UNIVERSIDADE
DE VIGO

Escola **S**uperior de **E**nxeñaría **I**nformática

Memoria del Trabajo de Fin de Grado que presenta

D. Diego Enrique Fontán Lorenzo

para la obtención del título de Graduado en Ingeniería Informática

Framework de automatización de auditorías Red Team

Julio, 2021
Trabajo Fin de Grado Nº: EI 20/21-10
Tutor/a: Miguel Ramón Díaz-Cacho Medina
Área de conocimiento: Ingeniería de sistemas y automática
Departamento: Ingeniería de sistemas y automática

A mi madre, la cual nunca pensó que llegaría este día...

Índice general

Índice de figuras

Índice de tablas

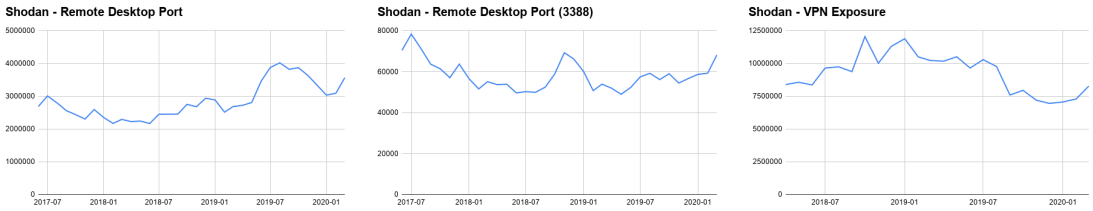
1 | Introducción

En este capítulo se detalla el marco contextual sobre el que se presenta la realización del trabajo, así como la motivación asociada al mismo. Se describen después los objetivos perseguidos, la descripción técnica de la solución propuesta y los aspectos legales. Finalmente, se expone la estructura completa de la documentación.

1.1. Marco contextual

La seguridad informática se puede definir como el conjunto de técnicas que permiten a la organización asegurar la confidencialidad, integridad y disponibilidad de su sistema de información.

La pandemia y el confinamiento han obligado a las empresas a digitalizarse. El uso de tecnologías de acceso remoto como RDP (*Remote Desktop Protocol*) y VPN (*Virtual Private Network*) ha crecido en un 41 % y 33 % (*Fig. 1.1)*, respectivamente, desde el inicio del brote del coronavirus (COVID-19) [3].



(a) RDP expuesto. (b) RDP expuesto. (alt.) (c) VPN expuesto.

Figura 1.1: Cantidad de dispositivos según el servicio expuesto. Fuente: *Shodan*.

La actual pandemia también ha tenido un gran impacto en la ciberseguridad. Los ataques informáticos contra corporaciones, así como las estafas en línea, se dispararon en más de un 400 % en marzo de 2020 en comparación con los meses anteriores, según la firma de abogados internacional *Reed Smith* [7], mientras que *Google* reveló que estaba bloqueando más de 18 millones de correos electrónicos de malware y phishing relacionados con el COVID-19 cada día [12].

Los ciberdelincuentes siguen consiguiendo comprometer los datos y sistemas corporativos con relativa facilidad y de forma regular. Esto se debe a la falta de perfiles cualificados

que sean capaces de realizar auditorías de seguridad (o *pentestings*), con el fin de asegurar los activos de la compañía [2]. Además, las tácticas empleadas por los ciberdelincuentes son cada vez más sofisticadas, haciendo que este tipo de ejercicios se vuelvan a su vez más complejos y extensos. *Gartner* ha proyectado que las empresas invirtieron más de 123.000 millones de dólares en ciberseguridad en 2020 (*Fig. 1.2)* y prevé que esa cifra aumente a 170.400 millones de dólares en 2022 [6].

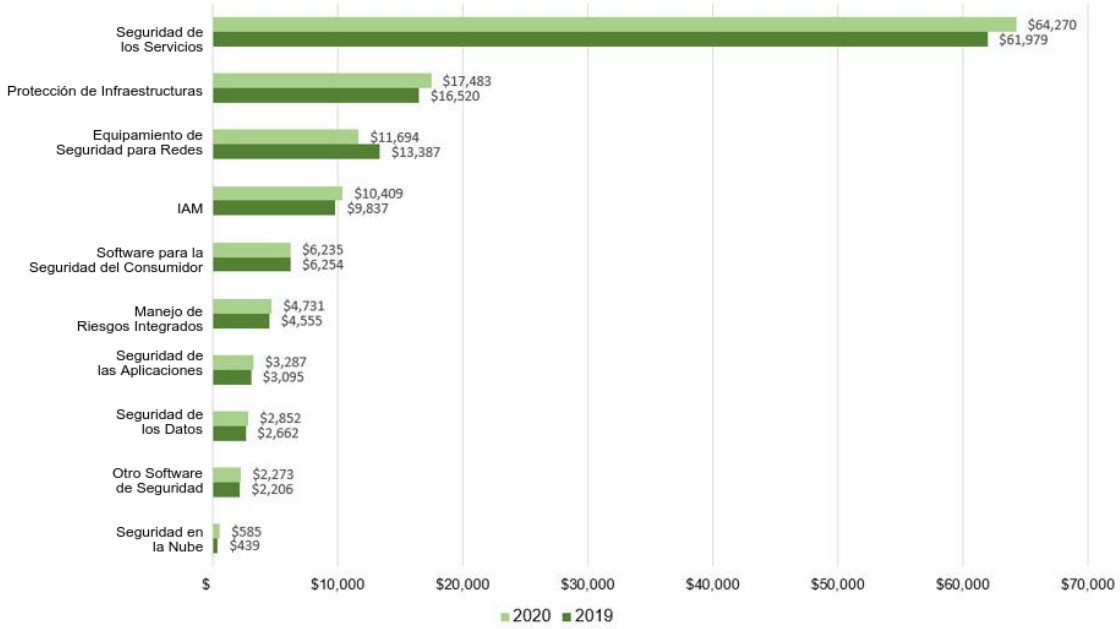


Figura 1.2: Gasto mundial en ciberseguridad (en millones de dólares). Fuente: *Forbes*.

