



UNIVERSIDAD DE JAÉN
Escuela Politécnica Superior (Jaén)

Trabajo Fin de Grado

DESARROLLO DE HERRAMIENTA SOFTWARE PARA DETECTAR VULNERABILIDADES MEDIANTE LA INYECCIÓN DE DATOS ALEATORIOS Y SECUENCIALES

Alumno: Andrea Serrano Urea

Tutor: Prof. D. Manuel José Lucena López

Dpto: Informática

Junio, 2018



Universidad de Jaén
Escuela Politécnica Superior de Jaén
Departamento de Informática

Don Manuel José Lucena López, tutor del Proyecto Fin de Carrera titulado:
Desarrollo de herramienta software para detectar vulnerabilidades mediante la
inyección de datos aleatorios y secuenciales, que presenta Andrea Serrano Urea,
autoriza su presentación para defensa y evaluación en la Escuela Politécnica
Superior de Jaén.

Jaén, Junio de 2018

El alumno:

Los tutores:

Andrea Serrano Urea

Manuel José Lucena López

Índice

Índice de figuras	5
Índice de tabas	6
1. Introducción.....	7
1.1. Objetivos	7
1.2. Motivación.....	7
1.3. Resultados esperados.....	8
2. Fuzzing.....	9
2.1. Definición	9
2.2. Etapas.....	9
2.3. Componentes	10
2.4. Tipos de generadores.....	11
2.5. Ventajas	12
3. Estructura y planificación del proyecto	13
3.1. Metodología	13
3.2. Tareas	14
3.3. Estimación de costes.....	14
4. Tecnologías utilizadas	16
4.1. Lenguaje de programación.....	16
4.2. Entorno de desarrollo	16
4.3. Bibliotecas utilizadas	16
Tkinter	18
PyQt	19
PyGTK.....	19
PyForms	19
Electron	20
5. Diseño del proyecto	20
5.1. Especificación de requisitos	20
5.2. Diagrama de casos de uso.....	21
5.3. Diagrama de actividad.....	22
6. Desarrollo del proyecto.....	22
6.1. Primera iteración	22
6.2 Segunda iteración.....	25

El protocolo HTTP y las peticiones GET	25
Implementación.....	26
Pruebas con Electron.....	30
6.3 Tercera iteración.....	31
Tipos de vulnerabilidades	31
Implementación.....	34
6.4 Cuarta iteración	38
Códigos HTTP.....	38
Implementación.....	40
6.5 Quinta iteración	43
6.6 Sexta iteración	47
File format fuzzing.....	48
Preparación del entorno e integración del fuzzer	48
Desarrollo del fuzzer	50
6.7 Séptima iteración.....	51
Objetivos	51
Formato del estudio	51
Perfil de los sujetos de pruebas	52
Pruebas a realizar	53
Métricas a las que se atenderá	54
Resultados	54
Conclusiones.....	55
7. Conclusiones.....	55
8. Anexos	57
8.1. Manual de uso	57
8.2. Manual del programador	57
Estructura del programa.....	58
Archivo de configuración (config.json)	58
La clase Fuzzer	58
Crear un fuzzer paso a paso	60
Bibliografía.....	64

Índice de figuras

FIGURA 1. DIAGRAMA DE ETAPAS DE FUZZING.	10
FIGURA 3. CASO DE USO “ANALIZAR APLICACIÓN”	21
FIGURA 4. DIAGRAMA DE ACTIVIDAD	22
FIGURA 6. ASPECTO DEL MENÚ A TRAVÉS DE LA CONSOLA.....	23
FIGURA 7. PROTOTIPO PARCIAL HECHO CON TKINTER	25
FIGURA 8. PETICIÓN GET.....	26
FIGURA 9. DETECCIÓN DE PARÁMETROS GET EN URL	27
FIGURA 10. SEGMENTOS EN URL 1.....	29
FIGURA 11. SEGMENTOS EN URL 2.....	29
FIGURA 12. SEGMENTOS EN URL 3.....	29
FIGURA 13. PARÁMETROS MÁS COMUNES USADOS EN FORMAT STRING ATTACKS.....	33
FIGURA 14. NUEVO ARCHIVO DE CONFIGURACIÓN	36
FIGURA 15. NUEVA SECCIÓN PAYLOAD EN CLI.....	37
FIGURA 16. ASPECTO DEL FUZZER DE URL EN LA NUEVA ARQUITECTURA 1.....	47
FIGURA 17. ASPECTO DEL FUZZER DE URL EN LA NUEVA ARQUITECTURA 2.....	47
FIGURA 18. CÓDIGO ANTES DE AÑADIR FILE FORMAT FUZZER	49
FIGURA 19. CÓDIGO DESPUÉS DE AÑADIR FILE FORMAT FUZZER	49
FIGURA 20. CONTENIDO DE LA CARPETA FUZZERS DESPUÉS DE AÑADIR EL FUZZER SAMPLE	60

Índice de tabas

TABLA 1. PLANIFICACIÓN GENERAL Y ESTIMACIÓN DEL TIEMPO	14
TABLA 2. PRESUPUESTO HARDWARE.....	15
TABLA 3. LIBRERÍAS EXTERNAS Y JUSTIFICACIÓN DE USO.....	17
TABLA 5. OPCIONES DE LANZAMIENTO DE LA APLICACIÓN DESDE LA CONSOLA DE COMANDOS.....	22
TABLA 4. CARACTERES EN RUTAS SEGÚN SO.....	32
TABLA 5. VULNERABILIDADES QUE SE VAN A BUSCAR Y EN QUÉ PARTES DE LA URL	34

1. Introducción

1.1. Objetivos

Durante el presente proyecto va a desarrollar una herramienta para la detección de vulnerabilidades en otras aplicaciones mediante la técnica de fuzzing, es decir, inyección de datos malformados en masa para buscar defectos en el control de las entradas.

Aunque existen numerosas herramientas que emplean esta técnica, no son muchas las que resultan fáciles y rápidas de configurar y ejecutar, especialmente para aquellos programadores que no están familiarizados con el testing. Este proyecto pretende constutir una herramienta de fácil uso para dichos programadores, acercándolos así a esta práctica de manera amigable.

1.2. Motivación

Uno de los aspectos más importantes a la hora de desarrollar software es controlar de manera concienzuda los datos que recibimos por parte de los usuarios, no solo para prevenir errores ingenuos sino también la explotación maliciosa de los mismos. Sin embargo, dada la magnitud de los programas que se desarrollan hoy en día, es difícil y costoso realizar comprobaciones exhaustivas de las entradas (*inputs*) de nuestros programas. Esto convierte cada interacción del usuario con el sistema en un potencial vector de ataque.

Es por esto que surge la necesidad de disponer de un método automatizado en la medida de lo posible para asegurarnos de que las entradas de nuestro programa son robustas, incluso si reciben datos no previstos. Y no solo automatizado, sino además exhaustivo, ya que para un humano es complicado *imaginar* todas las combinaciones de datos. De este concepto nace la técnica *fuzzing testing*, una técnica black-box que trata de encontrar bugs en implementaciones inyectando datos aleatorios o deliberadamente incorrectos ¹.

El primer *fuzzer* fue desarrollado en 1989 por Barton Miller y algunos de sus estudiantes en la Universidad de Wisconsin. Su objetivo era comprobar la robustez de algunas aplicaciones para UNIX, sin preocuparse de implicaciones a nivel de seguridad. A día de hoy se puede encontrar su trabajo en <http://www.cs.wisc.edu/~bart/fuzz/>.² No fue hasta 1999 que esta técnica ganó popularidad. En la Universidad de Oulu nació PROTOS, que consistía en una serie de *test suites* destinados a comprobar la seguridad de ciertos protocolos. PROTOS encontró muchas vulnerabilidades, lo cual dio un empujón a la fama del *fuzzing testing*.³

Desde entonces han ido surgiendo herramientas y frameworks para realizar *fuzzing testing*. La mayoría de estas herramientas tienden a ser complicadas de configurar, ya que es difícil combinar una técnica que ofrece tantas posibilidades con una interfaz sencilla y válida para cualquier aplicación objetivo. En general, se centran en proporcionar utilidades que ayuden a los programadores a crear su propio *fuzzer*, lo cual establece de entrada una barrera de conocimiento y de tiempo al programador.

Para contrarrestar dichas barreras, el propósito de este proyecto es desarrollar una aplicación de escritorio de uso simple que realice *fuzzing testing* a aplicaciones de escritorio y web. Además, la arquitectura de la aplicación se enfocará para que la incorporación de módulos personalizados sea sencilla y amplíe la funcionalidad inicial.

1.3. Resultados esperados

Al término del proyecto se espera tener una aplicación de escritorio que permita realizar *fuzzing testing* empleando un tiempo razonablemente corto en su aprendizaje y configuración. Además, en caso de que el objetivo sea un sitio web, se comprobará su vulnerabilidad frente a técnicas ya conocidas como inyección SQL y XSS.

Los resultados de cada ejecución variarán de formato y contenido, pero normalmente consistirán en uno o varios archivos de log donde se especifica cada prueba y su resultado.

Por último, la integración de nuevos módulos de *fuzzing* deberá ser intuitiva para el programador que quiera extender la funcionalidad de esta herramienta, alargando así su vida útil.

2. Fuzzing

2.1. Definición

Fuzzing se describe como un método para descubrir vulnerabilidades en el software mediante la inyección de datos inesperados y monitorizando las excepciones y reacciones del programa. Esto se realiza normalmente de manera automatizada o semi automatizada.⁴ Pese a que es un enfoque muy simple, esta técnica ha mostrado ser ampliamente efectiva para descubrir vulnerabilidades en un amplio rango de sistemas.

Es importante señalar que esta técnica es bastante útil como primer análisis en el que se trata de cubrir vulnerabilidades básicas o indicios de las mismas, pero por sí mismo no suele dar resultados directos. La idea es detectar de manera rápida y automatizada posibles *crashes* e inestabilidades que más adelante pueden ser estudiados y explotados con mayor especialización.²⁶

2.2. Etapas

El proceso se divide normalmente en las siguientes fases⁶:

- **Identificación del objetivo.** En esta fase se estudia la aplicación que queremos poner a prueba para elegir la herramienta de fuzzing que más se adapte a nuestras necesidades.
- **Identificación de las entradas.** Se estudian los puntos por los que el usuario interactúa con la aplicación y qué clase de valores podrían desencadenar un fallo.
- **Generación de datos.** Una vez se conocen las entradas al programa, se preparan los datos para poner a prueba la aplicación. Se ha de escoger entre crear nuevos datos o utilizar vectores de ataque ya existentes. Sea cual sea el método, debería ser lo más automatizable posible.

- **Inyección de datos.** Es la fase donde se inyectan los datos, también de manera automática. Si no lo fuera, no estaríamos haciendo *fuzzing*.
- **Monitorización de excepciones.** Es fundamental recoger los resultados de la forma más explícita posible. Ante una excepción, se debe almacenar el momento, dato que la provocó, y toda la información que sea relevante para poder reproducir ese fallo y estudiarlo.
- **Determinar las capacidades de explotación.** Se analizan los errores detectados, su gravedad y su posible solución.

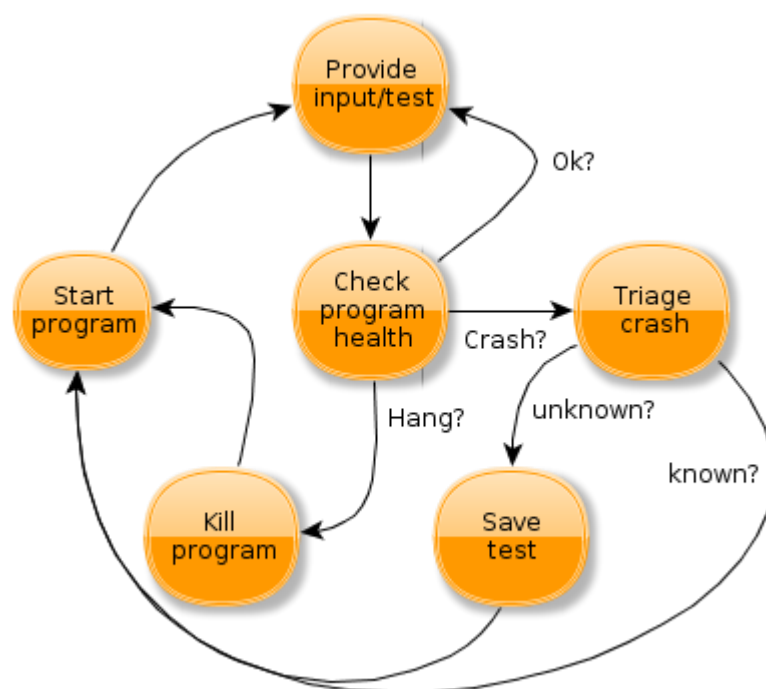


FIGURA 1. DIAGRAMA DE ETAPAS DE FUZZING.

En la Figura 2.1 podemos ver un diagrama que representa las fases de inyección y monitorización. Se puede ver que es un proceso iterativo y no requiere intervención humana una vez se ha iniciado el proceso.

