking tiene interés en enfocar un proyecto en las técnicas de fuzzing empleadas para el descubrimiento de vulnerabilidades.

El Fuzzing consisten en proporcionar datos inválidos, inesperados o aleatorios a programas de ordenador de forma automática o semiautomática con el fin de detectar posibles anomalías. Una vez realizadas las pruebas, el *fuzzer* debe interpretar la salida generada por la anomalía provocada, conocido como *crash* o fallo. Una vez detectado el fallo se puede reproducir con la información obtenida del *fuzzer* y si es posible realizar una filtración de información de la memoria, provocar una denegación de servicio o inyectar código, entre otras.

Para llevar a cabo estas pruebas se emplean *fuzzers*, herramientas automáticas encargadas de realizar las pruebas y reportar los fallos encontrados. Están compuestas de tres componentes principales: el generador de entradas inválidas, el mecanismo de envío de las entradas generadas y un sistema de monitorización encargado de evaluar la salida del programa. Existen multitud de *fuzzers* aunque los más conocidos son Libfuzzer, Honggfuzz y American Fuzzy Lop (AFL).

1.1. Motivación

Con la importancia de las Tecnologías de la Información y de la Comunicación (TIC) en la sociedad actual cada vez más se valora la disponibilidad, integridad y confidencialidad de la información que albergan estos sistemas. Garantizar los servicios ofrecidos es una obligación y por ello cada vez es más frecuente realizar auditorías en busca de vulnerabilidades para poder solucionarlas antes de que un atacante pueda aprovecharlas.

Al realizar una auditoría o un test de intrusión es importante localizar posibles fallos en los desarrollos propios o en software de terceros para poder explotarlos y tratar de comprometer lo máximo posible el sistema examinado. En caso de que una vulnerabilidad disponga de un parche para solucionar una vulnerabilidad de un producto de software, se deberá actualizar para evitar que se vuelva a producir ese fallo. En algunos casos,

un atacante podría emplear una vulnerabilidad no conocida y, por consiguiente, sin un parche que lo solucione. Una vulnerabilidad de este tipo se conoce como *0-day* y afectaría a cualquier sistema con la vulnerabilidad.

El descubrimiento de vulnerabilidades suele iniciarse con el empleo de técnicas de *fuzzing* para localizar posibles fallos de forma automática y, más tarde, tratar de replicar el fallo y comprobar si es posible llegar a afectar a la disponibilidad, integridad o confidencialidad de la información que alberga un determinado software. Para ello es importante realizar una buena fase de reconocimiento y descubrimiento de posibles fallos por lo que conviene conocer, entender y emplear herramientas como los *fuzzers*.

1.2. Objetivo

El proyecto tiene como objetivo dar una visión global de las técnicas más modernas de *fuzzing* estudiando sus componentes, sus ventajas y desventajas y los tipos de pruebas de *fuzzing* que se pueden realizar sobre un determinado producto de software. Para realizar un estudio completo, en primer lugar es necesario conocer los conceptos más básicos, analizar el estado del arte en las técnicas de *fuzzing* y, por último, aplicar los conceptos teóricos en el descubrimiento de fallos en productos de software de prueba y reales.

Para llevar a la práctica los conceptos teóricos del proyecto se estudiarán y emplearán tres *fuzzers*: American Fuzzy Lop, Libfuzzer y HonggFuzz. Estas herramientas se emplearán para detectar fallos en programas obtenidos de páginas webs como Rode0day con el fin de demostrar la utilidad de estas herramientas a la hora de detectar vulnerabilidades. Este tipo de webs ofrecen una serie de programas de software para que la comunidad realice pruebas de *fuzzing* sobre ellos y envíe la entrada que ha provocado el fallo. De esta forma será posible poner en práctica los conceptos estudiados en el proyecto.

1.3. Organización

La documentación del proyecto sigue una línea progresiva de menos técnico a más técnico con la intención de que se pueda realizar una lectura completa sin necesidad de buscar información adicional para su completa comprensión.

El capítulo 2 establece los conocimientos básicos a los que se va a hacer referencia a lo largo de todo el proyecto. Está compuesto por una explicación de porqué aplicar técnicas de *fuzzing* para evaluar software es útil y de que se componen las herramientas que se emplean para ello, los *fuzzers*. Después se introducen los algoritmos que emplean para realizar las pruebas con el mejor resultado posible y las técnicas que existen para encontrar fallos en software.

En el capítulo 3 se comentan las herramientas de *fuzzing* de propósito general más empleadas para realizar pruebas a productos software y en el capítulo 4 se van a realizar pruebas con los tres *fuzzers* mencionados en el capítulo anterior con el objetivo de encontrar fallos en el mismo software. De esta forma será posible comparar los resultados y analizarlos.

Finalmente, en el capítulo 5 se comentaran los resultados obtenidos durante el proyecto y que permitirán establecer unas conclusiones junto a las líneas de trabajo futuro en el capítulo 6.

El documento incluye un diccionario de términos en el anexo A y un desglose del tiempo dedicado a cada uno de los *fuzzers* empleados en el anexo B. Además, se incluyen detalles adicionales y código creado durante las fases del proyecto.

Capítulo 2

Contexto

En esta sección se van a comentar los motivos por los que emplear herramientas de *fuzzing* es útil y práctico. También se hace mención de la estructura de los *fuzzers* y los tipos de algoritmos que implementan para optimizar la detección de fallos. Por último, se incluyen las técnicas de *fuzzing* según el nivel de información que se tiene de un determinado programa software.

2.1. ¿Por qué fuzzear?

El *fuzzing* [DoDAGRMG] es solo una de muchas técnicas para encontrar vulnerabilidad en software. Existen métodos alternativos como revisiones de código, análisis estático o test unitarios pero el *fuzzing* ha tenido una gran acogida debido a los buenos resultados que ha demostrado ser capaz de obtener.

Los motivos principales para emplear el *fuzzing* en la detección de vulnerabilidades de productos software radica principalmente en su simplicidad y en su efectividad, siendo detectadas múltiples vulnerabilidades conocidas.

Además, para hacer *fuzzing* en herramientas de software no necesariamente se requiere del código del producto a analizar y sería posible realizar pruebas sobre productos comerciales que no permiten acceso a su código. No tener acceso al código puede suponer un esfuerzo adicional al realizar las pruebas de *fuzzing* pero es perfectamente viable analizar y encontrar fallos en estos productos.

Por otro lado, otra ventaja que ofrecen los *fuzzers* es que se realizan muchas pruebas en muy poco tiempo y algunos de ellos tratan de abarcar la mayor parte posible del código de un programa en pruebas. Al tratarse de pruebas automáticas tienen ventaja sobre los análisis manuales realizadas por personas que son más caras tanto en tiempo como en esfuerzo ya que, para realizarlas correctamente, se van a necesitar distintos equipos capaces de desarrollar pruebas para hacer fallar un programa.

Hay que tener en cuenta también la posibilidad de emplear la misma herramienta para analizar software de comportamientos totalmente diferentes. Un ejemplo podría ser el empleo del mismo *fuzzer* para realizar pruebas sobre distintos navegadores.

A pesar de que se realicen pruebas automáticas y la mayor parte del trabajo la realiza el *fuzzer*, se ha demostrado que tienen pocos falsos positivos. Es cierto que los *fuzzers* cuentan con la limitación de que no son capaces de distinguir entre aquellos fallos que puedan ser benignos y aquellos que puedan llegar a ser explotados. Por ello, los procesos de *fuzzing* suponen una fase previa al descubrimiento de vulnerabilidades.

2.2. ¿Qué es un fuzzer?

Un *fuzzer* [DoDAGRMG] es una herramienta muy simple que se puede resumir en que recibe datos de entrada, los procesa realizando alteraciones de esa entrada y genera un fichero de salida. Cada una de estas partes pueden implementarse con distinto nivel de complejidad según el propósito del *fuzzer* y pueden soportar también distinto nivel del integración. Los principales componentes componentes se introducen a continuación:

- **Generador de entradas:**, también conocido como el generador de *fuzzing*, es responsable de generar las entradas que se emplearán para analizar el sistema a probar. Estas entradas pueden generarse empleando múltiples algoritmos, cada uno con sus ventajas y sus desventajas, y que afectan directamente en la efectividad y eficiencia del *fuzzer*. Finalmente, es el componente encargado de enviar las entradas generadas al mecanismo de entrega.
- **Mecanismo de entrega:** El mecanismo de entrega recibe las variaciones de los datos del generador de entradas y se los entrega al sistema bajo pruebas. Este apartado está vinculado con el tipo de información que recibe el sistema bajo pruebas y con la forma en la que lo recibe. Los *fuzzers* de propósito general suelen centrarse en enviar la información por un fichero o por STDIN.
- **Sistema de monitorización:** Este sistema es el encargado de observar el funcionamiento del proceso que realiza las pruebas sobre el objetivo con el fin de detectar si se ha producido algún error. Es una sección crítica en el proceso de fuzzing ya que afecta directamente en si un *fuzzer* es capaz de detectar un fallo determinado.

2.3. Conceptos básicos

En esta sección se incluyen los conceptos empleados a la hora de hablar de los *fuzzers* con el fin de incluirlos en las explicaciones que se realicen durante todo el proyecto. Estos conceptos son vitales para comprender el proyecto.

2.3.1. Casos de prueba

Teniendo en cuenta los componentes que forman un *fuzzer*, comentados en la sección 2.2, y teniendo en cuenta la simplicidad en su funcionamiento, los ficheros de entrada o casos de prueba deben de ser lo más simples posibles. De esta forma, el *fuzzer* procesará esa entrada y realizará sus alteraciones tratando de alcanzar un objetivo que en algunos

casos puede especificar el usuario, como por ejemplo los caminos que más instrucciones tengan, los caminos que más comparaciones tienen o los caminos con más saltos, entre otros.

2.3.2. Cobertura de código

La cobertura de código o *code coverage* hace referencia a la cantidad de código de un programa que un *fuzzer* puede ejecutar y analizar. La mayoría de programas mantienen estructuras dentro del código empleando bloques condicionales o bucles a través de los cuales el *fuzzer* tiene que encontrar las opciones condicionales que le van a permitir avanzar a lo largo del programa.

En la búsqueda de fallos, tener una buena cobertura de código puede ser decisivo para conseguir casos de éxito y la principal dependencia son los casos de prueba que se le entregan al fuzzer. Existen herramientas capaces de monitorizar el flujo que sigue un *fuzzer* y reportar la información que se ha necesitado para acceder a una determinada función nueva para emplearla posteriormente como caso de prueba.

La generación de datos para la cobertura de código es un paso previo importante antes de iniciar un proceso de *fuzzing* para maximizar el alcance y la efectividad de un *fuzzer*. Un ejemplo en la que los casos de prueba son vitales para la cobertura de código puede ser cuando para alcanzar un determinado bloque de instrucciones se necesita realizar operaciones complejas que un *fuzzer* no va a poder alcanzar sin ayuda. Una forma de solucionarlo podría ser la modificación de dicho bloque para que sea alcanzable por el *fuzzer* o, sin necesidad de modificar el código, empleando casos de prueba óptimos obtenidos por herramientas de cobertura de código.

2.3.3. Instrumentación

Los fuzzers emplean componentes software o hardware para obtener información durante la ejecución de un determinado proceso. Se entiende como *feedback-guided fuzzing* a la posibilidad de recuperar información durante la ejecución de un determinado programa con el objetivo de optimizar y aumentar la cobertura del código. Para ello se deben añadir instrucciones o hooks para registrar el flujo de ejecución. Esta técnica se conoce como instrumentación.

Existen dos tipos de instrumentación por software: estática y dinámica. En la instrumentación estática se realiza la inclusión de código directamente sobre el binario aunque es poco común. La instrumentación dinámica se realiza en tiempo de ejecución y su principal ventaja es que no es necesario tener el código.

Cuando se posee el código del programa la mejor opción es realizar la instrumentación empleando el compilador que ofrece el *fuzzer* en caso de que lo tenga. AFL y Honggfuzz incluyen herramientas de compilación para hacer la instrumentación simple. La instrumentación con Libfuzzer se realiza indicando opciones específicas al compilador de clang siendo similar en el caso de Honggfuzz aunque con las herramientas de instrumentación que incluye este *fuzzer* el proceso se vuelve transparente.

Además de la instrumentación software sobre el código o sobre el binario compilado, es posible realizar instrumentación hardware en la que se emplea contadores de rendimiento que son registros del procesador que contabilizan eventos hardware como instrucciones ejecutadas, fallos de cache o saltos que no se han predecido.

2.3.4. Ficheros de salida

Los ficheros de salida que generan los *fuzzers* contienen principalmente la entrada que ha provocado que se produzca el fallo de un determinado programa. La estructura de los elementos que genera un *fuzzer* depende del mismo pero generalmente van a incluir un directorio con las entradas que ha provocado los fallos junto con información del fallo producido (en caso de que se hayan producido), casos de prueba generados o información del rendimiento del *fuzzer*.

Esta información se podrá emplear posteriormente para analizar el fallo y poder solucionarlo. Siempre que se produce un fallo se deriva una alteración del flujo del programa. En algunos casos, estos fallos pueden ser analizados y llegar a ser explotados. Un ejemplo podría ser la vulnerabilidad Heartbleed [MIT] con CVE asignado (CVE-2014-0160) que pudo ser detectada por un *fuzzer* y explotada para leer datos sensibles de la memoria del objetivo.

2.3.5. Modo persistente

El funcionamiento normal de un *fuzzer* consiste en crear un nuevo proceso con fork() al que se le envía un único caso de prueba. Para evitar la sobrecarga y ralentización que ocasiona la llamada a sistema execve para crear nuevos procesos y enviarles el nuevo caso de prueba, es posible emplear el modo persistente para que, desde un mismo proceso, se ejecute un bucle que realiza varias ejecuciones sin el coste temporal que supone crear nuevos procesos. De esta forma es posible optimizar la velocidad de ejecución hasta 10 veces. Esta optimización tiene un precio ya que es más fácil que el fuzzer se desvíe por fugas de memoria accidentales o condiciones de denegaciones de servicio en el código sobre el que se realizan las pruebas.

2.3.6. Harness o wrappers

Los *fuzzers* tratan de encontrar fallos en programas que reciben datos de entrada pero, en muchos casos, estos programas emplean librerías que solo pueden ser invocadas si se importan en un determinado programa y que también puede ser objeto de pruebas. Un *fuzzer* por si solo no puede realizar pruebas sobre librerías o funciones concretas por lo que se suele generar un envoltorio o *wrapper*, conocido como *harness*, que se emplea para la llamada a funciones concretas del código de un programa.

En el caso de que se quieran realizar las pruebas sobre una librería, el *harness* se emplea para importar la librería, obtener los datos de entrada que va a enviar el *fuzzer* y enviárselos a la función que se quiere analizar.

La realización de pruebas empleando *harness* no solo se limita a librerías y puede depender del tipo de función sobre la que se realice el *fuzzing*, además de los parámetros que recibe. El empleo de *harness* es necesario en muchos casos en los que la variable que va a recibir la entrada del *fuzzer* no asigna los datos de STDIN o de un fichero. Para realizar el *fuzzing* a una función que emplee este tipo de parámetros, como por ejemplo una variable de entorno, se debería generar un *harness* que recibiese la información de STDIN o de un fichero y asignase esa información a la variable de interés para después llamar al programa objeto de *fuzzing*. El uso de esta técnica supone una caída del rendimiento del *fuzzer* pero permite realizar *fuzzing* sobre variables que no se adaptan al funcionamiento de un *fuzzer* de propósito general.

2.3.7. Address Sanitizer (ASAN)

Address Sanitizer [Gooa], conocido como ASAN, es una herramienta desarrollada por Google que consiste en un módulo de instrumentación en tiempo de compilación y una librería empleada en tiempo de ejecución que reemplaza la función malloc.

ASAN funciona en arquitecturas x86, ARM, MIPS (32 y 64 bits) y PowerPC64. Los sistemas operativos soportados son Linux, Darwin (OSX e iOS), FreeBSD y Android. En el resto de arquitectura y sistemas operativos ASAN puede funcionar correctamente pero no se encuentran en soporte.

La principal ventaja que ofrece ASAN es que es capaz de detectar fallos de memoria y dar información detallada de los mismos. En procesos de *fuzzing* es importante monitorizar la memoria y tener información detalla de si el sistema en pruebas realiza una correcta gestión de la misma. A pesar de ello, ASAN produce una clara caída en el rendimiento. Las vulnerabilidades que ASAN es capaz de detectar son las siguientes:

- Use after free (dangling pointer dereference)
- Heap buffer overflow
- Stack buffer overflow
- Global buffer overflow
- Use after return
- Use after scope
- Initialization order bugs
- Memory leaks

2.4. Clasificación de fuzzers

Como se ha comentado en la sección 2.2, el generador de entradas es el componente encargado de crear las variaciones sobre las entradas que se entregarán finalmente al

programa en pruebas a través del mecanismo de entrega. Existen múltiples *fuzzers* con implementaciones totalmente diferentes de su generador de entradas aunque principalmente pueden dividirse en dos categorías principales, mutativos o generativos:

2.4.1. Mutativos o genéticos

Un *fuzzer* mutativo toma una serie de datos y produce alteraciones sobre ellos para obtener un nuevo conjunto de datos. Estos *fuzzers* tienen mucha menos inteligencia asociada a ellos que los *fuzzers* generativos pero también incluyen menos esfuerzo humano para su comprensión.

Este tipo de *fuzzer* pueden ser empleados en casos en los que el sistema de pruebas recibe datos altamente estructurados. Un motivo importante por el que utilizar *fuzzers* mutativos es que permiten aprovechar fácilmente estructuras internas complejas desde el fichero de entrada que se le envía. Por ello, es mucho más simple implementar un *fuzzer* mutativo que realice pruebas en caminos profundos de un sistema que un *fuzzer* generativo.

Una desventaja de este tipo de *fuzzers* es que suelen estar limitados en la cobertura del código según el fichero de entrada que se emplee y es posible que no se realicen pruebas a funcionalidad para la que no hay ejemplos disponibles.

2.4.2. Generativos

Un *fuzzer* generativo [DoDAGRMG] se caracteriza por generar nuevas entradas para el sistema con sus propios recursos. El contenido y estructura de esta entrada puede variar desde datos aleatorios sin estructura hasta datos altamente estructurados dependiendo del punto de entrada del *fuzzer* y la profundidad deseada dentro del sistema de pruebas.

La principal diferencia con un *fuzzer* mutativo es que se debe aplicar un esfuerzo adicional para comprender los formatos y estructuras del sistema en pruebas. A pesar de ello, si se realiza un entendimiento completo de la estructura, un *fuzzer* generativo puede alcanzar una gran cobertura con mucha eficiencia. Por el contrario, si no se comprende la estructura de los datos de entrada, la cobertura de código y los fallos que se puedan detectar serán peores.

Por lo general, los *fuzzers* generativos son más potentes que los mutativos pero requieren de mayor esfuerzo, lo que puede resultar en una limitación, por ejemplo, en el caso de realizar pruebas sobre protocolos de red. Un *fuzzer* mutativo podría realizar alteraciones sobre una captura de un paquete de red para realizar pruebas a un sistema que implementa un protocolo de red, como por ejemplo FTP, mientras que para un *fuzzer* generativo debería de comprender primero una serie de protocolos de red antes de realizar las pruebas, como el protocolo DNS, TCP, etc.

Los *fuzzers* mutativos o generativos puede estar caracterizados por las siguientes técnicas:

Caché ajena

Los *fuzzers* de caché ajena (*Cache-oblivious* en inglés) no tienen en cuenta el tipo y la estructura de la información que recibe el sistema en pruebas. En el caso de un *fuzzer* mutativo de caché ajena, el generador de entradas selecciona bytes del caso de pruebas inicial y los reemplaza de forma aleatoria, generando una cadena de bytes aleatorios.

Basados en plantilla

Los *fuzzers* con algoritmos basados en plantilla (*Template-based* en inglés) tienen un mínimo conocimiento del sistema en pruebas y de los datos que recibe. Este conocimiento se almacena en una plantilla para formar un caso de prueba estructurado. Tanto los mutativos como los generativos pueden emplear estas plantillas para incluir los nuevos valores generados para evitar que sean rechazados por el sistema en pruebas. Cuando se están realizando análisis de sistemas que requieran de datos de entrada altamente estructurados, el empleo de estas plantillas puede incrementar la profundidad del análisis e incrementar así la eficiencia.

Basados en bloque

Los *fuzzers* con algoritmos basados en bloque (*Block-based* en inglés) hacen referencia a los primeros *fuzzers* que aparecieron. Los primeros *fuzzers*, como los basados en plantilla, representaban los datos como campos y secuencias de caracteres. Un ejemplo es SPIKE [Imm] que representaba todos los datos como bloques anidados de diferentes tipos, haciendo más fácil la construcción de funciones que operaban con estos datos en el sistema en pruebas. Comúnmente se empleaba para analizar protocolos de red.

Gramaticales

Los *fuzzers* con algoritmos gramaticales (*Grammar-based* en inglés) incorporan una gramática que forma parte de los datos de entrada que recibe un sistema de pruebas. Generalmente las gramáticas se encuentran en los *fuzzers* generativos aunque es posible emplearlas para los *fuzzers* mutativos. Un ejemplo podría aplicar para realizar pruebas en compiladores de código.

Basados en heurísticas

Los *fuzzers* que incorporan heurísticas (*Heuristic-based* en inglés) son capaces de tomar decisiones más inteligentes que aquellos que emplean modificaciones aleatorias sobre los datos de entrada. Estas heurísticas típicamente intentan explotar errores comunes de transformación de Unicode, UTF-8 o ASCII, desbordamiento de enteros o errores *off-by-one* entre otros.

2.5. Técnicas de fuzzing

El *fuzzing* permite la posibilidad de analizar cualquier tipo de sistema aunque cuanta mayor sea la información de la que se dispone mayor cobertura se va a poder realizar sobre un determinado programa. Por ello, existen tres tipos distintos de análisis y que supondrán diferencias en el proceso de *fuzzing* y en los resultados que se puedan obtener. Estas técnicas se clasifican según la cantidad de información de la que se dispone de un sistema de pruebas. Se clasifican en pruebas de caja negra o *black box*, de caja gris o *grey box* o de caja blanca o *white box*.

2.5.1. Black box o Caja negra

Las pruebas de caja negra o *black box* hace referencia al *fuzzing* tradicional. Se refiere a aquellas de las que no se dispone más que del binario del programa o de la librería y no se tiene nada de información de su funcionamiento más que la entrada que se le envía y la salida que se obtiene.

Existen *fuzzers* que pueden operar sin instrumentar el código y pueden cumplir su propósito generando entradas, enviándolas al programa objetivo y analizando el resultado que provocan sobre el mismo. A pesar de ser posible, el proceso de *fuzzing* se ve notablemente afectado en el rendimiento y será más lento que en el caso de disponer del código y poder realizar la instrumentación.

Este tipo de pruebas se suele realizar para comprobar si sería posible identificar fallos y suele estar relacionado con herramientas con licencia. Un fabricante puede estar interesado en que no se conozca su sistema y sea más complicado encontrar errores. Si el objetivo es localizar los fallos y corregirlos, este tipo de pruebas no es la más óptima.

2.5.2. White box o Caja blanca

A diferencia de la caja negra, la caja blanca dispone del código del programa por lo que el *fuzzer* podría aplicar modificaciones en tiempo de compilación para monitorizar el sistema e identificar las alteraciones que se produzcan para poder centrarse en el mejor camino.

Disponiendo del código, el proceso de *fuzzing* se puede centrar en determinadas funciones en vez de en todo el programa. Disponer del código permite realizar un estudio y optimizar de esta forma la búsqueda de vulnerabilidades en funciones sensibles del código y adaptar el funcionamiento del *fuzzer* para analizar variables fuera del alcance del mismo, como por ejemplo en variables de entorno.

Un enfoque de caja blanca puede permitir encontrar las vulnerabilidades de forma más eficiente aunque requerirá de mucho tiempo analizar y comprender el código del sistema sobre el que se aplicarán las pruebas.

2.5.3. Grey box o Caja gris

Comúnmente, en las pruebas de software, siempre se han considerado dos categorías: caja negra y caja blanca. En una no se dispone de ninguna información y en la otra se

dispone del código de la herramienta sobre la que se realizan las pruebas. Con el paso de los años, los investigadores decidieron considerar una nueva categoría, llamada caja gris. En las pruebas de *fuzzing* de caja gris, el *fuzzer* dispone de alguna información del sistema de pruebas y no se basa simplemente en la entrada y en salida [Hje].

Un ejemplo sería en el caso de que el *fuzzer* aplicase ingeniería inversa para obtener, por ejemplo, las funciones exportadas por una determinada librería y fuese capaz de focalizar las pruebas en estas funciones. A pesar de contar con alguna información, un *fuzzer* podría encontrar caminos de un programa por los que es incapaz de continuar ya que no dispone apenas de información del sistema en pruebas.

Capítulo 3

Estado del arte

Con el paso de los años, las técnicas de *fuzzing* han ido evolucionando desde *fuzzers* simples de caja negra que hacían uso de entradas aleatorias a *fuzzers* modernos centrados en la instrumentación del código optimizando tanto las entradas que emplea como el análisis de los resultados.

Los *fuzzers* han mejorado pero el cambio principal se centra en la cantidad de información que poseen estas herramientas. Por ello, cuanta más información se tiene del programa o la librería sobre el que se van a realizar las pruebas mayor será la cobertura del código, la velocidad del *fuzzer* y su efectividad.

En esta sección se van a comentar tres de los *fuzzers* de propósito general más empleados por la comunidad y que han sido capaces de detectar múltiples vulnerabilidades importantes de los últimos años.

3.1. American Fuzzy Lop (AFL)

American Fuzzy Lop [MZ], conocido como AFL, es un *fuzzer* orientado a la seguridad y detección de fallos que emplea un técnica de instrumentación en tiempo de ejecución y algoritmos genéticos para generar casos de prueba que produzcan estados inconsistentes en los binarios objetivo. Esta técnica mejora la cobertura del código del sistema en pruebas.

Es uno de los *fuzzers* de propósito general más conocidos y empleados para la detección de fallos en productos software ya que es una herramienta sólida, rápida, sofisticada y se puede emplear junto con otras herramientas más específicas. Durante su uso, el usuario obtiene una ventana con información que detalla el estado del *fuzzer*, del binario, las técnicas que se están empleado y los resultados obtenidos en tiempo real.

En este sección se comentan conceptos generales de AFL, los detalles de la pantalla de estado así como técnicas de optimización que implementa el *fuzzer*.

3.1.1. Conceptos generales

En esta sección se van a detallar los aspectos generales de AFL, incluyendo las herramientas que ofrece, la estructura del directorio de salida y las opciones que ofrece de *fuzzing* en paralelo y de integración con ASAN.

Componentes

Los *fuzzers* de propósito general están pensados para trabajar de la forma más simple y óptima posible. AFL trata de cumplir siempre estas premisas y por ello ofrece herramientas de instrumentación para hacerlo posible. Por ello, se incluyen las herramientas afl-gcc, afl-g++, afl-clang y afl-clang++ que le indican de forma automática las opciones de compilación a cada compilador para realizar la instrumentación.

AFL, además de las herramientas de compilación para instrumentar un determinado binario y de afl-fuzz, ofrece distintas utilidades como afl-tmin, empleado para optimizar los casos de pruebas; afl-cmin, empleado para minimizar los ficheros de entrada manteniendo los caminos encontrados; herramientas de visualización como afl-plot o afl-whatsup, útil para recopilar el estado de todas las instancias de AFL que se encuentran en ejecución. Una descripción más detallada de cada componente se encuentra disponible en el anexo D.

Estructura del directorio de salida

AFL va a generar una estructura en el directorio de salida en el que se va a almacenar información relevante del proceso de *fuzzing*. Destacan tres directorios: *queue*, hangs y *crashes*. La información que almacena cada directorio es la siguiente:

- **queue:** Este directorio contiene los casos de prueba para cada camino de ejecución distinto junto a todos los ficheros de entrada especificados en la carpeta de *input*. Estos ficheros se pueden volver a emplear para analizar el funcionamiento del programa con las alteraciones que ha generado AFL aunque es recomendable emplear la herramienta afl-cmin para minimizar estos ficheros y mantener la cobertura del código.
- **hangs:** El contenido del directorio hace referencia a los casos de prueba que han provocado que el programa produzca *time-out*. El valor por defecto para que se considere una caída es de un segundo o el valor especificado en el parámetro -t.
- **crashes:** Este directorio contiene los casos de prueba que han provocado que el programa envíe una señal de fallo (SIGSEGV, SIGILL, SIGABRT). El nombre de cada fichero contiene información del fallo que se ha producido.