1.2. Motivación

La seguridad informática es un campo cambiante y en constante crecimiento. La complejidad a la hora de realizar ejercicios de intrusión requiere que los auditores tengan acceso a un amplio repertorio de herramientas, donde cada una de las cuales generalmente está enfocada en tareas concretas dentro de la totalidad de la auditoría.

Además, el autor de este documento también ha experimentado la dificultad a la que se enfrenta un docente al intentar formar nuevos perfiles dentro del campo de la ciberseguridad.

Esto se debe a que las soluciones *software* actuales requieren de conocimientos informáticos avanzados, tienen una funcionalidad muy específica o sólo están disponibles para arquitecturas de sistemas concretos.

1.3. Objetivo

Este Trabajo de Fin de Grado se plantea con el objetivo de crear un proyecto de código abierto enfocado a la seguridad informática, que automatice y enlace varias de las etapas más habituales durante un *pentesting* (listado de subdominios y servicios, comprobación de filtrados de información, credenciales por defecto, vulnerabilidades conocidas...), y que su uso sea válido tanto en auditorías reales como en entornos educativos, con el fin de simplificar y desmitificar las actividades relacionadas con el mundo de la ciberseguridad, acorde a lo detallado en el apartado 1.2.

1.3.1. Objetivos específicos

Además de llevar a cabo el proyecto, existen una serie de especificaciones que ha de cumplir el programa. Dichos requisitos, detallados a continuación, son necesarios para asegurarse de que la funcionalidad es acorde a la motivación inicial por parte del autor.

1.3.1.1. Manejo visual e intuitivo

Debido a que una de las motivaciones de este proyecto es acercar el mundo de la ciberseguridad a aquellos perfiles ajenos al sector, es necesaria la opción de poder prescindir de la habitual línea de comandos. Una alternativa es reemplazarla por una interfaz gráfica, teniendo como preferencia un diseño similar a un lenguaje de programación visual. Como ejemplo de lenguajes visuales, se encuentran aquellas basadan en bloques (véase *Scratch*[1] o *AppInventor*[2]), o basadas en nodos (*Node-RED*[3] o *vvvv*[4]), entre otros, siendo este último la preferencia.

Además, se pretende que al aproximarse a este concepto, la aplicación ofrezca una experiencia de usuario similar a la de "crear recetas", pudiendo compartirlas y reduciendo así el tiempo requerido para realizar auditorías de seguridad.

1.3.1.2. Estandarización de tareas

Se requiere que la aplicación ejecute tareas repetitivas, pertenecientes a las diferentes etapas de una auditoría de ciberseguridad, que interactúan directamente con servicios y dispositivos externos.

Para ello, será necesaria la creación de una **Interfaz de Programación de Aplicaciones** (*API* en adelante). Proporcionará una manera sencilla de realizar dichas tareas mediante llamadas estandarizadas a la aplicación, independientemente del servicio o dispositivo auditado.

[1] `https://scratch.mit.edu/`
[2] `https://appinventor.mit.edu/`
[3] `https://nodered.org/`
[4] `https://vvvv.org/`

1.3.1.3. Distribución sencilla

Acorde a la motivación mencionada en el apartado 1.3.1.1, se requiere que la distribución de la aplicación sea lo más sencilla posible. Para ello, se tendrá como objetivo la compilación de todo el entorno en un único ejecutable, sin dependencias, y con soporte para los sistemas operativos y arquitecturas más populares.

Este aspecto también concuerda con la ideología de crear un programa que sirva de reemplazo para gran parte de las soluciones existentes actualmente en el mercado, al poder ejecutar varias de las tareas habituales durante una auditoría de seguridad desde una misma plataforma que no dependa de programas externos.

1.4. Descripción técnica

En este punto se especifican, de manera breve, los componentes que formarán la aplicación con el fin de contextualizar los siguientes capítulos. Los requisitos serán explicados en profundidad en los apartados 2.4 y 4, así como la arquitectura y el diseño de la misma (apartados 5 y 6, respectivamente). La solución propuesta en este documento constará, según la naturaleza de sus componentes, de varios elementos.

Si aplicamos una categorización basada en soluciones software con interfaz gráfica, podemos diferenciar la aplicación en dos partes: la **interfaz** (o *frontend*) y el **servidor** (o *backend*)[5].

Por otro lado, siguiendo una categorización enfocada en la utilidad, podemos diferenciar tres entidades (*Fig. 1.3)*:

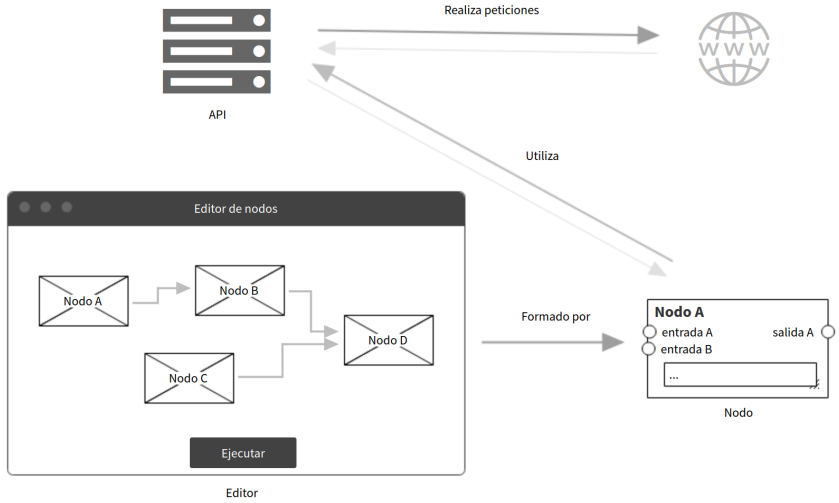


Figura 1.3: *Mockup* según la utilidad de los componentes.