2.3. Componentes

Un fuzzer puede ser normalmente descompuesto en tres componentes fundamentales: un generador *fuzz*, un mecanismo de inyección o envío de datos y un sistema de monitorización.¹⁷

- **Generador *fuzz*:** Se encarga de generar las entradas que se inyectarán en el Sistema a Prueba (SAP). El generador emite una serie de *test inputs* y estos se mandan al SAP mediante el mecanismo de inyección. Normalmente se intenta que el generador produzca salidas que sean apropiadas para el SAP, aunque esto puede lograrse desde diversas aproximaciones, cada una con sus ventajas e inconvenientes.
- **Sistema de inyección:** Recoge la salida del generador *fuzz* y lo introduce en el SAP para que sea procesada. La forma del sistema de inyección es totalmente dependiente del SAP.
- **Sistema de monitorización:** Observa cómo el SAP procesa las entradas que le llegan del sistema de inyección, tratando de registrar los errores que lanza (si hubiese).

2.4. Tipos de generadores

Existe una taxonomía¹⁹ que clasifica los *fuzzers* de acuerdo a dos ejes:

- Si los datos generados mediante la mutación de información existente o si parten totalmente de cero y se construyen mediante ciertas normas.
 - **Mutacional.** El *fuzzer* toma datos ya existentes y lo distorsiona, cambiando algunas partes para crear un nuevo *input set*. Si bien este tipo de *fuzzers* tienen mucha menos *inteligencia* que los generadores, también implican mucho menos esfuerzo humano para programarlos. Son adecuados cuando el SAP acepta datos altamente estructurados.
 - **Generativo.** Crea de cero nuevos *input sets*. El contenido puede variar desde lo puramente aleatorio hasta datos altamente estructurados. Se debe incorporar la máxima información posible sobre el SAP para obtener la máxima eficacia, lo que normalmente se traduce en un coste extra de esfuerzo.
- Tipo de *inteligencia* que usa el generador para mutar o generar datos.
 - **Inconsciente.** Ignora el tipo y la estructura de la información que el SAP espera recibir. El *fuzz generator* se limita a generar o mutar bytes aleatorios.

- **Basado en plantilla.** Se aplica un mínimo de conocimiento sobre la estructura aceptada por el SAP. Esta plantilla se utiliza para crear nueva información o malformar algún *test case* original.
- **Basado en bloques.** Se refiere a la representación de la información. Los fuzzers originales trataban los datos como simples cadenas de caracteres, mientras que los *fuzzers* basados en bloques representan toda la información como bloques de datos de varios tipos.
- **Basados en gramáticas.** Cubren al menos parte de lenguaje de entrada aceptado por el SAP. Las gramáticas normalmente se usan en *fuzzers* generativos, aunque son potencialmente aplicables en los mutacionales. Un ejemplo serían fuzzers que generan código fuente para poner a prueba algún compilador.
- **Basados en heurísticas.** Tratan de tomar decisiones *inteligentes* y no únicamente aleatorias. Típicamente tratan de explotar errores comunes como transformación entre Unicode/UTF-8/ASCII o *integer overflow*.

2.5. Ventajas

Esta técnica es una entre las muchas que hay para detectar vulnerabilidades, como la revisión de código manual, test unitarios o el *beta testing*. Por ello es interesante enumerar las ventajas que posee la técnica de *fuzzing testing* y en qué casos es más recomendable aplicarla¹⁸.

- **Funciona.** A lo largo de la historia se ha demostrado la sorprendente eficacia de este método a la hora de descubrir vulnerabilidades no obvias que habían sido pasadas por alto mediante otros métodos. Por ejemplo, un tercio de los bugs de Windows 7 que se descubrieron entre 2007 y 2010 fueron descubiertos por el SAGE *fuzzer* (bugs que pasaron inadvertidos en el proceso normal de control de calidad).

- **No requiere acceso al código fuente.** El enfoque más clásico es *black-box*, esto es, un ataque sin información del código de la aplicación. Es por ello que ha sido usado para descubrir vulnerabilidades *zero-day* en productos comerciales.
- **Es simple.** Es una técnica con un planteamiento extremadamente sencillo, que a su vez se puede extender en complejidad tanto como se necesite.
- **Los test manuales son caros.** Al ser una técnica automatizada, los costes disminuyen considerablemente. En especial si se comparan con el coste de desarrollar *test cases* manualmente.
- **Los humanos tenemos puntos ciegos.** La práctica de la ingeniería del software ha demostrado que los *test cases* son más efectivos si no los escribe el programador original, ya que un posible punto ciego durante el desarrollo puede reproducirse a la hora de pensar los tests. Por lo tanto, al utilizar una técnica automática y aleatoria cubrimos un amplio espectro de las posibles entradas del programa sin incurrir en un gasto extra de dinero, esfuerzo y tiempo por parte del equipo de desarrollo.
- **Portabilidad.** Normalmente un *fuzzer* sirve para varios sistemas.

3. Estructura y planificación del proyecto

3.1. Metodología

Este proyecto se realizará bajo la metodología Scrum. Scrum es una metodología de desarrollo ágil que se caracteriza por las siguientes características¹⁰:

- El desarrollo es incremental. Cada entrega del proyecto consiste en un producto funcionando con una funcionalidad nueva.
- Confianza en la auto organización.
- Las fases de desarrollo se solapan continuamente.

Se realizarán divisiones de las actividades en *sprints* con duración de 2 semanas. Para cada *sprint* se preparará una batería de tareas que deben ser completadas durante la duración del mismo. Dichas tareas dependen del resultado del *sprint* anterior, por lo que no se puede planificar con antelación.

3.2. Tareas

Las tareas presentadas en la Tabla 1 describen funcionalidades generales que la aplicación deberá tener.

Id	Descripción	Prioridad	Horas estimadas
1	Interfaz gráfica	6	80
2	Aplicar fuzz a URL con parámetros GET	10	60
3	Aplicar fuzz a aplicación de escritorio	8	95
4	Intérprete de resultados automatizado	7	45
5	Pruebas de usabilidad	6	15
6	Recopilación y creación de payloads con los que realizar los tests	9	15
7	Redacción de documentación técnica	7	30
8	Interfaz por línea de comandos	10	30
			370

TABLA 1. PLANIFICACIÓN GENERAL Y ESTIMACIÓN DEL TIEMPO

El tiempo de redacción de la memoria se ha contemplado al realizar la estimación de tiempo.

El proyecto se dividiría así en 8 grandes partes diferenciadas, que a su vez se descompondrán en pequeñas tareas o hitos que formarán parte de los sprints.

3.3. Estimación de costes

A continuación se realizará una estimación de costes de desarrollo y mantenimiento posterior.

3.1.1. Software

Los gastos de software son aquellos que se producen en la adquisición de licencias software.

- Microsoft Windows 10: 0€ (Adquirido mediante licencia de estudiante)
- PyCharm Community: 0€ (Versión gratuita del software PyCharm)
- Sublime Text: 0€ (Software libre)
- GitHub: 0€ (Evaluación gratuita ilimitada)

- Visual Paradigm: 0€ (Licencia de estudiante)

La suma total de los gastos de software son **0€** al trabajar con herramientas *open source* y con herramientas comerciales adquiridas mediante licencias de estudiante.

3.1.2. Hardware

El Hardware requerido para el desarrollo de este proyecto es un ordenador de sobremesa valorado en **800€** con un tiempo estimado de vida de 7 años. En él se instalará un servidor local en el cual alojaremos sitios web que sirvan para poner a prueba la herramienta en desarrollo. Además, será la misma estación de trabajo donde se desarrollará el *fuzzer*.

Al equipo se le debe sumar los costes de los periféricos necesarios para interactuar con el ordenador, los cuales se recogen en la siguiente tabla.

Equipo	Coste en €	Años de vida útil
Ratón	50	5
Teclado	35	10
Monitor principal	215	7
Monitor secundario	100	3
	400	

TABLA 2. PRESUPUESTO HARDWARE

Como se puede ver, el coste total derivado del *hardware* asciende a **1200€** (800€ el equipo + 400€ los periféricos), aunque este valor será amortizado a lo largo de su vida útil.

3.1.3. Personal

En el desarrollo del proyecto participará una única persona, que actuará de analista y de programadora. El precio de la hora de trabajo de una persona con este perfil se estima en unos 25€/hora, mientras que la estimación de horas dedicadas al proyecto es un total de 340 horas.

Por lo tanto, la paga total que corresponde al programador es de **9250€** divididos en 5 pagas mensuales, lo cual equivale a un sueldo de **1850€ brutos al mes**.

3.1.4. Otros gastos

Este proyecto no incurre en otros gastos. Sin embargo, una posible ampliación del mismo podría derivar en la necesidad de alojar una base de datos, en cuyo caso se deberían calcular e incluir aquí los costes del alojamiento.

4. Tecnologías utilizadas

4.1. Lenguaje de programación

La aplicación será desarrollada en Python 3.6 como aplicación de escritorio. Los motivos de la elección son los siguientes.

- **Conocimiento previo del lenguaje.** La familiaridad con Python permite ahorrar tiempo de aprendizaje y un mejor aprovechamiento de sus características.
- **Multiplataforma.** Ahorra el problema de garantizar la compatibilidad entre distintas plataformas.
- **Multitud de librerías útiles.** La comunidad ha desarrollado múltiples librerías y frameworks que facilitan el fuzzing en Python.
- **Sencillez de sintaxis.** Una de las características de Python es que con pocas líneas se consiguen grandes funcionalidades. Además, es bastante legible de cara a que otro programador quiera modificar el código para adaptarlo a sus necesidades.

4.2. Entorno de desarrollo

Para programar se utilizará como IDE PyCharm de JetBrains, ya que ofrece un potente *debugger* y autocompletado con los que el desarrollo será más sencillo y ágil. Así mismo, para edición de textos se utilizará el editor Atom, el cual permite trabajar de manera fluida y personalizada.

4.3. Bibliotecas utilizadas

En la Tabla 2 se presenta una lista resumen de las bibliotecas que se utilizarán en el proyecto, así como una breve descripción de su utilidad dentro del mismo.

Nombre	Enlace	Justificación
Subprocess	--	Lanzar la ejecución de otros programas para su análisis
Argparse	--	Valida y procesa los parámetros de entrada al llamar al programa a través de una consola de comandos
Os	--	Proporciona acceso a funcionalidades del SO con independencia de cuál sea el mismo
Sys	--	Acceso a los parámetros de entrada de la aplicación mediante CLI
Time	--	Proporciona la hora local para usarla como semilla
JSON	--	Procesa el archivo de configuración de la aplicación
Re	--	Expresiones regulares para extraer el esquema de los URIs
Lxml	--	Necesario para exportar resultados en formato XML
Random	--	Generación de números aleatorios
Shutil	--	Mover archivos entre carpetas
Colorama	Colorama	Permite dar color al texto de la consola independientemente del SO
Console Progress Bar	ProgressBar	Libería que permite incluir barras de progreso fácilmente en la consola
Requests	Requests	Facilita el proceso de realizar peticiones HTTP y procesar la respuesta
Figlet	PyFiglet	Generación de banners ASCII

TABLA 3. LIBRERÍAS EXTERNAS Y JUSTIFICACIÓN DE USO

Se puede comprobar que tan solo han sido necesarias 4 bibliotecas que no se distribuyen junto a Python. Es necesario asegurarse de que dichas bibliotecas se encuentran instaladas en las máquinas donde ejecutemos el programa.

4.3.1. Entorno gráfico

La elección de una biblioteca para programar la GUI es fundamental, ya que marcará la calidad del resultado final y el tiempo invertido en su construcción. De la amplia oferta de bibliotecas existentes, se han seleccionado para su estudio las siguientes: PyQt, PyGTK, Tkinter y Pyforms.

A lo largo de las iteraciones de este proyecto se pondrán a prueba estas 5 bibliotecas con el objetivo de conocer de primera mano su funcionamiento antes de tomar la decisión de cuál utilizar para la interfaz gráfica.

Tkinter

Tkinter es la biblioteca estándar de facto para construir interfaces gráficas en Python.

Está compuesta por varios *widgets* básicos que se estructuran de manera jerárquica. Entre dichos *widget* se encuentran cajas de texto, botones, listas desplegables y rótulos¹³. Para ubicar los elementos, Tkinter dispone de varios gestores que en principio no deben ser mezclados entre sí¹².

- Grid: Especificamos la fila y columna donde queremos posicionar el *widget*. Tkinter se encarga de asignar el espacio y las posiciones.
- Place: Posicionamiento absoluto del *widget* indicando sus coordenadas x e y. Desaconsejado ya que al redimensionar la ventana, cambiar de plataforma o trasladar el código a otro lenguaje tendremos que rehacer el código.
- Pack: Organiza los widgets en cajas verticales y horizontales.

La apariencia de los *widgets* es clásica y no ofrece posibilidades de personalización suficientes.

PyQt

PyQt es un framework multiplataforma que se distribuye bajo licencia comercial. Une el framework de C++ (Qt) con Python, manteniendo su compatibilidad con todas las plataformas.

No solo es una herramienta para desarrollar GUIs, sino que también incluye abstracciones para crear sockets, hilos, trabajar con Unicode, expresiones regulares, bases de datos SQL, OpenGL, XML, un navegador web plenamente funcional, un sistema de ayuda y un framework multimedia.

Además, Qt (y PyQt por extensión) incluye una interfaz gráfica, Qt Designer, en la que se puede diseñar de manera visual la aplicación y después obtener el código.^{ref1}

PyGTK

PyGTK es una biblioteca construida sobre GTK+, la cual proporciona una gran variedad de utilidades y elementos gráficos. Se ha utilizado en aplicaciones de relevancia como BitTorrent o Anaconda.

Las aplicaciones resultantes son totalmente multiplataforma, esto es, no requieren ninguna configuración especial para ejecutarlas en Windows, Linux, MacOS o cualquier otra plataforma.