Además, se incluye datos útiles para generar gráficos del proceso de *fuzzing* para ver de forma visual los resultados obtenidos. Con la herramienta afl-plot se pueden generar estos gráficos.

Fuzzing en paralelo

AFL cuenta con la opción de lanzar el *fuzzer* tantas veces como núcleos tiene el procesador. Teniendo en cuenta que cada núcleo tiene dos hilos, AFL podrá funcionar con tantos procesos en paralelo como núcleos tiene el procesador por dos. Para lanzar AFL múltiples veces y realizar tareas de *fuzzing* sobre objetivo de forma sincronizada existen las opciones -M (*master*) y -S (*slaves*).

La diferencia principal entre un maestro y un esclavo es que el maestro realizará validaciones deterministas mientras que los esclavos procederán a las alteraciones de los ficheros de entrada. Contar con un maestro no es necesario ya que es posible que no se quiera realizar *fuzzing* determinista para lo que se podrán lanzar todas las instancias con la opción -S, siendo útil en aquellos casos en los que el sistema objetivo es complejo y lento o cuando se trata de un proceso con alta paralelización.

En el caso de paralelizar el proceso, la estructura del directorio de salida se ampliará y se incluirán los resultados de cada proceso en un directorio propio con el nombre asignado en la opción -M o -S. El interior de cada una de estos directorios mantiene la estructura que si se emplease AFL con un único proceso.

AFL Address Sanitizer

AFL implementa la posibilidad de emplear ASAN, comentado en la sección 2.3, mediante los compiladores afl-gcc y afl-clang que integra. Cuando se vaya a compilar el código de un programa o de una librería, es posible añadir la opción AFL_USE_ASAN=1 tanto a afl-gcc como a afl-clang.

La principal característica de AFL es que reserva una gran cantidad de memoria virtual. En arquitecturas de 32 bits la memoria reservada oscila entre 600MB y 800MB, una cantidad asequible en la actualidad. En cambio, en arquitecturas de 64 bits, la memoria que requiere puede alcanzar los 20TB en las nuevas versiones y a menos que se indique la cantidad de memoria con gran precisión, no se tendrá protección contra fallos de *Out-of-Memory*.

Por ello, se recomienda emplear ASAN en arquitecturas de 32 bits y, en caso de emplearse una arquitectura de 64 bits, compilar el programa o la librería para arquitecturas de 32 bits (gcc -m 32).

3.1.2. Pantalla de estado

El estado de las pruebas de *fuzzing* se puede observar en los paneles y subpaneles que ofrece AFL. Cada una de las secciones se va actualizando continuamente durante el proceso y permanece operativo hasta que el usuario lo cancela. La información que aporta AFL se desglosa en varios apartados.

Información temporal

Esta sección muestra información temporal del proceso e indica el tiempo total desde que se inició el *fuzzer* y el tiempo que ha transcurrido desde el último descubrimiento.

La sección de los descubrimientos se divide en caminos, fallos y caídas únicas.

Esta sección es importante ya que, en caso de que AFL no encuentre nuevos caminos pasado un tiempo desde el inicio, mostrará un mensaje *(odd, check syntax!)* que indica que no se está invocando correctamente al binario objetivo. Otro motivo por el que se mostrará el mensaje es el tamaño de la memoria reservada ya que, si la memoria es limitada, pueden provocarse fallos al tratar de acceder a ella.

Resultados globales

Esta sección da información sobre las pruebas en el programa objetivo, indicando si es conveniente parar el *fuzzer*. En primer lugar se muestra el número de veces que AFL ha aplicado todas las modificaciones interesantes sobre el caso de prueba y se lo ha enviado al programa objetivo, volviendo al inicio. El número de ciclos completados se muestra en una sucesión de colores que van desde el magenta durante las primeras iteraciones, pasando por el amarillo, azul y finalmente verde, indicando que no se han encontrado nuevas acciones durante un tiempo.

El resto de resultados muestran los caminos encontrados junto a los fallos y caídas únicas que se han podido producir mientras se realizaba el *fuzzing* sobre el objetivo.

Progreso del ciclo

Este apartado muestra información sobre el ciclo actual. Muestra el identificador del caso de prueba que se está realizando junto al número de entradas que ha preferido evitar porque producían *time-out*.

En ocasiones aparecerá un "*" en la primera línea lo que indica que el camino que se está procesando no está marcado como favorable, es decir, dicho camino deberá esperar una serie de vueltas para ser favorable.

Mapa de cobertura

La línea de densidad del mapa muestra información sobre cuantas tuplas de ramas se han acertado en proporción a las ramas que el mapa de bits puede mantener. El valor de la izquierda muestra la entrada actual y el valor de la izquierda muestra el valor para el *corpus* de la entrada completa.

Si los primeros valores son anómalos, pueden deberse a tres motivos: el programa es muy simple, no está correctamente instrumentado o que el programa abandona la ejecución prematuramente con los casos de prueba. En caso de haber valores inesperados se resaltará con color rosa.

El segundo valor muestra la variabilidad en las tuplas de aciertos vistos en el binario. Si cada una de las ramas escogidas se elige un número fijo de veces para todas las entradas, aparecerá el valor "1.00". Si es posible disparar los aciertos para cada rama, el indicador llegará hasta "8.00" (cada bit del mapa de ocho es un acierto) aunque la probabilidad es muy pequeña.

Progreso de etapa

Este apartado de AFL muestra lo que está realizando el *fuzzer*. En la primera línea se muestra la técnica que está aplicando y en la siguientes se muestra información acerca de las ejecuciones.

Las técnicas que emplea AFL son las siguientes:

- **Calibration:** Fase anterior a iniciar las pruebas en la que se examina el camino de ejecución para detectar anomalias, establecer la velocidad de las ejecuciones y más. Se realiza rápidamente también en caso de que se haya hecho algún descubrimiento.
- **Trim L/S:** Otra fase anterior al inicio de las pruebas en la que el caso de prueba se minimiza de forma que todavía se obtenga el mismo camino de ejecución. El tamaño (L de *length* en inglés) y el paso (S de *stepover* en inglés), se eligen en función del tamaño del fichero.
- **Bitflip L/S:** Alteraciones de bits de forma determinista. Se modifican L bits en un momento dado, a lo largo de todo el fichero con incrementos de tamaño S. Las variantes de L/S son: 1/1, 2/1, 4/1, 8/8, 16/8, 32/8.
- **Arith L/8:** Aritmética determinista. AFL intenta restar y sumar enteros pequeños hasta valores 8, 16 y 32 bits. El paso (S) es siempre de 8 bits.
- **Interest L/8:** Sobrescritura del valor determinista. AFL cuenta con una lista de valores de 8, 16 y 32 bits "interesantes" para probar. El paso (S) es siempre de 8 bits.
- **Extras:** Inyección determinista de términos de un diccionario. Puede aparecer como *user* o *auto*, dependiendo si se ha indicado el uso de un diccionario determinado (opción -x) o uno auto-generado. También aparece *over* o *insert* dependiendo de cuando las palabras del diccionario sobrescriban datos existentes o si son insertados empleando su tamaño según los datos restantes.
- **Havoc:** Esta técnica realiza ciclos con datos de longitud fija sobre los que se apilan modificaciones. Las operaciones que intentan en esta fase incluyen *bit flips*, enteros aleatorios, borrado de bloque, duplicado de bloque junto a operaciones relacionados con el diccionario en caso de que se indique el empleo de uno.
- **Splice:** Última técnica que se realiza cuando se han aplicado todas las técnicas anteriores en un ciclo sin encontrar nuevos caminos. Es equivalente a *havoc* excepto en que primero añade dos valores aleatorios de la entrada en un punto arbitrario.
- **Sync:** Esta técnica solo se aplica cuando se ha lanzado AFL indicando un proceso maestro (-M) y uno o más esclavos (-S). La herramienta analiza la salida de otros *fuzzers* e importa casos de prueba.

Hallazgos en profundidad

La sección muestra varias métricas como el número de caminos más óptimos desde el punto de vista del *fuzzer* según un algoritmo de minimización implementado en el código y el número de casos de prueba que han dado mayor cobertura. En esta sección se muestran también el número de fallos totales junto a los fallos únicos.

Por último se muestra el número total de *time-outs*. Es importante diferenciar entre un *time-out* y una caída del programa ya que el *fuzzer* incorpora una diferencia entre los tiempos para decidir en que caso se trata de un *time-out* o de una caída.

Estrategia

Los primeros seis campos de esta sección tratan de analizar en tiempo real cuantos caminos se han capturado, en proporción al número de ejecuciones, para cada una de las estrategias de *fuzzing* que aplica AFL. Todos estos campos sirven para validar la utilidad de los distintos enfoques que aplica AFL.

La estrategia *trim* es diferente del resto. El primer porcentaje muestra el ratio de bytes eliminados de los ficheros que recibe AFL. El segundo número se corresponde con el número de ejecuciones que se han necesitado para lograr esa meta. Por último, el tercer porcentaje muestra la proporción de bytes que, a pesar de no ser posible su eliminación, se ha considerado que no tienen efecto y se han excluido de algunos de las etapas más costosas del proceso de *fuzzing*.

Geometría del camino

El primer campo muestra la profundidad del camino alcanzado en el proceso. Se muestra como números consecutivos en los que el nivel uno hace referencia a los casos de prueba iniciales; el nivel dos a aquellos casos de prueba que pueden obtener con técnicas de *fuzzing* tradicional; y el nivel tres para el uso de casos de prueba generados anteriormente en rondas siguientes.

El segundo campo indica las entradas que todavía no han sido procesadas seguido del indicador que muestra las entradas favorecidas pendientes que AFL quiere incluir en la cola del ciclo. Aquellas entradas que no son favorables deberán esperar unos cuantos ciclos para ser empleadas.

Los campos *own finds* e *imported* muestran los nuevos caminos encontrados durante el intento actual y los importados por otras instancias de AFL cuando se realiza *fuzzing* paralelo, respectivamente.

El último campo es indicativo de que el proceso es estable. Muestra la consistencia de las trazas observadas. Si un sistema en pruebas se comporta de la misma manera con los mismos datos de entrada, el campo *stability* mostrará 100 %. Cuando el valor es más bajo pero se muestra en morado, trata de informar de que el proceso no tiene muchas probabilidades de ser negativo. Si el valor aparece marcado en rojo indica que AFL tiene dificultades para diferenciar entre efectos significados y efectos "fantasma" al modificar el fichero de entrada. En el anexo C se muestra información del motivo por el que la estabilidad puede ser baja durante el *fuzzing*.

Aquellos caminos en los que se ha detectado un comportamiento variable se almacenan en un directorio dentro de la carpeta *queue* del directorio de salida indicado a AFL para que estén accesibles fácilmente para su análisis futuro.

Todas las secciones en conjunto ofrecen gran información sobre el estado global del proceso y muestra de forma muy visual si algo no está yendo bien y es necesario revisar el binario objetivo. A continuación se muestra la pantalla de estado de AFL durante la búsqueda de la vulnerabilidad Heartbleed:

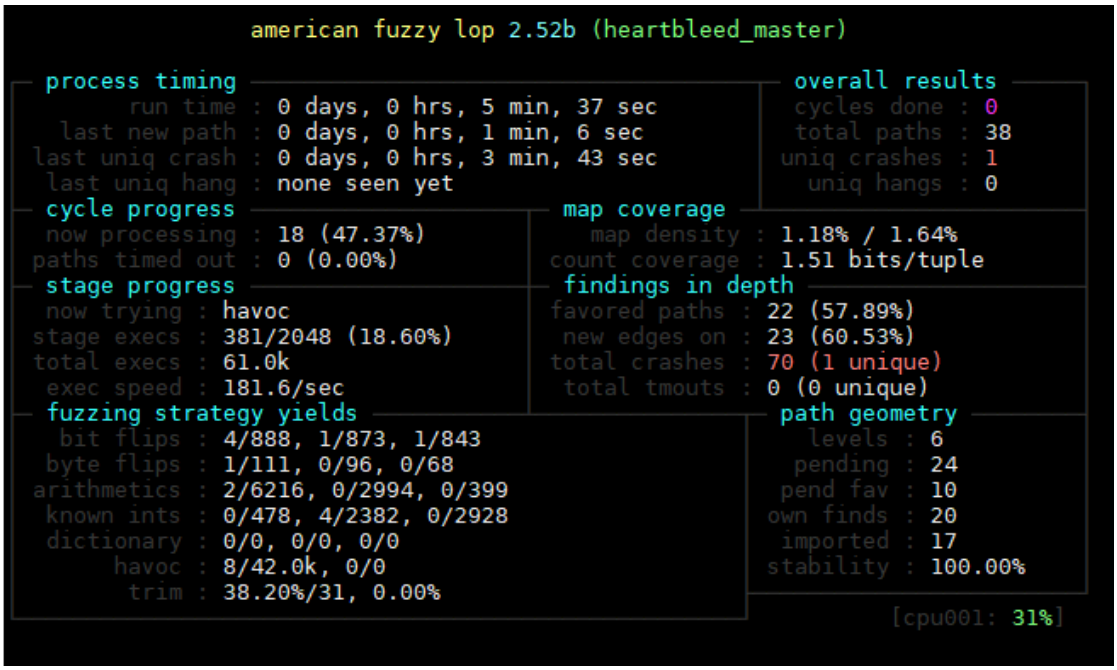


Figura 3.1: Pantalla de AFL durante la búsqueda de la vulnerabilidad Heartbleed

3.1.3. Modo persistente

El modo persistente consiste en la inclusión en el código de un bucle interpretado por el *fuzzer* para realizar pruebas sobre un determinado programa optimizando la ejecución evitando el inicio de procesos con fork() y la llamada con execve(). Esta optimización supone un riesgo ya que es posible que la ejecución se descarrile debido a fugas de memoria o condiciones de denegación de servicio en el código. AFL cuenta con esta opción y es posible emplear el modo persistente de la siguiente forma:

```
1  int main(int argc, char** argv) {
2
3    while (__AFL_LOOP(1000)) {
4
5      /* Reset state. */
```

```
6       memset(buf, 0, 100);
7
8       /* Read input data. */
9       read(0, buf, 100);
10
11      /*Call to a function or library */
12    }
13  }
```

Mediante __AFL_LOOP es posible indicarle a AFL que realice ese bucle un número de veces determinado sin realizar fork() de forma que se ahorrará tiempo al no emplear esa llamada de sistema con tanta frecuencia.

Dado que se van a realizar iteraciones sobre el mismo bucle empleando las mismas variables, es recomendable eliminar el contenido que se haya podido introducir en la memoria asignada a cada variable para evitar que queden residuos entre cada iteración. Así, durante cada iteración, el *fuzzer* enviará información al programa a través de STDIN que se almacenará en la variable *buf*, encargada de almacenar el contenido que se empleará en la llamada a la función bajo pruebas.

Este modo permite optimizar el proceso de *fuzzing* hasta diez veces más que si se emplease la instrumentación sin persistencia.

3.1.4. Modo LLVM

El proyecto LLVM [oIb] se inició como una investigación de la Universidad de Illinois con la meta de proveer un sistema moderno de compilación con la intención de dar soporte a la compilación estática y dinámica de lenguajes de programación arbitrarios. Desde entonces, LLVM ha crecido como un proyecto compuesto por múltiples subproyectos alguno de los cuales se emplean en entornos de producción tanto comerciales como *open source* además de emplearse en investigaciones académicas.

AFL incorpora la posibilidad de emplear LLVM lo que permite emplear las utilidades afl-clang-fast y afl-clang-fast++. Su uso permite optimizar el rendimiento de las tareas de *fuzzing*. Requiere disponer de clang en la máquina en la que se vayan a realizar las pruebas.

El modo LLVM se basa en la realización de optimizaciones en tiempo de compilación. La mayoría de los compiladores de C tienen una arquitectura dividida en tres partes: *Frontend*, *Optimizer* y *Backend*. El *Frontend* debe interpretar el código de entrada y elaborar un Abstract Syntax Tree (AST) dependiente del lenguaje para representar el código de entrada. El *Optimizer* es el responsable de transformar el código y mejorarlo en su tiempo de ejecución mientras que el *Backend* o generador de código "mapea" el código producido por el *Optimizer* con el conjunto de instrucciones de la arquitectura objetivo.

El aspecto más positivo de este diseño es que es posible emplear un mismo optimizador para distintos lenguajes de programación ya que sería necesario elaborar un *Frontend* para cada lenguaje que genere una representación común del código independientemente

del lenguaje. El *Backend* recibirá también una entrada genérica del *Optimizer* y de este modo generar el conjunto de instrucciones para cada arquitectura de forma independiente. A continuación se incluye una figura con la arquitectura de LLVM y el propósito para el que se diseñó:

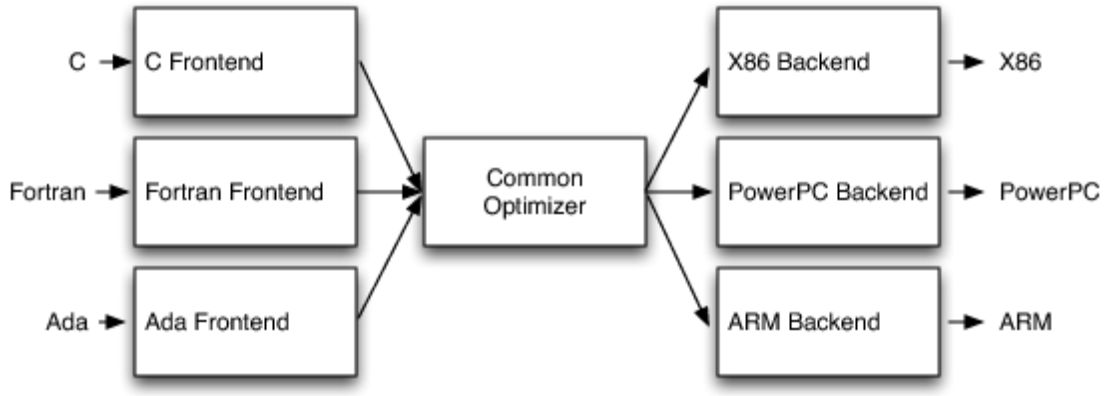


Figura 3.2: Arquitectura de LLVM

3.1.5. QEMU

QEMU [Bel] es un emulador de procesadores basado en la traducción dinámica de binarios de una arquitectura a la arquitectura del sistema que realiza la emulación. Su principal virtud es la capacidad de virtualización dentro de un sistema operativo, ya sea GNU/Linux, Windows, o cualquiera de los sistemas operativos admitidos. Además, ofrece la posibilidad de ejecutar binarios de Linux o BSD compilados para arquitecturas diferentes, lo que se conoce como *User-mode emulation*.

AFL aprovecha el *User-mode emulation* para instrumentar la salida de binarios de los que no se dispone del código y no ha sido posible realizar la instrumentación en tiempo de compilación. El empleo de esta técnica afecta al rendimiento del *fuzzing* debido a los coste de la emulación.

Para aplicar *fuzzing* en binarios de los que no se dispone del código es necesario instalar una modificación de QEMU. AFL incluye el directorio qemu_mode/ en el que se incluye un *script* para descargar, configurar y compilar QEMU. El *script* puede realizar la instalación de QEMU para una arquitectura concreta indicando la opción CPU_TARGET, útil para ejecutar programas de arquitecturas concretas o para emplear binarios de 32 bits en arquitecturas de 64 bits.

3.2. HonggFuzz

Honggfuzz [Goob] es otro *fuzzer* orientado a la seguridad, evolutivo y simple de usar. Es un proyecto multiplataforma mantenido por Google y que funciona en sistemas operativos Linux, FreeBSD, NetBSD, Android, Windows, OSX e iOS.

Se trata de otro *fuzzer* de propósito general, muy conocido por la comunidad, y que también muestra en tiempo real los descubrimientos realizados junto al estado del *fuzzer*.

A continuación se incluyen los detalles generales de Honggfuzz, una descripción de la pantalla de estado, más simple que la que muestra AFL, además de la posibilidad de emplear el modo persistente y su compatibilidad con otros sistemas operativos.

3.2.1. Conceptos generales

En esta sección se incluyen las características principales de Honggfuzz que lo hacen destacar entre otros *fuzzers* de propósito general además de su principal peculiaridad, la instrumentación por software y por hardware junto a un descripción de la pantalla de estado.

Características

Las características principales de Honggfuzz que lo hacen destacar como *fuzzer* de propósito general son las siguientes:

- **Multi-thread y multi-proceso:** No es necesario lanzar múltiples veces el *fuzzer* ya que trata de acceder a todos los núcleos disponibles. Honggfuzz tiene una opción para indicar el número de threads que por defecto son NUCLEOS/2.
- **Es rápido:** Especialmente si se emplea el modo persistente. Una función simple puede invocarse 1 millón de veces por segundo en un procesador moderno.
- **Múltiples fallos encontrados:** Honggfuzz ha sido empleado para encontrar, por ejemplo, vulnerabilidades críticas en OpenSSL.
- **Empleo de interfaces de bajo nivel para monitorizar los procesos (ptrace):** A diferencia de otros *fuzzers*, encontrará y notificará señales del sistema ocultas.
- **Fácil de usar:** Con una cuerpo de entrada básico, como un fichero de 1 byte, Honggfuzz generará entradas más óptimas con su algoritmo evolutivo.
- **Múltiples métodos de instrumentación:** Honggfuzz permite realizar la instrumentación por software (como AFL) o por hardware.
- **Portable:** Puede emplear con procesos remotos o autónomos como Apache httpd.

Instrumentación

Honggfuzz es capaz de realizar la instrumentación empleando componentes software y hardware. Por un lado, cuenta con opciones para realizar la instrumentación del código de un determinado programa por software con los compiladores gcc y clang. Para instrumentar con gcc o con versiones de clang anteriores a la 4.0 se puede emplear la opción -finstrument-functions mientras que para realizar la instrumentación con

versiones modernas de clang se deben emplear las opciones *-fsanitize-coverage=trace-pc-guard,indirect-calls,trace-cmp*. Para facilitar la instrumentación Honggfuzz cuenta con el directorio hfuzz_cc/ donde se encuentran las herramientas para este propósito.

Para la instrumentación por hardware es necesario tener en cuenta el tipo de procesador. Honggfuzz ofrece varios tipos de instrumentación según las especificaciones de la CPU. Por un lado, los procesadores Intel cuentan con registros especiales como los Branch Trace Store (BTS) o el subsistema Process Trace (PT). A continuación se dan detalles de ambas funcionalidades:

- **Branch Trace Store (BTS).** Estos registros permiten almacenar e identificar pares de saltos y guardar aquellos pares únicos conociendo desde donde se salta y hacia donde. Para emplear esta opción Honggfuzz incluye el parámetro *linux_perf_bts_edge*.
- **Subsistema Intel Process Trace (PT).** Permite la monitorización de la ejecución con muy poco impacto que se centra en cada thread empleando utilidades hardware dedicadas de forma que cuando se termina la ejecución, el software que lo emplea pueda recuperar la traza y reconstruir el flujo del programa. Honggfuzz ofrece esta posibilidad con el parámetro *linux_perf_ipt_block*. Esta opción es más rápida que emplear los registros BTS pero los resultados serán menos precisos.

En cierto modo, los procesadores Intel cuentan con mayores posibilidades empleando este *fuzzer* pero Honggfuzz incluye una serie de opciones para realizar la instrumentación por hardware de forma que sea funcionalidad para procesadores Intel, AMD y otras arquitecturas:

- **Contador de instrucciones.** Este modo trata de maximizar el número de instrucciones para cada proceso en cada iteración. El contador se obtiene mediante el subsistema perf de Linux. Para ello se incluye el parámetro *linux_perf_instr* para indicar al *fuzzer* que se emplee esta instrumentación.
- **Contador de saltos.** Este modo trata de maximizar el número de saltos que realiza la CPU durante la ejecución del proceso en cada iteración. Honggfuzz emplea el parámetro *linux_perf_branch* para indicar al *fuzzer* que se emplee esta instrumentación.

Ficheros de salida

Este *fuzzer* no cuenta con una estructura definida de directorios y, por defecto, almacena los ficheros de salida en el directorio actual. A pesar de ello, cuenta con una opción para indicarle un directorio de trabajo (*workspace*). En ese directorio, en caso de haber sido especificado, aparecerá un reporte con un resumen de la actividad y los fallos detectados, cada uno en un fichero independiente. El nombre de los ficheros de salida tiene el siguiente formato:

- **SIGSEGV, SIGILL, SIGBUS, SIGABRT, SIGFPE.** Nombre de la señal que ha provocado que el proceso finalice. Si se emplea ptrace() es la señal entragada al proceso.
- **PC junto al valor.** Solo si se emplea ptrace(). En arquitecturas de x86 es el valor del registro EIP y para arquitecturas de x86-64, el registro RIP.
- **STACK y su firma.** Firma del *stack* basado en *stack-tracing*.
- **ADDR junto al valor.** Incluye el valor de la dirección siginfo_t.si_addr. No tiene significado para la señal SIGABRT.
- **INSTR e información de la última instruccion.** Instrucción en ensamblador encontrada en el último Contador de Programa (PC). Únicamente en arquitecturas x86 y x86-64. No tiene utilidad para la señal SIGABRT.

El reporte final con los fallos que se han producido durante el proceso de *fuzzing* muestra la información más relevante como el nombre del fallo con los valores indicados anteriormente, el PID del proceso, la señal recibida, la dirección de fallo, la instrucción y una firma del stack junto con los parámetros especificados:

```
================================================================
TIME: 2018-10-25.15:20:51
================================================================
FUZZER ARGS:
 mutationsPerRun : 6
 externalCmd     : NULL
 fuzzStdin       : TRUE
 timeout         : 10 (sec)
 ignoreAddr      : (nil)
 ASLimit         : 0 (MiB)
 RSSLimit        : 0 (MiB)
 DATALimit       : 0 (MiB)
 targetPid       : 0
 targetCmd       :
 wordlistFile    : /home/fuzz/sergio/hongfuzz/ntpq.dict
 dynFileMethod:
 fuzzTarget   : /home/fuzz/sergio/ntp-4.2.2/ntpq/ntpq
ORIG_FNAME: [DYNAMIC]
FUZZ_FNAME: ./SIGSEGV.PC.445180.STACK.badbad0c3cdf5d27.CODE.1.ADDR.0x7fffffff000.INSTR.mov____%r13b,-0x1(%r15).2018-10-25.15:20:51.32246.fuzz
PID: 32246
SIGNAL: SIGSEGV (11)
FAULT ADDRESS: 0x7fffffff000
INSTRUCTION: mov____%r13b,-0x1(%r15)
STACK HASH: badbad0c3cdf5d27
STACK:
 <0x0000000000000000> [[UNKNOWN]():0 at UNKNOWN]
================================================================
```

Figura 3.3: Información del fichero resumen de Honggfuzz

Integración con otros fuzzers

Honggfuzz ofrece la posibilidad de integrar su generador de entradas en otro *fuzzer* o emplear un generador de entradas externo para trabajar sobre un determinado programa o librería.

En el caso de querer emplear su generador de entradas, ofrece la posibilidad de aplicar las técnicas de bit o byte *flipping* con las opciones *-mb* y *-mB* respectivamente y el número de bits o bytes que se quieren alterar con la opción *-r* sobre un determinado fichero de entrada.

Por otro lado, con la opción *-c* es posible emplear un generador de entradas externo y que Honggfuzz emplee la salida generada recibiéndola por STDIN o leyendo el fichero generado.

3.2.2. Pantalla de estado

Al igual que AFL, Honggfuzz también cuenta con una pantalla que informa del estado del *fuzzer*. Principalmente, la pantalla se divide en dos secciones, en la parte superior se incluye la información del proceso y del estado del *fuzzer* y, en la parte inferior, un log con los eventos que se producen, incluidos los fallos detectados.