[5]Puede ser objeto de debate definir a qué categoría pertenece cada uno de los elementos que conforman la aplicación. En este documento, se atribuyen al *frontend* todos aquellos elementos desarrollados mediante el uso de *HTML*, *CSS* y/o *JavaScript*. El resto, se asocian con el *backend*.

- La ***API***, encargada de manejar las peticiones y proporcionar los datos referentes a sistemas externos a la aplicación.

- Los **nodos**, encargados de encapsular tareas concretas, de las cuales se pueden obtener datos a través de ciertos valores de entrada y/o parámetros de configuración (similar a la idea de *función* dentro de la programación).

- El **editor de nodos**, encargado de interpretar los nodos, interconectarlos entre si, manejar el flujo de datos y controlar la ejecución del programa.

Además, los componentes (así como la solución final) deberán cubrir una serie de características técnicas detalladas a continuación.

1.4.1. Interfaz de Programación de Aplicaciones / API

La **API** es, posiblemente, una de las partes más delicadas en cuanto a su diseño se refiere. Será la encargada de interactuar con los activos auditados, por lo que su implementación deberá ser precisa para evitar conexiones innecesarias y/o pérdidas en el contenido de los datos recibidos. Un mal diseño no sólo ralentizaría el servicio, sino que podría sobreexponer al auditor, además de causar daños sobre los activos.

Muchas de las tareas habituales durante una auditoría de ciberseguridad no requieren de una conexión activa. Es por ello que las peticiones realizadas deberán cerrarse una vez finalizadas. En el caso de que se exija más información sobre el mismo activo se podrán reusar, siempre que sea posible, las respuestas entre las tareas que auditen características similares. De esta forma se consigue evitar la sobrecarga.

La ejecución de las rutinas encargadas de realizar las conexiones se estandarizarán usando el protocolo *HTTP*. Ésto se podrá conseguir mediante la creación de un servicio que tramite cada una de las peticiones y devuelva los resultados en formato *JSON*.

Para abarcar de manera satisfactoria estos problemas, se pretende utilizar un lenguaje de programación con acceso a librerías que permitan la creación de tramas a bajo nivel. De esta forma, se logrará obtener un mayor control sobre los datos enviados.

Por otro lado, la velocidad de ejecución es un tema a tener en cuenta, dado que se realizarán múltiples tareas en un periodo de tiempo limitado. Por lo tanto, el lenguaje seleccionado debe ser compilado (dado que ofrece un mayor rendimiento frente a las alternativas interpretadas), y con soporte para modelos de computación no secuenciales (*concurrencia* o *paralelismo*, por ejemplo).

También se pretende minimizar el riesgo de errores durante la ejecución, por lo que el lenguaje ha de ser *tipado* (variables con tipos definidos) y que ofrezca mecanismos de gestión de excepciones.

Ejemplos de lenguajes de programación que cumplen con estos requisitos son, entre otros: *C/C++*, *Rust* o *Golang*.

1.4.2. Editor de nodos

Este componente tendrá que encargarse de interpretar correctamente los nodos que lo conforman. Permite, a su vez, realizar operaciones de conexión y desconexión entre dos o más nodos, así como la personalización de los valores asociados a cada uno de ellos.

El diseño seleccionado para el editor seguirá el paradigma de programación de flujo de datos (o *dataflow programming*), el cual modela un programa como un gráfico dirigido en el que los datos fluyen entre las operaciones.

Aunque existen varios patrones de diseño relacionados con este paradigma, se valorarán principalmente dos de ellos para la implementación: los basados en un único flujo de datos unidimensional (totalmente lineal), y los basados en nodos interconectados entre sí, formando un gráfico bidimensional con varios flujos de datos simultáneos (*Fig. 1.4)*.

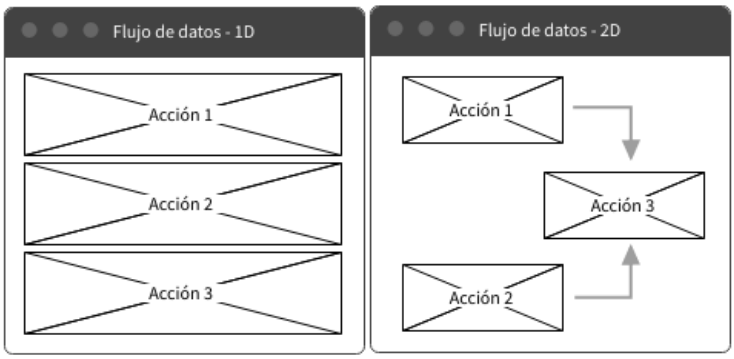


Figura 1.4: *Mockup* según el patrón de diseño.

Las aplicaciones con diseños unidimensionales[6] son más intuitivas al compartir los datos en una única dirección, pero exige que todas las acciones sean evaluadas en el orden definido, existiendo un solo flujo de ejecución.

Por el contrario, las soluciones con diseños bidimensionales ofrecen un mayor control sobre el orden de ejecución, pero su implementación requiere de un mayor esfuerzo, debido a que es necesario controlar excepciones tales como bucles infinitos, acciones huérfanas o flujos de datos independientes, entre otras (*Fig. 1.5)*.