Otra característica distintiva de PyGTK son la facilidad de uso y prototipado, apoyo a la accesibilidad y la capacidad para trabajar con textos multilingües en aplicaciones localizadas¹⁴.

PyGTK es software libre, por lo que se puede usar, modificar y distribuir con pocas restricciones (licencia LGPL).

PyForms

PyForms es una capa de Python construida sobre PyQt, OpenGL y otras bibliotecas. Permite la ejecución de GUIs de escritorio, web y CLI sin requerir la modificación del código. Está pensado como un grupo de reglas y metodologías que ayudan al desarrollador a mantener su código breve, limpio, reusable y legible.¹⁵

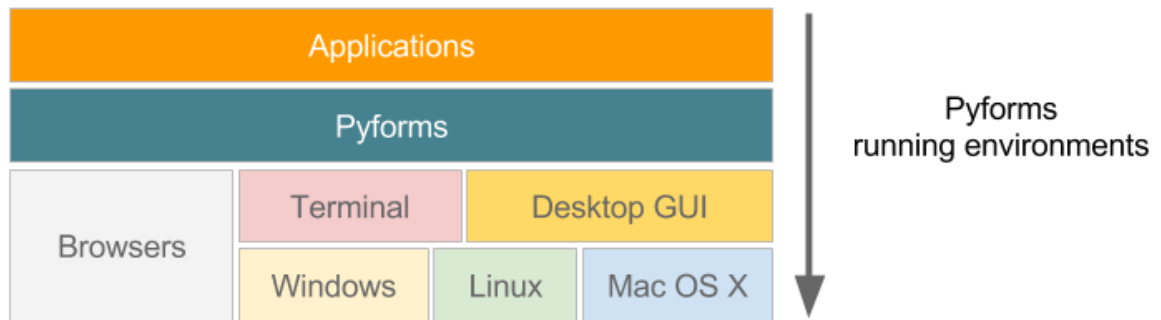


FIGURA 2. ESTRUCTURA DE CAPAS Y PyFORMS

Electron

Electron es una biblioteca de código abierto para crear aplicaciones de escritorio multiplataforma utilizando HTML, CSS y JavaScript. Para ello, combina las tecnologías de Chromium y Node.js. A día de hoy, Electron es una biblioteca muy utilizada tanto por nuevas empresas como por compañías ya establecidas.¹⁶

5. Diseño del proyecto

5.1. Especificación de requisitos

5.1.1. Historias de usuario

Dado que solo un tipo de rol (programador) va a utilizar esta herramienta, se omitirá su mención al comienzo de cada historia de usuario.

- Quiero comprobar si una URL es vulnerable a un ataque mediante inyección SQL o XSS para protegerla debidamente.
- Quiero intentar ejecutar un programa con diversas combinaciones de parámetros para conocer su robustez al gestionar situaciones no esperadas.
- Quiero reproducir una condición de error para poder estudiar el motivo del mismo.
- Quiero importar mis propios vectores de ataque para utilizar mi conocimiento sobre la aplicación y sus potenciales puntos débiles.

- Quiero obtener dos informes de resultados, uno resumido y otro detallado, para sacar conclusiones en función de mis necesidades.
- Quiero alternar el tema de la interfaz entre oscuro y claro para adaptarlo a mis preferencias y condiciones lumínicas.
- Quiero personalizar los parámetros de fuzzing para adaptar la operación a mis necesidades y restricciones.

5.1.2. Requisitos no Funcionales

- Quiero prescindir del manual de uso para poder usar la aplicación de manera estándar para no invertir demasiado tiempo en la fase de test.
- Quiero tener acceso a la documentación técnica que me ayude a implementar mis propios módulos para extender su funcionalidad.
- Quiero que el formato de los informes de resultados sea amigable y permitan reconocer rápidamente las zonas de interés.

5.2. Diagrama de casos de uso

A continuación se presenta el siguiente diagrama de casos de uso que describe el funcionamiento de la aplicación.

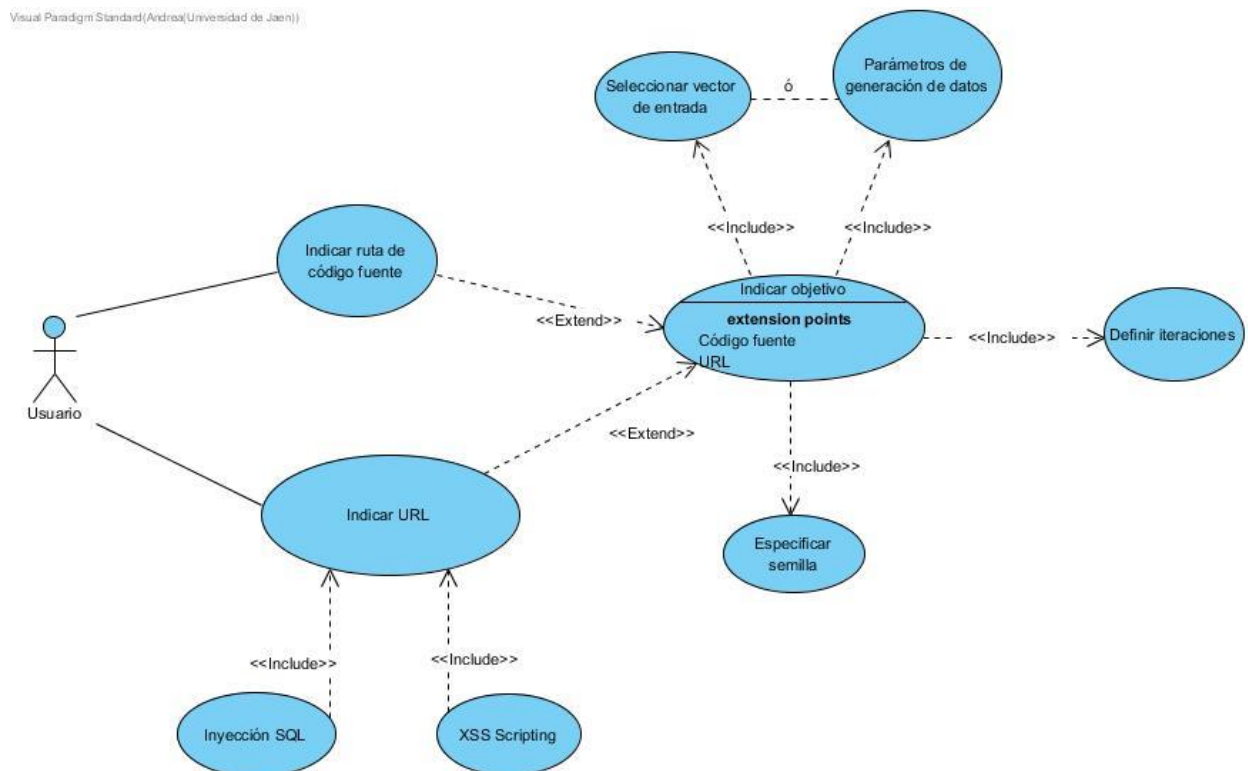


FIGURA 3. CASO DE USO "ANALIZAR APLICACIÓN"

5.3. Diagrama de actividad

El siguiente diagrama de actividad detalla el flujo de acción de la aplicación.

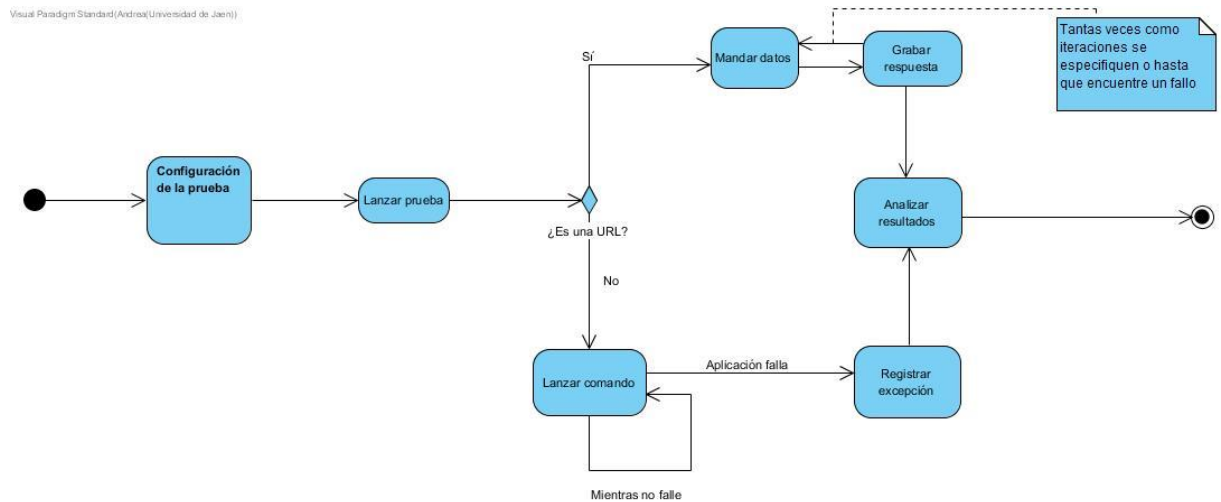


FIGURA 4. DIAGRAMA DE ACTIVIDAD

Se divide en tres grandes pasos:

- Configuración de la prueba
- Realización de la prueba (este paso es automático)
- Análisis de resultados (este paso es semi automático)

6. Desarrollo del proyecto

6.1. Primera iteración

La duración de la primera iteración será de dos semanas y el objetivo es preparar una interfaz por consola de comandos que permita especificar la configuración de la prueba. Adicionalmente, se comenzará a hacer pruebas de interfaz gráfica en Tkinter para explorar las opciones que ofrece la biblioteca.

6.1.1. Consola de comandos

Por consola se ha de llamar al programa de las siguientes maneras:

Comando	Función
<code>python fuzzuja.py</code>	Lanza la aplicación en modo CLI
<code>python fuzzuja.py -g</code>	Lanza la aplicación en modo GUI

TABLA 5. OPCIONES DE LANZAMIENTO DE LA APLICACIÓN DESDE LA CONSOLA DE COMANDOS

Por lo tanto, los pasos que hay que seguir para cumplir esta tarea son:

- Navegación por menú parametrizado.
- Lectura de archivo de configuración JSON para cargar los valores predeterminados.
- Configuración de parámetros a través del menú por consola.

Al ejecutar la orden *python fuzzuja.py* podremos ver el siguiente menú:

```
Bienvenido a FuzzUJA.  
  
1. Objetivo (Actualmente: http://lxml.de/objectify.html)  
2. Payloads  
3. Alterar payload aleatoriamente (Actualmente: No)  
4. Ficheros de salida  
5. Iniciar prueba (puede tardar varios minutos)  
6. Salir  
Tu elección:
```

FIGURA 6. ASPECTO DEL MENÚ A TRAVÉS DE LA CONSOLA

Se divide en apartados a los que se puede acceder escribiendo el número correspondiente y pulsando *enter*. Podemos indicar la configuración desde aquí o bien configurando un archivo json llamado **config.json** que se encuentra en el directorio raíz del proyecto y cuya estructura será dependiente del fuzzer o fuzzers del sistema.

Si se editan sus valores antes de iniciar el programa, solo necesario poner en marcha el mismo sin tener que navegar por el menú.

Los parámetros que se pueden configurar son los siguientes:

- **Objetivo***: URI objetivo del fuzzer.
- **Payload***: Archivo del que se van a extraer los valores con los que se van a realizar los test. Dichos valores deben aparecer listados cada uno en una línea. En este proyecto se van a recopilar una serie de payloads que se ubicarán en la carpeta *payloads*. Si el usuario desea incluir su propio payload es necesario que lo coloque dentro de esta carpeta. En el

menú, al seleccionar la opción de seleccionar payload, se listarán todos los disponibles dentro de esa carpeta.

- **Random:** Si se van a atacar sistemáticamente todas las posibles entradas (n) o si se van a escoger aleatoriamente un porcentaje de ellas (s). Por defecto es **n**.
 - **Fichero de salida:** Ruta y del archivo que va a contener los resultados de la prueba. Por defecto es *fuzzresults.txt* y se alojará en la carpeta raíz del proyecto.
- Parámetros que necesitan configuración manual.

Es importante señalar que los parámetros que indiquemos durante la ejecución del programa en ningún caso serán almacenados en el archivo de configuración. Éste es solo de lectura y sirve para ahorrarnos pasos a la hora de configurar la prueba.

6.1.2. Interfaz gráfica

Durante este *sprint* no se contempla la implementación de la interfaz gráfica al ser una tarea de baja prioridad, sino su planificación y estudio de los distintos frameworks que se pueden elegir para esta tarea. En este caso, se realizarán pruebas con Tkinter. El objetivo es conseguir abrir una ventana que contenga grandes apartados:

- **Configuración:** Donde se pueden establecer los parámetros de la prueba.
- **Resultados:** Campo donde se van imprimiendo los resultados durante la prueba. Esta misma información se guardará en un archivo para una consulta más cómoda por parte del usuario.

Tras varias horas de prueba con Tkinter se ha llegado a la conclusión de que no es la mejor opción para este proyecto debido a que es una biblioteca de estética desfasada y de código poco legible. Es necesario configurar muchas opciones a mano para ubicar los elementos donde se desea, y aún así los resultados se alejan de un resultado profesional.