En comparación con AFL, Honggfuzz no muestra tanta información aunque sí que incluye los aspectos importantes que pueden servir para detectar el mal funcionamiento del *fuzzer*.

En la parte superior se puede observar el tiempo total que lleva el *fuzzer* en funcionamiento junto a la versión empleada. Dentro de los dos paneles principales, en el superior, los campos de información son los siguientes:

- **Iteraciones:** Número de veces que se ha ejecutado el binario.
- **Modo:** Puede indicar *Feedback Driven Dry Run* o *Feedback Driven Mode*.
- **Objetivo:** El binario sobre el que se están realizando las pruebas.
- **Threads:** Número de procesos en ejecución junto al número total de procesos que se podrían ejecutar al mismo tiempo. Aparece también un indicador de la capacidad de la CPU total.
- **Velocidad:** Número de ejecuciones por segundo junto a la media total.
- **Fallos:** Número de fallos detectados indicando cuales son únicos, cuales se han incluido en una lista negra y cuales están verificados.
- **Time-outs:** Número de veces que el programa no ha enviado respuesta tras recibir la entrada. Por defecto, el *time-out* es de diez segundos.
- **Tamaño de la entrada:** Tamaño de la entrada actual, bytes totales generados y el número de ficheros iniciales que se incluyen en el directorio de entrada.
- **Actualización de la cobertura:** Última actualización de la cobertura del *fuzzer*.
- **Cobertura:** Información sobre la cobertura del código del programa objetivo, incluyendo los caminos con mejores resultados y las instrucciones de comparación encontradas.

En la parte inferior se pueden observar un listado de todos los eventos que ha detectado el *fuzzer* y acciones que ha tomado para optimizar el proceso.

A continuación se muestra un ejemplo de uso de Honggfuzz y la información que muestra:

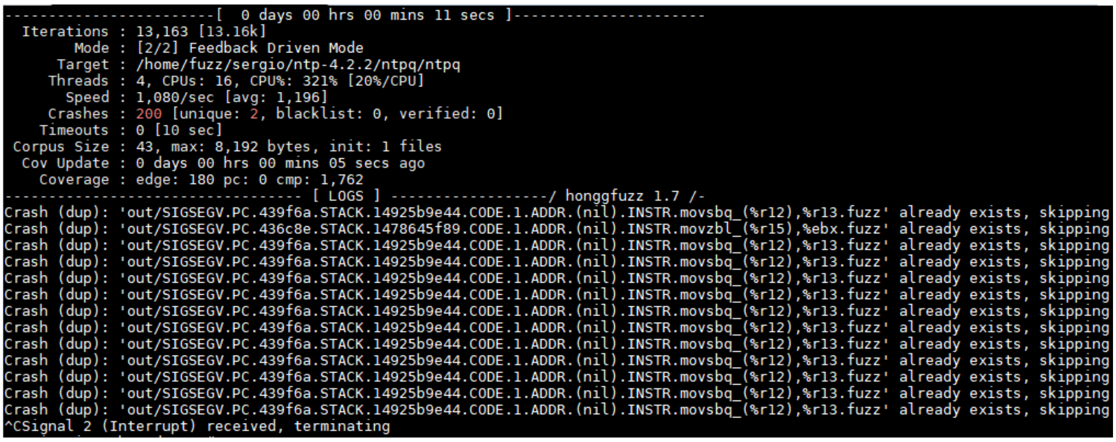


Figura 3.4: Pantalla de Honggfuzz durante la búsqueda de una vulnerabilidad

3.2.3. Modo persistente

Honggfuzz, de igual modo que AFL, cuenta con la posibilidad de realizar el proceso de *fuzzing* en modo persistente sin necesidad de crear nuevos procesos empleando las llamadas a sistema fork() y execve() incluyendo un bucle interpretado por el *fuzzer* que consigue optimizar el proceso reiniciando el estado en cada iteración.

Cuenta con tres formas distintas de añadir persistencia a un binario:

- **Estilo ASAN.** Es necesario emplear la función LLVMFuzzerTestOneInput que recibe un *buffer* de datos y el tamaño. Una fragmento simple de código iría incluido en la siguiente función:

```
int LLVMFuzzerTestOneInput(uint8_t *buf, size_t len) {
        /* Llamada a la función de interés*/
        return 0;
}
```

- **HF_ITER.** Se emplea la función HF_ITER para obtener nuevas entradas. Un ejemplo de código sería el siguiente:

```
#include <inttypes.h>

extern HF_ITER(uint8_t** buf, size_t* len);

int main(void) {
  for (;;) {
        size_t len;
        uint8_t *buf;

```

```
10              HF_ITER(&buf, &len);
11
12              /* Llamada a la función de interés */
13          }
14      }
```

- **Modo de instrumentación.** Tal como se habla en la sección 3.2.1, existen opciones para indicar la instrumentación por software o por hardware. El modo persistente no es excluyente y es posible emplear este modo junto con las opciones de instrumentación comentadas en dicha sección.

3.2.4. Compatibilidad

Honggfuzz implementa una serie de funcionalidades que lo hacen muy útil a la hora de utilizarlo como *fuzzer* en sistemas de pruebas especiales. Esto se debe a la posibilidad de incluir generadores de entradas externos y a la compilación cruzada, lo que permite emplear Honggfuzz en navegadores o en Android.

Android

Desde la version 0.6, Honggfuzz soporta el sistema operativo Android (con cross-compilación con NDK) empleando tanto ptrace() como señales del sistema. Cuando ptrace() está habilitado, el motor de Honggfuzz evita que las señales monitorizadas lleguen al *debugger* ya que el análisis en tiempo de ejecución se ve afectado.

Entre los requisitos se encuentra el NDK versión r16 con la API de Android 24 (Nougat), Libunwind disponible en Github (*commit* [bc8698f]) y Capstone versión 3.0.4.

El uso de ptrace() y la interfaz de señales del sistema se encuentra implementada en arquitecturas armeabi, armeabi-v7a, arm64-v8a, x86 y x86_64 aunque en el caso de la arquitectura arm64-v8a falla la dependencia Libunwind en objetivos de 32 bits.

Un aspecto importante del *fuzzing* en dispositivos Android es la compilación cruzada o Cross-Compiling que consiste en compilar el código de una determinada aplicación para un arquitectura diferente de la que posee la máquina que realiza la compilación. Para posibilitar la compilación cruzada en dispositivos Android, Honggfuzz incluye el directorio third_party/android/scripts para facilitar y automatizar la configuración de las dependencias y el proceso de compilado.

La compilación se realiza con el comando make según la arquitectura del dispositivo objetivo. El fichero make cuenta con la opción android-all para realizar la compilación cruzada de todas las arquitecturas disponibles o de especificar con el parametro ANDROID_APP_ABI la arquitectura de interés.

Navegadores

Honggfuzz incorpora también un modo *batch* para realizar fuzzing sobre aquellos procesos que no pueden reiniciarse cada vez que se quieran enviar nuevas entradas. Para

ello, Honggfuzz incluye el parámetro -p para indicarle el PID del proceso al que se debe unir (attach) aunque también es posible unirse a *threads*.

Un ejemplo claro es el caso de Apache. En primer lugar se debe iniciar apache en modo *debug* para que no genere procesos hijos (opción -X). Para el *fuzzing* de navegadores se suelen emplear herramientas propias implementando generalmente *fuzzers* generativos basados en plantilla. En caso de que se quiera realizar el análisis de cabeceras, Honggfuzz permite emplear la opción -c para indicarle un generador de entradas externo y con la opción -p indicarle el PID de Apache para unirse a ese proceso. Una posible forma de enviar las entradas generadas podría ser con la herramienta netcat (nc).

3.3. Libfuzzer

Libfuzzer [oIa] es un *fuzzer* basado en la instrumentación del código de un determinado programa que implementa un algoritmo evolutivo para la generación de las entradas. Este *fuzzer* se enlaza con el sistema en pruebas y le provee de casos de prueba iniciales o modificados desde un punto de entrada conocida como la "función objetivo". De esta forma, Libfuzzer analiza a que áreas del código se accede y genera mutaciones de los datos de entrada para maximizar la cobertura del código. La información sobre la cobertura del código se obtiene de la instrumentación que provee LLVM SanitizerCoverage [oIc].

Actualmente se encuentra en desarrollo por lo que es necesario disponer de una versión reciente de clang. Libfuzzer viene incluido por defecto en clang a partir de la versión 6.0.

3.3.1. Conceptos generales

En esta sección se incluyen los conceptos generales de Libfuzzer siendo, algunos de ellos, comunes a otros *fuzzers* como AFL o Honggfuzz.

Fuzz target

La primera fase para emplear Libfuzzer sobre un determinado programa consiste en implementar una *fuzz target*, es decir, una función que recibe datos en un puntero a bytes junto con el tamaño del contenido al que apunta y que realiza cualquier operación interesante con estos datos empleando las funciones del sistema de pruebas.

Un ejemplo de *fuzz target* es el siguiente:

```
1 extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t
  ↪ Size) {
2   DoSomethingInterestingWithMyAPI(Data, Size); // Función de interés
3   return 0;  // Valores distinto de cero están destinados a uso
    ↪ futuro
4 }
```

Esta función no depende de Libfuzzer y puede emplearse con otros *fuzzers* como AFL. A continuación se enumeran un conjunto de situaciones a tener en cuenta sobre una *fuzz target*:

- El motor de *fuzzing* ejecutará la *fuzz target* múltiples veces con entradas de datos diferentes en un mismo proceso.
- Debe tolerar cualquier tipo de entrada, es decir, entradas vacías, entradas grandes, entradas malformadas, etcétera.
- No debe finalizar con ninguna entrada.
- Esta función puede emplear hilos pero idealmente debe realizar un *join* al final de la función.
- Debe ser lo más determinista posible. En caso de ser indeterminista, como por ejemplo tomar decisiones con datos no obtenidas de la entrada, el proceso de *fuzzing* sessrá ineficiente.
- Debe ser rápida. Es necesario evitar complejidades de orden cúbico o superior y no emplear la memoria de forma excesiva.
- Idealmente, no se debe modificar ningún estado global aunque no es estricto.
- Cuanto mayor específico sea el procesamiento de la entrada mejor. En caso de querer emplear distintas extensiones de fichero, es conveniente emplear una *fuzz target* para cada una.

Para emplear Libfuzzer sobre un programa objetivo se debe emplear el parámetro -fsanitize=fuzzer en tiempo de compilación. Es posible emplear AddressSanitizer(ASAN), UndefinedBehaviourSanitizer (UBSAN) o ambos. Existen otros parámetros para analizar la memoria como MemorySanitizer(MSAN) pero su soporte es experimental. Un ejemplo de compilación empleando clang es el siguiente:

```
# Empleo de Libfuzzer
clang -g -O1 -fsanitize=fuzzer mytarget.c
# Empleo de Libfuzzer con ASAN
clang -g -O1 -fsanitize=fuzzer,address mytarget.c
```

Todos los *fuzzers* basados en instrumentación como Libfuzzer requieren de un conjunto de ficheros de entrada sobre los que realizar modificaciones y enviar al programa objetivo. El *fuzzer* realizará mutaciones aleatorias basadas en estas entradas y, en caso de que una de las mutaciones encuentre un camino nuevo, esa mutación se almacenará en el directorio de entradas para su uso futuro. Libfuzzer puede funcionar sin introducirle un directorio de entradas con casos de prueba pero el *fuzzing* es mucho menos eficiente. En caso de emplear un conjunto de entradas complejas es importante minimizar este conjunto para optimizar las tareas del *fuzzer*. Para minimizar un conjunto de entradas se puede emplear la opción -merge=1:

```
./target -merge=1 nuevo_directorio_inputs inputs_complejas
```

Esta opción también puede emplearse para añadir nuevos elementos interesantes a un directorio de entradas. Únicamente se añadirán los ficheros de entrada que permitan descubrir nuevos caminos en el programa objetivo. Para iniciar una *fuzz target* instrumentada se puede ejecutar el siguiente comando:

```
./target inputs -dict=dict.txt -jobs=4 -max_length=1024
```

En el ejemplo anterior se emplea un diccionario y se inicia el *fuzzer* con cuatro procesos. El parámetro de -max_length=1024 indica el tamaño máximo de los ficheros de entrada con los que puede trabajar LibFuzzer.

Fuzzing en paralelo

Cada proceso de Libfuzzer emplea un único hilo a menos que el sistema en pruebas inicie sus propios hilos. A pesar de ello, es posible iniciar múltiples procesos de Libfuzzer en paralelo con un directorio de entradas compartido. Así, los descubrimientos hechos por cada uno de los procesos iniciados serán compartidos con el resto de instancias de Libfuzzer, a menos que se deshabilite con la opción -reload=0.

Para iniciar varias instancias de Libfuzzer se debe emplear la opción -jobs=N, la cual indica que se deben iniciar N instancias. Estas tareas van a iniciar una serie de trabajadores que van a ocupar aproximadamente la mitad de la capacidad de cómputo disponible. Con la opción -workers=M se puede limitar el número de trabajadores. Por ejemplo, en una máquina con doce núcleos, si se establecen treinta tareas, por defecto se obtendrán seis trabajadores (mitad de los núcleos del procesador).

Diccionarios

Como el resto de *fuzzers*, Libfuzzer también permite el empleo de ficheros con palabras clave o secuencias de bytes que puedan ayudar en el descubrimiento de caminos favorables. En algunos casos, el empleo de diccionarios puede mejorar la velocidad de búsqueda. La sintaxis de un fichero diccionario es similar a la de AFL.

Monitorización de instrucciones CMP

Empleando el parámetro -fsanitize-coverage=trace-cmp en tiempo de compilación, incluido por defecto al emplear el parámetro -fsanitize=fuzzer, Libfuzzer recopilará todas las instrucciones CMP y guiará las mutaciones según los argumentos que reciben las instrucciones de comparación. El empleo de esta opción supone una perdida en el rendimiento pero es común que mejore los resultados.

Perfil de valor

En tiempo de compilación, cuando se está realizando la instrumentación, si se indica el parámetro -fsanitize-coverage=trace-cmp y un parámetro extra en tiempo de ejecución

-use_value_profile=1, Libfuzzer recopilará los valores de los parámetros de instrucciones de comparación y tratará esos nuevos valores como nueva cobertura.

La implementación actual realiza las siguientes tareas:

1. El compilador instrumenta todas las instrucciones CMP con un *callback* que registra los dos parámetros de la instrucción de comparación.

2. La función de *callback* manipula los argumentos de la instrucción de comparación y emplea el resultado para establecer un conjunto de bits.

3. Cada nuevo bit observado en el conjunto de bits se tratará como nueva cobertura.

El empleo de estas opciones tanto en tiempo de compilación como en tiempo de ejecución puede descubrir nuevos casos de entrada potencialmente favorables para la detección de fallos pero supone una penalización sobre el rendimiento además de producir casos de prueba de mayor tamaño.

3.3.2. Información del estado

Este *fuzzer*, por defecto, se ejecuta de forma indefinida, al menos hasta que se encuentra un fallo. Cualquier fallo o error en memoria se notificará, finalizará el proceso de *fuzzing* y el caso de prueba que provoca el fallo se almacenara en disco con un identificador asignado para fallos (*crash-sha1*), fugas de información (*leak-sha1*) o caída (*timeout-sha1*).

Al iniciar Libfuzzer se muestra información con los parámetros con los que se inicia el *fuzzer*. Entre esta información se encuentra el valor del parámetro -seed=N, los módulos cargados, el tamaño máximo de los casos de prueba que por defecto es -max_len=64 y los datos de los casos de prueba iniciales. El resto de información tiene asignado un código de evento e información estadística. Los código de eventos son los siguientes:

- **READ:** El *fuzzer* ha leído todos los datos de las muestras iniciales.
- **INITED:** El *fuzzer* ha completado la fase de inicialización que incluye la ejecución de cada caso de prueba sobre el código de pruebas.
- **NEW:** El *fuzzer* ha sido capaz de generar una nueva entrada capaz de encontrar nuevos caminos del sistema en pruebas. El fichero generado se almacenará en el directorio de trabajo.
- **REDUCE:** El *fuzzer* ha sido capaz de encontrar una entrada reducida que dispara caminos previamente encontrados. Esta opción se puede deshabilitar con el parámetro -reduce_inputs=0).
- **PULSES:** Mensaje empleado para notificar al usuario de que el *fuzzer* sigue trabajando.

- **DONE:** El *fuzzer* ha terminado ya que ha superado el valor especificado en el parámetro -runs o el tiempo de trabajo establecido por el parámetro -max_total_time.
- **RELOAD:** El *fuzzer* está realizando una carga periódica de las entradas del directorio de trabajo. Esto permite conocer las entradas descubiertas por otros procesos de *fuzzing* en paralelo.

Cada línea de salida que genera el *fuzzer* muestra las siguientes estadísticas:

- **cov:** Número total de bloques o *edges* cubiertos al ejecutar las entradas introducidas o generadas.
- **ft:** Libfuzzer emplea diferentes señales para evaluar la cobertura de código: cobertura de *edges*, contador de *edges*, valores de los perfiles, pares invocador/invocado indirectos, etcétera. Estas señáles combinadas se llaman características (*features*).
- **corp:** Número de entradas en memoria y su tamaño en bytes.
- **lim:** Límite asignada a las entradas. Se incrementa en el tiempo hasta alcanzar el valor del parámetro -max_len.
- **exec/s:** Número de ejecuciones por segundo.
- **rss:** Consumo de memoria actual.

Para los nuevos eventos, la salida por pantalla también incluye información de las operaciones de mutación que han producido la nueva entrada. El indicador L muestra el tamaño de la nueva entrada en bytes y el indicador MS cuenta y muestra un listado de las operaciones realizadas durante la mutación para generar la nueva entrada.

3.3.3. Compatibilidad con AFL

Libfuzzer puede emplearse en conjunto con AFL. Ambos *fuzzers* esperan los datos de entrada de un directorio, un fichero por cada entrada de prueba. Es posible iniciar ambas herramientas empleando el mismo directorio de entradas, uno detrás de otro. Un ejemplo de su empleo conjunto es el siguiente:

```
1 afl-fuzz -i testcases -o output -- ./bin/program @@
2 ./fuzz-target testcases output
```

Periódicamente es interesante reiniciar ambos *fuzzers* de forma que puedan compartir los elementos encontrados por cada uno. No existe una forma simple de iniciar ambos *fuzzers* en paralelo mientras comparten los elementos descubiertos dentro de su directorio de trabajo.

Capítulo 4

Búsqueda de fallos en software

Desde sus orígenes, los *fuzzers* han sido empleados por la comunidad de investigadores de seguridad para encontrar vulnerabilidades en el código de herramientas y aplicaciones. En la actualidad, estas herramientas se siguen empleando para detectar vulnerabilidades que, en muchos casos, han sido críticas y han afectado a multitud de sistemas como la vulnerabilidad conocido como Heartbleed en OpenSSL. En esta sección se van a realizar pruebas con los *fuzzers* expuestos en la sección 3 con el fin de realizar una comparativa de los resultados obtenidos con cada uno de ellos en la sección 5.

4.1. CVE-2009-0159

Esta vulnerabilidad consiste en un *buffer overflow* en la función cookedprint() en el fichero ntpq/ntpq.c en las versiones de NTP anteriores a la 4.2.4p7-RC2. Explotar con éxito esta vulnerabilidad podría permitir a los servidores NTP remoto ejecutar código a través de una respuesta específica. El código de *Common Vulnerability and Exposures* que se le asignó a esta vulnerabilidad es CVE-2009-0159 [Dat].

La versión escogida para realizar las pruebas es la 4.2.2 y se puede obtener de la web de la Universidad de Delaware [oD]. Al tratarse de una vulnerabilidad de *buffer overflow* en una determinada función, en caso de localizarse el fallo, el programa fallará enviando una señal SIGSEGV que cualquier *fuzzer* puede detectar.

En todo proceso de *fuzzing* de caja blanca, es decir, aquel del que es posible obtener toda la información del sistema que se va a probar, se debe de adaptar el código para hacerlo compatible con el funcionamiento del *fuzzer*. En este caso, dado que se conoce la función que contiene el fallo, el código se puede adaptar para que directamente, desde la función main(), se invoque a la función vulnerable cookedprint() adaptando los parámetros para recibir la entrada del *fuzzer* y emplearla al llamar a la función objetivo.

Una forma simple de controlar la entrada del *fuzzer* consiste en leer de STDIN con read(). Será necesario incluir variables para almacenar los valores que introduzca el *fuzzer* para enviarlos posteriormente a la función vulnerable. La función cookedprint() recibe un puntero a un *buffer* que recibirá la información de STDIN. Este tipo de variables reservan una cantidad de la memoria que contiene información que puede provocar inconsistencias

en el proceso de *fuzzing*. Por ello, es necesario reiniciar la memoria reservada para evitar problemas empleando la función memset(), siendo común asignarle un valor de cero.

El código adaptado para emplear con los *fuzzers* AFL y Honggfuzz sería el siguiente:

```
1  int main(int argc, char *argv[])
2  {
3      //Declaracion de variables
4      int datatype;
5      int length;
6      char data[1024 * 16] = {0};
7      int status;
8      FILE *fp;
9
10     //Asignacion de valores a las variables
11     datatype = 0;
12     length = 0;
13     status = 0;
14
15     //Poner la memoria a ceros
16     memset(data, 0, 1024*16);
17
18     //Leer entrada de stdin (fuzzer)
19     read(0, &datatype, 1);
20     read(0, &status, 1);
21     length = read(0, data, 1024 * 16);
22
23     //Funcion vulnerable
24     cookedprint(datatype, length, data, status, stdout);
25     return 0;
26 }
```

En el caso de Libfuzzer el código se deberá adaptar para establecer una función que el *fuzzer* pueda interpretar. En la sección 4.1.3 se incluye el proceso seguido para adaptar el funcionamiento para el uso de Libfuzzer.

El código adaptado puede emplearse en múltiples *fuzzers* de propósito general, como AFL y Honggfuzz, realizando la instrumentación en tiempo de compilación con las herramientas que ambos incluyen. Ha sido posible descubrir el fallo con AFL y con Honggfuzz obteniendo unos resultados diferentes con cada uno de ellos durante un tiempo de aproximadamente tres horas y empleando únicamente cuatro hilos del procesador.

Analizar el funcionamiento del código cuando esté disponible es un requisito indispensable para optimizar la búsqueda de fallos y ayudar a aumentar la cobertura del código. Una buena forma de llevar esto a cabo es mediante el uso de diccionarios. Para elaborar un diccionario simple que pueda servir para acceder a ciertos bloques condicionales

puede consistir en la selección de cadenas de texto del código. Los resultados empleando un diccionario incrementan considerablemente los resultados obtenidos.

4.1.1. Detección con AFL

Partiendo del código adaptado para realizar pruebas sobre la función en la que se encuentra el fallo, se ha instrumentado ntpq para que AFL pueda realizar las pruebas sobre esta herramienta. Este fallo es posible encontrarlo sin emplear diccionarios pero el tiempo de procesamiento es mayor. Para apreciar las diferencias se han aplicado ambas técnicas sobre el mismo binario instrumentado.

Sin diccionario

Con el código adaptado se ha realizado la instrumentación por software empleando afl-clang:

```
1 CC=afl-clang AFL_HARDEN=1 ./configure && make
```

A la hora de iniciar un *fuzzer* sobre un programa en pruebas es importante disponer de unos buenos casos de prueba. Para localizar este fallo, se puede emplear un fichero simple con una cadena de caracteres "A" que AFL irá transformando hasta localizar el fallo.

Por otro lado, no solo es necesario contar con unos buenos casos de prueba ya que el empleo de un diccionario puede disparar el número de caminos encontrados y que el *fuzzer* podrá explorar. En este caso, el empleo de un diccionario marca una gran diferencia a la hora de encontrar fallos. Ha sido posible encontrar el fallo con el código adaptado con diccionario y sin diccionario.

En primer lugar se ejecutó la prueba empleando paralelización con cuatro procesos. Una vez instrumentado el programa se puede iniciar el proceso de *fuzzing* de la siguiente forma:

```
1 afl-fuzz -i in -o out -M master-ntpq -- ./ntp-4.2.2/ntpq/ntpq
2 afl-fuzz -i in -o out -S slave-ntpq -- ./ntp-4.2.2/ntpq/ntpq
3 afl-fuzz -i in -o out -S slave2-ntpq -- ./ntp-4.2.2/ntpq/ntpq
4 afl-fuzz -i in -o out -S slave3-ntpq -- ./ntp-4.2.2/ntpq/ntpq
```

El tiempo de *fuzzing* se ha fijado en tres horas. Realizando las pruebas sin diccionario un *slave* encontró dos fallos únicos, uno que provoca un fallo de *Segmentation Fault* y un falso positivo que se desechó.

La pantalla de estado del *slave* que detectó el fallo es el siguiente:

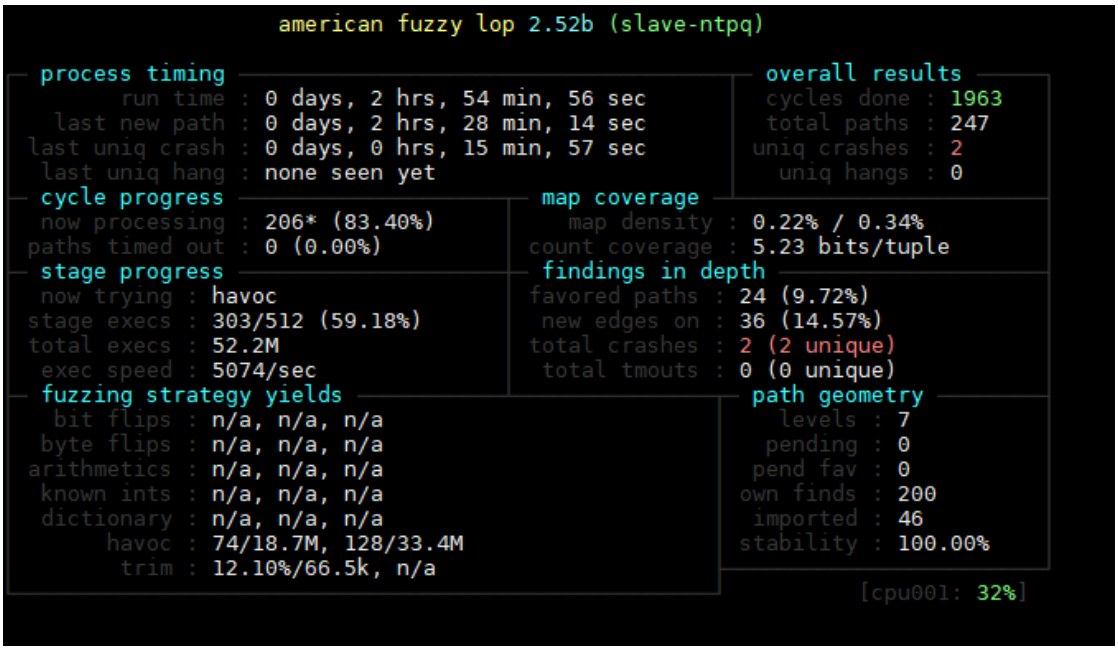


Figura 4.1: Estado de AFL en el slave que detecta el fallo sin diccionario

Como se ha comentado en la sección 3.1.1, AFL generará una estructura de directorios con la información que ha encontrado el *fuzzer*. En este caso, la información de interés se va a encontrar en el directorio crashes/. En ese directorio se encontrarán los ficheros de entrada que ha empleado AFL para conseguir el fallo. Estos ficheros se pueden enviar a la adaptación del programa y comprobar que el fallo es real, observando la señal SIGSEGV:

```
pM-pM-pM-pM-pM-pM-pM-pM-pM-kM-pM-pM-pM-pM-pM-pM-pM-pM-pM-pM-pM-pM-pM-pM-
pM-pM-pM-NM-pM-pM-pM-pM-pM-pM-pM-pM-pM-pM-pM-pM-pM-pM-pM-p,
M-^B=, t=, m^AM-;^]^QT=f=, ^PM-^?M-^?M-^@^AM-"^AM-^@^Xn=^MM-^@m, M-^B
=,
m^RM-""^Uf=fTm^BM-t^AM-"^]mmmmmmXm^FM-t, M-^@M-^?f[ ^FM-t=, t=, K^A=,
M-^@=, mM-^@^AM-^D^P=, M-d^ZA^BM-P^VM-^JM-^J=,
^ZJM-04444m^AM-^?f[m^FM-q^AL^]^Ff=, M-t=~M-ZU^AM-#^]QBM-r, t^B=, =,
M-^B=, t=, M-^P^AM-"^]^Qd=f=, ^VM-^?M-^?M-^@^AM-"M-^@M-^@^Fn=^MM-^@m,
Segmentation fault
fuzz@fuzz:~/sergio/afl-training/challenges/ntpq$ cat out/slave-ntpq/c
rashes/id\:000001\,sig\:11\,src\:000230\,op\:havoc\,rep\:16 | ./ntp-4
.2.2/ntpq/ntpq
```

Figura 4.2: Validación del fallo encontrado por AFL sin diccionario

Con diccionario

Una vez que se ha generado un diccionario, el descubrimiento de caminos se dispara y es posible encontrar el fallo en segundos. Para emplear un diccionario con AFL se

emplea la opción -x. La instrumentación se realiza de la misma forma que en el caso sin diccionario y se puede emplear el mismo *corpus* simple empleado anteriormente:

```
1 afl-fuzz -i in -o out -x ntpq.dict -- ./ntp-4.2.2/ntpq/ntpq
```

De esta forma, en dos minutos ha sido posible encontrar 30000 fallos de los cuales 46 son únicos. En la imagen 4.3 se muestra el estado de AFL cuando se canceló el proceso de *fuzzing*.