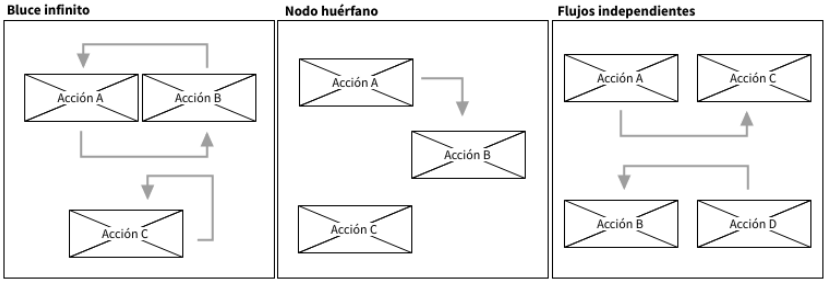


Figura 1.5: *Mockup* de errores comunes al desarrollar diseños bidimensionales.

[6]Un ejemplo de esta implementación es *Cyberchef* (`https://gchq.github.io/CyberChef/`)

Para la implementación de este diseño, es necesario que el lenguaje de programación seleccionado conste de librerías que permitan el desarrollo de interfaces altamente personalizables. Los lenguajes como *Golang* o *Rust*, mencionados en el apartado 1.4.1, carecen de dichas librerías[7], por lo que también se valorará realizar la interfaz mediante lenguajes web.

Los nodos deberán seguir un estándar y ofrecer la capacidad de definir nuevos a través del uso de archivos parametrizados que consten, por ejemplo, de los siguientes valores:

- **Identificador**: Cadena única usada para referenciar al nodo.
- **Nombre**: Texto identificativo mostrado en el editor.
- **Categoría**: Grupo al que pertenece según su funcionalidad.
- **Descripción**: Documentación relativa a su utilidad.
- **Etiquetas**: Palabras descriptivas por las que filtrar al nodo.
- **Entradas**: Valores de entrada requeridos.
- **Salidas**: Valores resultantes de la ejecución del nodo.
- **Controles**: Parámetros de configuración.
- **Código**: Método utilizado para generar los valores de salida.

Los flujos de datos también deberán poder exportarse e importarse mediante ficheros, los cuales especificarán, al menos:

- **Identificador**: Cadena usada para referenciar los tipos de nodos soportados.
- **Nodos**: Resumen de los nodos presentes en el editor, así como las conexiones entre ellos.

1.4.3. Binario precompilado multiplataforma

De acuerdo al objetivo detallado en el punto 1.3.1.3, la aplicación ha de ser fácil de distribuir. Para ello, será necesario que el lenguaje seleccionado sea capaz de procesar plantillas *HTML*. Dichas plantillas formarán parte del código fuente, con la intención de reducir el número de archivos que formen la aplicación a un único fichero ejecutable. Además, esto requiere que sea un lenguaje compilado, al igual que en el apartado 1.4.1.

Por otro lado, el programa ha de ser multiplataforma y sin dependencias de librerías o programas externos.

Existen compiladores (como el nativo de *Golang*) que permiten generar binarios teniendo como objetivo cualquier sistema operativo y plataforma, independientemente de la arquitectura del sistema anfitrión.

[7] Algunos de ellos ni siquiera ofrecen soporte gráfico, como *Rust*. (`https://www.areweguiyet.com/`)

1.5. Aspectos legales

A continuación se detallan los aspectos legales asociados al proyecto. Debido a la naturaleza del mismo, **el uso de la aplicación resultante puede ser un acto constitutivo de delito**, según la regulación de cada país.

En España, se entiende por **Delito Informático** todo aquel ataque que se produce contra el derecho a la intimidad, delitos de descubrimiento y revelación de secretos mediante el apoderamiento y difusión de datos reservados registrados en ficheros o soportes informáticos. (*Art.197-201 Código Penal* [8]).

Concretamente, considera delito de **sabotaje informático** a toda aquella acción que produzca daños mediante la destrucción o alteración de datos, programas o documentos electrónicos contenidos en redes o sistemas informáticos (*Art. 263* [8]), y delito de **fraude informático** a las estafas a través de la manipulación de datos o programas para la obtención de un lucro ilícito (*Art. 248*[8]).

Además, en la reforma realizada en 2010 [9], la detección de vulnerabilidades informáticas se considera acto de delito. Cualquier persona que detecte vulnerabilidades en aplicaciones o sitios web, sin consentimiento explícito de los interesados, puede acabar con consecuencias legales. En otras palabras, **realizar un acceso no consentido, sin autorización, será considerado hecho válido de sanción, aunque no exista intención de cometer un delito**. El usuarios que comete el acto puede ser sancionado con una pena de prisión de entre seis meses y dos años.

Las personas jurídicas dedicadas a la seguridad informática deben conocer esta ley, y todas las consecuencias que pueden ocurrir en el incumplimiento de la misma. Es recomendable que cualquier individuo al cargo de una auditoría de seguridad sea informado y tenga consciencia de ello, entendiendo el ámbito en el que se encuentra y a las leyes que está sujeto. Actualmente, el Código Penal no hace distinción entre un investigador de seguridad que actúa sin consentimiento y un delincuente que se aprovecha de las vulnerabilidades para obtener beneficio propio.

Respecto al uso de la aplicación producto del proyecto, **el autor de este documento no se hace responsable en caso de que se presenten cargos penales contra cualquier individuo o corporación que utilice la herramienta en contra de las leyes estipuladas**, así como de los daños causado por un mal uso de la misma. Es responsabilidad del usuario final obedecer todas las leyes aplicables.

Se recomienda que su uso sea limitado a entornos controlados, tales como entornos educativos, y/o pruebas de intrusión bajo autorización previa de los interesados.