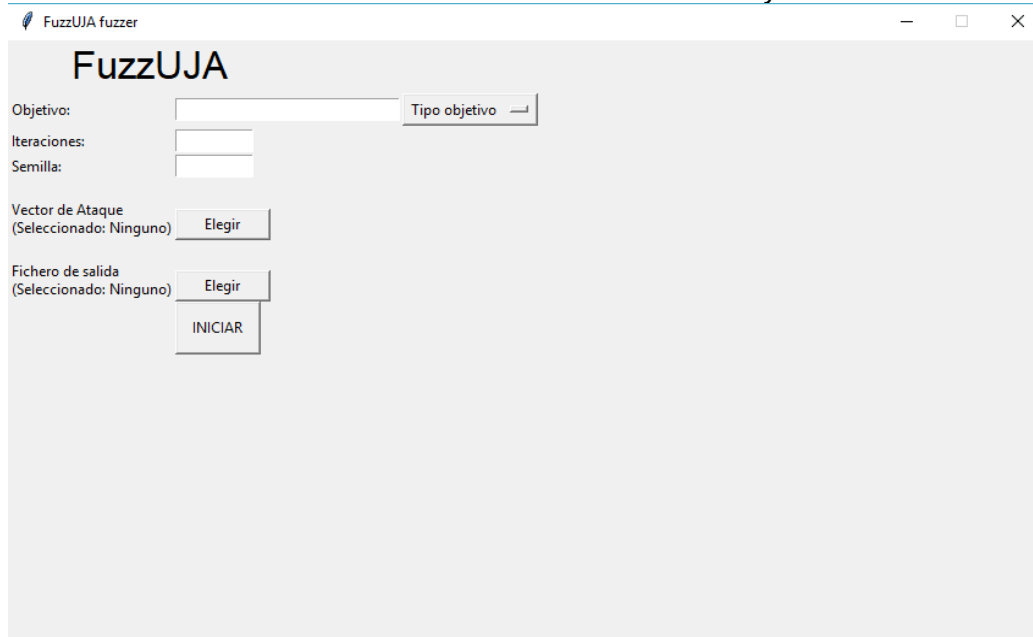


FIGURA 7. PROTOTIPO PARCIAL HECHO CON TKINTER

Por lo tanto, en siguientes iteraciones se estudiará el uso de otro Framework con el que realizar la interfaz gráfica.

6.2 Segunda iteración

La segunda iteración del proyecto durará dos semanas y tiene por objetivo realizar la inyección de datos a URL.

6.2.1 Inyección de datos

En este punto de la aplicación, es posible lanzarla y almacenar los parámetros de lanzamiento. A continuación se espera realizar la inyección de datos a las URL sin preocuparnos por ahora de los resultados. Esta funcionalidad se divide en dos subtarefas mutuamente dependientes:

- Conexión con la URL, reconocer sus parámetros y deformarlos.
- Recopilación de vectores de ataque.

El protocolo HTTP y las peticiones GET

El protocolo de transferencia de hipertexto (**HTTP**) es el protocolo de comunicación que permite las transferencias de información en la World Wide Web.

Los mensajes HTTP son texto plano y tienen la siguiente estructura²⁰.

- Línea inicial
 - Para las peticiones se envía la acción requerida por el servidor, la URL y la versión HTTP que soporta el cliente.
 - Para las respuestas se envía la versión de HTTP, el código de respuesta (indica el estado de la petición) y la frase asociada a la respuesta en sí.
- Cabeceras con metadatos.
- Cuerpo del mensaje (opcional). Es la información que intercambian el cliente y el servidor.

La figura 6.5 muestra cómo sería una petición GET mediante el protocolo HTTP.

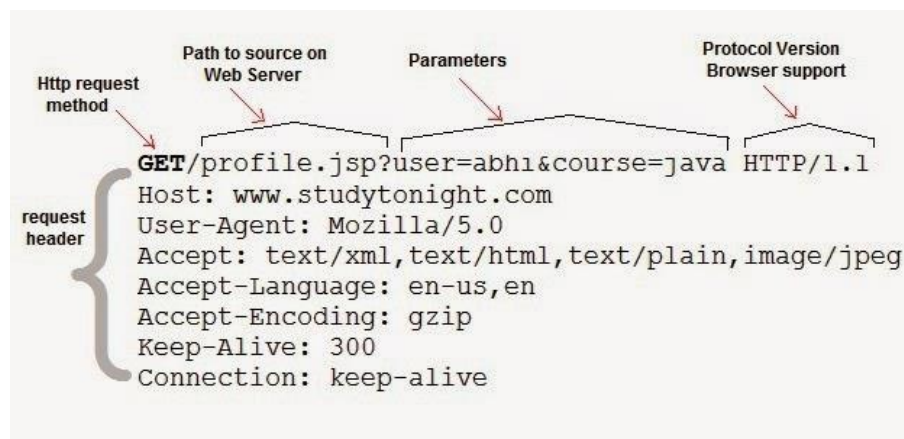


FIGURA 8. PETICIÓN GET

En este proyecto se pretende utilizar peticiones GET para poner a prueba aplicaciones web. Al llevar los parámetros a través de la URL, es muy importante sanear y validar correctamente los mismos en el servidor para evitar código malicioso o malformado.

Un ejemplo del peligro que entrañan las URL podría ser la vulnerabilidad [CVE-2008-0016](#), la cual se explotaba provocando un *buffer overflow* mediante el envío de una URL lo suficientemente larga.²¹

Implementación

Esta parte de la iteración se divide en las siguientes subtarefas:

- Detección de segmentos y parámetros de la URL indicada

- Realización de peticiones GET alterando sus distintas partes

La detección de parámetros es trivial: Se toma la subcadena que empieza a partir del carácter `?` y esta se divide usando `&` como carácter separador. A su vez, cada una de las partes obtenidas se dividen usando `=` como separador, y se construye un diccionario con los pares obtenidos.

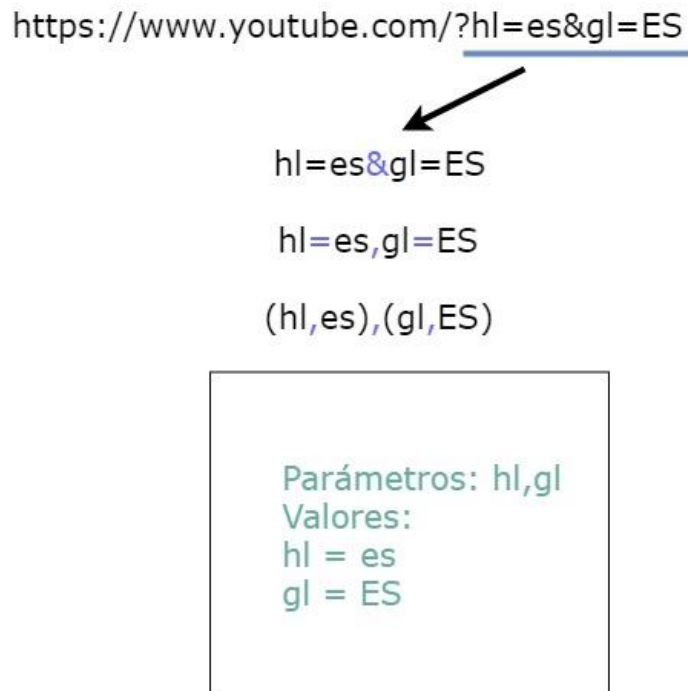


FIGURA 9. DETECCIÓN DE PARÁMETROS GET EN URL

Por otro lado, todo el conjunto de la URL se puede entender como la unión de las siguientes partes²²:

[dominio] / [ruta] / [pagina] . [extensión] ? [parámetros]

Y cada una de ellas se debe tratar de forma separada con valores que a menudo han llevado a alguna vulnerabilidad o bien con valores aleatorios para comprobar cómo gestiona el servidor las peticiones no esperadas. A continuación, se desgranar las partes de la URL y sus posibles ataques²³:

- **Ruta.** Las vulnerabilidades más frecuentes descubiertas en la ruta son *buffer overflows* y *directory traversals*. Los *overflows* pueden ser

descubiertos mediante la inyección de un gran volumen de información (cadenas de caracteres muy largas), mientras que los *directory traversals* se descubren mandando sucesivos “../”.

- **Página.** Se suelen buscar nombres comunes de páginas importantes que no han sido correctamente aseguradas o, nuevamente, *buffer overflows*.
- **Extensión.** Los archivos de páginas web normalmente tienen extensiones que permiten identificar la tecnología o lenguajes empleados (.php, .asp...). Además, se han descubierto algunas vulnerabilidades haciendo peticiones con extensiones no reconocidas por el servidor.
- **Parámetros.** Los parámetros se dividen en tres partes: nombre del parámetro, separador y valor.
 - **Nombre del parámetro:** Probando nombres comunes se pueden descubrir variables que el servidor procesa (como aquellas que se encuentran en campos ocultos). Además, se pueden mandar nombres de parámetro malformados para comprobar cómo el servidor maneja esa situación.
 - **Valor:** La mejor manera de tratar el valor es dependiendo del tipo de dato esperado por el servidor. Por ejemplo, si el valor original es un número, es razonable intentar mandar números muy grandes, o muy pequeños, o el valor 0, o valores negativos. Además, ofrecen otra posibilidad de intentar provocar un *buffer overflow*.
 - **Separador:** Si se tratan de sustituir o malformar estos caracteres (?, =, &) es posible descubrir errores.

Este fuzzer se vale de expresiones regulares y manipulación de cadenas de caracteres para reconocer los segmentos de una URI y así poder procesarlos por separado. Estos son los resultados obtenidos con distintas URL.

- <https://www.ece.cmu.edu/~ganger/712.fall02/papers/p761-thompson.pdf>

```
* URI: https://www.ece.cmu.edu/~ganger/712.fall02/papers/p761-thompson.pdf
* Dominio: https://www.ece.cmu.edu/
* Ruta: /~ganger/712.fall02/papers/
* Página: p761-thompson.pdf
* Parámetros: {}
```

FIGURA 10. SEGMENTOS EN URL 1

- <https://mail.google.com/mail/u/0/#inbox>

```
* URI: https://mail.google.com/mail/u/0/#inbox
* Dominio: https://mail.google.com/
* Ruta: /mail/u/0/
* Página: #inbox
* Parámetros: {}
```

FIGURA 11. SEGMENTOS EN URL 2

- <http://www.elladodelmal.com/2018/04/como-entrenar-tu-ai-para-jugar-video.html?m=1>

```
* URI: http://www.elladodelmal.com/2018/04/como-entrenar-tu-ai-para-jugar-video.html?m=1
* Dominio: http://www.elladodelmal.com/
* Ruta: /2018/04/
* Página: como-entrenar-tu-ai-para-jugar-video.html
* Parámetros: {'m': '1'}
```

FIGURA 12. SEGMENTOS EN URL 3

El siguiente paso es construir nuevas URL alterando el valor de los parámetros y realizar las peticiones de nuevo.

Los datos que se envíen pueden tener multitud de formas y orígenes. Por suerte, existen diccionarios de ataque que contienen los valores que son más susceptibles de generar un fallo. El primer diccionario de ataque que se va a utilizar es [este](#), y aunque contiene 1057 valores, se van a remarcar algunos de ellos:

- [illegible]

- `phpinfo()`. Si el servidor es PHP y se permite la inyección de código se volcaría información tal como la versión del mismo.
- `"</script>alert(1);</script><!--"`. Esta cadena y sus derivados pueden realizar ataques XSS.
- `../../../../../../../../../../../../../../../../../../../../etc/passwd.` Esta cadena y derivadas serviría para detectar *directory traversals*.

Además, se incluyen caracteres susceptibles de provocar alteraciones tales como `$` o `%ff` entre otros.

La inyección básica de datos se hará en la parte de la URL llamada ruta. Para ello, se descompone la URL en segmentos y se vuelve a componer sustituyendo la ruta por el valor de ataque. La petición se realiza y se graba el código HTTP devuelto. Este proceso se refinará en sucesivas iteraciones del proyecto.

Pruebas con Electron

Durante esta iteración también se ha investigado a fondo la opción de Electron como framework para desarrollar la interfaz gráfica. A pesar de no haber llegado a desarrollar un prototipo, se ha descartado esta opción debido a los siguientes factores:

- **Dependencias:** Utilizar este framework implica una serie de dependencias (sin ir mas lejos Node.js) que implican el aprendizaje de nuevas tecnologías no contempladas inicialmente para el proyecto. Esto acarrearía un retraso en el desarrollo que se traduciría en un aumento de los costes o en una disminución de la calidad.
- **Arquitectura:** Utilizar Electron implicaría un cambio de arquitectura en la aplicación que acarrearía retrasos en el desarrollo por tener que adaptar todo lo que hay desarrollado hasta ahora a un paradigma distinto.
- **Rendimiento.** Electron ejecuta, resumidamente, un navegador sobre un servidor Node.js. Esto requiere unos recursos bastante elevados teniendo en cuenta que el proyecto pretende ser simple, portable y ejecutable en máquinas desfasadas si fuese necesario.

Por lo tanto, este framework queda descartado para el desarrollo de este proyecto. Sin embargo, si la ambición del mismo creciera y fuera una prioridad aportar

un aspecto moderno y especialmente agradable al usuario, sería un fuerte candidato a tener en cuenta.

6.3 Tercera iteración

Esta iteración durará tres semanas y los objetivos a cumplir durante la misma son los siguientes:

- Fuzzing de distintas partes de la URL por separado.
- Estructuración de payloads según su tipo y objetivo.
- Presentar el primer prototipo de interfaz gráfica.