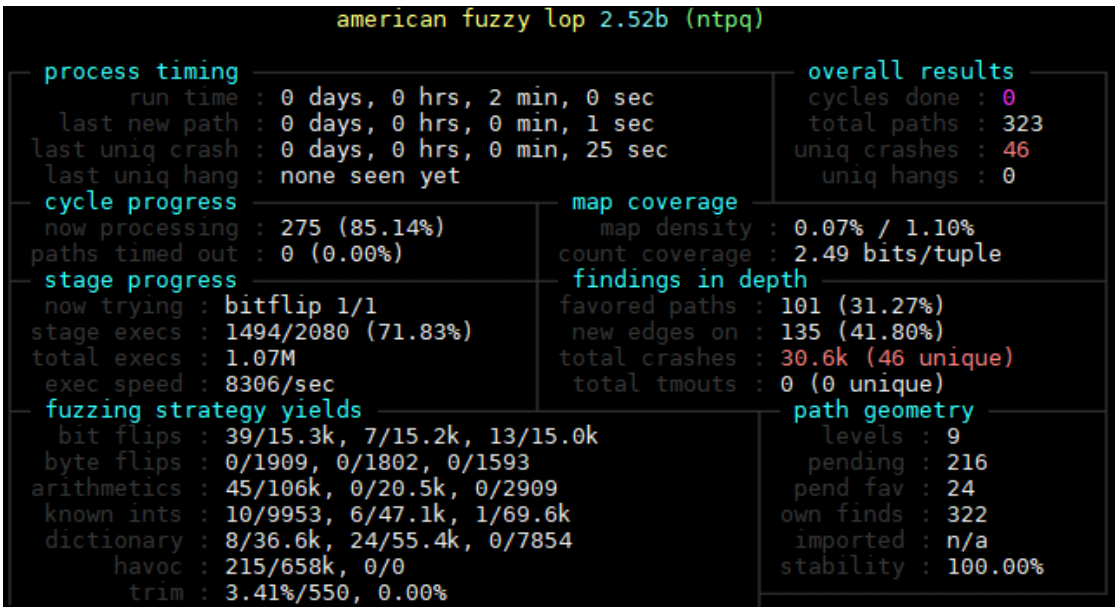


Figura 4.3: Estado de AFL detectando múltiples fallos con diccionario

De igual manera que se ha hecho antes, una vez obtenidos los fallos es necesario validarlos. Por ello, se ha elegido uno de los fallos y se ha enviado por STDIN al programa objeto de pruebas. A continuación se muestra una imagen recibiendo la señal SIGSEGV con uno de los fallos obtenidos:

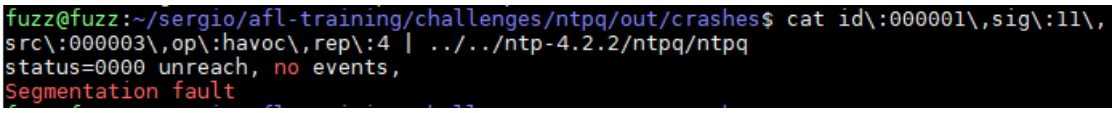


Figura 4.4: Validación del fallo encontrado por AFL con diccionario

El empleo de un diccionario simple con cadenas de texto del programa en pruebas ha reducido considerablemente el tiempo que requiere el AFL para detectar el fallo además de haber encontrado mayor número de fallos únicos.

4.1.2. Detección con Honggfuzz

De igual manera que en el apartado anterior se van a realizar las pruebas con Honggfuzz. Honggfuzz permite también el empleo de diccionarios y va a ser capaz de detectar los fallos igual que AFL en un tiempo similar. Por ello, lo interesante va a ser la comparación en el caso de que no se emplee un diccionario.

Sin diccionario

El tiempo de *fuzzing* se ha establecido en tres horas y se ha empleado el mismo código aunque se ha instrumentado con las opciones que requiere Honggfuzz, incluidas en la herramienta hfuzz-clang. Dado que el código ya se encuentra adaptado para realizas las pruebas sobre la función cookedprint(), hay que realizar la instrumentación del código de la siguiente forma:

```
1 CC=hfuzz-clang ./configure && make
```

Una vez instrumentado, se debe lanzar Honggfuzz con los mismos procesos para que las pruebas se apliquen con las mismas condiciones (opción -n) y como la entrada se recibe de STDIN, se le indica al *fuzzer* con la opción -s. Ya esta posible iniciar las pruebas sin incluir un diccionario:

```
1 honggfuzz -f in/ -n 4 -s -W out/ -- ./ntp-4.2.2/ntpq/ntpq
```

En esta prueba, los resultados obtenidos muestran que este *fuzzer* ha sido capaz de detectar un fallo único en un tiempo mucho inferior al tiempo establecido de las pruebas sin emplear diccionario. El estado de la pantalla de Honggfuzz muestra un gran número de fallos de los cuales uno es único:

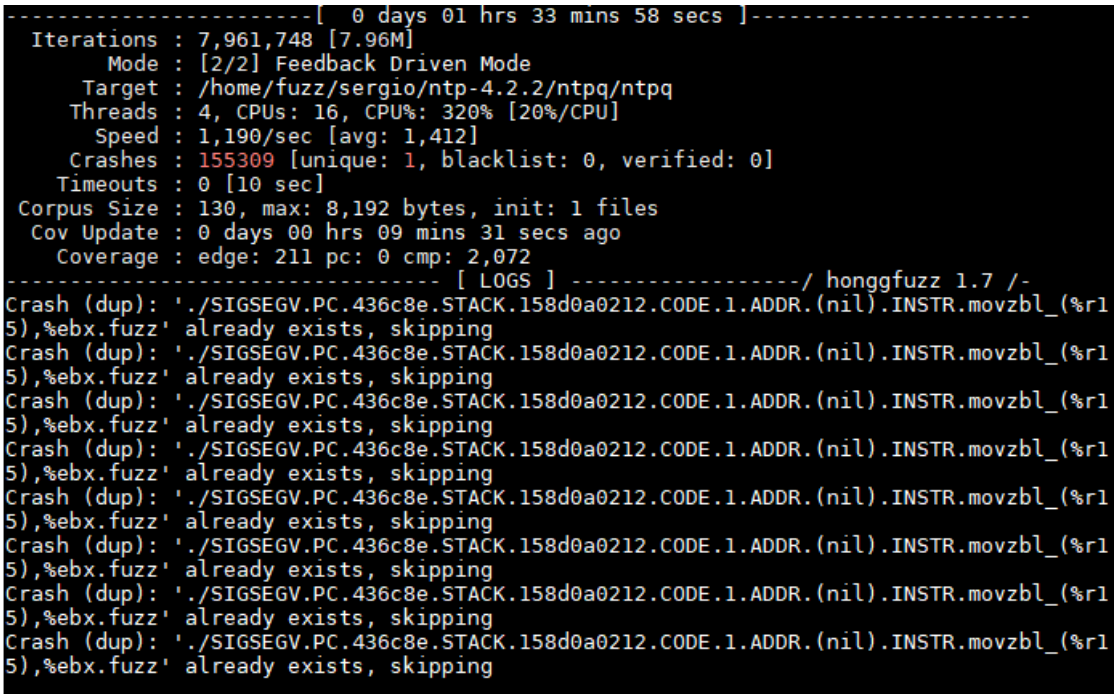


Figura 4.5: Estado de Honggfuzz cuando se detecta el fallo sin diccionario

Una vez obtenido el fallo es necesario validarlo. Por ello, se ha enviado por STDIN al programa objeto de pruebas obteniendo la señal SIGSEGV que indica su validez:

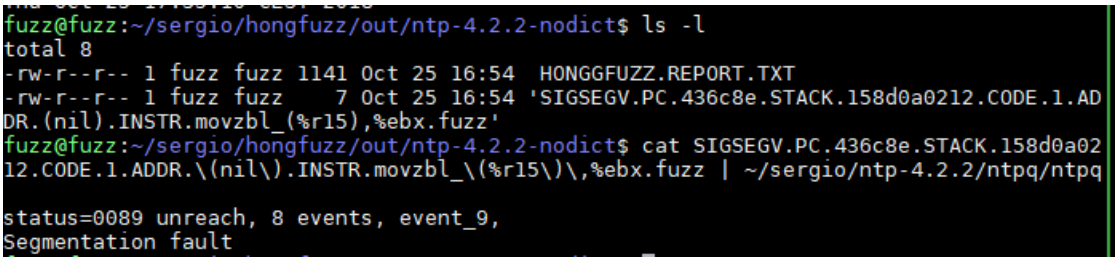


Figura 4.6: Validación del fallo encontrado por Honggfuzz sin diccionario

Honggfuzz almacena en ficheros que han provocado los fallos en el directorio que se especifica con la opción -W. En caso de que no se haya especificado esta opción, la ruta que emplea el *fuzzer* es el directorio actual. Como se ha comentado en la sección 3.2.1, Honggfuzz incluye un fichero de *log* con la información del proceso.

Con diccionario

El empleo del diccionario elaborado en la fase de preparación y que también se ha empleado en AFL para encontrar un gran número de fallos únicos ha servido para replicar el proceso con Honggfuzz. Empleando el mismo binario instrumentado de ntpq

y asignando también un tiempo de dos minutos, se ha iniciado el proceso llamando al *fuzzer* de la siguiente forma:

```
honggfuzz -f in/ -w ntpq.dict -n 1 -s -W out/ -- ./ntp-4.2.2/ntpq/ntpq
```

De esta forma, los fallos encontrados se disparan considerablemente. En tan solo dos minutos se han producido casi 2000 fallos y un total de 193 únicos. Al finalizar el tiempo establecido, la pantalla muestra la siguiente información:

```
-----------------------[  0 days 00 hrs 02 mins 00 secs ]----------------------
  Iterations : 59,743 [59.74k]
        Mode : [2/2] Feedback Driven Mode
      Target : /home/fuzz/sergio/ntp-4.2.2/ntpq/ntpq
     Threads : 1, CPUs: 16, CPU%: 100% [6%/CPU]
       Speed : 475/sec [avg: 497]
     Crashes : 1873 [unique: 193, blacklist: 0, verified: 0]
    Timeouts : 0 [10 sec]
 Corpus Size : 133, max: 8,192 bytes, init: 198 files
  Cov Update : 0 days 00 hrs 00 mins 04 secs ago
    Coverage : edge: 545 pc: 0 cmp: 5,429
--------------------------------- [ LOGS ] ------------------/ honggfuzz 1.7 /-
Crash (dup): './SIGSEGV.PC.4450d0.STACK.15cf95efdd.CODE.1.ADDR.(nil).INSTR.movzbl_(%r$
2),%r15d.fuzz' already exists, skipping
Crash (dup): './SIGSEGV.PC.436c8e.STACK.144ddc556c.CODE.1.ADDR.(nil).INSTR.movzbl_(%r$
5),%ebx.fuzz' already exists, skipping
Crash (dup): './SIGSEGV.PC.43a011.STACK.14925b9e44.CODE.1.ADDR.0xfffffffffffff00.INS$
R.movzwl_(%rax,%r13,2),%r14d.fuzz' already exists, skipping
Crash (dup): './SIGSEGV.PC.436c8e.STACK.144ddc556c.CODE.1.ADDR.(nil).INSTR.movzbl_(%r$
5),%ebx.fuzz' already exists, skipping
Crash (dup): './SIGSEGV.PC.4450d0.STACK.140c120d7f.CODE.1.ADDR.(nil).INSTR.movzbl_(%r$
2),%r15d.fuzz' already exists, skipping
Crash (dup): './SIGSEGV.PC.435445.STACK.1508d2719f.CODE.1.ADDR.(nil).INSTR.movzbl_(%r$
5),%ebx.fuzz' already exists, skipping
^CSignal 2 (Interrupt) received, terminating
Terminating thread no. #0
All threads done
Summary iterations:59793 time:120 speed:498
```

Figura 4.7: Estado de Honggfuzz detectando múltiples fallos con diccionario

Se ha realizado la misma validación que anteriormente enviando el contenido de uno de los ficheros generados por el *fuzzer* a través de STDIN al programa vulnerable verificando que es un fallo válido:

```
fuzz@fuzz:~/sergio/hongfuzz/out/ntp-4.2.2$ cat 'SIGSEGV.PC.445180.STACK.badbad0cbab42c
93.CODE.1.ADDR.0x7fffffff000.INSTR.mov____%r13b,-0x1(%r15).2018-10-25.15:19:53.3087.f
uzz' | ~/sergio/ntp-4.2.2/ntpq/ntpq
status=00be unreach, 11 events, event_14,
Segmentation fault
```

Figura 4.8: Validación del fallo encontrado por Honggfuzz con diccionario

Como se puede apreciar, el empleo del diccionario generado anteriormente ha permitido detectar un gran número de fallos únicos en un tiempo muy reducido. El indicador de los fallos únicos no es correcto ya que el *fuzzer* solo es capaz de detectar el fallo buscado, por lo que no sería único.

4.1.3. Detección con LibFuzzer

Libfuzzer requiere de un trabajo adicional para hacer viable la ejecución de las pruebas de *fuzzing*. El funcionamiento de este *fuzzer* es similar al modo persistente de AFL y Honggfuzz, es decir, se debe implementar código para realizar las pruebas sobre la sección concreta del código del programa objetivo de forma que sea interpretado e instrumentado por el *fuzzer* en tiempo de compilación. A pesar de requerir un trabajo adicional, las pruebas estarán mucho más optimizadas.

En este caso particular, dado que se conoce la función que provoca el fallo, se debe implementar un código que invoque a la función cookedprint. Disponiendo del código es posible realizar multitud de modificaciones pero para que el tiempo dedicado a implementar el código sea lo mínimo posible, es posible cargar el binario generado en la memoria del proceso con un enlace dinámico sin tener que modificar el código de esta versión de NTP y depender de generar un nuevo programa que acceda a la función cookedprint. Para generar este código, se ha empleado la herramienta LIEF.

LIEF

LIEF [Qua], *Library to Instrument Executable Formats*, es una herramienta que permite analizar y modificar binarios en formato ELF, PE, MachO, y formatos Android como DEX o ART, entre otros. En muchos casos estos formatos comparten características como las secciones, los símbolos o los puntos de entrada, por lo que LIEF puede trabajar sobre ellos e incluye soporte en C, C++ y Python.

La arquitectura de este proyecto es la siguiente:

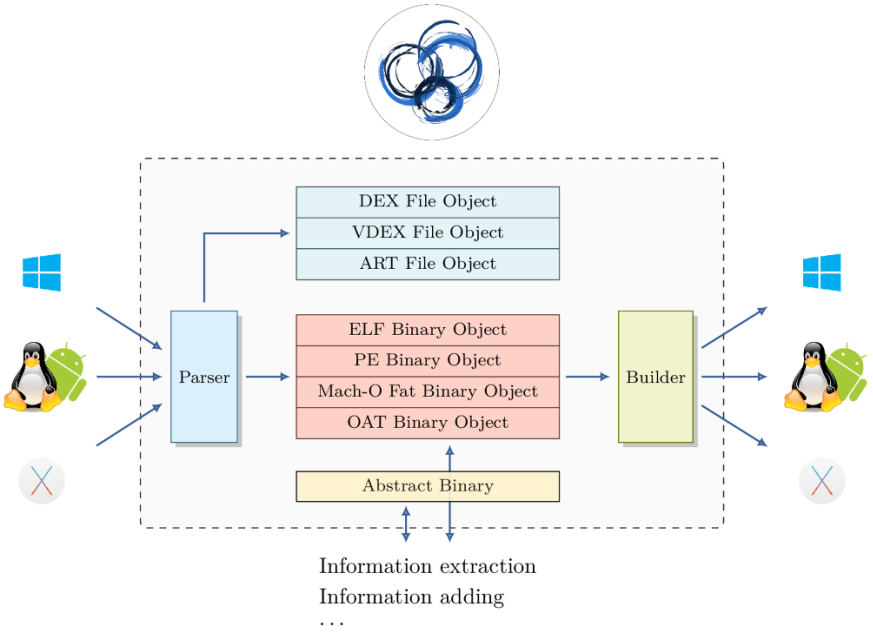


Figura 4.9: Arquitectura de LIEF

La API que ofrece es simple de emplear y es posible modificar un binario para exportar una función que anteriormente no se encontraba disponible fuera del propio programa. Por ello, LIEF va a permitir exportar la función cookedprint de una versión compilada de ntpq para que esté accesible desde programas externos, como la implementación del programa creado para las pruebas de *fuzzing* con Libfuzzer.

Para llevar esta labor a cabo es necesario conocer el punto de inicio de la función cookedprint, esto se puede realizar con la herramienta readelf de la siguiente forma:

```
fuzz@fuzz:~/sergio/libfuzzer/ntp-4.2.2/ntpq$ readelf -a ./ntpq | grep cookedprint
   852: 000000000012a210 14418 FUNC    LOCAL  DEFAULT   14 cookedprint
fuzz@fuzz:~/sergio/libfuzzer/ntp-4.2.2/ntpq$
```

Figura 4.10: Dirección de la función cookedprint en el binario ntpq

Una vez conocida la dirección de la función de interés ya es posible emplear la herramienta LIEF para exportar la función cookedprint en una nueva modificación del binario. Para ello, en primer lugar se debe emplear el *parser* de LIEF sobre el binario de interés. Una vez importado el binario, se debe emplear la dirección de la función encontrado en la figura 4.10 para exportar la función cookedprint. Finalmente, se crea un nuevo fichero ejecutable con la función cookedprint exportada. El código Python implementado para realizar lo descrito anteriormente es el siguiente:

```
1  import lief
2
3  binary = lief.parse("ntpq")
4  binary.add_exported_function(0x12a210, "cookedprint")
5  binary.write("ntpq.modified")
```

De esta forma, el binario ntpq.modified dispone de la función cookedprint exportada lo que va a permitir acceder a ella creando un enlace dinámico de este binario en la memoria de un programa implementado para detectar la vulnerabilidad con Libfuzzer.

Implementación del código

Disponiendo de una modificación del binario ntpq con la función de interés accesible por otro programa externo, ha sido posible implementar un código que añada esta modificación con un enlace dinámico en la memoria del programa desarrollado y sobre el que se van a recibir los casos de prueba generados por el *fuzzer* que se enviarán a la función vulnerable. En el anexo E se incluye el código implementado.

A la hora de implementar un objetivo de *fuzzing* para Libfuzzer se debe tener en cuenta como va a funcionar este *fuzzer* y considerar como se realiza el acceso a los distintos recursos del programa para evitar fallos que se puedan producir durante el funcionamiento del *fuzzer*.

El funcionamiento de Libfuzzer es similar al modo persistente de otros *fuzzers* como AFL o Honggfuzz, Libfuzzer accederá a una función de forma iterativa optimizando así su funcionamiento. Este acceso iterativo afecta directamente en el uso de la memoria

por lo que se debe de gestionar correctamente para que el *fuzzer* no finalice en caso de sobrepasar un límite establecido. Es importante considerar el empleo de variables globales para evitar realizar tareas de forma repetitiva sin ser necesario.

Para cargar el binario modificado con LIEF se puede emplear la interfaz de carga de enlaces dinámicos. Esta interfaz permite cargar un binario en la memoria del programa con *dlopen* y asignarle una función de salida al finalizar la invocación de una determinada función. En este caso, cuando se termine la llamada a cookedprint, se procederá a eliminar el enlace dinámico con *dlclose*. En caso de cargar un binario en memoria es de vital importancia eliminarlo con *dlclose* para evitar problemas en la gestión de la memoria.

Una vez realizado el enlace dinámico y siendo posible acceder a la función de interés, se debe controlar los casos de prueba que genere el *fuzzer*. En el caso de Libfuzzer, los datos se envían en un puntero junto con el tamaño del contenido al que apunta dicho puntero. Ese contenido generado por el *fuzzer* no tiene porque finalizar en un byte nulo por lo que es necesario copiar este contenido en una nueva sección de memoria (reservada con *malloc*). Cuando ya no es necesario emplear esta sección reservada, se debe liberar con *free*.

La función cookedprint se debe definir en el propio programa para que esté accesible y sea posible emplear la interfaz de creación del enlace dinámico *dlsym*.

Detección del fallo

Una vez que se ha implementado el programa encargado de crear el enlace dinámico y hacer uso de la función de interés, se debe compilar con los parámetros que requiere el *fuzzer*. Para compilar el proyecto de NTP se deben incluir los parámetros -fPIC en CFLAGS y -pie en LDFLAGS para establecer el uso de Código con Posición Independiente (Position Independt Code). El establecimiento de estos parámetros se puede realizar tanto en el propio fichero Makefile como incluirlos como variables de entorno en el proceso de compilación al ejecutar el comando make.

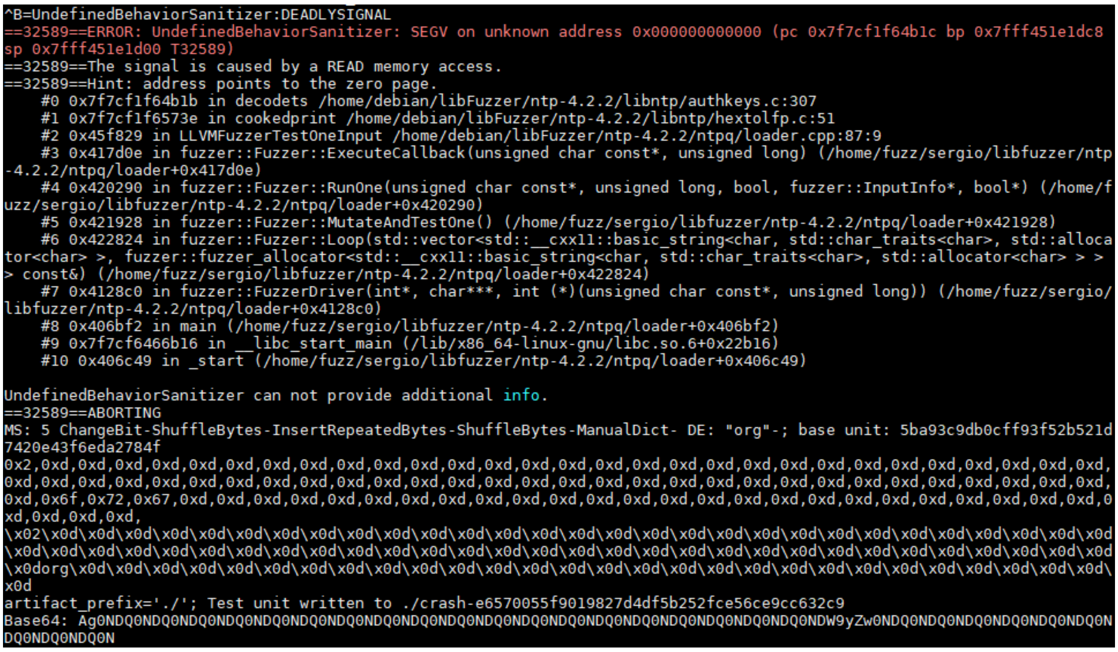
Una vez compilado se ha obtenido el binario modificado con la función cookedprint empleando LIEF y se ha generado el correspondiente *fuzz-target* para Libfuzzer. La compilación del programa implementado para realizar el *fuzzing* sobre la función cookedprint se ha compilado de la siguiente forma:

```
1 clang++ -DUSE_LIBFUZZER -fsanitize=fuzzer -g -O1 -ldl -no-pie
  ↪ loader.cpp -o loader
```

Una vez compilado y realizada la instrumentación sobre el programa, ha sido posible iniciar el *fuzzer* con diccionario y un caso de prueba inicial simple, idéntico al empleado con AFL y Honggfuzz. Para iniciar el *fuzz-target* implementado se ha ejecutado el siguiente comando:

```
1 ./loader testcases/ -dict=dict.ntpq
```

Libfuzzer es capaz de detectar en cuestión de segundos el fallo y que el proceso lance la señal de SIGSEGV:



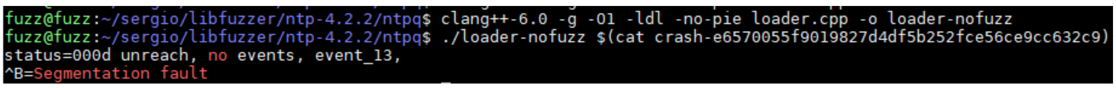
```
^B=UndefinedBehaviorSanitizer:DEADLYSIGNAL
==32589==ERROR: UndefinedBehaviorSanitizer: SEGV on unknown address 0x000000000000 (pc 0x7f7cf1f64b1c bp 0x7fff451e1dc8
sp 0x7fff451e1d00 T32589)
==32589==The signal is caused by a READ memory access.
==32589==Hint: address points to the zero page.
    #0 0x7f7cf1f64b1b in decodets /home/debian/libFuzzer/ntp-4.2.2/libntp/authkeys.c:307
    #1 0x7f7cf1f6573e in cookedprint /home/debian/libFuzzer/ntp-4.2.2/libntp/hextolfp.c:51
    #2 0x45f829 in LLVMFuzzerTestOneInput /home/debian/libFuzzer/ntp-4.2.2/ntpq/loader.cpp:87:9
    #3 0x417d0e in fuzzer::Fuzzer::ExecuteCallback(unsigned char const*, unsigned long) (/home/fuzz/sergio/libfuzzer/ntp
-4.2.2/ntpq/loader+0x417d0e)
    #4 0x420290 in fuzzer::Fuzzer::RunOne(unsigned char const*, unsigned long, bool, fuzzer::InputInfo*, bool*) (/home/f
uzz/sergio/libfuzzer/ntp-4.2.2/ntpq/loader+0x420290)
    #5 0x421928 in fuzzer::Fuzzer::MutateAndTestOne() (/home/fuzz/sergio/libfuzzer/ntp-4.2.2/ntpq/loader+0x421928)
    #6 0x422824 in fuzzer::Fuzzer::Loop(std::vector<std::__cxx11::basic_string<char, std::char_traits<char>, std::alloca
tor<char> >, fuzzer::fuzzer_allocator<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > >
> const&) (/home/fuzz/sergio/libfuzzer/ntp-4.2.2/ntpq/loader+0x422824)
    #7 0x4128c0 in fuzzer::FuzzerDriver(int*, char***, int (*)(unsigned char const*, unsigned long)) (/home/fuzz/sergio/
libfuzzer/ntp-4.2.2/ntpq/loader+0x4128c0)
    #8 0x406bf2 in main (/home/fuzz/sergio/libfuzzer/ntp-4.2.2/ntpq/loader+0x406bf2)
    #9 0x7f7cf6466b16 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x22b16)
    #10 0x406c49 in _start (/home/fuzz/sergio/libfuzzer/ntp-4.2.2/ntpq/loader+0x406c49)

UndefinedBehaviorSanitizer can not provide additional info.
==32589==ABORTING
MS: 5 ChangeBit-ShuffleBytes-InsertRepeatedBytes-ShuffleBytes-ManualDict- DE: "org"-; base unit: 5ba93c9db0cff93f52b521d
7420e43f6eda2784f
0x2,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,
0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,
0xd,0x6f,0x72,0x67,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0xd,0
xd,0xd,0xd,0xd,
\x02\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d
\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d
\x0dorg\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\x0d\
x0d
artifact_prefix='./'; Test unit written to ./crash-e6570055f9019827d4df5b252fce56ce9cc632c9
Base64: Ag0NDQ0NDQ0NDQ0NDQ0NDQ0NDQ0NDQ0NDQ0NDQ0NDQ0NDQ0NDQ0NDQ0NDQ0NDQ0NDQ0NDQ0NDQ0NDW9yZw0NDQ0NDQ0NDQ0NDQ0NDQ0NDQ0N
DQ0NDQ0NDQ0N
```

Figura 4.11: Detección del fallo con Libfuzzer con diccionario

Tal como aparece en la imagen anterior, el *fuzzer* incluye la entrada generada que provoca el fallo en un fichero dentro del directorio desde el que se inicia el proceso de *fuzzing*. Para comprobar si realmente el fallo es válido y aprovechando las condiciones de compilación incluido en el programa implementado, es posible compilar el programa y ejecutarlo sin necesidad de iniciar el *fuzzer*. Para compilarlo de esta forma se ha ejecutado el siguiente comando:

```
clang++ -g -O1 -ldl -no-pie loader.cpp -o loader-nofuzz
```

Para comprobar si el fichero detectado provoca el fallo buscado y teniendo en cuenta que recibe el contenido por STDIN, para reproducir el fallo sin el *fuzzer* se puede ejecutar el programa de la siguiente forma:



```
fuzz@fuzz:~/sergio/libfuzzer/ntp-4.2.2/ntpq$ clang++-6.0 -g -O1 -ldl -no-pie loader.cpp -o loader-nofuzz
fuzz@fuzz:~/sergio/libfuzzer/ntp-4.2.2/ntpq$ ./loader-nofuzz $(cat crash-e6570055f9019827d4df5b252fce56ce9cc632c9)
status=000d unreach, no events, event_13,
^B=Segmentation fault
```

Figura 4.12: Validación del fallo encontrado por Libfuzer

4.2. Rode0day

Rode0day [AFDG] es un proyecto de colaboración entre el Instituto de Tecnología de Massachusetts (MIT), el laboratorio MIT Lincoln y la Universidad de Nueva York

(NYU) que trata de promover la comprensión de los fallos en el *software*. Este proyecto trata de que se conozcan las fortalezas y debilidades de las técnicas de búsqueda de fallos mediante evaluaciones concretas. También trata de que la evaluación sea barata y frecuente, garantizando que los resultados son relevantes y novedosos.