1.6. Organización de la documentación

A continuación se detalla la documentación relativa al proyecto, así como su estructura.

1.6.1. Memoria del proyecto

Se trata del documento actual. En él se recogen formalmente todos los aspectos del proyecto, desde sus primeras fases, así como las decisiones arquitectónicas tomadas, el proceso completo de desarrollo, la documentación generada y las conclusiones a las que se han llegado como consecuencia de la realización del ejercicio.

Además, se adjuntan tres anexos correspondientes a:

- **Anexo A**: La bibliografía consultada durante la investigación inicial.

- **Anexo B**: El manual de usuario para la correcta instalación y uso del producto desarrollado.

- **Anexo C**: Resumen de los controles típicos durante una auditoría de seguridad web según la metodología descrita en la *OWASP Web Security Testing Guide*[10].

1.6.2. Documentación del código

El código de la aplicación se ha comentado detallando las funcionalidades de las partes más importantes.

Es posible visualizar de forma *online* a la documentación de las clases definidas en el apartado 6.3. La *URL* de acceso es: `https://pkg.go.dev/github.com/cosasdepuma/masterchef`.

Por otro lado, se pone a disposición del usuario el archivo *swagger.yml*. Esto permite documentar las llamadas a la *API*, usando la tecnología descrita en el apartado 3.6.

1.6.3. Repositorio público

Debido a que éste es un proyecto de código abierto, todos los archivos a los que se hacen referencia en esta memoria se encuentran alojados en un repositorio público. Se puede consultar y realizar un seguimiento, a través de: `https://github.com/CosasDePuma/Masterchef`.

2 | Planificación y seguimiento

En este capítulo se detalla el la metodología de trabajo seguida para la correcta realización del mismo, así como la planificación inicial, la descripción de las fases, la recogida de requisitos y las desviaciones respecto al plan sufridas durante el proceso de desarrollo.

2.1. Metodología de trabajo

A continuación se explican las metodologías utilizadas a lo largo de la vida del desarrollo de este proyecto.

Dichas metodologías han sido escogidas acorde a una serie de características derivadas de la naturaleza de la aplicación desarrollada, así como las habilidades y recursos disponibles por parte del autor de este documento.

2.1.1. Metodología *eXtreme Programming*

Para el desarrollo del proyecto se ha optado por implementar la metodología de Programación Extrema (o *eXtreme Programming*).

Catalogada como una metodología ágil, tiene como base la simplicidad y adaptabilidad, en vez de la previsibilidad. Esto permite que sea más natural hacer cambios sobre la marcha en los requisitos, en vez de tratar de definirlos todos al comienzo del proyecto. Es decir, conforme avance el proyecto, se adaptará el código a las necesidades que vayan surgiendo.

2.1.1.1. Características

Sus principales características son las siguientes:

- Se diseña y construye una base sólida sobre la que añadir nuevas características, mediante un **desarrollo iterativo e incremental**.

- Las capturas de requisitos deben ser escritas de manera breve y usando lenguaje natural por el cliente. A cada una de estas capturas se le conoce con el nombre de

historias de usuario. Normalmente generan una nueva funcionalidad del sistema y deben ser independientes entre sí.

- Apuesta por mantener el **código lo más simplificado posible**, realizando una refractorización del mismo siempre que el desarrollador lo considere adecuado.

- Antes de desarrollar cualquier funcionalidad nueva, se debe establecer un conjunto de objetivos, así como una **batería de pruebas unitarias**, preferiblemente automatizadas.

- Antes de generar una nueva característica, es necesario **corregir todos los errores detectados en la iteración actual**, además de revisar y documentar cada una de las pruebas unitarias.

- Se prioriza generar la **documentación de manera** integrada en los archivos fuente del *software* y de forma simultánea al desarrollo, de manera que no existan iteraciones con segmentos de código sin documentar.

- La **integración del cliente como parte del equipo** es necesaria para saber si el requerimiento cumple con las expectativas.

- Se recomienda la **programación en parejas**, donde ambos integrantes comparten las actividades y responsabilidades referentes a un mismo código o proyecto.

2.1.1.2. Limitaciones

Si bien la metodología expuesta encaja con el proyecto propuesto, no todas las características descritas en el apartado 2.1.1.1 se ajustan a las restricciones inherentes a un Trabajo de Fin de Grado.

En este caso concreto, el desarrollo de la aplicación no sigue la recomendación de la programación en parejas. Todo el proceso ha sido llevado a cabo por una única persona (el autor de este documento).

Por otro lado, debido a que el cliente interesado en el producto no puede definirse con exactitud[1], se ha designado que sea el tutor del proyecto quien se encargue este rol, teniendo en cuenta las limitaciones de horarios y medios del mismo. El resto de roles pertenecientes a un equipo basado en esta metodología (*Tester*, *Coach*, *Manager*...) recaen bajo el autor del trabajo.

Por último, también será el autor quien genere gran parte de las historias de usuario, dado que se trata de un proyecto propio, el cual no ha sido propuesto por el tutor ni la Universidad.

[1]Dependiendo de la opinión personal de cada lector, el cliente podría considerarse la propia Universidad, el tutor del proyecto o el público objetivo del trabajo, entre otros.

2.1.2. Anexo: Metodología *SCRUM*

Debido a que la documentación generada usando la metodología propuesta en el apartado 2.1.1 no contempla de manera detallada la planificación del trabajo mediante la distribución en fases y tareas propias de un Trabajo de Fin de Grado, se ha optado por complementar este paradigma con otro proceso alternativo: la metodología *SCRUM*.

2.1.2.1. Características

Las características implementadas en este proyecto son:

- El desarrollo incremental de los requisitos del proyecto se organiza en **bloques temporales cortos y fijos**.