Tipos de vulnerabilidades

Como se ha introducido en la iteración anterior, cada parte de una URL tiene una serie de vulnerabilidades que han de ser buscadas de manera distinta.

Directory traversals

Esta vulnerabilidad permite acceder a archivos y directorios que son almacenados fuera del directorio raíz de la web. Manipulando las variables que hacen referencia a archivos con secuencias de “../” y derivados, o bien usando rutas absolutas, es posible acceder a archivos tales como código fuente o archivos de configuración.

Para su testeo, una buena heurística es buscar nombres de variables interesantes en las URL. Ejemplos de estos nombres son²⁴:

- <http://example.com/getUserProfile.jsp?item=ikki.html>
- <http://example.com/index.php?file=content>
- <http://example.com/main.cgi?home=index.htm>

A continuación, se dotan a esas variables de rutas absolutas o relativas para intentar encontrar alguna vulnerabilidad que nos permita movernos por los archivos del servidor sin autorización.

A la hora de construir las rutas es importante tener en cuenta el sistema operativo del servidor, ya que determinará los caracteres que se usan como separadores.

	Windows OS's shell	Unix-like OS	Mac OS
Directorio root	"<letra de disco>:\."	"/"	"<letra de disco>:"
Separador de directorios	"\" o "/"	"/"	","

TABLA 4. CARACTERES EN RUTAS SEGÚN SO

Buffer Overflows

Los errores de *buffer overflow* se caracterizan por sobrescribir fragmentos de memoria al agotarse la memoria asignada a un recurso específico. Estos errores suelen provocar excepciones y *segmentation faults*, y provocan la detención del programa de manera inesperada.

Existen tres tipos de *buffer overflows*.

- **Heap Overflow.** En el heap se almacenan datos dinámicamente y variables globales. Si se sobrepasa el espacio del heap asignado, la información sobrescribe otras zonas de memoria asignadas a otros procesos. Esto se traduce en que un atacante podría sobrescribir punteros de funciones y direcciones.
- **Stack Overflow.** Ocurre cuando una variable de tamaño no fijo es copiado en un buffer que sí tiene tamaño fijo sin la correspondiente comprobación de los límites de tamaño. Es una vulnerabilidad muy grave que permite al atacante tomar control del registro IP, y por lo tanto, ejecutar código arbitrario en la máquina.
- **Format String.** Esta vulnerabilidad hace que información enviada como string sea evaluada como un comando, permitiendo ejecutar código ajeno a una aplicación. Esto ocurre a través de las funciones que formatean strings. Estas utilizan ciertos caracteres especiales (%s, %x, %n) que sirven para recibir parámetros.

Parameters	Output	Passed as
%%	% character (literal)	Reference
%p	External representation of a pointer to void	Reference
%d	Decimal	Value
%c	Character	
%u	Unsigned decimal	Value
%x	Hexadecimal	Value
%s	String	Reference
%n	Writes the number of characters into a pointer	Reference

FIGURA 13. PARÁMETROS MÁS COMUNES USADOS EN FORMAT STRING ATTACKS

Inyección SQL

Este ataque consiste en mandar consultas SQL arbitrarias desde el cliente de la aplicación. Un ataque exitoso de este tipo puede volcar información sensible, modificarla o realizar operaciones administrativas. Para mostrar cómo este ataque funciona, imaginemos que tenemos el siguiente código en un servidor PHP:

```
<?php
```

```
$query = "SELECT * FROM users WHERE username='".$_POST['user']."' AND  
pass='".$_POST['pass']."'";
```

```
mysqli_query($connection, $query);
```

```
?>
```

Si las variables `$_POST['user']` y `$_POST['pass']`, que son los valores que manda el usuario a través de cualquier interfaz, fueran los siguientes:

- User = foo

Pass = “ OR 1 = 1 La query que se mandaría a la base de datos sería:

“SELECT id FROM users WHERE username='foo' AND pass='' OR 1 = 1”

Y por darse que 1=1 es siempre verdadero, devolvería la información del usuario aunque las credenciales fueran erróneas.

Cross-site scripting

Abreviado como XSS, este tipo de vulnerabilidad es propia de las aplicaciones web. Permite inyectar código malicioso escrito en JavaScript u otros como VBScript. Esta vulnerabilidad se da al no validar correctamente los datos de entrada que se utilizan en una aplicación, o bien por no sanear las entradas. XSS se puede utilizar para robar información delicada, secuestrar sesiones de usuario o comprometer el navegador.

Implementación

A la vista de las vulnerabilidades presentadas anteriormente, es necesario concretar qué se va a buscar en qué secciones de la URL. De alguna manera, se podrían probar todas las combinaciones. Este enfoque sería muy exhaustivo, pero comprometería el rendimiento del *fuzzer*. La siguiente tabla muestra la selección de vulnerabilidades y partes de las URL donde se van a buscar.

Parte de la URL	Vulnerabilidades
Dominio	--
Ruta	Directory traversals, Heap Overflow,
Página	Directory traversals
Extensión	--
Parámetros	Directory traversals, Format String, Heap Overflow, Inyección SQL, XSS

TABLA 5. VULNERABILIDADES QUE SE VAN A BUSCAR Y EN QUÉ PARTES DE LA URL

El dominio queda fuera de la prueba porque si lo alteramos estaríamos atacando una aplicación web distinta, mientras que la extensión de la página es solo un elemento que aporta información para más ataques. Por otro lado, los parámetros son

el vector de entrada del que más se abusa, ya que sus distintos usos dentro de una aplicación web aportan una gran variedad de potenciales ataques.

Para realizar los ataques de inyección SQL y XSS, se preparará una serie de archivos que contengan líneas que son susceptibles de poner evidencia este tipo de ataques. Estos se incluirán sustituyendo los valores de los parámetros.

De igual manera se hará con los directory traversals, pese a que estos pueden programarse para construirlos durante la ejecución del test, pero si ya se encuentran alojados en un archivo, ahorramos recursos y la velocidad del test aumenta. Dichos valores se colocarán en los valores de los parámetros, en el nombre de la página y en cada segmento de la ruta. Esto es, si tenemos la ruta */folder1/folder2/folder3/*, cada valor de ataque se utilizará como */ataque/*, */folder1/ataque* y */folder1/folder2/ataque*.

Los caracteres especiales para los *buffer overflows* también se almacenarán en un archivo, pero se combinarán aleatoriamente en cada ejecución (aumentando su frecuencia u orden en caso de *format string*, o aumentando el tamaño en el caso del *heap overflow*).

Por último, aunque no aparece mencionado en la tabla ya que no es un ataque con nombre propio, se intentará comprometer los parámetros en función de su naturaleza (inferida a través de su nombre y el valor). Por ejemplo, si el valor que tiene un parámetro es un número, se probará a sustituir por números positivos, negativos y el 0.

Para los *directory traversals* se ha creado un script que a su vez ha generado un fichero que contiene sucesivas líneas con “*../*” y “*..*” concatenados. En total hay 15 niveles de profundidad por cada uno. Por otro lado, se ha recopilado una serie de archivos y comandos que se pueden aprovechar al realizar un ataque *directory traversal* y estos se han almacenado en un archivo aparte. El objetivo es combinar cada nivel de profundidad con cada posible ataque. El motivo de hacerlo así es que en los payloads ya disponibles por internet se utilizan rutas ya fijadas que llevan a algún archivo, pero no tienen en cuenta las distintas combinaciones que pueden llevar a descubrir dónde está el archivo o sitio vulnerable. Además, al separar la profundidad

del directorio y el archivo/comando objetivo, se pueden hacer modificaciones en ambas partes de manera modular.

Los valores que podrían desencadenar *heap overflows* se han extraído del archivo del que ya disponemos, y se han recopilado en uno nuevo de forma manual. En él se incluyen cadenas de caracteres muy largas, así como números muy grandes, y cadenas que pueden desencadenar *format string attacks*.

En cuanto a la inyección SQL y el XSS, se ha procedido de manera análoga: Recopilar vectores de ataque de diversas fuentes de internet para obtener dos ficheros separados, cada uno con entradas maliciosas que puedan desencadenar una vulnerabilidad de ese tipo.

Como se puede deducir, en este punto del desarrollo el esquema del programa cambia. Ya no se usa un vector de ataque para todo, sino que existen distintos vectores de ataque según el objetivo y la vulnerabilidad. Como consecuencia de esto, el menú será distinto. Si invocamos a la opción de modificar el payload, podremos hacerlo en función del objetivo. Es decir, podremos elegir el archivo que queremos utilizar según cada vulnerabilidad, en lugar de uno para todo.

Esto obliga a modificar la estructura del archivo de configuración, que pasa a tener varios valores dentro del campo "payload".

```
{
  "target": "none",
  "payload": {
    "traversal_depth": "directory_traversals.txt",
    "file_commands": "file_commands.txt",
    "overflow": "heap_overflow.txt",
    "sql_injection": "sql_injection.txt",
    "xss": "xss.txt"
  },
  "random": "No",
  "output": "fuzzresults.txt"
}
```

FIGURA 14. NUEVO ARCHIVO DE CONFIGURACIÓN

A continuación se detalla cada campo:

- ***traversal_depth***: Es el archivo que ha de contener los valores para navegar por directorios. Esto es, la sucesión de “../” y derivados.
- ***file_commands***: Es el archivo que contiene comandos y nombres de archivos que, combinados con el fichero anterior, dan lugar a *traversal attacks*.
- ***overflow***: En este archivo se encuentran los ataques que pueden desencadenar *heap overflow* y *format string attacks*.
- ***sql_injection*, *xss***: Como su propio nombre indican, contienen cadenas que pueden detectar esas vulnerabilidades respectivamente.

Para ahorrar pasos al usuario, se han establecido por defecto los valores de estos campos, ya que podría ser muy tedioso querer realizar una prueba rápida y tener que pararse a escribir cada uno de los ficheros requeridos. La figura 6.13 muestra el nuevo aspecto de la sección *payload* en el menú de consola.

```
Configurar payloads
1. Navegación a través de directorios ( Actualmente: directory_traversals2.txt )
2. Archivos y comandos ( Actualmente: file_commands.txt )
3. Heap overflow y format string attack ( Actualmente: overflow.txt )
4. Inyección SQL ( Actualmente: sql_injection.txt )
5. XSS ( Actualmente: xss.txt )
6. Terminar
Tu elección:
```

FIGURA 15. NUEVA SECCIÓN PAYLOAD EN CLI

Una vez implementado el sistema de fuzzing descrito anteriormente surge un problema. Dependiendo del tamaño de los vectores de ataque, la prueba puede durar varios minutos. En el caso de este programa actualmente, una ejecución sobre la siguiente URL:

https://elpais.com/economia/2017/03/27/actualidad/1490624613_488525.html?id_externo_rsoc=fbads_ep_harvardaulas

Tarda de media unos 15 minutos. En futuras iteraciones se tratará esta característica, puesto que presenta varias alternativas. Entre ellas encontramos

- Reducir los vectores de ataques con heurísticas.
- Dejar a decisión del usuario el tiempo máximo.
- Limitar el tamaño máximo de payload.
- Utilizar hilos paralelos.

6.4 Cuarta iteración

Durante esta iteración vamos a desarrollar la última parte crítica del *fuzzer*: El análisis de las respuestas para determinar si existe una potencial vulnerabilidad o no al atacar una URL. Además, se presentará los resultados en el formato que el usuario especifique.

Dicho análisis se hace después de la fase de ataque. Primero se realizarán todos los test, se almacenarán los datos de respuesta pertinentes y se analizarán al acabar para ofrecer las conclusiones al usuario. El fichero de salida del programa, por otro lado, consistirá en los datos recabados sin procesar para un análisis más detallado del usuario.

Códigos HTTP

El protocolo HTTP devuelve una serie de códigos numéricos que contienen información acerca del estado de la petición realizada. Estos pueden informar de éxito, error, redirección o simplemente ofrecer información neutra²⁵. Esta será nuestra primera fuente de información para conocer el efecto de nuestro ataque.

Existe una gran cantidad de códigos de estado HTTP, pero no todos aportan información significativa para nuestros propósitos. A continuación se listan y detallan los más importantes y relacionados con esta actividad:

- **2XX**: Los códigos que empiezan por un 2 indican que la petición ha tenido éxito.
 - **200**. “OK”. La solicitud ha tenido éxito, que en caso de las peticiones GET significa que se ha obtenido el recurso y se encuentra en el cuerpo del mensaje HTTP.
- **3XX**. Indican que se ha producido una redirección.

- **300.** Indica que la petición tiene más de una posible respuesta y un usuario o agente debe elegir alguna.
- **4XX.** Indican que ha habido un error por parte del cliente.
 - **400.** El servidor no pudo interpretar la solicitud, probablemente por una sintaxis inválida que proviene del cliente.
 - **401.** Requiere autenticarse para obtener la respuesta.
 - **403.** Igual que 401, pero el cliente no tiene los permisos necesarios y por tanto no puede autenticarse.
 - **404.** El contenido solicitado no se encuentra.
 - **407.** Similar a 401, pero necesita hacerse mediante un proxy.
 - **410.** El contenido solicitado ha sido borrado del servidor.
 - **413.** La respuesta es demasiado grande.
 - **414.** La URI mandada es más larga de lo que el servidor puede interpretar.
 - **429.** El usuario ha mandado demasiadas peticiones en un tiempo determinado.
- **5XX.** Son errores que se producen en el servidor.
 - **500.** Es un error que se da cuando el servidor encuentra una situación que no sabe manejar. A menudo es un fallo de código.
 - **503.** El servidor no puede manejar la petición, ya sea porque está en mantenimiento o por un desbordamiento de trabajo.
 - **508.** Se ha detectado un bucle infinito al procesar la petición.