La metodología del proyecto se basa en la publicación mensual de programas en los que se han inyectado vulnerabilidades empleando la tecnología LAVA [BDG] que relaciona una entrada con el fallo que provoca. LAVA emplea un sistema preciso y completo capaz de localizar los datos controlados por el usuario en un programa y los emplea para inyectar vulnerabilidades así como otras modificaciones benignas. La calidad del resultado es altamente notable y útil para validar el grado de detección de las herramientas de descubrimiento de vulnerabilidades.

Aprovechando las muestras publicadas, se han realizado pruebas de caja negra y de caja blanca empleando American Fuzzy Lop, Honggfuzz y Libfuzzer. Dado que algunas pruebas no disponen de código, no será posible emplear Libfuzzer ya que es un *fuzzer* orientado a la instrumentación en tiempo de compilación.

Rode0day plantea su proyecto como una competición con el objetivo de realizar proyectos de *fuzzing* que se prolonguen en el tiempo. Multitud de investigadores han empleado este proyecto para generar su propios *fuzzers*, modificar *fuzzers* de propósito general como AFL y tratar de comparar los resultados obtenidos.

Una vez conseguido un fallo, se debe enviar empleando la API que ofrece. Para automatizar la subida de fallos se ha creado un *script* en Python que puede recibir un fichero único o un directorio y mostrará por pantalla que fallos han sumado puntos. El código está disponible en el apéndice F.

Las secciones siguientes muestran la detección de fallos sobre un motor de Javascript centrado en la portabilidad en el que han inyectado fallos empleando la tecnología LAVA y del que solo se dispone del binario compilado por lo que las pruebas serán de caja negra. Por otro lado, se ha realizado *fuzzing* sobre una modificación del comando *file* de los sistemas Unix, empleado para detectar el formato de un determinado fichero. En este caso si que se dispone del código por lo que será posible instrumentar el código en tiempo de compilación y recibir mayor información del *fuzzer*.

4.2.1. Duktape

Duktape [Vaa] es un motor de Javascript compacto, de fácil integración en proyectos C/C++, portable y que se puede incluir en cualquier plataforma ya que no asume capacidades de los sistemas operativos. Este motor recibe un fichero de entrada que es interpretado por el motor Javascript. De esta forma, se pueden emplear un *fuzzer* para localizar posibles fallos en la gestión y manejo de estos ficheros de entrada.

Dado que solo se dispone del binario no será posible emplear Libfuzzer ya que requiere del código para poder instrumentar y aplicar el *fuzzing*. Por ello, las pruebas se han centrado en los modos *dumb* o *fuzzing* "tonto" como se empleaba en los *fuzzers* de la vieja escuela. Este tipo de *fuzzing* se van a enviar las entradas generadas al binario en pruebas pero no va a ser posible analizar y mejorar la cobertura del código.

A pesar de no poder emplear la totalidad de la funcionalidad de los *fuzzers* modernos si que va a ser posible emplear diccionarios y ayudar al *fuzzer* en la generación de las entradas obteniendo mejores resultados. A continuación se comentan los resultados obtenidos con diccionario y sin él tanto para AFL como para Honggfuzz. El tiempo de *fuzzing* se ha fijado en seis horas que, a pesar de ser un tiempo muy reducido, permite comparar los resultados obtenidos por ambos *fuzzers*. Para aprovechar parte de la potencia del procesador se ha decidido iniciar los *fuzzers* con cuatro hilos cada uno.

AFL

AFL puede emplearse para realizar *fuzzing* sobre binarios compilados y que no es posible instrumentar. De esta forma se va a emplear el generador de entradas y el mecanismo de entrega pero el *fuzzer* no va a poder emplear todo su potencial. Para iniciar AFL en modo *dumb* se puede emplear la opción -n.

En los primeros ciclos de un proceso de *fuzzing*, AFL trata de localizar el mayor número de caminos posibles. En este caso, debido a la poca información disponible del binario, el *fuzzer* solo puede encontrar dos caminos e iterar de forma infinita encontrando el mismo fallo. Esto se debe a que es posible que en el sistema en pruebas se evalúe una condición compleja que AFL no es capaz de sobrepasar y, al estar empleándose el modo *dumb*, se consideren los fallos únicos aunque no lo sean ya que AFL no puede distinguir unos de otros.

El problema anterior va a ocurrir si se emplea un diccionario o sin él ya que no va a ser posible pasar la condición que bloquea el proceso. A pesar de ello se ha generado un diccionario simple con palabras clave de Javascript y, por otro lado, un diccionario con las cadenas de texto del binario ya que es posible que estas cadenas puedan sobrepasar condiciones complejas que el *fuzzer* no va a ser capaz de generar él mismo. La táctica de emplear las cadenas de texto imprimibles de un binario es muy común ya que aporta ayuda al proceso de *fuzzing* de forma rápida y mejorando los resultados considerablemente.

Para lanzar AFL empleando cuatro hilos del procesador empleando el modo *dumb* se deben iniciar cada proceso de forma manual y no es posible emplear las opciones master-slave (-M y -S) ya que no son compatibles con el modo *dumb*. Por ello, para iniciar cuatro instancias de AFL, se pueden ejecutar los siguientes comandos:

```
1 afl-fuzz -n -x dict.txt -i inputs/ -o out/afl1/ -- ./bin/duk @@
2 afl-fuzz -n -x dict.txt -i inputs/ -o out/afl2/ -- ./bin/duk @@
3 afl-fuzz -n -x dict.txt -i inputs/ -o out/afl3/ -- ./bin/duk @@
4 afl-fuzz -n -x dict.txt -i inputs/ -o out/afl4/ -- ./bin/duk @@
```

Se puede observar el empleo del diccionario generado y que cada instancia cuenta con su propio directorio de salida. En el caso de iniciar AFL con master-slave se podría indicar el mismo directorio ya que el *fuzzer* se encargaría de la estructura del directorio de salida. El empleo de los caracteres "@@" se emplea para indicar que AFL no recibe el fichero de entrada por STDIN sino que debe acceder y leer el fichero correspondiente de la ruta especificada en la opción -i.

Con diccionario, el resultado obtenido tras seis horas de *fuzzing* en uno de los procesos iniciados es el siguiente:

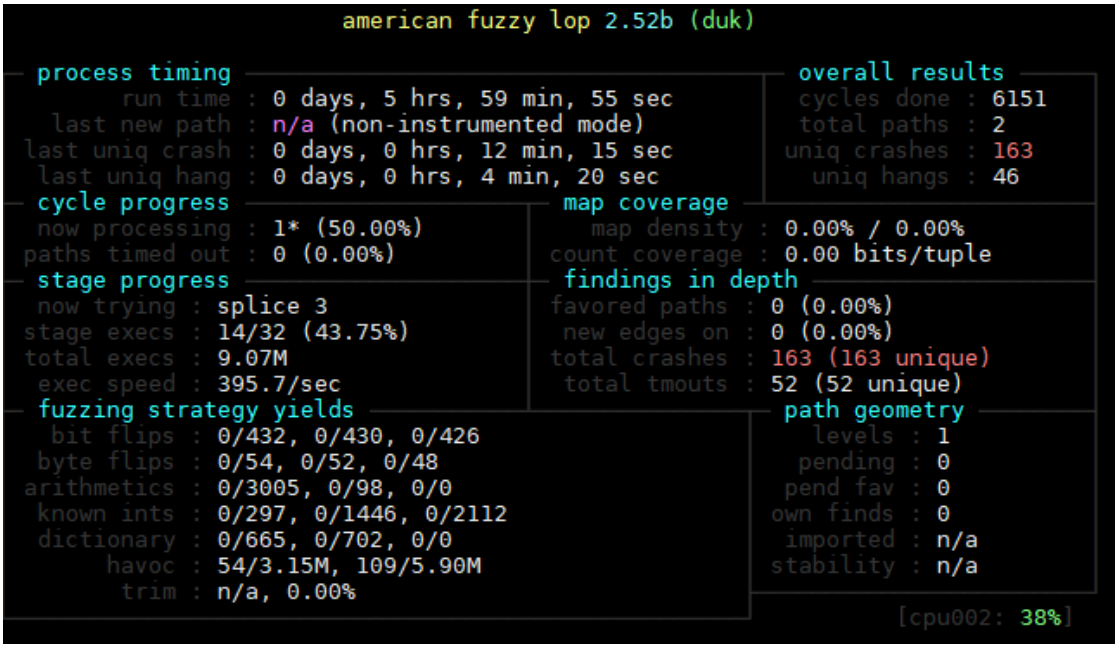


Figura 4.13: Estado de AFL al finalizar las pruebas en Duktape con diccionario

Para iniciar AFL sin emplear diccionario se realiza lo mismo que en el paso anterior pero sin emplear la opción -x:

```
1 afl-fuzz -n -i inputs/ -o out/afl1/ -- ./bin/duk @@
2 afl-fuzz -n -i inputs/ -o out/afl2/ -- ./bin/duk @@
3 afl-fuzz -n -i inputs/ -o out/afl3/ -- ./bin/duk @@
4 afl-fuzz -n -i inputs/ -o out/afl4/ -- ./bin/duk @@
```

El resultado obtenido tras aplicar el mismo proceso sin diccionario durante seis horas es el siguiente:

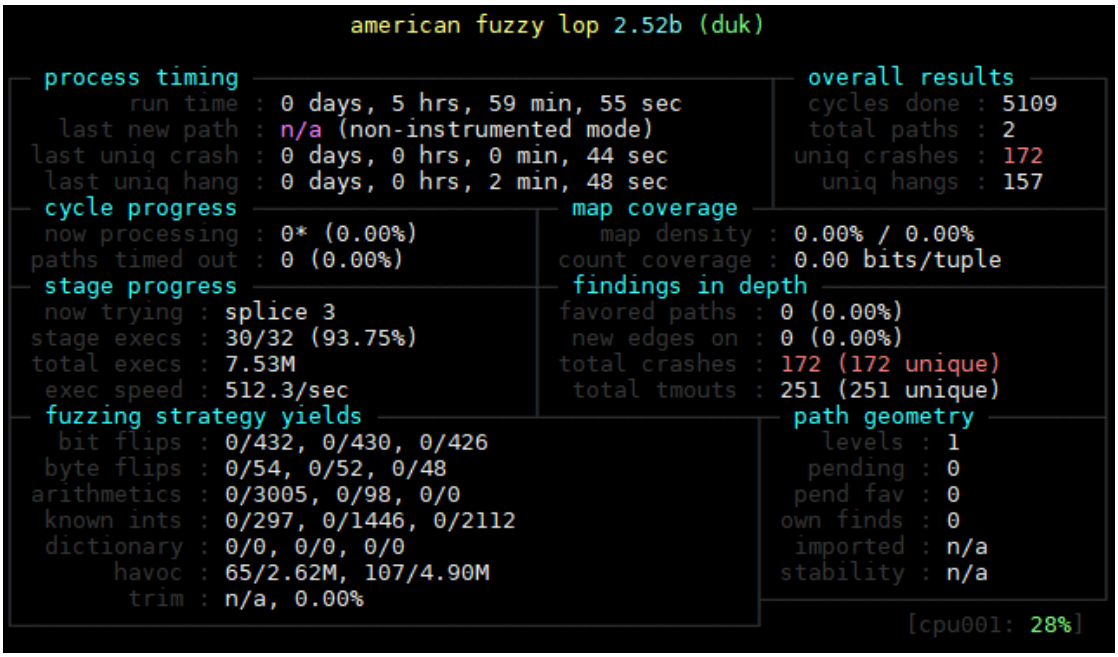


Figura 4.14: Estado de AFL al finalizar las pruebas en Duktape sin diccionario

Se puede observar que si se emplea un diccionario, a pesar de realizar las pruebas durante el mismo periodo, el número de ejecuciones por segundo es menor lo que influye directamente en los fallos que detecta. En este caso, la diferencia de ejecuciones y fallos no afecta al resultado final ya que el fallo detectado es el mismo en todos los casos.

Como se menciona en la sección 3.1.5, AFL incorpora un parche para QEMU que permite mejorar la calidad del *fuzzing* en aquellos binarios de los que no se dispone del código para realizar la instrumentación en tiempo de compilación. De esta forma, las pruebas realizadas sobre el binario de Duktape mejoran considerablemente y AFL es capaz de encontrar nuevos caminos y, por lo tanto, nuevos fallos únicos.

Para indicar a AFL que emplee QEMU se emplea la opción -Q, disponible si se ha instalado y aplicado el parche para que AFL pueda optimizar su uso. En este caso, a diferencia del modo *dumb* en el que no se pueden emplear las opciones de master-slave, en el modo QEMU si que se pueden emplear la organización de maestro y esclavo. Para iniciar AFL con QEMU con cuatro hilos se han ejecutado estos comandos:

```
1 afl-fuzz -Q -i inputs/ -o out/qemu/ -x dict.txt -M master -- ./bin/duk
  ↪ @@
2 afl-fuzz -Q -i inputs/ -o out/qemu/ -x dict.txt -S slave1 -- ./bin/duk
  ↪ @@
3 afl-fuzz -Q -i inputs/ -o out/qemu/ -x dict.txt -S slave2 -- ./bin/duk
  ↪ @@
4 afl-fuzz -Q -i inputs/ -o out/qemu/ -x dict.txt -S slave3 -- ./bin/duk
  ↪ @@
```

El tiempo de pruebas se ha fijado también en seis horas para realizar la comparación con los casos en modo *dumb* pero, en la figura 4.15, se puede observar que es un tiempo insuficiente para que se hayan realizado pruebas que se puedan considerar concluyentes.

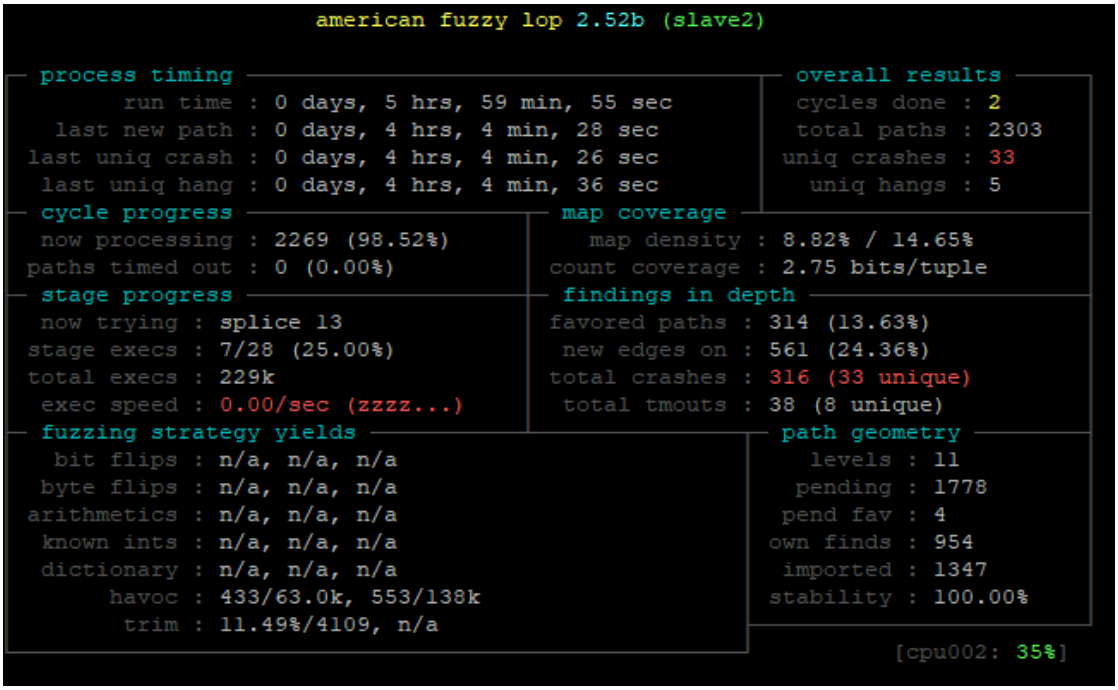


Figura 4.15: Estado de AFL al finalizar las pruebas en Duktape en modo QEMU

Un aspecto importante al emplear QEMU es la velocidad de las pruebas. En la figura anterior se puede observar que el número de ejecuciones por segundo es muy reducido debido al impacto que supone emplear la emulación con QEMU pero los fallos detectados son de mayor calidad que los obtenidos en el modo *dumb*.

Los resultados obtenidos a lo largo de este apartado se analizan en la sección 5.2.1.

Honggfuzz

Honggfuzz también puede emplearse para realizar procesos de *fuzzing* sobre binarios compilados que no es posible instrumentar. El modo se conoce como *static* y se diferencia del modo *feedback-driven fuzzing* en que únicamente se va a emplear el generador de entradas y el mecanismo de entrega. El modo estático de Honggfuzz es equivalente al modo *dumb* de AFL.

La información que muestra este *fuzzer* en tiempo real es reducida y no va a ser necesario iniciar varias instancias ya que Honggfuzz, con la opción -n, permite iniciar el proceso con múltiples hilos en el mismo comando. El tiempo del proceso se ha establecido en seis horas y se han realizado las pruebas con diccionario y sin diccionario. El diccionario empleado es idéntico que en las pruebas realizadas con AFL.

Para iniciar Honggfuzz en modo estático con cuatro hilos y sin diccionario se debe ejecutar el siguiente comando:

```
honggfuzz -n 4 -x -f inputs/ -W output/ -- ./bin/duk ___FILE___
```

Sin diccionario, Honggfuzz es capaz de encontrar un fallo único. De igual manera que ocurre con AFL, al no emplear un diccionario, el número de ejecuciones por segundo incrementa siendo posible obtener mayor cantidad de fallos. Al emplearse el modo estático el *fuzzer* no diferencia los fallos por lo que los considera todos únicos y disponer de un gran número de fallos no supone que todos sean interesantes. En la figura 4.16 se muestra el estado del *fuzzer* tras seis horas de pruebas sin diccionario:

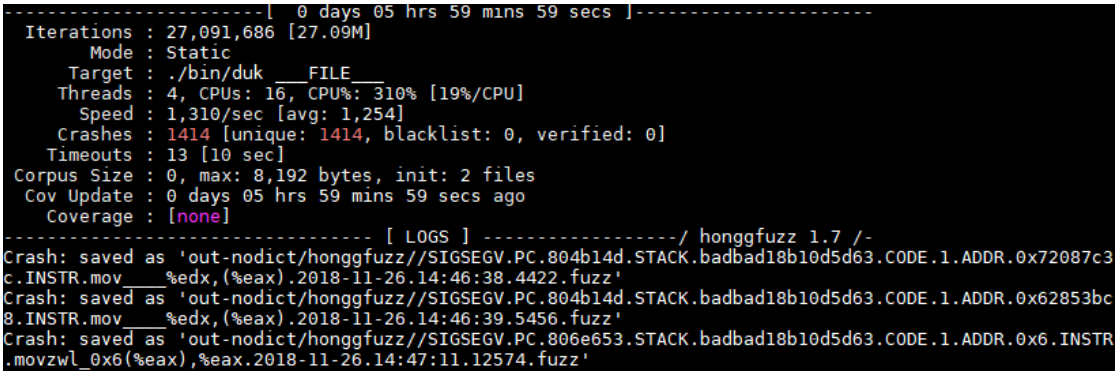


Figura 4.16: Estado de Honggfuzz al finalizar las pruebas en Duktape sin diccionario

Honggfuzz ofrece información justa para conocer el estado del *fuzzer*. Con la experiencia obtenida al realizar las mismas pruebas con AFL se puede deducir que encuentra el mismo fallo que AFL en modo *dumb* y permanece bloqueado sin encontrar nuevos caminos. El número de ejecuciones por segundo es más elevado y por ello se obtendrá un mayor número de fallos que el *fuzzer* considera únicos pero en realidad se trata del mismo.

En el caso de emplear un diccionario se debe iniciar el proceso de la siguiente forma:

```
honggfuzz -n 4 -x -f inputs/ -W output/ -w dict.txt -- ./bin/duk
↪ ___FILE___
```

Los resultados obtenidos al emplear diccionario van a ser idénticos que en el caso de no emplearlo ya que, cada intento del *fuzzer* va a ser independiente y no podrá sobrepasar la posible condición que evita que el proceso continúe. De igual manera que al no emplear diccionario, el tiempo del proceso se ha establecido en seis horas. En la figura 4.17 se muestran los fallos encontrados con Honggfuzz al emplear diccionario.

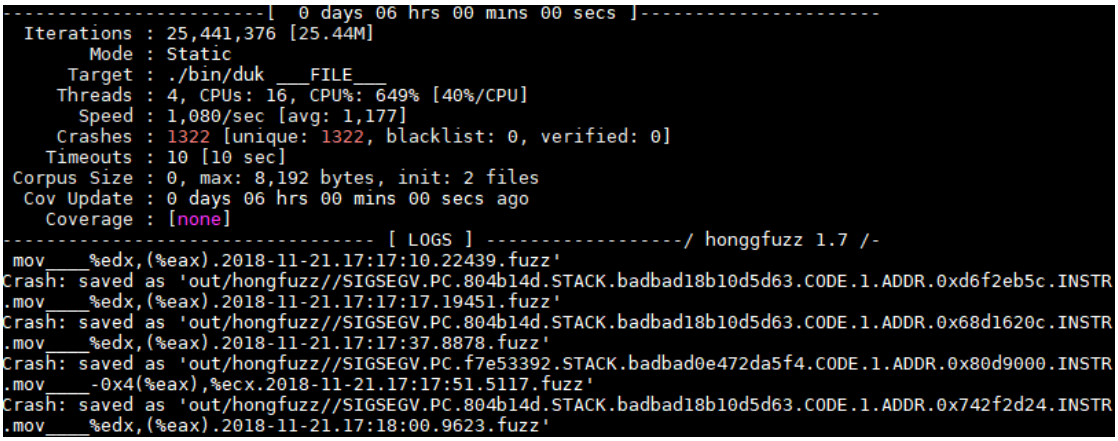


Figura 4.17: Estado de Honggfuzz al finalizar las pruebas en Duktape con diccionario

En la sección 5.2.1 se incluyen los resultados obtenidos por Honggfuzz con el binario Duktape.

4.2.2. FileS2

El comando *file* [mp] en sistemas Unix permite detectar el tipo y el formato de un fichero validando el archivo en tres fases, mostrándose el resultado en la primera que logra identificar el fichero. Las fases son las siguientes:

- **Sistema de archivos:** Estas pruebas evalúan el resultado obtenido al emplear la llamada de sistema *stat*. Se comprueba si el fichero está vacío o si es un tipo de fichero especial como pueden ser los *sockets*, enlaces simbólicos o *pipes*.
- **Magic bytes:** Estos bytes se encuentran en los primeros bytes del fichero y hacen referencia a formatos particulares de ficheros. Estos bytes indican al sistema UNIX el tipo de fichero según el fichero ubicado en /usr/share/misc/magic.mgc. Este fichero puede alterarse indicando con la opción -m la ruta de un fichero alternativo.
- **Pruebas de sintaxis:** Esta prueba se realiza en último lugar y busca en el fichero el tipo de sintaxis que emplea en los caracteres imprimibles, es decir, únicamente se aplicará esta prueba sobre ficheros de texto.

Esta prueba de Rode0day dispone del código por lo que va a ser posible realizar la instrumentación en tiempo de compilación y emplear AFL, Honggfuzz y Libfuzzer. Al realizar la instrumentación, el proceso dispondrá de mayor información del sistema en pruebas y la calidad del *fuzzing* se verá incrementada.

Las condiciones del proceso se han fijado en seis horas para los tres *fuzzers* empleando cuatro hilos del procesador para cada *fuzzer*. Al realizar la instrumentación se localiza un mayor número de caminos para explorar por lo que el tiempo empleado finalmente se ha incrementado a un día. En todos los casos se ha empleado un diccionario con las cadenas de texto obtenidas del binario una vez instrumentado.

AFL

Al disponer del código, se ha podido instrumentar en tiempo de compilación incrementando las posibilidades de AFL para encontrar comportamientos interesantes. A diferencia que en el modo *dumb* o al emplear QEMU, AFL optimiza su funcionamiento analizando los caminos favorables debido a la información que recibe en cada ejecución. El número de ejecuciones no se va a ver afectado por la emulación con QEMU, aumentando también las ejecuciones por segundos del *fuzzer* produciendo un proceso de mayor calidad y mucho más rápido.

Para compilar el código modificado de *file* con la tecnología LAVA se puede indicar el compilador modificando la variable de entorno CC antes de ejecutar el comando *make*:

```
1 CC=afl-gcc make
```

Una vez que se ha instrumentado el código y se ha obtenido el binario compilado ya es posible iniciar el proceso de *fuzzing* empleando un enfoque de caja blanca. De igual manera que en pruebas anteriores, se ha empleado un diccionario formado por las cadenas de texto obtenidas del binario compilado. El tiempo de las pruebas se ha establecido en seis horas. Estas pruebas se han realizado con cuatro hilos del procesador empleando las opciones de AFL -M y -S. Para iniciar los procesos de AFL se han ejecutado los siguientes comandos:

```
1 afl-fuzz -i inputs -o output -x dict.txt -M master-fileS2 --
  ↪ ./src/src/file -m src/magic/magic.mgc @@
2 afl-fuzz -i inputs -o output -x dict.txt -S slave1-fileS2 --
  ↪ ./src/src/file -m src/magic/magic.mgc @@
3 afl-fuzz -i inputs -o output -x dict.txt -S slave2-fileS2 --
  ↪ ./src/src/file -m src/magic/magic.mgc @@
4 afl-fuzz -i inputs -o output -x dict.txt -S slave3-fileS2 --
  ↪ ./src/src/file -m src/magic/magic.mgc @@
```

En la figura 4.18 se muestra el estado de uno de los procesos una vez finalizadas las pruebas.