- Se establecen **tiempos máximos para lograr objetivos**.

- **Se agrupan las tareas independientes en *sprints***, que a su vez harán de iteraciones.

Dichas características son aplicadas solamente para la documentación del proyecto, dado que el desarrollo se llevará a cabo respecto a lo mencionado en el apartado 2.1.1.

2.2. Planificación inicial

El desarrollo de la aplicación se ha dividido en ocho partes, con una estimación temporal asociada a cada una de ellas en función de la carga de trabajo (*Tab. 2.1)*, con jornadas de ocho horas y semanas laborables de cinco días.

Iteración	**Estimación** (en horas)
Investigación inicial	32.0
Especificación de requisitos	16.0
Diseño del *software*	24.0
Sprint 1: Implementación de la interfaz	80.0
Sprint 2: Implementación de la *API*	48.0
Sprint 3: Implementación de los nodos	40.0
Sprint 4: Integración del sistema	24.0
Elaboración del documento	96.0
Total	360.0

Tabla 2.1: Estimación temporal según la planificación inicial.

2.2.1. Diagrama de Gantt

A continuación se muestra la planificación temporal mediante un cronograma (*Fig. 2.1)*. El eje horizontal está asociado con el marco del tiempo (en días) y, el vertical, con cada una de las iteraciones.

Figura 2.1: Diagrama de Gantt.

2.3. Descripción de las fases

En este punto se procede a detallar cada una de las iteraciones descritas en el apartado 2.2. Además, por cada iteración, se definirán una serie de subtareas junto a su estimación temporal.

2.3.1. Investigación inicial

Primera fase del proceso de desarrollo mediante la cual se realizará el estudio del arte, tomando como antecedentes aquellos proyectos que compartan características similares a la idea propuesta, ya sea en su motivación o en su implementación. Durante esta fase también se valorarán todos los conceptos relativos a lenguajes de programación, protocolos, patrones de diseño y metodologías que puedan ser objeto de interés durante el desarrollo.

Las tareas asociadas a esta fase son:

Tarea	**Estimación** (en horas)
Investigación sobre *software* similar	8.0
Investigación técnica	12.0
Investigación legal	4.0
Documentación relativa a ciberseguridad	8.0
Total	32.0

Tabla 2.2: Tareas y estimación temporal de la investigación inicial.

2.3.2. Especificación de requisitos

La fase de especificación de requisitos consiste en la creación de las historias de usuario iniciales, las cuales formarán el núcleo de la aplicación. También se encargará de definir las herramientas necesarias para la realización del trabajo.

Las tareas asociadas a esta fase son:

Tarea	**Estimación** (en horas)
Definición de las historias de usuario	8.0
Definición de tecnologías y lenguajes	4.0
Planificación de los *sprints*	4.0
Total	16.0

Tabla 2.3: Tareas y estimación temporal de la especificación de requisitos.

2.3.3. Diseño de la aplicación

En esta fase se valorará la arquitectura inicial de la aplicación. Es necesaria para concretar las bases sobre las que se comenzará a desarrollar el programa. Aún así, al tratarse de la metodología *eXtreme Programming*, es posible que ésta se transforme a lo largo de todo el proceso de desarrollo.

Las tareas asociadas a esta fase son:

Tarea	**Estimación** (en horas)
Diseño de la arquitectura	12.0
Bocetaje de posibles flujos de datos	8.0
Bocetaje de la interfaz	4.0
Total	24.0

Tabla 2.4: Tareas y estimación temporal del diseño de la aplicación.

2.3.4. *Sprint* 1 - Implementación de la interfaz

El primer *sprint* está dedicado a la creación de la interfaz visual. Se encargará de implementar cómo interpretará la aplicación el flujo de datos y todo lo referente a la interacción con el usuario.

A pesar de tratarse de una aplicación híbrida, y al contrario que en los procesos de desarrollo de aplicaciones para escritorio (donde la interfaz generalmente es lo último en implementarse), se ha optado por una aproximación similar al proceso de desarrollo web, en el que el *frontend* es el protagonista.

Las tareas asociadas a esta fase son:

Tarea	**Estimación** (en horas)
Creación de pruebas unitarias	8.0
Diseño del modelo de flujo de datos	36.0
Implementación del flujo de datos	16.0
Diseño de la interfaz gráfica	20.0
Total	80.0

Tabla 2.5: Tareas y estimación temporal de la implementación de la interfaz.

2.3.5. *Sprint* 2 - Implementación de la *API*

El segundo *sprint* está dedicado a la creación de la *API*. Se encargará de crear el servidor *HTTP*, definir el enrutamiento e implementar las peticiones a terceros.

Las tareas asociadas a esta fase son:

Tarea	**Estimación** (en horas)
Creación de pruebas unitarias	8.0
Implementación del servidor	12.0
Implementación del *router*	12.0
Implementación de los *middlewares*	4.0
Implementación de las acciones relativas a terceros	12.0
Total	48.0

Tabla 2.6: Tareas y estimación temporal de la implementación de la *API*.

2.3.6. *Sprint* 3 - Implementación de los nodos

El tercer *sprint* consta de la definición de un estándar asociado a la creación de los nodos, así como la propia creación de varios nodos de ejemplo.

Los tipos de nodos resultantes fruto de esta iteración formarán parte del cuerpo de la aplicación, pero se dedicará más tiempo y recursos a una sólida estandarización, con el objetivo de proporcionar a la comunidad la habilidad de diseñar, crear y compartir sus propios nodos.

Las tareas asociadas a esta fase son:

Tarea	**Estimación** (en horas)
Creación de pruebas unitarias	8.0
Estandarización de los nodos	16.0
Implementación de los nodos	8.0
Creación de varios nodos de ejemplo	8.0
Total	40.0

Tabla 2.7: Tareas y estimación temporal de la implementación de los nodos.