Cada uno de estos códigos pueden aportar información de valor para empezar a tirar del hilo y encontrar finalmente una vulnerabilidad. Cuáles nos interesan más o menos dependen de nuestro objetivo, así como la manera de interpretarlos.

Cabe señalar que, en la práctica, los desarrolladores de aplicaciones web rara vez son totalmente precisos con los códigos de estado HTTP. Los códigos que normalmente se devuelven en aplicaciones normales son 200, 403, 404 y 500. Una excepción a esto son las REST APIs, las cuales suelen devolver códigos detallados sobre las peticiones que se les realiza. Es por esto que este filtro aportará más información si nuestro objetivo es una REST API en lugar de un sitio web convencional.

En esta iteración el foco será presentar unos resultados basados en el código devuelto por la petición HTTP. En futuras iteraciones sería posible realizar un análisis más detallado de las respuestas, por ejemplo, *parseando* el cuerpo de la respuesta para detectar palabras o patrones clave.

Implementación

Para almacenar los resultados se han dispuesto dos clases específicas para el *fuzzer* de URLs. Una de ellas constituirá la respuesta de la petición en sí, con la siguiente información:

- **URL objetivo.** La URI completa que se ha utilizado para la petición.
- **Código de estado HTTP.** El código de estado de la respuesta.
- **Contenido.** El contenido (cuerpo) del mensaje de respuesta.
- **Tiempo de respuesta.** El tiempo en segundos que ha tardado en procesarse la petición.

La otra será el contenedor de respuestas, donde se irán almacenando todas las respuestas obtenidas y al final realizará un análisis de todas ellas, generando a su vez un fichero de salida con los resultados.

En esta iteración, el análisis se basa en clasificar cada petición según el código HTTP que devuelve, aunque el contenido de la respuesta se almacena para su posterior análisis en otra iteración.

Para plasmar los resultados, es pertinente hacerlo en un formato que permita su importación a otro programa donde se pueda analizar de la manera más cómoda posible. Es por ello que no solo se van a utilizar formatos estandarizados y ampliamente usados, sino que el usuario va a poder elegir cuáles quiere de entre los siguientes.

JSON

JSON²⁷ es un formato de texto ligero para el intercambio de datos. Es muy ligero de analizar con un parser, por lo que está altamente indicado para transmitir datos muy voluminosos. Permite los siguientes tipos de datos:

- Números, incluidos negativos y con parte fraccional (separada por puntos)
- Cadenas de texto, rodeadas entre comillas dobles.
- Booleanos, true o false.
- Null (null)
- Array, metiendo los datos entre corchetes y separándolos por comas.
- Objetos. Se ponen entre llaves y consisten en pares de <nombre>:<valor> separados por comas.

El archivo generado por esta herramienta será un array que contendrá los siguientes valores:

```
[  
  "objetivo":<url objetivo>,  
  "servidor":<servidor>,  
  "respuestas":{  
    <respuesta 1>,  
    <respuesta 2>,  
    ...  
    <respuesta n>  
  }  
]
```

Las respuestas, a su vez, serán objetos con la siguiente estructura:

```
{  
  "url":<url probada>,  
  "codigo":<codigo HTTP>,  
  "tiempo":<tiempo en segundos>  
}
```

XML

Es un meta-lenguaje²⁸ que, entre sus usos, se encuentra el almacenamiento de datos de forma legible. Es importable a casi todas las plataformas, entre las que se

encuentran bases de datos, hojas de cálculo y procesadores de texto. El esquema del archivo resultante sería el siguiente.

```
<objetivo>URL</objetivo>
<servidor>Servidor</servidor>
<peticiones>
  <peticion>
    <url>url mandada</url>
    <codigo>codigo HTTP</codigo>
    <tiempo>tiempo en segundos</tiempo>
  </peticion>
  ...
</peticiones>
```

CSV

Este formato es abierto y extremadamente sencillo. Representa los datos en forma de tabla, donde las columnas se separan por un valor concreto (habitualmente comas o puntos y comas) y las filas por saltos de línea.

Para nuestra herramienta tiene una desventaja, y es que no todos los datos de resultados encajarían en una única tabla. Los valores de objetivo y servidor quedarían fuera de este formato, por lo que se optará por incluirlos en cada fila al principio aunque suponga replicar la información. Por otro lado, el usuario podrá elegir el separador que prefiera para los datos.

Al final, el esquema de nuestro archivo CSV sería el siguiente (suponiendo que el separador es punto y coma):

```
objetivo;servidor;url;codigo;tiempo
objetivo;servidor;url;codigo;tiempo
objetivo;servidor;url;codigo;tiempo
```

...

Texto plano

Es de esperar que en ocasiones el usuario desee analizar por sí mismo el archivo o parte del mismo. Para ello, los anteriores formatos pueden presentar algún problema de legibilidad, por lo que se preparará también un formato no tan estructurado pero más cómodo de leer por un humano.

En este caso, la información se agrupará según su código HTML devuelto, ya que este enfoque es el que es potencialmente más útil para el usuario.

El formato provisional de este archivo sería el siguiente:

```
Objetivo: <url>
Servidor: <servidor>
Peticiones:
<codigo> <explicacion>
-> <url> (<segundos> segundos)
-> <url> (<segundos> segundos)
...
<codigo> <explicacion>
-> <url> (<segundos> segundos)
-> <url> (<segundos> segundos)
...
...
```

En futuras iteraciones sería posible sustituir este archivo por una página HTML con estilos adecuados que permita una cómoda visualización de los datos.

6.5 Quinta iteración

Una vez tenemos una herramienta funcional es el momento de refactorizar el código para que su ampliación sea sencilla, y adaptar el código que teníamos hasta ahora al nuevo formato. Estos serán los objetivos durante esta iteración.

La forma de invocar el programa cambia ligeramente. Si antes simplemente se escribía el nombre del programa principal, ahora es necesario indicar mediante un parámetro qué fuzzer queremos ejecutar. La llamada al programa se haría de la siguiente manera.

python fuzzuja.py -t <opcion>

Donde <opcion> es uno o varios caracteres que identifican el fuzzer que queremos ejecutar.

Por ese motivo el archivo de configuración cambiará su estructura. El nuevo archivo JSON consistirá en un array de objetos, donde cada objeto corresponde a un tipo de *fuzzer* distinto. Cada *fuzzer* tendrá unas características únicas, por lo que es decisión del programador añadir unos atributos u otros. Sin embargo, para que la aplicación funcione deberá tener, al menos, los siguientes atributos.

- **Target.** Es el objetivo del fuzzer. Este puede ser un número, una cadena de caracteres o un objeto con sus características particulares.
- **Option.** Es un nuevo atributo que indica mediante qué opción se invocará ese fuzzer. Por ejemplo, si **option** vale “u”, para ejecutar ese fuzzer deberemos escribir **python fuzzuja.py -u**. Por motivos obvios, este valor no debe estar repetido en ningún fuzzer.
- **Attempts.** Es el número de iteraciones que realizará el fuzzer antes de detenerse.

El código de la aplicación se ha reorganizado de manera que el programador apenas deba editar unas pocas líneas del código ya existente. La mayor parte del fuzzer residirá en otro archivo que deberá reunir una serie de requisitos para poder funcionar acorde con la existente aplicación.

Las modificaciones que deberá realizar el programador serán en el archivo **cli.py** y son las siguientes:

- Añadir una sentencia import para añadir el archivo que contiene su fuzzer.

- En la función **createFuzzer(config)** deberá añadir una sentencia de la siguiente forma: **elif fuzzer == '<opcion>'** donde <opcion> es el valor **<option>** del archivo de configuración JSON.
- Dentro de dicho condicional se debe devolver una instancia del fuzzer. Si este ha sido creado correctamente, el programa estaría listo para ejecutarlo.

A partir de aquí, la creación del fuzzer queda limitada a la imaginación del programador, siempre y cuando se respete la filosofía del fuzzing. Basándonos en las fases que se llevan a cabo al realizar fuzzing, se ha definido una clase de la cual debe heredar todo fuzzer que vaya a existir en la aplicación. Esta clase se llama **Fuzzer** y se encuentra en la carpeta **fuzzers**. Es una clase abstracta que contiene los métodos necesarios para que el fuzzer funcione. Es responsabilidad del programador reescribirlos en su fuzzer particular.

La especificación completa de cómo crear un fuzzer se redactará en el manual para el programador, por lo que por ahora solo se comentarán los métodos fundamentales de la clase y cuáles deberían ser sus propósitos.

- **Menu:** Este método es opcional. Su función es permitir la configuración de la prueba durante la ejecución del fuzzer. El programador es responsable de mostrar mensajes en pantalla y recoger la entrada del usuario para modificar la configuración de la prueba. De no sobrecargar este método, la configuración será la especificada en el archivo de configuración.
- **Setup.** Método opcional que se llama antes de iniciar la prueba. Puede servir para preparar e inicializar las variables necesarias que no se encuentran en el archivo de configuración.
- **GenerateInput.** Este método se encarga de devolver el dato que se va a inyectar en la iteración. Su fuente (aleatorio, secuencial...) y características(texto, número, dato binario...) son libres.
- **Fuzz.** Este método es el que inyecta el dato, recoge la respuesta, la procesa si es necesario, y la devuelve.

- **Save.** La función de este método es almacenar la respuesta en el formato y medio necesarios. Puede ser en un archivo de texto, en una estructura de datos o simplemente imprimir la respuesta por pantalla.
- **Analyze.** Este método, que también es opcional, realiza un procesamiento de todas las respuestas obtenidas y eventualmente presenta los resultados al usuario.

Se puede, y de hecho se recomienda, crear las funciones extra que el programador necesite.

En cuanto a adaptar el fuzzer ya creado, es necesario adaptar la funcionalidad anterior a la nueva estructura. El primer paso es plantear cómo se realizará la generación de datos. Para poder realizar un recorrido exhaustivo de cada vector de entrada se va a construir un objeto que contenga cada URL que debería ser probada. La función *generateInput* devolverá y borrará cada vez una URL de dicho objeto. En este caso, estableceremos un número considerable de iteraciones y haremos que se acceda cada vez a una URL aleatoria. Otra opción hubiera sido calcular la cantidad de posibles URL que se probarían y establecer el límite a dicha cantidad, pero no se llevará a cabo ya que es más sencillo y eficiente realizar varias pruebas aleatorias.

En cuanto a la ejecución de la prueba en sí, en cada iteración se realizará una petición HTTP GET con la URL devuelta por *generateInput*, y devolveremos como respuesta un objeto que contenga la URL accedida, el código de respuesta y el tiempo de la misma.

Por último, para almacenar y analizar las respuestas utilizaremos las clases que ya se diseñaron en iteraciones anteriores para analizar los resultados. Estas se encontrarán en un archivo propio ubicado en la carpeta **analyzers**, cuyo propósito es albergar las clases empleadas para procesar la información.

En el proceso de adaptación se ha perdido la potencia para disponer de una interfaz agradable. No es imposible, pero lleva más tiempo y es una tarea de menor prioridad que puede quedar relegada a iteraciones posteriores. Por otro lado, la funcionalidad del otro fuzzer se ha imitado sin mayor problema, cambiando solo la cantidad de iteraciones que se realizan.

Este es el aspecto del *fuzzer* de URLs basado en la nueva arquitectura de código.

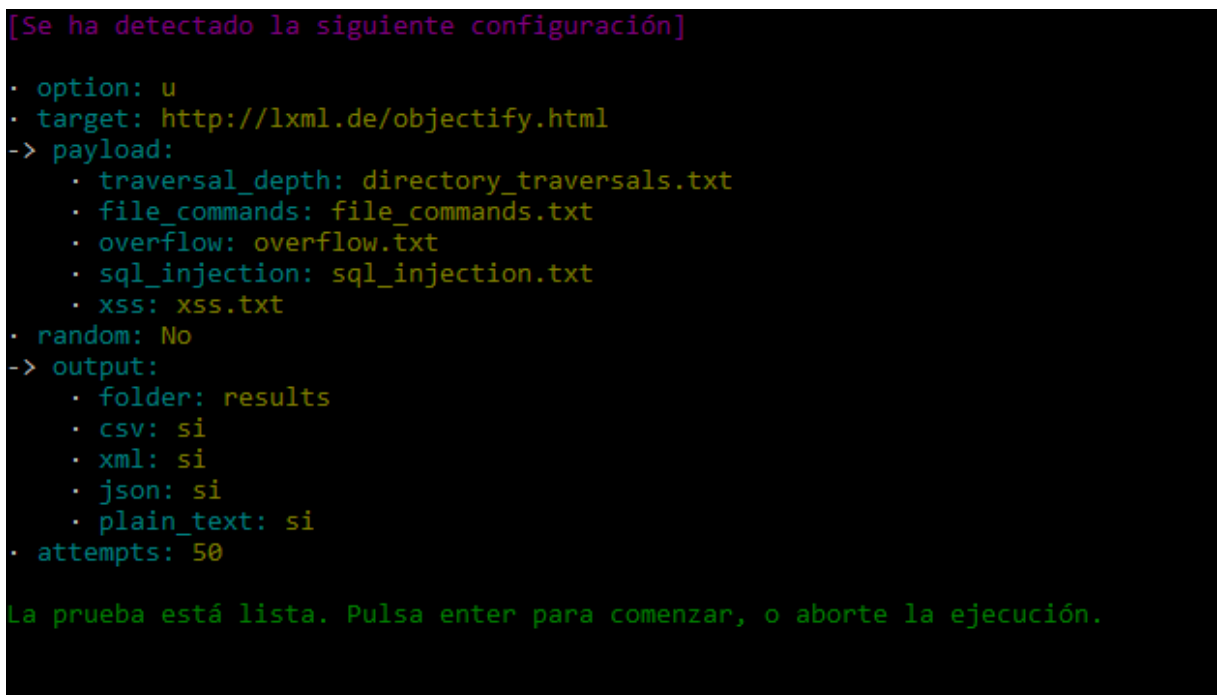
A terminal window showing a menu for the URL Fuzzer. The title 'URL Fuzzer' is displayed in a large, stylized, green font at the top. Below it, a numbered list of options is shown in yellow and green text: 1. Objetivo (Actualmente: http://lxml.de/objectify.html), 2. Payloads, 3. Ficheros de salida, 4. Iniciar prueba (puede tardar varios minutos), and 5. Salir. At the bottom, the text 'Tu elección:' is displayed in yellow.

```
URL Fuzzer

1. Objetivo (Actualmente: http://lxml.de/objectify.html)
2. Payloads
3. Ficheros de salida
4. Iniciar prueba (puede tardar varios minutos)
5. Salir

Tu elección:
```