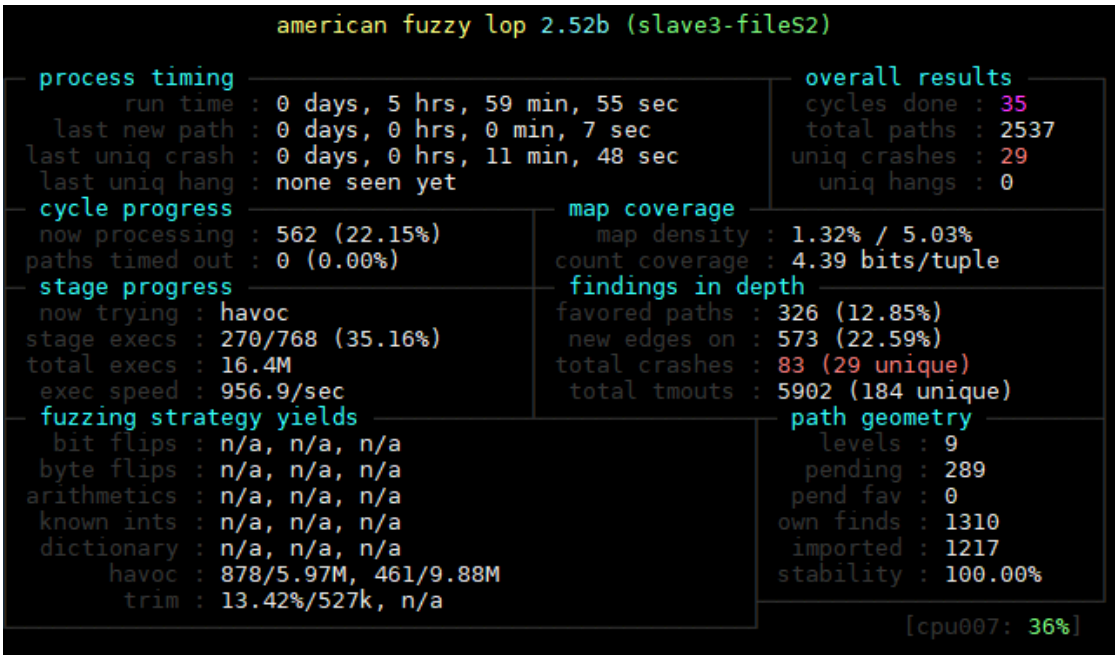


Figura 4.18: Estado de AFL al finalizar las pruebas en fileS2 durante seis horas

En la pantalla de estado de AFL tras las pruebas se puede apreciar que se han detectado múltiples fallos en los caminos explorados por el *fuzzer*. Durante las pruebas se ha observado que la estabilidad no varía y la profundidad en la geometría de los caminos es amplia. Estos valores deben tenerse en cuenta de cara a futuros intentos. Observando los ciclos realizados y los caminos pendientes se puede comprobar que el tiempo asignado a las pruebas no es suficiente y el proceso no ha terminado. A pesar de que el proceso no esté completo se pueden observar unos valores prometedores.

Teniendo en cuenta los valores anteriores y para aprovechar por completo las cualidades de AFL se ha incrementado el tiempo de *fuzzing* a un día de forma que los resultados van a ser más prometedores y con posibilidades de encontrar muchos más fallos únicos. En la figura 4.19 se incluye una imagen de los resultados obtenidos con uno de los procesos de AFL tras un día de pruebas sobre el binario modificado de *file*.

Los resultados obtenidos en las pruebas realizadas durante un día muestran un estado de AFL mucho más completo tanto en el número de ciclos realizados como en los fallos detectados. AFL indica con el color amarillo que el proceso de *fuzzing* no ha finalizado pero terminará pronto. Al aumentar el tiempo de pruebas ha sido posible incrementar la cobertura del código, realizar más ciclos y, por consiguiente, encontrar más fallos que en el caso de realizar pruebas durante un tiempo de seis horas.

Los resultados obtenidos se analizan en la sección 5.2.2.

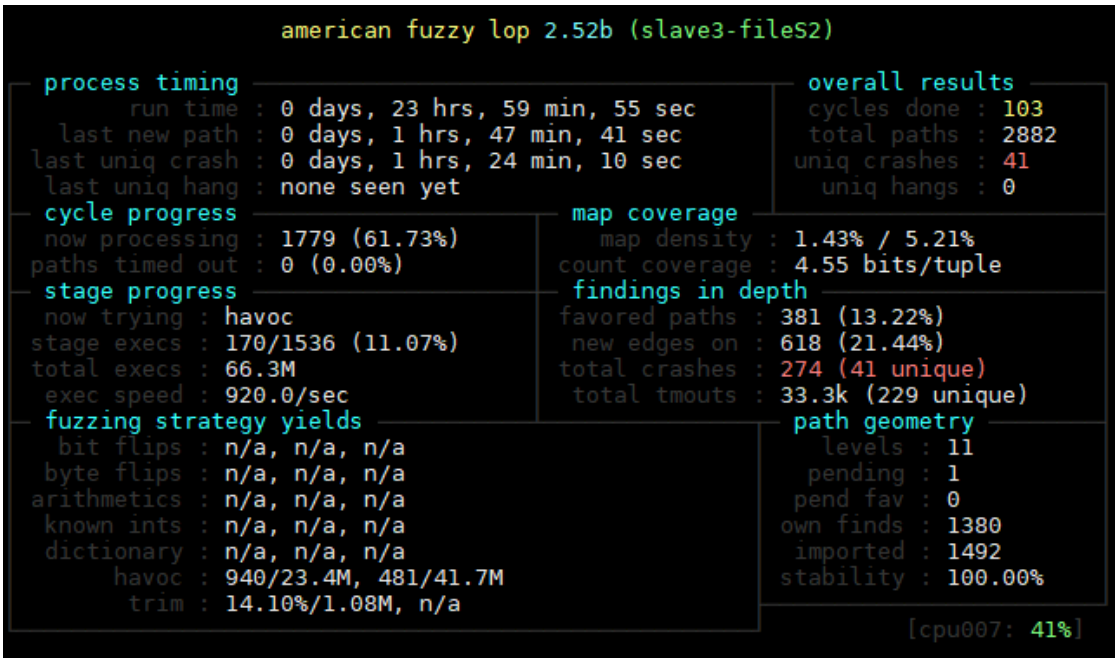


Figura 4.19: Estado de AFL al finalizar las pruebas en fileS2 durante un día

Honggfuzz

Tanto AFL como Honggfuzz incluyen herramientas para realizar la instrumentación del código en tiempo de compilación, facilitando así la generación de un binario que el *fuzzer* es capaz de monitorizar. Por ello, a la hora de realizar la compilación del proyecto con Honggfuzz se puede emplear el siguiente comando:

```
1 CC=hfuzz-gcc make
```

Una vez obtenido el binario de FileS2 instrumentado por Honggfuzz ya es posible iniciar el proceso de *fuzzing*. Como en el caso de AFL, el tiempo se ha establecido en seis y veinticuatro horas, empleando cuatro hilos del procesador y haciendo uso del mismo diccionario que el utilizado para la detección de fallos con AFL. De esta forma, las condiciones son idénticas y es posible apreciar las diferencias entre ambos *fuzzers*. Para iniciar Honggfuzz con las condiciones anteriores se ha ejecutado el siguiente comando:

```
1 honggfuzz -f inputs -W outputs -w dict.txt -n 4 -- ./rebuilt/bin/file
  ↪  -m rebuilt/share/misc/magic.mgc
```

En la figura 4.20 se puede observar la pantalla de estado de Honggfuzz tras un periodo de una hora de pruebas. El programa objetivo se ha podido instrumentar correctamente ya que aparecen las indicaciones completas en la sección de *Coverage* pero es incapaz de actualizar la cobertura, es decir, no puede localizar nuevos caminos favorables sobre los que realizar pruebas.

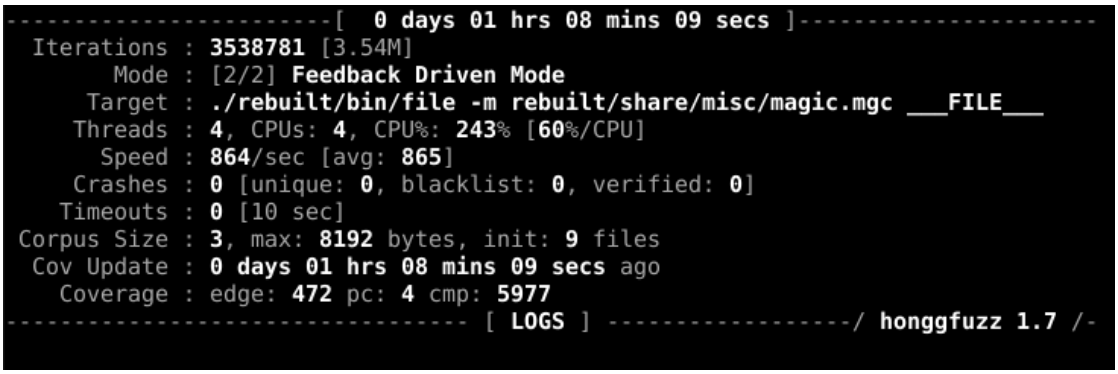


Figura 4.20: Estado de Honggfuzz tras una hora de pruebas

Debido a que no ha sido capaz de realizar actualizaciones sobre la cobertura, Honggfuzz no ha encontrado fallos sobre el binario FileS2.

Libfuzzer

Algunos *fuzzers*, como es el caso de Libfuzzer, ofrecen la posibilidad de obtener un gran rendimiento aunque para lograrlo puede ser necesario adaptar considerablemente el código. En la sección 4.9 se propone la modificación de un binario como forma de acceder a sus funciones. Este método se intentó replicar para el caso de FileS2 pero no fue viable por la cantidad de dependencias que requieren las estructuras que emplean las funciones principales que realizan la operación de identificar el tipo del fichero.

En cualquier proceso de *fuzzing* se deben considerar varios factores a la hora de realizar pruebas sobre una función. Uno de ellos es el tiempo necesario para estudiar el código, lo que está directamente relacionado con la selección del *fuzzer* más adecuado para el proyecto a analizar. En el caso de Libfuzzer, las pruebas han de realizarse siempre en modo persistente por lo que es necesario introducir la función de entrada para el *fuzzer* y adaptar su estructura al código del programa. El proyecto FileS2, al tener que enviar el nombre del fichero por argumento, el caso de prueba de Libfuzzer se debe introducir en un fichero o, en un caso más óptimo, incorporar el caso de prueba generado en el contenido del fichero que se lee con una llamada a *fopen* o similar mediante un *hook* o alteración del funcionamiento de la llamada.

El código del proyecto FileS2 es complejo y requiere dedicar gran cantidad de tiempo para realizar pruebas sobre el mismo. A diferencia de AFL o Honggfuzz, capaces de trabajar con ficheros, Libfuzzer no lo permite. Por ello, no es el *fuzzer* más óptimo para encontrar fallos en FileS2 sin invertir una gran cantidad de tiempo que puede evitarse con el empleo de otros *fuzzers*.

El rendimiento que se podría obtener con Libfuzzer posiblemente fuese superior al de AFL aunque, en este caso, se ha priorizado el tiempo requerido en la adaptación del código sobre el rendimiento que puede brindar un determinado *fuzzer*.

Capítulo 5

Resultados

Los *fuzzers* comentados en la sección 3 han sido de encontrar los fallos esperados en todas las pruebas realizadas sobre los distintos programas seleccionados empleando técnicas tanto de caja negra como de caja blanca. De esta forma, en caso de disponer del código, se ha podido instrumentar para realizar procesos de *fuzzing* guiados obteniendo resultados más prometedores que en aquellas programas de los que no se disponía del código.

Durante las pruebas realizadas ha sido posible utilizar características propias de cada *fuzzer* obteniendo resultados prometedores que han permitido conocer las fortalezas y debilidades de cada uno de ellos.

En esta sección se incluyen los resultados obtenidos en las fases del proyecto de búsqueda de vulnerabilidades en la versión vulnerable de NTP y en dos de las muestras que se incluyeron en Rode0day durante el mes de noviembre.

5.1. CVE-2009-0159

En esta sección se incluyen los resultados obtenidos durante el proceso de *fuzzing* sobre la versión 4.2.2 de NTP que cuenta con una vulnerabilidad en su función *cookedprint*. Ha sido posible localizar esta vulnerabilidad con todos los *fuzzers* tras aplicar ciertas modificaciones sobre el fichero con el código de ntpq. Finalmente se ha realizado una comparativa de los resultados obtenidos teniendo en cuenta el número de ejecuciones por segundo, el número de fallos o el número de fallos únicos que han sido capaces de encontrar cada uno de los *fuzzers* al emplear un diccionario con palabras clave y sin él.

5.1.1. Resultados obtenidos con AFL

AFL ha demostrado que es capaz de encontrar el fallo empleando un diccionario, lo que reduce el tiempo dedicado considerablemente, además de ser capaz de detectar la vulnerabilidad sin necesidad de usar un diccionario, aumentando así el tiempo del proceso.

Una vez adaptado el código de NTP para recibir la entrada generada por el *fuzzer* y con el binario instrumentado se pueden iniciar las pruebas con AFL. Dado que la vulnerabilidad existe en la versión analizada se tiene certeza de que AFL encontrará el fallo por lo que se debe considerar el tiempo que va a requerir localizar dicha vulnerabilidad y hacer fallar al programa.

Para reducir el tiempo que requieren las pruebas es posible emplear un diccionario con palabras clave y constantes del fichero ntpq. En la figura 4.3 se puede observar el estado de AFL tras dos minutos de realización de pruebas cuando se le ha indicado que emplee un diccionario con palabras clave que han permitido que se disparen las condiciones necesarias para encontrar el fallo.

A pesar de haber encontrado el fallo, los indicadores como los caminos encontrados, los caminos pendientes y los ciclos realizados hacen prever que no se ha completado el proceso de *fuzzing*. Los resultados obtenidos por AFL han permitido que el programa finalice enviando una señal SIGSEGV.

Detectar la vulnerabilidad de NTP sin emplear diccionario requiere de un proceso de *fuzzing* mayor ya que se deben superar condiciones que AFL es capaz de sobrepasar aunque, al no disponer de ayuda, es más costoso. En la figura 4.1 se puede observar que en un tiempo inferior a tres horas AFL ha sido capaz de encontrar el fallo y los ciclos realizados indican que las pruebas pueden concluir. En la fase sin diccionario se realizaron intentos con cuatro procesos de AFL disponiendo de un *master* y un *slave* de los cuales solo detectó el fallo uno de ellos. El motivo que ha impedido que se detectase el fallo en más de un esclavo está directamente relacionado con los niveles de profundidad que cada uno de ellos ha sido capaz de encontrar. Incrementando el tiempo es posible que se hubiese detectado el mismo fallo en todos los procesos iniciados.

La diferencia de tiempos es considerable para detectar el mismo fallo por lo que se destaca la importancia y el valor que añade el empleo de un diccionario. Por otro lado, comparando los caminos encontrados en las figuras anteriores se puede observar que se dispone de un mayor número de caminos al emplear un diccionario, aumentando de esta forma la posibilidad de encontrar fallos sobre cualquier programa.

El funcionamiento de los *fuzzers* depende primordialmente de las alteraciones que realizan sobre los casos de prueba iniciales. En el caso de AFL, se realiza alteraciones aleatorias por lo que en pruebas independientes no tienen porque encontrarse los mismos caminos o detectarse los mismos fallos. De los resultados anteriores se puede deducir que es muy importante asignar tiempos de *fuzzing* coherentes según el programa que se esté analizando y que los diccionarios son un gran apoyo a la hora de explorar caminos y encontrar fallos.

Con la utilidad afl-plot de AFL es posible generar gráficos que detallan el estado de un proceso a lo largo de las pruebas realizadas. De esta forma se puede visualizar el número de ejecuciones por segundo, los fallos y caídas en relación a los niveles de profundidad alcanzados en el programa y los ciclos realizados junto a los caminos encontrados.

En la figura 4.1 se muestra el estado del *fuzzer* durante el proceso de *fuzzing*. Por otro lado, en la figura 5.1 se puede observar que se ha profundizado hasta el séptimo nivel y ha sido posible encontrar dos fallos sin detectar ninguna caída.

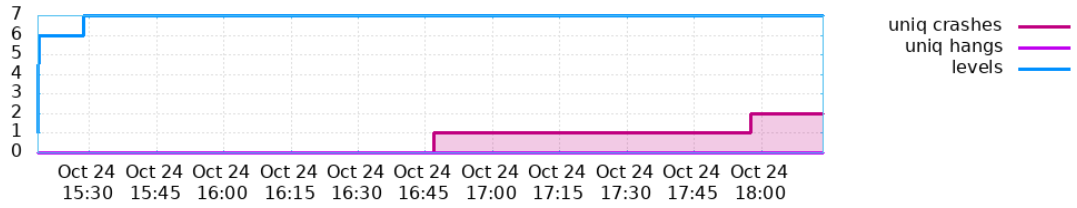


Figura 5.1: Gráfico con información de los fallos encontrados

Para conocer el número de ejecuciones por segundo del proceso se puede generar un gráfico como el que aparece en la figura 5.2. En el caso del *slave* sin diccionario el número de ejecuciones se mantiene prácticamente constante a lo largo del tiempo.

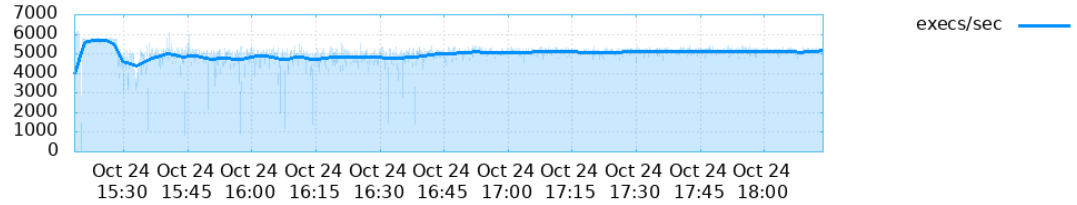


Figura 5.2: Gráfico con las ejecuciones por segundo realizadas

Por último, es posible obtener información de los caminos encontrados por AFL con el gráfico 5.3. En este gráfico se puede observar que poca información es legible debido a que el número de ciclos realizado es superior al que realmente necesita AFL para terminar el proceso. Por ello, es importante ajustar el tiempo de pruebas a lo que realmente requiere el *fuzzer*.

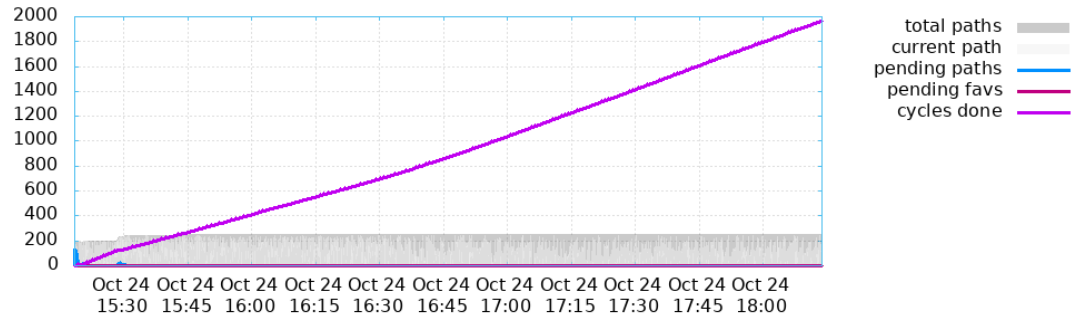


Figura 5.3: Gráfico con los caminos encontrados y los ciclos realizados

Existen diferencias entre los gráficos de estado de un proceso que ocupa el rol de *master* que otro que actue como *slave* ya que las tareas que realiza cada rol son distintas. Por ejemplo, el número de ejecuciones va a ser inferior en un *master* además de los niveles de profundidad y, por consiguiente, los fallos encontrados. La diferencia se debe a que

un *master* realiza alteraciones deterministas sobre los casos de prueba mientras que un *slave* únicamente realiza alteraciones aleatorias.

5.1.2. Resultados obtenidos con Honggfuzz

Honggfuzz también ha sido capaz de detectar el fallo en la versión vulnerable de NTP al emplear un diccionario y sin él. Las pruebas con Honggfuzz han compartido las mismas condiciones que las de AFL por lo que se ha empleado el mismo tiempo de *fuzzing* y los mismos núcleos del procesador.

El funcionamiento de los *fuzzers* de propósito general suelen compartir el mecanismo de entrega y disponen de la posibilidad de instrumentar en tiempo de compilación el código del programa objetivo para optimizar lo máximo posible la detección de fallos. Esta similitud entre los *fuzzers* ha permitido reutilizar el código adaptado en la sección 4.1 e instrumentarlo con la utilidad *hfuzz-clang*. Partiendo de las mismas condiciones de configuración, los casos de prueba iniciales empleados en Honggfuzz han sido los mismos que con AFL.

Al emplear Honggfuzz con diccionario, en un periodo de dos minutos, ha sido posible detectar cerca de doscientos fallos. En la figura 4.7 se puede observar el estado final del *fuzzer*. Los nombres de los ficheros de salida de Honggfuzz tienen información con la señal que ha provocado el fallo, la dirección del contador de programa o la instrucción en la que se produce el fallo por lo que se puede observar en la propia pantalla de estado si el fallo es interesante.

La información que muestra Honggfuzz en su pantalla de estado es reducida pero es posible conocer cuando se ha producido la última ampliación de la cobertura del código siempre que el binario esté instrumentado correctamente. A pesar de ello, no es posible conocer cuantos caminos se han detectado, cuantos se han explorado, cuantos ciclos se han realizado o cuantos caminos son favorables.

En el caso de no emplear un diccionario, el *fuzzer* requiere de mayor tiempo para detectar la vulnerabilidad. El tiempo total se ha establecido en tres horas con cuatro núcleos del procesador y Honggfuzz ha sido capaz de detectar el fallo en menos de dos horas sin recibir ayuda de un diccionario. En la figura 4.5 se puede observar que se ha detectado el fallo pero el *fuzzer* realiza pruebas similares provocando el mismo fallo multitud de veces. Si se continúan las pruebas durante el tiempo de *fuzzing* establecido el estado del *fuzzer* no se ve apenas alterado ya que se ha detectado muchas veces el mismo fallo pero la cobertura del código no se ve actualizada después de localizar el fallo. En la figura 5.4 se incluye la imagen con los fallos detectados por Honggfuzz tras tres horas de pruebas.

La identificación del fallo que provoca la señal SIGSEGV se ha detectado rápidamente pero ha permanecido el resto del tiempo asignado a las pruebas bloqueado sin actualizar la cobertura debido a que no se ha empleado ningún diccionario. Por ello, ha sido incapaz de continuar avanzando para encontrar nuevos caminos que permitan identificar posibles fallos.

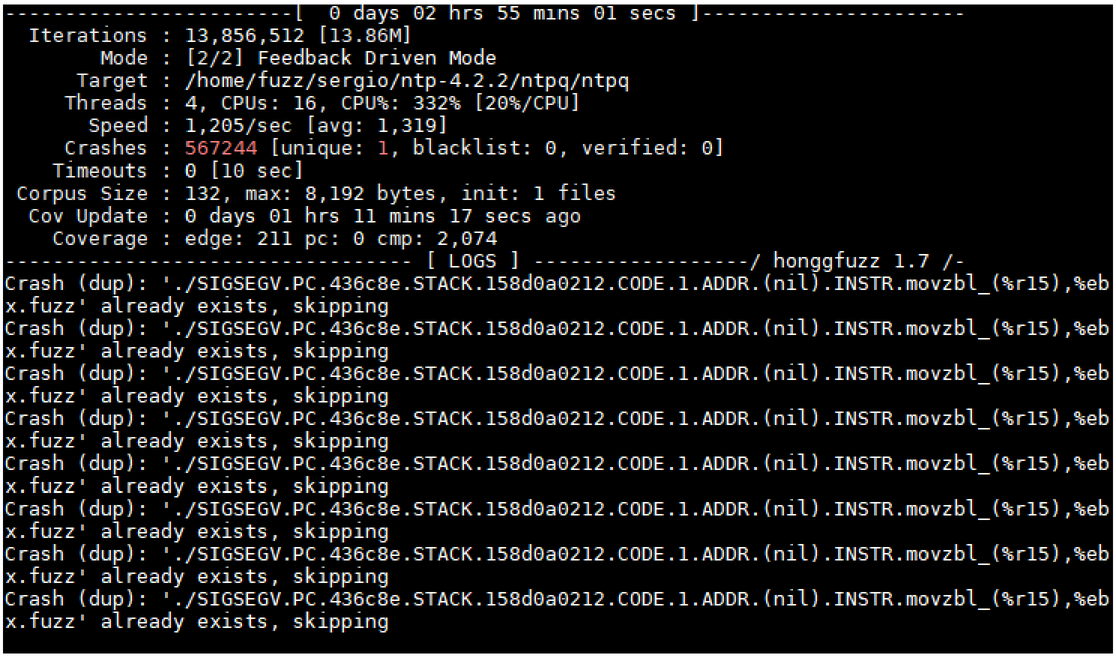


Figura 5.4: Estado final de Honggfuzz detectando múltiples fallos sin diccionario

Honggfuzz ofrece la posibilidad de realizar instrumentación por hardware empleando registros del procesador. Al emplearse un procesador AMD, no ha sido posible comprobar la eficacia de este tipo de instrumentación.

5.1.3. Resultados obtenidos con Libfuzzer

Las pruebas relacionadas con la detección del fallo sin diccionario con este *fuzzer* no se han realizado ya que al hacer uso de tareas en paralelo, la evolución del proceso de *fuzzing* se incluye en ficheros de *log* pero como la función cookedprint muestra información por pantalla, se producen unos ficheros con gran cantidad de contenido. Dado que las pruebas con otros *fuzzers* se han realizado con cuatro procesos, Libfuzzer generaría cuatro ficheros de gran tamaño. Para evitarlo, sería necesario controlar el contenido o investigar en el código si no se afecta al flujo del programa implicando que sea necesario invertir mayor tiempo en detectar el mismo fallo sin diccionario.

La detección del fallo con este *fuzzer* ha requerido de un mayor esfuerzo y el empleo de técnicas que con AFL y Honggfuzz no han sido necesarias. A pesar de ello, ha sido posible localizar la vulnerabilidad y observar una traza de la pila de llamadas de la ejecución corroborando el éxito del proceso.

5.1.4. Comparativa de los resultados obtenidos

Todos los *fuzzers* han sido capaces de detectar la vulnerabilidad después de la modificación aplicada en el código de ntpq para enfocar las pruebas en la función cookedprint.

A pesar de ello, se puede evidenciar que el *fuzzing* con AFL muestra más información y un mayor número de utilidades que pueden permitir optimizar futuros intentos. Una clara evidencia de ello es la posibilidad de generar gráficos y contar con gran cantidad de información en la pantalla de estado en cada momento del proceso.

Con todas las tecnologías se han realizado las pruebas empleando un diccionario, agilizando la detección del fallo. En todos los casos se debe a que el *fuzzer*, al recibir ayuda, es capaz de encontrar un mayor número de caminos para explorar en menor tiempo incrementando así los niveles de cobertura del código. Al realizar las pruebas sin diccionario se ha comprobado que todos los *fuzzers* han necesitado más tiempo e incluso algunas de las instancias del *fuzzer* han sido incapaces de encontrar los caminos y la profundidad requerida para detectar la vulnerabilidad.

Honggfuzz ha sido capaz de detectar el fallo con mayor rapidez al no emplear diccionario y permanece bloqueado sin actualizar la cobertura del código tras su detección. AFL es capaz de detectar el fallo pero ha requerido de mayor tiempo.