2.3.7. *Sprint* 4 - Integración del sistema

El último *sprint* corresponde a la interconexión de los módulos desarrollados en las etapas anteriores del desarrollo. Será el encargado de unificar la interfaz gráfica con el servidor, además de preparar los archivos necesarios para su correcta distribución.

Las tareas asociadas a esta fase son:

Tarea	**Estimación** (en horas)
Creación de pruebas unitarias	8.0
Interconexión de los módulos	12.0
Creación de los archivos y *scripts* de distribución	4.0
Total	24.0

Tabla 2.8: Tareas y estimación temporal de la integración del sistema.

2.3.8. Elaboración de la documentación

Tras el desarrollo del proyecto, el último marco temporal está reservado para la maquetación del documento actual, así como de los Anexos y demás archivos referentes a la documentación del Trabajo de Fin de Grado.

Las tareas asociadas a esta fase son:

Tarea	**Estimación** (en horas)
Memoria del proyecto	80.0
Anexos	16.0
Total	96.0

Tabla 2.9: Tareas y estimación temporal de la elaboración del documento.

2.4. Historias de usuario

Durante la tarea descrita en el apartado 2.3.2 y en relación a la metodología usada en este proyecto (ver apartado 2.1), los requisitos de la aplicación se han definido usando **historias de usuario**, que actuarán a su vez de *Product Backlog*[2] (relativo a la metodología *SCRUM*). A continuación, se detallan cada una de ellas de manera numerada siguiendo el formato **HUXX**, donde *XX* corresponde a un número de identificación arbitrario[3].

Todas las historias de usuario deben comenzar por el texto "Deseo ..." y tienen una prioridad asociada del 1 al 100 (siendo 1 la más baja y 100 la más alta).

Los requisitos iniciales especificados son:

ID	HU01
Título	Deseo que la aplicación tenga interfaz gráfica
Descripción	La aplicación debe poder ser manejable mediante el uso del ratón
Prioridad	90
Iteración	*Sprint* 1

Tabla 2.10: Historia de usuario – 01.

ID	HU02
Título	Deseo que sea capaz de ejecutar tareas propias de un *pentesting*
Descripción	La aplicación debe poder realizar controles asociados a una auditoría
Prioridad	99
Iteración	*Sprint* 2

Tabla 2.11: Historia de usuario – 02.

ID	HU03
Título	Deseo que se puedan declarar varios activos a auditar
Descripción	La aplicación debe asociar tareas a uno o más objetivos concretos
Prioridad	25
Iteración	*Sprint* 2

Tabla 2.12: Historia de usuario – 03.

[2]Listado de todas las tareas que se pretenden hacer durante el desarrollo del proyecto.

[3]El orden de las historias de usuario no viene determinado por su orden de implementación. Para reflejar este aspecto, se ha optado por asociar cada una de ellas a los *sprints* definidos en el apartado 2.1.

ID	HU04
Título	Deseo tener una lista de controles de seguridad por defecto
Descripción	La aplicación debe proporcionar un catálogo de tareas habituales
Prioridad	60
Iteración	*Sprint* 3

Tabla 2.13: Historia de usuario – 04.

ID	HU05
Título	Deseo poder crear mis propios controles de seguridad
Descripción	La aplicación debe poder interpretar tareas definidas bajo un estándar
Prioridad	40
Iteración	*Sprint* 3

Tabla 2.14: Historia de usuario – 05.

ID	HU06
Título	Deseo poder ejecutar controles de manera condicional
Descripción	Las tareas deben poder compartir datos entre sí
Prioridad	70
Iteración	*Sprint* 1

Tabla 2.15: Historia de usuario – 06.

ID	HU07
Título	Deseo poder definir el momento en el que se ejecutan las tareas
Descripción	La aplicación debe soportar señales de evento
Prioridad	65
Iteración	*Sprint* 1

Tabla 2.16: Historia de usuario – 07.

ID	HU08
Título	Deseo poder configurar controles en base a parámetros
Descripción	Las tareas similares deben fusionarse en una única tarea personalizable
Prioridad	10
Iteración	*Sprint* 3

Tabla 2.17: Historia de usuario – 08.

ID	HU09
Título	Deseo poder guardar plantillas de auditorías
Descripción	La aplicación debe proporcionar un sistema de guardado
Prioridad	8
Iteración	*Sprint* 1

Tabla 2.18: Historia de usuario – 09.

ID	HU10
Título	Deseo poder compartir plantillas de auditorías
Descripción	La carga de plantillas debe estar sujeta a un control de versiones
Prioridad	5
Iteración	*Sprint* 1

Tabla 2.19: Historia de usuario – 10.

ID	HU11
Título	Deseo que el programa no tenga dependencias
Descripción	La ejecución no debe depender de archivos o programas de terceros
Prioridad	85
Iteración	*Sprint* 4

Tabla 2.20: Historia de usuario – 11.

ID	HU12
Título	Deseo que el programa sea multiplataforma
Descripción	El binario debe soportar sistemas operativos y arquitecturas populares
Prioridad	80
Iteración	*Sprint* 4

Tabla 2.21: Historia de usuario – 12.

ID	HU13
Título	Deseo poder auditar objetivos que no pertenezcan a mi red local
Descripción	La aplicación debe poder realizar peticiones a través de Internet
Prioridad	95
Iteración	*Sprint* 2

Tabla 2.22: Historia de usuario – 13.

ID	HU14
Título	Deseo poder recopilar información de fuentes abiertas
Descripción	La aplicación debe poder obtener datos de servicios concretos
Prioridad	20
Iteración	*Sprint* 2

Tabla 2.23: Historia de usuario – 14.