FIGURA 16. ASPECTO DEL FUZZER DE URL EN LA NUEVA ARQUITECTURA 1

A terminal window showing the configuration options for the URL Fuzzer. The title '[Se ha detectado la siguiente configuración]' is in pink. The configuration is listed in yellow and green text, including options for target, payload, random, output, and attempts. At the bottom, a green message states 'La prueba está lista. Pulsa enter para comenzar, o aborta la ejecución.'.

```
[Se ha detectado la siguiente configuración]

• option: u
• target: http://lxml.de/objectify.html
-> payload:
  • traversal_depth: directory_traversals.txt
  • file_commands: file_commands.txt
  • overflow: overflow.txt
  • sql_injection: sql_injection.txt
  • xss: xss.txt
• random: No
-> output:
  • folder: results
  • csv: si
  • xml: si
  • json: si
  • plain_text: si
• attempts: 50

La prueba está lista. Pulsa enter para comenzar, o aborta la ejecución.
```

FIGURA 17. ASPECTO DEL FUZZER DE URL EN LA NUEVA ARQUITECTURA 2

6.6 Sexta iteración

Una vez que el código ha sido preparado para admitir la fácil integración de nuevos *fuzzers*, llega la hora de ponerlo a prueba implementando un tipo distinto de

fuzzer. Como prueba de concepto se implementará un *file format fuzzer* sencillo que ilustre el modo de diseñar e integrar nuevos fuzzers en la aplicación.

File format fuzzing

Este tipo de *fuzzing* aprovecha la carga de ficheros como vector de entrada de las aplicaciones para intentar causar alteraciones. El planteamiento es sencillo: Muchas aplicaciones pueden abrir y leer ficheros, pero ¿Qué pasa si el fichero que reciben no tienen el formato, contenido o tamaño esperados?

Normalmente se utilizan dos enfoques distintos:

- Tomar un archivo de ejemplo y mutar parte del mismo.
- Generar archivos de cero a partir de ciertas reglas y/o gramáticas.

Para este caso utilizaremos el primer enfoque al ser más sencillo de entender e implementar.

Preparación del entorno e integración del fuzzer

Para comenzar, hay que preparar el entorno para ejecutar nuestro fuzzer. Crearemos en la carpeta **fuzzers** un archivo llamado **file_format_fuzzer.py** y dentro importaremos la clase padre de la que cada *fuzzer* debe heredar añadiendo en la primera línea **from fuzzers import fuzzer**. Dentro del mismo crearemos una clase llamada **FileFormat** que heredará de la clase padre y que tendrá un constructor que acepte los parámetros **config** y **results**, que son los básicos que necesita la clase padre.

A continuación, en el archivo de configuración de la aplicación, *config.json*, añadimos la configuración necesaria para nuestro fuzzer. En este caso tendrá los siguientes parámetros.

- **Option:** **f**
- **Target:** **“notepad.exe”**. El nombre del programa que vamos a poner a prueba es el Bloc de Notas de Windows.
- **Sample:** **“payloads/sample.txt”**. Es el archivo a partir del cual se van a hacer mutaciones en el fuzzer.

- **Time_to_wait: 0.5.** Es el tiempo en segundos que se esperará antes de detener el comando que se ha lanzado. Si transcurre dicho tiempo y ninguna excepción ha saltado, se pasa a una nueva ejecución con otro archivo.
- **Mutation_ratio: 0.3 .** Es el porcentaje del archivo que se va a alterar. Será un valor entre 0 (nada) y 1 (todo).
- **Attempts: 50.**

Esta configuración indica que llamando al fuzzer con la opción **f** haremos que se mute el 30% del archivo **sample.txt** de manera aleatoria y que se ejecute **notepad.exe** abriendo el mismo durante medio segundo. Esto se realizará 50 veces.

La última configuración necesaria se realiza en el archivo **cli.py**. En él se debe importar el archivo que contiene el fuzzer o directamente su clase. Además, en la función **createFuzzer** se debe añadir una sentencia elif que cree y devuelva el fuzzer deseado.

```
if fuzzer == 'u':  
    return URL(config)
```

FIGURA 18. CÓDIGO ANTES DE AÑADIR FILE FORMAT FUZZER

```
if fuzzer == 'u':  
    return URL(config)  
elif fuzzer == 'f':  
    return FileFormat(config)
```

FIGURA 19. CÓDIGO DESPUÉS DE AÑADIR FILE FORMAT FUZZER

Hecho esto ya sería posible lanzar la aplicación llamando al nuevo fuzzer, aunque no haría nada porque no hemos configurado los métodos necesarios.

Desarrollo del fuzzer

A partir de este momento solo es necesario modificar el código del archivo que acabamos de crear (**file_format_fuzzer.py**). Es posible añadir nuevos archivos para utilizarlos en nuestro fuzzer, pero no hará falta modificar el código que ya existía en la aplicación.

Cabe mencionar que para ahorrar detalles innecesarios no se implementará un menú para la modificación en línea de la configuración de la prueba, ya que no aporta nada al proceso de fuzzing propiamente dicho.

La generación de datos se dividirá en las siguientes fases:

- Leer el contenido del archivo de ejemplo.
- Realizar sustituciones aleatorias de unos bytes por otros hasta haber modificado el porcentaje del archivo indicado en la configuración.
- Volcar la nueva información en un nuevo archivo y devolver su ruta.

Para cada archivo generado se lanzará un subprocesso que llamará al programa seguido del nombre del archivo de entrada. Se esperará el tiempo especificado en el archivo de configuración, y se pueden dar los siguientes escenarios.

- Transcurre dicho tiempo y nada ha ocurrido, en cuyo caso se detiene la ejecución y se considera que el programa ha superado con éxito ese caso de prueba.
- El subprocesso termina con una excepción, en cuyo caso se recoge y se devuelve. Previamente se almacena el archivo empleado con un nombre único en la carpeta indicada en la configuración.

En la función que guarda los resultados escribe en el archivo de resultados (ubicado en la misma carpeta que los archivos) el nombre del archivo que provocó la excepción y el texto de la misma.

6.7 Séptima iteración

Durante esta iteración se va a llevar a cabo una prueba de usabilidad para descubrir posibles aspectos de la interfaz que necesiten ser mejorados para alcanzar el objetivo de la sencillez de uso.

Una prueba de usabilidad es aquella encargada de medir el grado de satisfacción que el usuario tiene respecto al uso de un producto (software en este caso) atendiendo a aspectos como la facilidad de uso o el grado en el que les permiten lograr completar sus tareas.

Son especialmente valiosas cuando una de las prioridades es lograr una aplicación de uso sencillo.

Objetivos

Los objetivos que se pretenden lograr con esta prueba son:

- Conocer los aspectos del diseño de la interfaz que pueden suponer mayor atasco al usuario
- Recoger las necesidades del usuario final para orientar la aplicación al mismo

Formato del estudio

Estas pruebas se realizarán de manera presencial en la biblioteca de la Universidad de Jaén, ya que es un lugar donde se reúnen muchos programadores y no les supondría un desplazamiento extra.

El programa estará preparado en un ordenador que se suministrará para la prueba, ya que la fase de instalación de la herramienta queda fuera del ámbito de estudio de esta prueba.

Habrà un moderador en cada prueba, la desarrolladora del proyecto, quien estará cerca para resolver las posibles dudas y anotar las reacciones del sujeto de prueba.

Se realizará un total de dos pruebas de usabilidad, cada una con un perfil de prueba distinto.

Perfil de los sujetos de pruebas

Se distinguen dos perfiles diferenciados:

- Programador/a de entre 0 y 2 años de experiencia (no necesariamente profesional) programando. Ha de ser una persona que ha tenido algún contacto con aplicaciones de corto o medio alcance, pero que no ha hecho nunca pruebas a su código o solo alguna vez aislada.
- Programador/a con al menos 1 año de experiencia profesional familiarizado con el testing de aplicaciones, aunque no haya hecho fuzzing concretamente.

El primer perfil busca medir la facilidad de uso y la intuitividad de la herramienta, mientras que con el segundo se busca conocer la utilidad de la aplicación y en qué medida cumple unas expectativas semi profesionales.

Para seleccionarlos se ha revisado un total de 20 CV de estudiantes de Ingeniería Informática de la Universidad de Jaén. Estos se han clasificado en tres grupos.

- Los que cumplen el primer perfil
- Los que cumplen el segundo perfil
- Los que no cumplen ninguno de ellos, en cuyo caso se descartan.

Quedó un total de 11 CV del primer perfil y 3 CV del segundo, de los cuales se seleccionó uno de cada grupo al azar. Sus datos personales se han anonimizado, a continuación se presentan sus cualidades profesionales:

Sujeto 1

- Años de experiencia: 6 meses de programación autodidacta.
- Experiencia en testing: Ninguna
- Experiencia en fuzzing: Ninguna

Sujeto 2

- Años de experiencia: 3 años de programación, de los cuales 9 meses han sido dentro de una empresa de desarrollo.

- Experiencia en testing: 3 meses realizando test unitarios en aplicaciones escritas en Java.
- Experiencia en fuzzing: Ninguna

Pruebas a realizar

Se han diseñado dos pruebas que serán llevadas a cabo por ambos sujetos de prueba.

Prueba 1

Título: Hacer fuzzing testing a la URL 'http://google-gruyere.appspot.com/376520606848610381617211426476648375834/' con el fuzzer de URL integrado en la herramienta

Descripción: Al usuario se le suministrará un ordenador con el fuzzer de URL ya arrancado. El usuario tendrá 20 minutos para realizar las siguientes acciones:

- Cambiar la URL objetivo
- Cambiar la carpeta de resultados a una a elegir por el usuario
- Exportar solo en formato JSON
- Realizar la prueba
- Localizar y abrir los resultados con un editor de texto

Prueba 2

Título: Implementar un nuevo fuzzer que testee URLs.

Descripción: El usuario recibirá el código fuente y el comando necesario para ejecutar el fuzzer. Éste deberá implementar en un máximo de una hora un fuzzer con las siguientes características.

- Que use como vector de ataque una URL cualquiera
- Que cambie cada parámetro (GET) numérico por un número aleatorio, y cada parámetro textual por un carácter aleatorio.

- Que muestre por pantalla la URL y los cambios que se han realizado respecto a la original. Mostrar el código HTTP recibido (no es necesario almacenar los resultados)

Además, el usuario recibirá toda la información necesaria para realizar peticiones con la librería de Python "requests" y recoger el código HTTP devuelto.

Métricas a las que se atenderá

- Tiempo de finalización de la prueba.
- Éxito o fallo en la prueba.
- Reacciones negativas del sujeto de pruebas (resoplidos, quedarse pensando un largo rato...)
- Partes de la prueba donde el usuario se atasca y por qué.

Resultados

Prueba 1

El sujeto nº 1 ha completado la prueba ocupando prácticamente todo el tiempo permitido (20 minutos). La navegación por el menú no parecía del todo cómoda, ya que solía entrar y salir de los menús antes de quedarse en uno fijo y realizar alguna acción.

Por otro lado, el sujeto nº 2 ha completado esta prueba en unos 7 minutos y no ha dado muestras de incomodidad con la interfaz. Ha comentado en voz alta, sin embargo, que la manera de elegir los formatos de entrada le parece extraña y que normalmente se usarían números en lugar de escribir el nombre del formato.

Prueba 2

El sujeto nº 1 no ha logrado completar la prueba. Parecía incómodo con tener que modificar código que él mismo no había hecho. No obstante, ha conseguido completar esa parte ocupando casi la mitad del tiempo total (30 de los 60 minutos totales). Al comenzar a implementar el fuzzer se ha atascado y no ha sido capaz de

completar los requisitos que se pedían. Ésto, sin embargo, es independiente del *fuzzer*. Probablemente la causa de esta situación es la poca experiencia programando sumado a que Python, según sus palabras, era un lenguaje que apenas había usado.

Por otro lado, el sujeto nº 2 ha completado la prueba, pero parecía visiblemente reacio a modificar código del *fuzzer* original. Parecía dudar en qué zonas debía modificar el código y de qué manera.