Libfuzzer también ha sido capaz de detectar la vulnerabilidad empleando diccionario, obteniendo el fallo en cuestión de segundos. Para poder localizar el fallo, ha sido necesario invertir un tiempo superior al dedicado con AFL o Honggfuzz y se ha necesitado emplear la herramienta LIEF para poder acceder a la función vulnerable sin invertir mucho tiempo en el análisis del código de NTP. Este *fuzzer* ha demostrado ser más rapido que AFL y Honggfuzz aunque ha requerido de un mayor tiempo para poder iniciar el proceso de *fuzzing*. Las pruebas de detección sin diccionario con Libfuzzer no se han realizado aunque el *fuzzer* es capaz de llegar a detectarlo.

La información que muestra la pantalla de estado de AFL es mucho más extensa que los otros *fuzzers* y al actualizarse continuamente es posible detectar tanto si se está produciendo una situación inesperada como si se está consiguiendo éxito en la pruebas.

Los nombres de los ficheros de salida de Honggfuzz incluyen mayor información que en el caso de AFL y Libfuzzer pero ninguno de ellos muestra una traza de la pila de llamadas al detectar un fallo como hace Libfuzzer. Los nombres de los ficheros pueden ayudar a diferenciar un fallo de otro y obtener información del fallo pero no es indispensable ya que, si el fichero obtenido produce un fallo, se debería comprobar con un *debugger* si el fallo es válido.

Finalmente, los resultados obtenidos por cada *fuzzer* se han agrupado teniendo en cuenta el empleo de diccionario, el número de ejecuciones medias por segundo, los fallos encontrados totales y los fallos clasificados por los *fuzzers* como únicos. La tabla de resultados es la siguiente:

	AFL		Honggfuzz		Libfuzzer	
	Con dicc.	Sin dicc.	Con dicc.	Sin dicc.	Con dicc.	Sin dicc.
Ejecuciones/s	8.3k	5.2k	497	1.4k	520k	-
Fallos	30.6k	2	1.8k	155.3k	1	-
Fallos únicos	46	2	193	1	1	-

Cuadro 5.1: Comparación de los resultados para NTP-4.2.2

Observando los fallos únicos indicados por AFL y Honggfuzz se obtiene información no fiable ya que solo existe una vulnerabilidad pero los *fuzzers* indican que se han identificado más de una. Estos resultados dependen de la valoración que hace el *fuzzer* de lo que considera un fallo único.

Las pruebas realizadas para detectar la vulnerabilidad clasificada con el identificador CVE-2009-0159 ha permitido familiarizarse con estas tecnologías, conocer su funcionamiento y comenzar a diferenciar sus ventajas e inconvenientes.

5.2. Rode0day

En el mes de noviembre, la competición de *fuzzing* Rode0day ofrece tres binarios compilados y el código de una aplicación. Para analizar las ventajas y desventajas de cada *fuzzer* se han realizado pruebas de caja negra sobre uno de los binarios y de caja blanca sobre el código de uno de los programas disponible.

En la sección 5.2.1 se incluye los resultados obtenidos durante los procesos de *fuzzing* llevados a cabo con AFL y Honggfuzz ya que ofrecen la posibilidad de realizar pruebas de caja negra. En el caso de Libfuzzer no es posible analizar binarios de los que el código no está disponible.

En la sección 5.2.2 se incluyen los resultados obtenidos por los tres *fuzzers* escogidos para el proyecto con los que ha sido posible instrumentar el código de la herramienta *file* de sistemas UNIX modificada por la tecnología LAVA.

5.2.1. Resultados del binario Duktape

Al no disponer del código del motor Javascript Duktape, las pruebas realizadas con AFL y Honggfuzz han sido de caja negra. A pesar de no disponer de información del sistema a analizar, ambos *fuzzers* han sido capaces de encontrar vulnerabilidades que se han podido reportar con éxito empleando la API que ofrece la web de Rode0day.

Resultados obtenidos con AFL

AFL ofrece más posibilidades para *fuzzear* binarios de caja negra sin depender del hardware que Honggfuzz. En primer lugar, las pruebas se han realizado empleando el modo *dumb* de AFL con cuatro procesos lo que permite conseguir un buen número de ejecuciones por segundo siendo capaz de encontrar múltiples fallos. En este modo de *fuzzing*, AFL no recibe nada de información del sistema en pruebas por lo que es incapaz de discernir entre un fallo y otro. En este modo, una vez finalizadas las pruebas se deberá comprobar si estos fallos son únicos. El tiempo dedicada a cada prueba se ha establecido en seis horas.

Las pruebas se han realizado sin enviar ayuda al *fuzzer* con un diccionario y empleando un diccionario generado con las cadenas de texto imprimibles del propio binario. Tal como se observa en la imagen 4.13, AFL es capaz de detectar en modo *dumb* y con ayuda de un diccionario múltiples fallos pero únicamente ha encontrado dos caminos posibles y un único nivel de profundidad en la aplicación.

Algo similar ocurre en el caso de no emplear diccionario. En la figura 4.14 se pueden observar unos resultados similares que en el caso de emplear diccionario aunque el número de ejecuciones por segundo es ligeramente superior al no emplear diccionario. Por ello, al hacer uso un diccionario extenso, el *fuzzer* debe trabajar con él por lo que se produce un impacto en el rendimiento que se aprecia en el número de ejecuciones por segundo.

Teniendo en cuenta los caminos encontrados, se puede prever que los fallos encontrados van a tratarse del mismo fallo único dado que solo se han encontrado dos caminos para explorar. El gran número de fallos encontrados se debe a la cantidad de ciclos realizados durante las seis horas de las pruebas por lo que se puede entender que el *fuzzer* ha estado realizando pruebas un tiempo superior al necesario.

Los *fuzzers* son herramientas muy útiles para detectar fallos, sobre todo al disponer de gran cantidad de información pero al no contar con información del sistema en pruebas es posible que se localice una condición compleja que no sea capaz de cumplimentar. Es por ello que las pruebas realizadas en modo *dumb* con AFL han quedado bloqueadas en una condición compleja que de ninguna manera va a ser capaz de sobrepasar.

Para superar los problemas relacionados con las pruebas de caja negra, el creador de AFL junto a la comunidad realizaron una investigación en QEMU para adaptar su funcionalidad con AFL para que fuese posible optimizar el *fuzzing* sobre programas de los que no se puede disponer del código para realizar la instrumentación. Tal como aparece en la figura 4.15, empleando QEMU se localizan más de dos mil caminos posibles alcanzando el nivel once en la cobertura. Un valor a tener en cuenta es la penalización en el número de ejecuciones por segundo pero merece la pena observando los fallos únicos conseguidos en las seis horas de pruebas. El tiempo asignado para QEMU es insuficiente y es posible obtener mejores resultados ampliando el tiempo de pruebas.

En la gráfico 5.5 se puede observar que los fallos encontrados y los niveles para explorar dejan de incrementarse en un cierto punto. Esto se debe a la información que se puede recopilar del gráfico 5.6 que muestra el número de ejecuciones por segundo. En el mismo momento que comienzan a descender las ejecuciones por segundo, los fallos detectados dejan de incrementar. Esto se debe al tipo de pruebas que realizan los procesos *slave* sobre los casos de prueba obtenidos del proceso *master*. Debido a que el *master* se encarga de realizar pruebas deterministas, las entradas que obtiene pueden incrementar de tamaño, impactando directamente sobre el tiempo que necesitan los procesos *slave* para completar las alteraciones no deterministas de los casos de prueba disponibles.

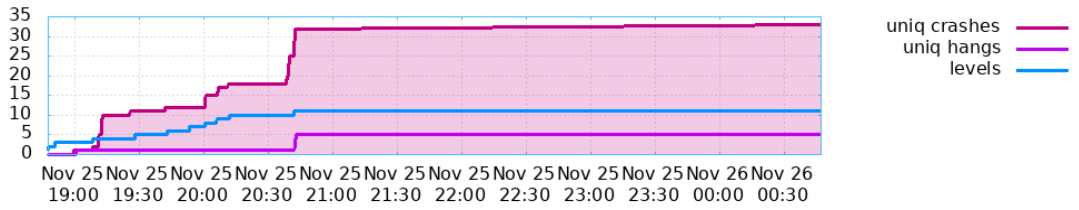


Figura 5.5: Gráfico con los fallos encontrados empleando QEMU en Duktape

Los gráficos anteriores también afectan a los datos que muestra el gráfico 5.7. En el

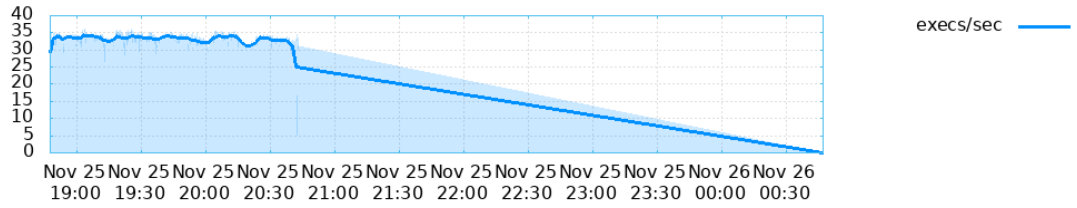


Figura 5.6: Gráfico con las ejecuciones por segundo empleando QEMU en Duktape

momento en el que las ejecuciones por segundo decrementan por el tipo de pruebas que se están realizando, los caminos pendientes y todo lo relacionado con ellos se mantiene constante.

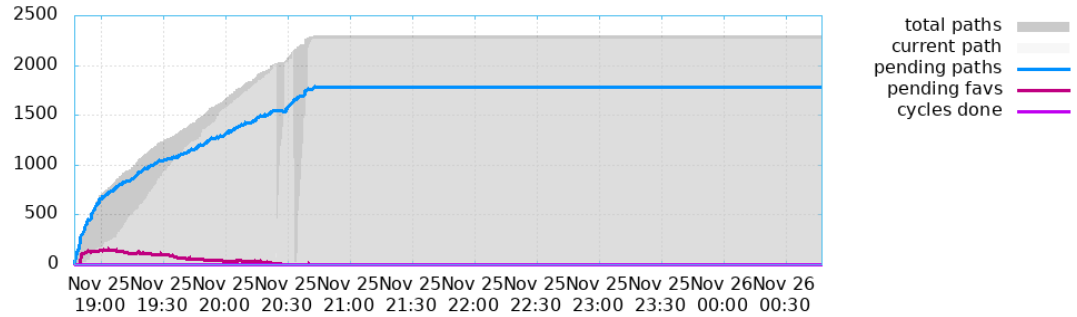


Figura 5.7: Gráfico con los caminos encontrados y los ciclos empleando QEMU en Duktape

Los resultados obtenidos al emplear QEMU son óptimos y ha permitido comprobar la utilidad de su uso además de confirmar que el sacrificio del número de ejecuciones mejora considerablemente los resultados obtenidos al final. A pesar de estos resultados, emplear unos casos de prueba iniciales es muy importante para que no se produzca el estancamiento del proceso de *fuzzing*.

Resultados obtenidos con Honggfuzz

De la misma forma que AFL permite emplear el *fuzzer* sin poder realizar instrumentación, Honggfuzz también permite realizar pruebas y emplear únicamente su generador de entradas y su mecanismo de entrega. Este tipo de *fuzzing* es clasificado por la herramienta como *static* y ha permitido encontrar un fallo. El tiempo de pruebas se ha fijado también en seis horas realizando las pruebas con el mismo diccionario que en el caso de AFL.

Del mismo modo que ocurre con AFL en modo *dumb*, en el modo estático de Honggfuzz el *fuzzer* no recibe información del sistema en pruebas por lo que considera el mismo fallo como único aunque esto no sea cierto. Las pruebas se han realizado con diccionario y sin diccionario y los resultados obtenidos permiten verificar que, de igual forma que

ocurre con AFL, si se emplea un diccionario el número de ejecuciones por segundo se ve reducido.

En la figura 4.16 se pueden observar los resultados obtenidos cuando no se ha empleado un diccionario con un número de ejecuciones por segundo inferior al de la figura 4.17 que contiene el estado del *fuzzer* al finalizar las pruebas con diccionario. En estas imágenes se puede volver a comprobar como el número de fallos en modo estático es directamente proporcional al número de ejecuciones por segundo que consigue Honggfuzz.

Honggfuzz no dispone de la posibilidad de emplear QEMU para optimizar los resultados como si que permite AFL pero este *fuzzer* cuenta con una ventaja sobre AFL y es que permite realizar la instrumentación por hardware. En el caso de emplear un procesador Intel se pueden controlar los registros BTS y el subsistema PT para instrumentar un binario del que no se dispone el código pero en este caso, al emplearse un procesador AMD Ryzen, no ha sido posible incorporar esta prueba.

Comparativa de los resultados obtenidos

Los *fuzzers* modernos están pensados para trabajar recopilando información del sistema de pruebas de forma que se pueda guiar al *fuzzer* y realizar una amplia cobertura en el código (*coverage-guided fuzzers*, en inglés). Por ello, la realización de pruebas de caja negra están mucho más limitadas.

A pesar de estas limitaciones, los *fuzzers* cuentan con herramientas como QEMU o la posibilidad de instrumentar los programas por hardware para recopilar información y optimizar así las pruebas y, por consiguiente, también los resultados. A continuación se incluye una tabla comparativa de los resultados obtenidos durante las pruebas realizadas sobre el binario Duktape de Rode0day:

	AFL			Honggfuzz		
	Con dicc.	Sin dicc.	QEMU	Con dicc.	Sin dicc.	QEMU
Ejecuciones/s	500	420	20	1.1k	1.3k	-
Fallos	563	635	316	1.3k	1.4k	-
Fallos únicos	1	1	7	1	1	-

Cuadro 5.2: Comparación de los resultados para el motor de Javascript Duktape

5.2.2. Resultados del binario FileS2

La prueba de Rode0day fileS2 tiene disponible el código por lo que se ha podido instrumentar con AFL y Honggfuzz y proveer de información a cada uno durante el proceso de *fuzzing*. Al realizar la instrumentación se va a encontrar un mayor número de caminos y se necesitará asignar un tiempo amplio para en las pruebas. En este caso, el tiempo asignado ha sido de un día para cada uno de los *fuzzers* y se han realizado las pruebas empleado cuatro núcleos del procesador.

En el caso de Libfuzzer y debido al tiempo necesario para estudiar y adaptar el código del programa para hacerlo funcional con el *fuzzer*, se ha considerado que no es la mejor opción para realizar las pruebas priorizando la comodidad que ofrecen los otros *fuzzers* seleccionados para el proyecto.

A pesar de no ser la forma más óptima de generar un diccionario, en todas las pruebas realizadas se ha empleado el mismo diccionario generado con las cadenas de texto imprimibles del binario instrumentado. De esta forma se le puede ofrecer ayuda adicional al *fuzzer*.

Resultados obtenidos con AFL

AFL ha demostrado ser un *fuzzer* genérico basado en instrumentación más que fiable. Con él, ha sido posible obtener fallos únicos válidos para la plataforma de Rode0day. Como ya se ha mencionado, asignar un buen tiempo de pruebas es de vital importancia para que el proceso de *fuzzing* sea satisfactorio. Durante las pruebas realizadas se han asignado dos intervalos de tiempo: uno de seis horas y otro de un día. En ambos se ha ayudado a AFL con un diccionario y ha sido posible obtener mejores resultados cuanto mayor ha sido el tiempo de pruebas.

Los resultados obtenidos tras seis horas se muestran en la figura 4.18. Estos resultados no son completos y se puede observar en las indicaciones del campo *cycles done* del apartado *overall results*. El color que indica los ciclos completos aparece en lila mientras que debería aparecer en color verde en caso de que el proceso hubiese terminado. Este es el indicador principal pero en la sección *path geometry* en su campo *pending* se indican los caminos por comprobar seguido de los que quedan pendientes y han sido marcados como favorables por AFL.

Por ello, es importante aumentar el tiempo de las pruebas y así poder obtener mejores resultados. En la figura 4.19 se encuentra la pantalla de estado de uno de los procesos de AFL tras un día de pruebas. En él se observa como todos los valores se incrementan. En un día de pruebas, AFL ha sido capaz de encontrar un mayor número de camino disponibles y ha podido profundizar más en el binario. Este aumento en la cobertura del sistema en pruebas puede ofrecer resultados más prometedores ya que a mayor número de caminos para explorar mayores serán las probabilidades de alcanzar una condición que provoque un fallo. La ampliación del tiempo de pruebas ha permitido obtener un mayor número de fallos a pesar de que el proceso no esté completo.

Para visualizar los resultados, a continuación se incluyen los gráficos con el estado de AFL en las pruebas realizadas por uno de los *slave* durante la prueba de un día. En la figura 5.8 se puede observar como en los primeros momentos de las pruebas se profundiza en el sistema en pruebas y se produce un estancamiento tras dos horas de pruebas. A pesar de ello, el número de fallos únicos detectados mantiene un incremento más o menos constante a lo largo de las pruebas.

En la figura 5.9 las ejecuciones por segundo fluctúan durante todo el proceso obteniéndose una media aproximada de ochocientas ejecuciones por segundo.

En la figura 5.10 se pueden observar la geometría de los caminos encontrados. Los caminos pendientes incrementan al iniciar el proceso y van descendiendo conforme incre-

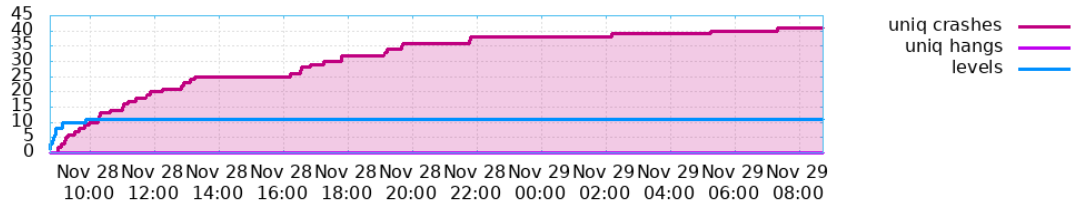


Figura 5.8: Gráfico con los fallos encontrados en FileS2

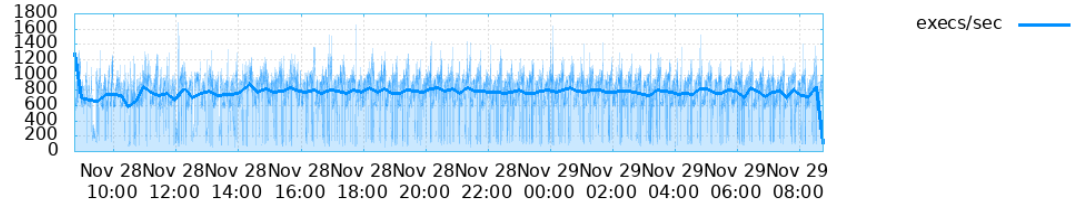


Figura 5.9: Gráfico con las ejecuciones por segundo en FileS2

mentan los ciclos. Debido al número total de caminos, la proporción del resto de valores se ve minúsculo.

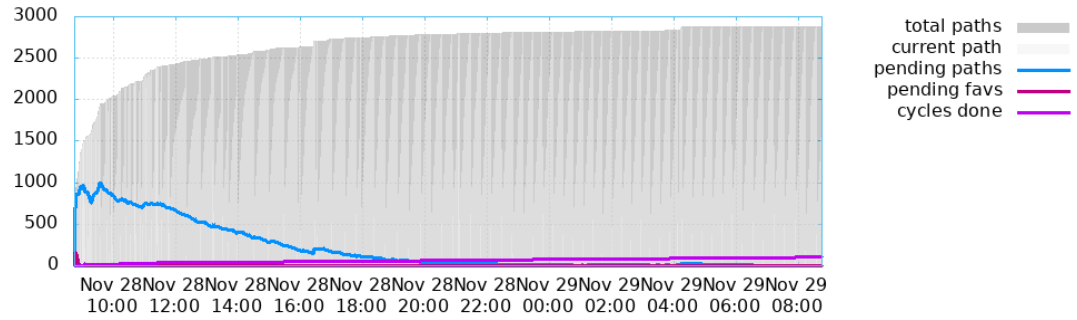


Figura 5.10: Gráfico con los caminos encontrados y los ciclos en FileS2

AFL ofrece una serie de variables de entorno que permiten añadir comprobaciones en la instrumentación durante el tiempo de compilación. Estas comprobaciones adicionales suponen una reducción del rendimiento por lo que se han realizado pruebas con el fin de comparar ambos resultados.

A la hora de instrumentar con AFL es posible incluir la variable de entorno AFL_HARDEN que añade opciones para bastionar el binario instrumentado. Esta opción añade automáticamente las opciones -D_FORTIFY_SOURCE=2 y -fstack-protector-all lo que permite identificar errores en memoria que no provocan un fallo. El coste sobre el rendimiento suele estar cerca del cinco por ciento. Realizando las pruebas durante un día los resultados obtenidos son peores que al no emplear la variable de entorno AFL_HARDEN.

Resultados obtenidos con Honggfuzz

Los resultados obtenidos con Honggfuzz no han sido todo lo óptimos que se podía esperar ya que, aunque se ha realizado correctamente la instrumentación, el *fuzzer* es incapaz de encontrar fallos en el programa.

Durante las pruebas realizadas, Honggfuzz no ha podido actualizar la cobertura la cual permanece equivalente al tiempo de trabajo. Por ello, el *fuzzer* no realiza correctamente la exploración y permanece bloqueado en una de las primeras condiciones encontradas.

A pesar de que los resultados en las pruebas realizadas sobre el binario de FileS2 no han sido todo lo óptimas que se podían esperar, Honggfuzz ha demostrado tener potencial como *fuzzer* de propósito general.

Resultados obtenidos con Libfuzzer

Debido a la gran cantidad de modificaciones que se deberían aplicar para poder iniciar el proceso de *fuzzing* con Libfuzzer y el tiempo ligado a realizar una implementación completa para el *fuzz-target*, se ha concluido que Libfuzzer no es la mejor opción para analizar el programa FileS2 ya que se deben tener en cuenta muchos factores para poder iniciar las pruebas de *fuzzing*.

El tratar de analizar el funcionamiento de un proyecto tan amplio puede ofrecer la posibilidad de encontrar una sección del código en el que incluir la entrada de los datos de Libfuzzer pero, en el caso de FileS2, al emplearse un gran número de estructuras en las llamadas a funciones no es posible emplear la misma técnica que en la sección 4.1.3 sin requerir un gran tiempo de estudio del código.

Con todo lo anterior se ha concluido que Libfuzzer no es el *fuzzer* más óptimo para analizar el funcionamiento de FileS2 en el caso de priorizar el tiempo sobre el rendimiento.

Comparativa de los resultados obtenidos

En las pruebas realizadas sobre FileS2 se han obtenido unos resultados que no son tal y como se podían esperar. Todos los *fuzzers* tienen sus ventajas e inconvenientes pero en el caso de las pruebas realizadas sobre FileS2 AFL ha permitido obtener mejores resultados.

AFL ha sido capaz de detectar fallos válidos para la plataforma de Rode0day demostrando la utilidad de los *fuzzers* basados en instrumentación. El número de ejecuciones por segundos obtenido por este *fuzzer* son las más reducidas pero observando los resultados se puede concluir que las pruebas han sido de mayor calidad.

En el caso de Honggfuzz, el programa se ha podido instrumentar correctamente pero los resultados no son nada favorables ya que no ha sido capaz de encontrar ningún fallo. En el número de ejecuciones por segundo supera ligeramente a AFL pero los resultados finales no son los esperados.

Finalmente, Libfuzzer podría haber sido un buen candidato para encontrar fallos sobre el sistema de pruebas pero, debido a la gran cantidad de trabajo que supone adaptar el código del proyecto y teniendo en cuenta los resultados obtenidos por AFL, se ha concluido que Libfuzzer no es la herramienta más óptima para este programa.

A continuación se muestra una tabla comparativa con los resultados obtenidos por cada uno de los *fuzzers*:

	AFL	Honggfuzz	Libfuzzer
Ejecuciones/s	800	865	-
Fallos	977	0	-
Fallos únicos	149	0	-

Cuadro 5.3: Comparación de los resultados para FileS2

5.3. Optimizaciones

En la actualidad, es común que los procesadores ofrezcan la posibilidad de emplear núcleos virtuales gestionados directamente por el sistema operativo. Esto quiere decir que por cada núcleo del procesador real, se obtendrían dos núcleos virtuales. Esta implementación de los procesadores se conoce como *Hyper-threading* [Int].

En los procesos de *fuzzing* se depende totalmente del procesador por lo que su implementación está directamente asociada a su rendimiento. Durante las pruebas realizadas se ha empleado un procesador AMD Ryzen con ocho núcleos físicos que al emplear *Hyper-threading* se convierten en dieciséis núcleos virtuales. Los núcleos virtuales del sistema operativo comparten los recursos de ejecución como son el motor de ejecución, las memorias caché y los buses.

En este proyecto el *Hyper-threading* ofrece la posibilidad de disponer de más núcleos pero que puede suponer un problema al compartir recursos. Para optimizar el uso del procesador se debe conocer qué núcleos son hermanos (*siblings*) y decidir si es conveniente emplear únicamente los núcleos físicos.

Los procesadores AMD gestionan los *siblings* de forma distinta a como lo hace Intel. Para conocer que núcleos virtuales se encuentran en el mismo núcleo físico se puede ejecutar el siguiente comando:

```
1 cat /sys/devices/system/cpu/cpu[0-9]*/topology/thread_siblings_list |
  ↪ sort -nu
```

El comando anterior permitirá conocer que núcleos son hermanos y comparten recursos. En el caso de Intel los núcleos suelen asociarse de forma no consecutiva mientras que en el caso de los procesadores AMD se asignan como consecutivos. Una limitación detectada de AFL con procesadores AMD se debe a que selecciona los núcleos virtuales en orden de disponibilidad por lo que dos procesos de AFL se ejecutarían consumiendo toda la capacidad de los dos núcleos virtuales disponibles en el mismo núcleo físico, compartiendo los recursos de ejecución. En el caso de un procesador Intel, AFL no contaría con esta limitación ya que no asigna los núcleos virtuales de forma consecutiva.

Una posible forma de solucionar esta restricción consiste en modificar el código de AFL para que tenga en cuenta esta limitación. Otra forma de gestionar los núcleos del

procesador sin necesidad de modificar AFL está relacionada con seleccionar el núcleo que se quiere emplear al iniciar una tarea. En Linux se puede seleccionar con el comando *taskset* que permite modificar el núcleo empleado por un proceso en ejecución. El comando para modificar un proceso de AFL con PID 17638 al núcleo cinco, teniendo en cuenta que los núcleos se numeran empezando por cero, sería el siguiente:

```
1 taskset -cp 4 17638
```

En el caso de Honggfuzz y Libfuzzer, la gestión de los núcleos virtuales de AMD no afecta ya que selecciona la capacidad total del procesador sin asignar el procesamiento a un núcleo único.

El modo persistente de los *fuzzers* permite mejorar el rendimiento de las pruebas mediante la repetición de las pruebas empleando un bucle interpretado por el *fuzzer* que se debe incluir en el código del sistema a probar. De esta forma, es posible evitar el tiempo que requiere iniciar cada proceso añadiendo las pruebas en una iteración del bucle que se va a instrumentar. En el modo persistente se debe tener especial cuidado en la manipulación que se haga de la memoria del proceso ya que, en caso de no reiniciar el estado, es posible que el *fuzzer* detecte fallos causados por trazas de pruebas anteriores, produciendo así falsos positivos.

5.4. Limitaciones

El *fuzzing* se ha demostrado como una técnica útil para la fase de pruebas de productos software. A pesar de ello, cuenta con una serie de limitaciones que si el sistema es lo suficientemente complejo, el investigador de seguridad va a necesitar aplicar mucho esfuerzo para conseguir éxito.

Los *fuzzers* se han empleado en la comunidad de los investigadores de seguridad como una herramienta inicial para encontrar fallos explotables en sistemas. De esta forma, una vez detectado el fallo, se debe analizar y conocer si es posible elaborar un *exploit* para comprometer el sistema. En muchos casos, es posible encontrar un fallo y que este no sea explotable. Por otro lado, los sistemas modernos cuentan con una gran cantidad de medidas de seguridad y es posible que la explotación de un determinado fallo no sea posible debido a que es necesario evadir todas estas medidas de seguridad. Una vez detectado el fallo, los requisitos para conseguir que sea explotable se incrementa y requiere de aplicar técnicas de ingeniería inversa que aumentan considerablemente el esfuerzo que debe realizar el investigador siendo posible que el tiempo dedicado no se materialice en la explotación del sistema.