Conclusiones

Parece que un aspecto mejorable de la herramienta es la integración de nuevos fuzzers, concretamente la modificación del código base de la herramienta. Una posible mejora sería eliminar la necesidad de modificar ese código e incluso añadir un asistente que cree y/o modifique los archivos necesarios.

7. Conclusiones

Como resultado de este proyecto hemos conseguido desarrollar una aplicación de escritorio multiplataforma que permite realizar dos pruebas *fuzzing* distintas y que además permite una fácil integración de nuevas pruebas.

Se ha logrado satisfacer casi todos los objetivos del proyecto, siendo la excepción la inclusión de un modo de interfaz gráfica que sin duda hubiera contribuido a mejorar la usabilidad para aquellos menos experimentados en el manejo de la consola. Sin embargo, la funcionalidad de la aplicación está completa y resulta sencilla incluso a través de la consola.

Gracias a la realización de este proyecto se han adquirido sólidos conocimientos acerca de las diferentes técnicas que pueden emplear los atacantes para comprometer nuestro sistema. De esta manera no sólo mejora nuestra capacidad de detectarlas, sino también de prevenirlas cuando seamos nosotros los encargados de desarrollar software. Además, se ha puesto en práctica la metodología SCRUM, con lo que se ha experimentado lo que sería un desarrollo ágil.

Al ser el *fuzzing* una técnica con tantas posibilidades, esta herramienta es susceptible de albergar muchas características que por tiempo no han sido posibles durante el desarrollo de este proyecto.

La primera y más inmediata mejora que puede recibir es una interfaz gráfica para que los usuarios poco o nada acostumbrados al manejo de la consola sean capaces de utilizar este programa con comodidad.

Una mejora potencial que podría recibir el *fuzzer* de URLs sería el procesamiento del cuerpo del mensaje recibido como respuesta a cada petición. De esta manera se lograrían dos cosas fundamentales:

- Automatizar la detección de potenciales vulnerabilidades.
- Paliar el hecho de que muchos sitios web no devuelven códigos de estado HTTP acordes con lo que realmente ha pasado.
- Distinguir entre distintas respuestas que se reciben con el código HTTP 200 (éxito). Estas pueden descubrir accesos a lugares no permitidos o inyección de comandos.

Por último, una mejora sustancial que podría recibir esta herramienta es automatizar la inclusión de nuevos *fuzzers*, ya sea mediante un asistente por consola que edite y cree los archivos necesarios dados el nombre del *fuzzer*, o haciendo solo necesaria la creación de un archivo en el cual se ubicaría toda la información necesaria para cargarlo automáticamente.

8. Anexos

8.1. Manual de uso

El programa se distribuye en una carpeta que contiene el código fuente del mismo. En primer lugar hay que asegurarse de que en la máquina donde se va a ejecutar se encuentran instalados los siguientes paquetes e intérpretes. Para esto se recomienda instalar **pip**, un gestor de paquetes de python:

- **Python 3.6**
- **Colorama** (`pip install colorama`)
- **Console Progress Bar** (`pip install console-progressbar`)
- **Requests** (`pipenv install requests`)
- **PyFiglet**

Una vez que todos los requisitos se cumplen, a través de la consola hay que navegar hasta situarnos en la carpeta del proyecto y ejecutar `python fuzzuja.py -t <option>` donde `<option>` es un caracter de los siguientes:

- `u` para el fuzzer de URLs.
- `t` para el format file fuzzer.

Además de los anteriores, se podrán ejecutar los fuzzers que cada programador decida incluir.

En cuanto a la configuración de la prueba, esta se puede hacer cambiando los valores en el archivo **config.json** o bien desde el menú interactivo en la consola (de haberlo).

8.2. Manual del programador

En esta sección se ofrece un manual que detalla las características relevantes de esta herramienta y cómo se realizaría la integración de un nuevo *fuzzer* paso a paso.

Estructura del programa

Al inspeccionar el código fuente se puede encontrar la siguiente estructura de archivos y carpetas (sin tener en cuenta los archivos generados por el propio usuario).

- **Analyzers:** Carpeta que contiene archivos usados para el análisis de resultados.
- **Fuzzers:** Carpeta que contiene los fuzzers.
- **Payloads:** Carpeta que contiene los archivos de donde se sacan los datos para inyectar.
- **Cli.py:** Archivo desde el cual se gestiona la interfaz y se lanzan los distintos fuzzers.
- **Config.json:** Archivo de configuración del programa.
- **Fuzzuja.py:** Programa principal.

Archivo de configuración (config.json)

El propósito de este archivo es definir los parámetros de cada fuzzer. Como su propia extensión indica, su formato es JSON y consiste en un array de objetos, cada uno con las características de un *fuzzer* determinado. Los únicos atributos obligatorios que deben estar presentes en cada fuzzer son:

- **Attempts:** Número de iteraciones que realizará el fuzzer.
- **Target:** Aplicación o protocolo objetivo.
- **Option:** Opción con la que se llamará a ese *fuzzer* en particular.

No hay ningún límite de atributos ni su formato, el programador puede disponer de los que necesite.

La clase Fuzzer

En la carpeta **fuzzers** encontramos originalmente un solo archivo, **fuzzer.py**, donde se define la clase padre de la que cada fuzzer ha de heredar. Esta clase define una serie de métodos que no hacen *nada*, y deberán ser sobreescritos por el programador en la medida de lo posible. A continuación se detalla el uso que ha de darse a cada uno de ellos.

- **Constructor.** Cuando definamos nuestro propio constructor debemos inicializar el constructor padre pasándole como parámetros el objeto que contiene la configuración leída en el archivo **config.json**. Éste se almacenará en un atributo llamado **config**. Además, podemos definir tantos atributos adicionales como necesitemos.
- **Menu().** Es la función que albergará el menú de configuración de nuestro fuzzer. Es totalmente prescindible, ya que solo ofrece la posibilidad de alterar la configuración sin necesidad de modificar el archivo **config.json**. Si el programador decide implementarlo, debe saber que la función solo es llamada una vez, y que los cambios que efectúe se deberán aplicar al atributo del fuzzer *config*.
- **Setup().** Función que se llamará justo antes de iniciar las iteraciones del fuzzer. Puede ser usado para inicializar las variables necesarias, por ejemplo, para la generación de datos.
- **GenerateInput().** Esta función debe devolver un dato para inyectar. Sabiendo eso, cada programador debe diseñarla e implementarla de la manera que más se ajuste a sus necesidades. Algunos ejemplos serían tomar valores de un array e ir devolviéndolos secuencialmente, o generar un dato aleatorio en cada llamada.
- **Fuzz(inputData).** Esta función es la encargada de ejecutar la prueba y devolver los resultados. Acepta el parámetro **inputData**, el cual consiste en el dato generado por la función *generateInput()*. Nuevamente, el programador es responsable de utilizar el dato y ejecutar la prueba de la manera que necesite, ya sea haciendo alguna petición HTTP o lanzando algún subproceso.
- **Save(result).** Esta función acepta por parámetro el resultado de la función *fuzz(inputData)* y se encarga de procesarlo y almacenarlo como sea necesario.
- **Analyze().** Esta función es llamada al terminar todas las iteraciones del fuzzer. Su función sería realizar las últimas operaciones necesarias como análisis conjunto de todos los resultados obtenidos, creación de archivos de log o mostrar por pantalla las conclusiones finales.

- *Clear()*. Esta función no ha de ser sobrecargada. Sirve como ayuda al programador y su función es limpiar el contenido de la consola.
- *Banner()*. Función donde mostrar por pantalla un banner para la aplicación.

Cabe mencionar que cada una de estas funciones, a excepción de *clear()*, admiten un parámetro opcional pueden extender la utilidad y personalización de las mismas.

Crear un fuzzer paso a paso

A continuación se describen las acciones que se deben llevar a cabo para implementar un nuevo fuzzer en esta herramienta.

En primer lugar, crea un archivo en la carpeta **fuzzers** con la extensión **.py** y dale un nombre representativo según el tipo de *fuzzer* que sea. Ejemplo de *fuzzer sample*.

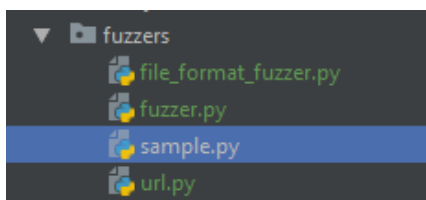


FIGURA 20. CONTENIDO DE LA CARPETA FUZZERS DESPUÉS DE AÑADIR EL FUZZER SAMPLE

Dentro del mismo crea una clase que herede de la clase **Fuzzer** ubicada en el archivo **fuzzers/fuzzer.py** (será necesario importarlo). El código que se añadiría sería algo así:

```
from fuzzers import fuzzer
class Sample(fuzzer.Fuzzer):
```

Siendo *Sample* el nombre del nuevo fuzzer.

Sobrecarga todos los métodos que sean necesarios atendiendo a la sección **La clase Fuzzer**. Los únicos obligatorios son

- a. Constructor

- b. *generateInput*
- c. *fuzz*
- d. *save*

Un posible resultado sería el siguiente:

```
class Sample(fuzzer.Fuzzer):
    def __init__(self, config):
        super().__init__(config)

    def generateInput(self, parameters=None):
        return 1

    def fuzz(self, inputData, parameters=None):
        try:
            print(inputData)
            return True
        except:
            return False

    def save(self, result, parameters=None):
        with open("log.txt", "a") as f:
            f.write(result)
```

Una vez está creado y definido el fuzzer, abre el archivo **config.json** y añade al array un objeto que contenga los atributos necesarios. Revisar la sección **Archivo de configuración** para conocer los atributos obligatorios. Un ejemplo para el caso anterior sería el siguiente:

```
{
    "option": "s",
    "target": "none",
    "attempts": 100
}
```

A continuación, abre el archivo **cli.py** y realiza las siguientes modificaciones.

1. Importa la clase del nuevo fuzzer
2. En la función *createFuzzer()* añade una sentencia *if/elif* que compruebe si la variable **fuzzer** es igual al caracter definido en **option** dentro del archivo **config.json**. De ser así, crear y devolver una instancia del *fuzzer* pasando por parámetro la variable **config**, que contiene los datos del archivo JSON para ese fuzzer. El archivo se vería de una manera similar a la siguiente:

...

```
from fuzzers.sample import Sample
init(autoreset=True)
```

```
def createFuzzer(config):
    fuzzer = config['option']
    try:
        # [2] add your custom if/elif statement below
        if fuzzer == 'u':
            return URL(config)
        elif fuzzer == 'f':
            return FileFormat(config)
        elif fuzzer == 's':
            return Sample(config)
        pass
    except Exception as err:
        print(Fore.LIGHTRED_EX + "[ERROR] Ha habido un error al
crear el fuzzer.")
        print(err)
        exit(1)
...
```

Una vez completados todos estos pasos, el nuevo *fuzzer* se ejecutaría con el siguiente comando:

python fuzzuja.py -t <option>

Donde <option> es el atributo **option** definido en el archivo de configuración.

Bibliografía

1. <https://www.owasp.org/index.php/Fuzzing>
2. <https://www.owasp.org/index.php/Fuzzing>
3. <http://bxi.es/Reversing-Exploiting/Fuzzing%20Brute%20Force%20Vulnerability%20Discovery.pdf> página 35
4. <https://www.ee.oulu.fi/research/ouspg/Protos>
5. <http://bxi.es/Reversing-Exploiting/Fuzzing%20Brute%20Force%20Vulnerability%20Discovery.pdf> página 35
6. <http://bxi.es/Reversing-Exploiting/Fuzzing%20Brute%20Force%20Vulnerability%20Discovery.pdf> página 34
7. <https://arxiv.org/pdf/1801.04589.pdf>
8. <http://bxi.es/Reversing-Exploiting/Fuzzing%20Brute%20Force%20Vulnerability%20Discovery.pdf> página 40
9. <https://wiki.python.org/moin/PyQt>
10. <https://es.wikipedia.org/wiki/Scrum>
11. <https://www.riverbankcomputing.com/software/pyqt/intro>
12. <http://zetcode.com/gui/tkinter/layout/>
13. <http://effbot.org/tkinterbook/tkinter-whats-tkinter.htm>
14. <http://www.pygtk.org>
15. <http://pyforms.readthedocs.io/en/latest/#it-offers>
16. <https://electronjs.org/docs/tutorial/about>
17. <http://www.dtic.mil/dtic/tr/fulltext/u2/a558209.pdf>
18. <http://www.dtic.mil/dtic/tr/fulltext/u2/a558209.pdf>
19. <http://www.dtic.mil/dtic/tr/fulltext/u2/a558209.pdf>
20. [https://es.wikipedia.org/wiki/Protocolo de transferencia de hipertexto](https://es.wikipedia.org/wiki/Protocolo_de_transferencia_de_hipertexto)
21. <https://www.mozilla.org/en-US/security/advisories/mfsa2008-37/>
22. (21 en realidad) fuzzing art of discovering pag 138
23. fuzzing art of discovering pag 138
24. [https://www.owasp.org/index.php/Testing_Directory_traversal/file_include_\(OTG-AUTHZ-001\)](https://www.owasp.org/index.php/Testing_Directory_traversal/file_include_(OTG-AUTHZ-001))
25. <https://developer.mozilla.org/es/docs/Web/HTTP/Status>
26. <https://kciredor.com/fuzzing-adobe-reader-for-exploitable-vulns-fun-not-profit.html>
27. <https://es.wikipedia.org/wiki/JSON>
28. [https://es.wikipedia.org/wiki/Extensible Markup Language](https://es.wikipedia.org/wiki/Extensible_Markup_Language)