Otras de las limitaciones con las que cuentan los *fuzzers* son aquellos fallos que no producen una señal de fallo del sistema operativo ya que no van a poder ser detectados. Un ejemplo de fallos no detectables son las elevaciones de privilegios, inyecciones SQL, la lectura arbitraria de ficheros, etcétera.

A pesar de ser proyectos con mucha funcionalidad y, en algunos casos, con una gran madurez, los *fuzzers* tratan de mostrar el progreso de las tareas de la mejor forma

posible pero, en algunos casos y dependiendo del *fuzzer*, la pantalla informativa no es lo suficientemente clara y es difícil conocer el progreso de la herramienta con exactitud.

A lo largo del proyecto se ha demostrado que los *fuzzers* pueden lograr el éxito y encontrar fallos realizando pruebas tanto de caja negra como de caja blanca o caja gris. Así, también ha quedado demostrado que los resultados obtenidos en el caso de disponer del código del sistema y pudiendo realizar la instrumentación del código son mejores por lo que, al emplear estas herramientas en sistemas complejos, se va a producir una gran dependencia del código. En muchos casos, como por ejemplo en los sistemas propietarios, no sera posible disponer del código de la aplicación y las pruebas estarán ligadas a la aplicación de técnicas de caja negra.

Capítulo 6

Conclusiones y trabajo futuro

Este proyecto se ha centrado en la detección de fallos en productos de software reales y a lo largo de las secciones de este documento ha sido posible mostrar la identificación de vulnerabilidades reales empleando *fuzzers* como AFL, Honggfuzz y LibFuzzer. El empleo de distintos *fuzzers* ha permitido evidenciar las ventajas y desventajas de cada uno de ellos así como posibles formas de mejorar tanto el rendimiento como los resultados.

Además de encontrar fallos en software real y detectar vulnerabilidades publicadas se han detectado fallos en las muestras publicadas en la competición de Rode0day en la que mediante la tecnología LAVA se inyectan fallos con el objetivo de poder validar la eficiencia y eficacia de los *fuzzers*.

Durante los procesos de *fuzzing* realizados ha sido posible conocer el funcionamiento de los *fuzzers* y como es posible mejorar los resultados empleando diccionarios con palabras clave que pueden aumentar la cobertura del código y descubrir nuevos caminos favorables en los sistemas en pruebas. Con una mayor cobertura sobre el código de una herramienta es posible detectar fallos que el propio *fuzzer*, sin contar con ayuda, podría no ser capaz de detectar.

La instrumentación del código y la monitorización continua del sistema en pruebas durante un proceso de *fuzzing* incrementa significativamente la calidad y los resultados de las pruebas. Contar con el código de una aplicación supone una gran ventaja con respecto a aquellos programas de los que solo se dispone del binario. La mayoría de los *fuzzers* incluyen un modo *dumb* para realizar pruebas de caja negra aunque es común que no sean capaces de superar ciertas condiciones complejas de la aplicación y permanezcan bloqueados hasta su interrupción. En el caso de AFL, estas limitaciones pueden reducirse empleando el modo usuario de QEMU permitiendo obtener información y mejorar el *fuzzing* en pruebas de caja negra.

Cuando no es posible instrumentar el código, Honggfuzz permite aprovechar el procesador para recuperar información de cada una de las ejecuciones de pruebas. Estas técnicas aprovechan especialmente registros de los procesadores Intel por lo que cuentan con ventaja sobre otros tipos de procesadores. AFL no dispone de la instrumentación hardware pero cuenta con una modificación de QEMU que, a pesar de afectar considerablemente al rendimiento, permite ofrecer información al *fuzzer* y mejorar los resultados

obtenidos en pruebas de caja negra. Algunos investigadores han sido capaces de optimizar el funcionamiento de QEMU e incrementar el número de ejecuciones por segundo hasta por cinco veces [Bio]. Por otro lado, LibFuzzer se centra únicamente en pruebas de caja blanca en las que es posible instrumentar el código aunque existe la posibilidad de emplear técnicas para adaptar el uso de Libfuzzer en pruebas de caja negra.

Libfuzzer ha demostrado una gran eficiencia a la hora de analizar binarios aunque, a la hora de realizar pruebas de *fuzzing* sobre algún programa, también se debe tener en cuenta el esfuerzo que requiere adaptar el código para ello. Tal como se ha mostrado en la sección 4.1.3, existen herramientas como LIEF capaces de modificar un determinado binario para poder acceder a una función y emplearla haciendo uso de un enlace dinámico, ahorrando gran cantidad de tiempo. En otras ocasiones no va a ser viable cargar un binario con un enlace dinámico y será necesario adaptar gran cantidad de código para poder centrar las tareas en determinadas funciones, como se ha podido observar en la sección 4.2.2. Así, se puede concluir que dependiendo el sistema sobre el que se quieran hacer las pruebas el *fuzzer* a elegir dependerá también del propio programa, realizando una valoración de si una pérdida de eficiencia es rentable si es posible ahorrar tiempo en la adaptación del código del programa. A pesar de que en el caso del programa FileS2 no sea la mejor opción sus resultados y su rendimiento son prometedores.

En las pruebas realizadas con Libfuzzer se ha podido emplear la herramienta LIEF que ha demostrado tener unos resultados favorables. Esta herramienta se puede integrar en cualquier proceso de *fuzzing* siendo posible ahorrar tiempo a la hora de realizar adaptaciones para iniciar las pruebas con cualquier *fuzzer*.

Los casos de prueba iniciales son uno de los aspectos más importantes de las pruebas. Proveer al *fuzzer* con unos casos de prueba robustos y de acuerdo a la funcionalidad que desempeña cada programa puede reducir el tiempo necesario para encontrar un fallo que de otra forma no podrían ser detectados. Debido a las alteraciones que realizan los *fuzzers* sobre estos casos de prueba, es importante que estén reducidos y optimizados evitando que el *fuzzer* sufra una perdida de rendimiento durante las modificaciones.

En este proyecto se ha analizado el comportamiento de los *fuzzers* detectando que las ejecuciones por segundo fluctúan según la etapa en la que se encuentre el proceso de pruebas. Los *fuzzers* realizan alteraciones deterministas y no deterministas sobre los casos de prueba iniciales. Las modificaciones que se van realizando sobre los ficheros de entrada alteran e incrementan el tamaño de los casos de prueba por lo que, en la fase de pruebas no deterministas, el tamaño de los ficheros puede aumentar considerablemente, produciendo la caída del número de ejecuciones por segundo. Para evitar estas fluctuaciones se deben optimizar lo máximo posibles los casos de prueba iniciales. A pesar de que se pueda producir una caída del número de ejecuciones por segundo, la calidad del *fuzzing* no está ligada únicamente a este indicador ya que existen valores más importantes como son los caminos encontrados, los caminos favorables encontrados o la estabilidad.

Los procesadores toman un papel fundamental en el *fuzzing* y debe tenerse en cuenta el *Hyper-threading* para optimizar lo máximo posible sus capacidades. Los procesadores modernos incluyen *Hyper-threading* con lo que es posible incrementar el número de

núcleos por parte del sistema operativo. De esta forma, un núcleo físico puede dividirse en dos virtuales compartiendo recursos de ejecución como la memoria caché o los buses por lo que bajará el rendimiento. En procesos de *fuzzing* es conveniente gestionar los núcleos libres para que no se disponga de núcleos reales libres y se ocupen núcleos virtuales. Según como gestione el *fuzzer* la capacidad de computo, puede ser conveniente deshabilitar el *Hyper-threading* y trabajar únicamente con núcleos reales.

A lo largo de este proyecto se han empleado *fuzzers* de propósito general que establecen las bases para la creación de nuevos *fuzzers* más específicos. Estos proyectos tienen una gran complejidad y muestran resultados favorables que permiten considerarlos como herramientas de vital importancia en la detección de fallos. Además, los *fuzzers* de propósito general son la evidencia de que las pruebas de *fuzzing* funcionan y deben incluirse en el desarrollo de cualquier proyecto. Como toda herramienta, los *fuzzers* tienen limitaciones y no es posible emplearlos de forma óptima en cualquier producto software pero pueden servir para sentar las bases para la creación de *fuzzers* específicos para la realización de pruebas sobre programas especiales como los navegadores.

Bibliografía

[AFDG] Rahul Sridhar Andrew Fasano, Tim Leek and Brendan Dolan-Gavitt. Rode0day, a continuos bug finding competition. https://rode0day.mit.edu.

[BDG] Engin Kirda Tim Leek Andrea Mambretti Wil Robertson Frederick Ulrich Ryan Whelan Brendan Dolan-Gavitt, Patrick Hulin. Lava: Large-scale automated vulnerability addition. Technical report. https://rode0day.mit.edu/static/lava.pdf.

[Bel] Fabrice Bellar. Qemu emulador open-source. https://www.qemu.org/.

[Bio] Andrea Biondo. Improving afl's qemu mode performance. https://abiondo.me/2018/09/21/improving-afl-qemu-mode/.

[Cat] Hugsy Blah Cats. Fuzzing arbitrary functions in elf binaries. https://blahcat.github.io/2018/03/11/fuzzing-arbitrary-functions-in-elf-binaries/.

[Dat] National Vulnerability Database. Cve-2009-0159. https://nvd.nist.gov/vuln/detail/CVE-2009-0159.

[DoDAGRMG] Duncan Grove Department of Defense. Australian Goverment. Richard McNally, Ken Yiu and Damien Gerhardy. Fuzzing: The state of the art. http://www.dtic.mil/dtic/tr/fulltext/u2/a558209.pdf.

[Gooa] Google. Address sanitizer, asan. https://github.com/google/sanitizers/wiki/AddressSanitizer.

[Goob] Google. Honggfuzz, security oriented fuzzer. https://github.com/google/honggfuzz.

[Hje] Isak Hjelt. The future of grey-box fuzzing. https://pdfs.semanticscholar.org/e83d/539cbab9ae4906c49e6daae8c6f952713c17.pdf.

[Imm] Dave Aitel. Immunity. An introduction to spike, the fuzzer creation kit. https://www.blackhat.com/presentations/Fbh-usa-02/bh-us-02-aitel-spike.ppt.

[Int] Intel. Intel hyperthreading techonology. https://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html.

[MIT] MITRE. Vulnerabilidad heartbled (cve-2014-0160). https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160.

[mp] Linux man page. File command, determine file type. https://linux.die.net/man/1/file.

[MZ] lcamtuf Michał Zalewski. American fuzzy lop, security oriented fuzzer. http://lcamtuf.coredump.cx/afl/.

[oD] University of Delaware. Ntp versión 4. https://www.eecis.udel.edu/~ntp/ntp_spool/ntp4/ntp-4.2/.

[oIa] University of Illinois. Libfuzzer, a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

[oIb] University of Illinois. Llvm compiler infrastructure. https://llvm.org/.

[oIc] University of Illinois. Sanitizercoverage, a simple code coverage instrumentation component. http://clang.llvm.org/docs/SanitizerCoverage.html.

[Qua] Quarkslab. Lief, a library to instrument executable formats. https://lief.quarkslab.com/doc/latest/index.html.

[Vaa] Sami Vaarala. Duktape - embeddable javascript engine with a focus on portability and compact footprint. https://duktape.org/.

Apéndice A

Diccionario de términos

En este anexo se establecen los conceptos fundamentales que se mencionan a lo largo de todo el proyecto. Es conveniente realizar la lectura del proyecto apoyándose sobre las siguientes definiciones:

- **Sistema en pruebas**: Programa sobre el que se van a realizar las pruebas de *fuzzing*.
- **Fuzz**: Proceso de generar datos de entrada aleatorios o pseudo-aleatorios que se van a enviar a un sistema en pruebas.
- **Fuzzer**: Herramienta empleada para procesador y aplicar modificaciones sobre las entradas recibidas y que serán enviadas a un sistema en pruebas.
- **Cobertura**: Cantidad de código que el *fuzzer* es capaz de abarcar y sobre el que se aplicarán las pruebas.
- **Instrumentación**: Hecho de añadir componentes de monitorización por el *fuzzer* en tiempo de compilación y que van a permitir mejorar el proceso dotando de cierta inteligencia a la herramienta.
- **Casos de pruebas o testcases**: Ficheros de entrada proporcionados o auto-generados por el propio *fuzzer* que serán entregados al sistema en pruebas.
- **Profundidad**: Condiciones o niveles superados por el *fuzzer* durante la realización de las pruebas según un determinado caso de prueba.
- **Camino**: Consecución de condiciones distintas que el *fuzzer* puede identificar según un determinado caso de prueba.
- **Diccionario**: Listado de palabras que el *fuzzer* puede emplear para incluir en los casos de pruebas.
- **Modo persistente**: Introducción de un bucle en el código del sistema en pruebas previo a la instrumentación de forma que es posible optimizar el proceso debido a que no es necesario reiniciar el proceso por cada prueba.

- **Caja negra o black box**: Pruebas sobre un sistema del que no se tiene información.
- **Caja blanca o white box**: Pruebas sobre un sistema del que se dispone del código.
- **Caja gris o grey box**: Pruebas sobre un sistema del que se tiene algo de información, como por ejemplo un listado de tipos de archivos que procesa.

Apéndice B

Planificación

Tal como estipula el Plan Bolonia (BOE-A-2007-18770) y con el reconocimiento del Trabajo Final de Master (TFM) con 8 créditos ECTS, el proyecto debe tener asignado un total de 200 horas asignando 25 horas por cada crédito ECTS tal y como reconoce la Universitat Oberta de Catalunya. Dado que el proyecto se realizará durante el primer semestre del año académico 2018-2019, se le deben asignar entre 12 y 16 horas semanales aproximadamente.

La mayor parte del tiempo del proyecto se ha asignado al estudio de los algoritmos empleados por las herramientas de *fuzzing*, a la instalación de dichas herramientas en un equipo de la organización, a la recolección de los binarios sobre los que se aplicarán las pruebas, a las pruebas sobre los binarios seleccionados y al análisis e interpretación de los resultados obtenidos por cada *fuzzer*.

Finalmente, el tiempo total dedicado se ha podido dividir entre los tres *fuzzers*: AFL, Honggfuzz y Libfuzzer. Además del tiempo dedicado a las pruebas y a la detección de fallos, también es importante considerar el tiempo dedicado a la documentación. En la figura B.1 se muestra un gráfico que resume el tiempo dedicado a cada elemento del proyecto.

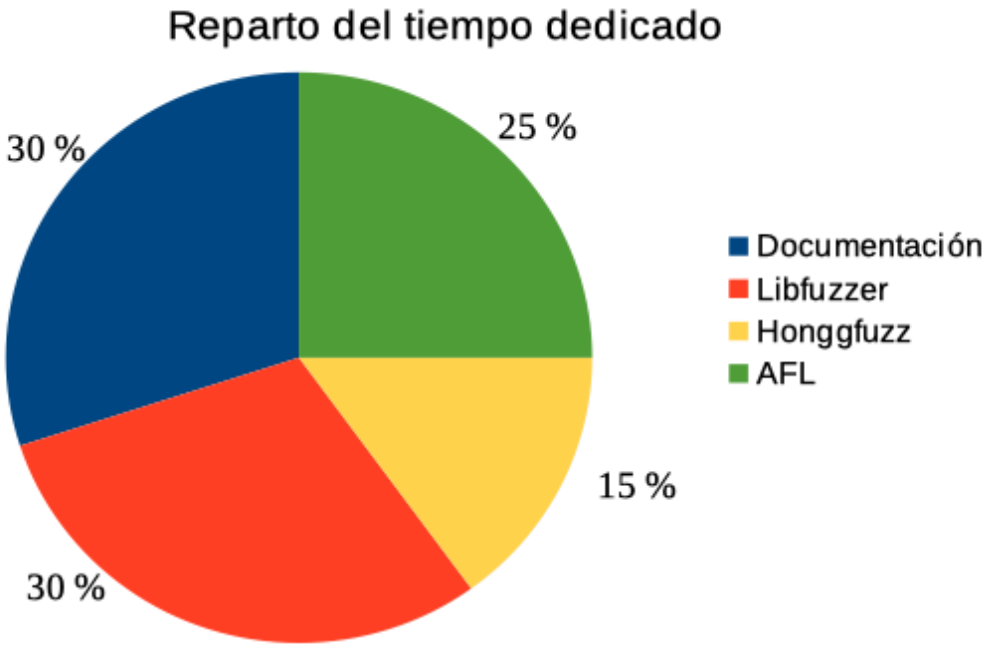


Figura B.1: Reparto del tiempo dedicado

Apéndice C

Estabilidad en AFL

Este anexo muestra los motivos por los por los que el valor del parámetro *stability* de AFL, en la esquina inferior derecha, pueden variar. Si el binario funciona correctamente se debería obtener una estabilidad del 100 % pero en algunas situaciones esto puede variar. Los motivos por lo que esto puede ocurrir son los siguientes:

- El empleo de memoria sin inicializar en conjunto con fuentes intrínsecas en el binario. En principio es inofensivo para AFL y podría ser indicativo de un fallo de seguridad.
- El intento de manipulación de recursos persistentes, como ficheros temporales u objetos compartidos en memoria. No es dañino pero se debe comprobar que el fuzzer no está fallando de forma prematura.
- Cuando se accede a funcionalidades diseñadas para comportarse de forma aleatoria. No afecta a AFL ya que es el comportamiento normal. Un ejemplo sería cuando se emplean funciones random().
- Cuando se emplean múltiples *threads* ejecutándose a la vez en un orden semi-aleatorio. Este problema es inofensivo mientras la estabilidad se mantenga por encima del 90 %. Este problema se puede solucionar de las siguientes maneras:
  - Empleando afl-clang-fast en el modo LLVM que incluye un modelo de análisis de *threads* propio que no es propenso a tener tantos problemas de concurrencia.
  - Comprobar si el binario objetivo puede ejecutarse sin *threads*. Muchos ficheros de configuración incluyen opciones para ello, como —without-threads, —disable-pthreads o —disable-openmp.
  - Reemplazar *pthreads* con GNU path que permite la opción de emplear un *scheduler* determinista.
- En el modo persistente, tener caídas en la estabilidad es normal ya que no todo el código se comporta de forma idéntica en el bucle __AFL_LOOP() aunque si la

estabilidad cae en exceso puede ser indicativo de que no se reinicia por completo el estado en cada iteración.

Apéndice D

Componentes de AFL

En este apéndice se incluyen un conjunto de herramienta que forman parte de AFL, conocidas como afl-utils, y que permiten optimizar y automatizar tareas trabajando con AFL. Para cada una de ellas se incluye un descripción de las tareas que pueden realizar.

- **afl-analyze:** Recibe un fichero de entrada e intenta de forma secuencial alterar bytes y observar el comportamiento del programa en pruebas. Después, incluye códigos de colores en el fichero de entrada en los cuales la sección es crítica y cuales no.
- **afl-cmin:** Esta herramienta intenta encontrar el subconjunto más pequeño de ficheros en el directorio de entradas que permita disparar el rango total de datos de los puntos instrumentados que han sido vistos en el conjunto inicial. Si se dispone de un conjunto de entrada de datos grande disponible para mostrar por pantalla, afl-cmin puede emplearse para eliminar ficheros redundantes. Hay que tener en cuenta que la herramienta no modifica los ficheros por si solo, para eso es necesario emplear afl-tmin.
- **afl-collect:**Recopila todos los ficheros con muestras de fallos de un directorio de trabajo de afl-fuzz (empleado por múltiples instancias de afl funcionando en paralelo) y los copia en un directorio único de fácil acceso para futuros análisis de fallos. Dispone de opciones más avanzadas como la eliminación de muestras de fallos inválidos.
- **afl-cron:** El propósito principal de afl-cron consiste en ejecutar múltiples herramientas de AFL de forma periódica.
- **afl-gotcpu:** Esta herramienta ofrece una medida bastante acertada de la prioridad del procesador. Es un complemento para la sección de la carga de la interfaz de afl-fuzz.
- **afl-minimize:** Ayuda a crear casos de prueba minimizados de muestras de una tarea de *fuzzing* en pararelo. Funciona de la siguiente manera:

1. Recolecta todas las muestras del directorio queue de un proceso de afl sincronizado.
2. Ejecuta afl-cmin de las muestras recogidas y almacena las muestras minimizadas.
3. Ejecuta afl-tmin en el resto de muestras para reducir su tamaño y almacena los resultados.
4. Realiza un comprobación para cada muestra y mueve los fallos o caídas del programa fuera de las muestras. Esta fase es útil antes de iniciar una nueva tarea paralelizada o antes de restaurar una sesión de afl-fuzz. Las muestras que puedan provocar un fallo en el programa objetivo se desplazarán al directorio *crashes*.

- **afl-multicore:** Inicia varias tareas de *fuzzing* en paralelo en *background* empleando *nohup*.
- **afl-multikill:** Finaliza todas las instancias de afl-fuzz pertenecientes a una sesión activa de afl-multicore no interactiva.
- **afl-plot:** Permite generar imágenes con *gnuplot* de los datos de salida. Creará tres archivos PNG y un archivo HTML que incluya los tres gráficos.
- **afl-showmap:** Herramienta que ejecuta el binario objetivo y muestra el contenido de la traza de *bitmap* de forma legible. Es útil para *scripts* para eliminar entradas redundantes y realizar otras comprobaciones. El código de salida es dos si el programa objetivo falla; uno si se produce un *timeout* y hay un problema ejecutándolo; o cero si la ejecución es exitosa.
- **afl-stats:** Muestra por pantalla estadísticas de *fuzzing* similares a la salida que genera afl-whatsup y opcionalmente publica *tweets* en Twitter con esta información. Esta herramienta es especialmente útil cuando se están realizando tareas de *fuzzing* en múltiples máquinas.
- **afl-sync:** Permite distribuir las entradas de múltiples instancias de afl-fuzz en distintos nodos. Permite realizar *backup*, restaurar o sincronizar directorios de trabajo de afl-fuzz a una máquina, desde una máquina o con una máquina remota.
- **afl-tmin:** Herramienta empleada para minimizar los casos de prueba. Selecciona un fichero de entrada e intenta eliminar tantos datos como sea posible mientras se intenta mantener los datos que permiten provocar el fallo o producir una salida consistente.
- **afl-vcrash:** Verifica que los fallos detectados por afl-fuzz realmente provocan un fallo en el binario objetivo y opcionalmente elimina las que no producen el fallo automáticamente.

- **afl-whatsup:** Resume el estado de cualquier instancia local sincronizada de afl-fuzz. Comprueba si el *fuzzer* está vivo, muestra el tiempo total de ejecución, el número de ejecuciones, los caminos pendientes y el número de fallos encontrados.

Apéndice E

Código para detectar la vulnerabilidad de NTP con Libfuzzer

El código empleado para realizar pruebas sobre la función cookedprint se ha obtenido del blog blahcat [Cat] y se ha adaptado para las pruebas de detección de la vulnerabilidad en NTP. En la sección 4.1.3 se realiza una explicación de las secciones más relevantes del programa.

El código adaptado es el siguiente:

```
1  /**
2   * Adaptación del código del blog https://blahcat.github.io/
3   * Realizado por @_hugsy_
4   */
5  #include <dlfcn.h>
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <stdint.h>
9  #include <alloca.h>
10 #include <string.h>
11
12 typedef int(*cookedprint_t)(int, int, char *, int, FILE *);
13
14 int is_loaded = 0;
15 void* h = NULL;
16
17 void CloseLibrary()
18 {
19         if(h){
20                 dlclose(h);
```

```
21                  h = NULL;
22          }
23          return;
24  }
25
26  #ifdef USE_LIBFUZZER
27  extern "C"
28  #endif
29  int LoadLibrary()
30  {
31          h = dlopen("./ntpq.so", RTLD_LAZY);
32          atexit(CloseLibrary);
33          return h != NULL;
34  }
35
36  #ifdef USE_LIBFUZZER
37  extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t
    ↪  Size)
38  #else
39  int main (int argc, char** argv)
40  #endif
41  {
42          char* code;
43          char* ptr = NULL;
44          int datatype, length, status;
45          FILE *fp;
46
47          if (!is_loaded){
48              if(!LoadLibrary()){
49                  return -1;
50              }
51              is_loaded = 1;
52          }
53
54  #ifdef USE_LIBFUZZER
55          if(Size==0)
56              return 0;
57  #else
58      if(argc < 2)
59          return 0;
60          char *Data = argv[1];
61          size_t Size = strlen(argv[1]);
62  #endif
```

```
63
64      if (Data[Size-1] != '\x00')
65      {
66          code = (char*)malloc(Size+1);
67          memcpy(code, Data, Size+1);
68          code[Size]='\x00';
69      }
70      else{
71          code = (char*) malloc(Size);
72          memcpy(code, Data, Size);
73      }
74          datatype = Data[0];
75          status = Data[1];
76          length = Size;
77
78          cookedprint_t cookedprint = (cookedprint_t)dlsym(h,
            ↪  "cookedprint");
79
80          cookedprint(datatype, length, code, status, stderr);
81          free(code);
82          return 0;
83  }
```

Apéndice F

Script de subida de fallos a Rode0day

En este apéndice se incluye el código empleado para automatizar la subida de fallos a Rode0day. El script recibe tres parámetros: la ruta de un fichero o un directorio con fallos, una clave de la API y el identificador de un reto. Conforme se realice la subida de ficheros se irán mostrando aquellos fallos que suman puntos junto a la puntuación conseguida total tras la aceptación de un determinado fallo.

```
1  import requests, optparse, time
2  from os import listdir
3  from os.path import isfile
4  from os.path import isdir
5
6  def main(file=None, dir=None, challenge_id=None, key=None):
7      if file is not None and isfile(file):
8          response = send_request(file, challenge_id, key)
9          parse_response(response)
10     elif dir is not None and isdir(dir):
11         if not dir.endswith('/'):
12             dir = dir + '/'
13         score = get_score(challenge_id,key)
14         directory_list = listdir(dir)
15         for file in directory_list:
16             response = send_request(dir + file, challenge_id, key)
17             score = parse_response(file, response, score)
18
19 def get_score(challenge_id, key):
20     #Send an empty file to get score
21     empty_file = 'empty.txt'
22     f = open(empty_file,'w+')
```

```
23      f.close()
24
25      score = 0
26      response = send_request('empty.txt', challenge_id, key)
27      for line in response.splitlines():
28          line = line.split(":")
29          if 'score' in line[0]:
30              score = line[1].replace(" ","")
31          if 'remain' in line[0]:
32              print "Requests available: %s" % (line[1].replace(" ",
                ↪  ""))
33      return score
34
35  def send_request(file, challenge_id, key):
36      response = ""
37      url = 'https://rode0day.mit.edu/api/1.0/submit'
38
39      multipart = {'input': open(file,'rb')}
40      data = {'challenge_id': challenge_id, 'auth_token': key}
41      r = requests.post(url, data=data, files = multipart)
42      #print file
43      if r.status_code == 200:
44          response = r.text
45      time.sleep(1)
46      return response
47
48  def parse_response(file, response, score):
49      new_score = score
50      for line in response.splitlines():
51          line = line.split(":")
52          if 'score' in line[0]:
53              tmp_score = line[1].replace(" ", "")
54              if score < tmp_score:
55                  new_score = tmp_score
56                  print "[*] Valid rode0day crash. File = %s - Score =
                    ↪  %s" % (file, new_score)
57      return new_score
58
59
60  def print_help(parser):
61      parser.print_help()
62      exit(-1)
63
```

```
64 if __name__ == "__main__":
65     parser = optparse.OptionParser()
66     parser.add_option('-f', '--file', action="store", help="File
       ↪  containing the crash to submit", dest="file", type="string")
67     parser.add_option('-d','--dir', action="store", help="Directory
       ↪  with crashes to submit", dest="dir",type="string")
68     parser.add_option('-k','--key', action="store", help="API key",
       ↪  dest="key", type="string")
69     parser.add_option('-c','--challenge', action="store",
       ↪  help="Challenge ID (included in info.yaml)",
       ↪  dest="challenge_id", type="string")
70     (opts, args) = parser.parse_args()
71     if (opts.file is not None or opts.dir is not None) and opts.key is
       ↪  not None and opts.challenge_id is not None:
72         main(opts.file, opts.dir, opts.challenge_id, opts.key)
73     else:
74         print_help(parser)
```