ID	HU15
Título	Deseo que los controles estén agrupados por categorías
Descripción	Las tareas deben pertenecer a una categoría concreta
Prioridad	2
Iteración	*Sprint* 3

Tabla 2.24: Historia de usuario – 15.

ID	HU16
Título	Deseo poder encontrar rápidamente un control concreto
Descripción	Las tareas se podrán filtrar por nombre o palabras clave
Prioridad	7
Iteración	*Sprint* 3

Tabla 2.25: Historia de usuario – 16.

ID	HU17
Título	Deseo poder visualizar los resultados de múltiples maneras
Descripción	La aplicación proporcionará diferentes tipos de nodos de salida
Prioridad	6
Iteración	*Sprint* 3

Tabla 2.26: Historia de usuario – 17.

ID	HU18
Título	Deseo que la aplicación cuente con mecanismos de protección
Descripción	La implementación deberá seguir ciertos estándares de seguridad
Prioridad	75
Iteración	*Sprint* 4

Tabla 2.27: Historia de usuario – 18.

2.5. Seguimiento de la planificación

El seguimiento se ha ejecutado utilizando la metodología *Kanban* [5], la cual también se denomina *sistema de tarjetas*, pues en su implementación más sencilla se usan tarjetas sobre un tablero segmentado en tres partes fundamentales:

- Tareas pendientes
- Tareas en curso
- Tareas finalizadas

Según la metodología descrita en el apartado 2.1.1, el segmento de *Tareas en curso* solamente deberá contener un máximo de una tarea. Además, se han agrupado las historias de usuario según el *sprint* al que pertenecen, mediante un código de colores, siguiendo la especificación del apartado 2.4 (*Fig. 2.2)*.

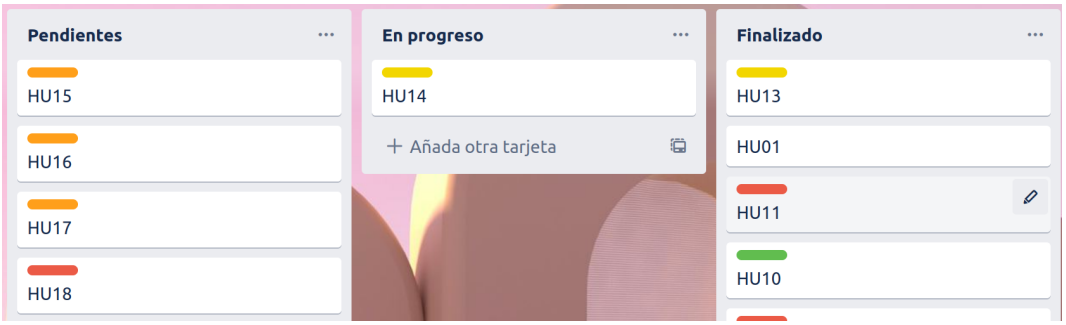


Figura 2.2: Estado del tablero *Kanban* durante el desarrollo.

2.6. Desviaciones respecto a la planificación inicial

A lo hora de desarrollar el proyecto, se han observado algunas irregularidades en relación a los tiempos marcados en el apartado 2.2. Dichas irregularidades son comunes al realizar aplicaciones debido a que la guía temporal que ofrece la planificación inicial, aunque contemple la corrección de errores y contratiempos, se trata solamente de una valoración subjetiva de la dimensión de las tareas.

A continuación se detallan las desviaciones respecto a la planificación que se han ido encontrado a lo largo del desarrollo.

2.6.1. Investigación inicial

Esta fase se consiguió completar un 400 % más rápido de lo previsto, debido a que el autor contaba con experiencia previa relativa a temas de ciberseguridad, así como sus conceptos legales, además del conocimiento sobre la existencia de herramientas similares.

2.6.2. *Sprint* 1 - Implementación de la interfaz

Durante la implementación de la interfaz, se encontraron varios errores asociados al paradigma de programación basado en flujos de datos, que retrasaron el *sprint* seis días laborables más de lo previsto (48 horas). Esto fue debido a una incompatibilidad entre el diseño y el desarrollo que obligó a tener que pausar la tarea en curso y rediseñar por completo el modelo de flujo de datos.

Para compensar este imprevisto, se optó por prescindir de la creación de pruebas unitarias al comienzo de cada *sprint*, reemplazándolas por pruebas manuales en paralelo con el desarrollo.

2.6.3. Elaboración de la documentación

Por último, se decidió dedicar 4 horas más a la elaboración de este documento, durante las cuales se diseñó una plantilla haciendo uso de herramientas especializadas en la creación de documentos científicos (ver apartado 3.7), con el fin de mejorar la calidad de la documentación.

2.7. Resultado de la planificación

Tras la realización del proyecto, se ha analizado el resultado de la planificación inicial (Apartado 2.2).

A pesar los imprevistos detallados en el apartado 2.6, la variación total del tiempo se redujo a cuatro horas de diferencia, excediendo la fecha de finalización un día más de lo previsto (*Tab. 2.28)*.

Iteración	**T. estimado** (en horas)	**T. real** (en horas)
Investigación inicial	32.0	8.0
Especificación de requisitos	16.0	16.0
Diseño del *software*	24.0	24.0
Sprint 1: Implementación de la interfaz	80.0	128.0
Sprint 2: Implementación de la *API*	48.0	40.0
Sprint 3: Implementación de los nodos	40.0	32.0
Sprint 4: Integración del sistema	24.0	16.0
Elaboración del documento	96.0	100.0
Total	360.0	364.0

Tabla 2.28: Estimación temporal real en comparación con la planificación inicial.

2.7.1. Diagrama de Gantt

A continuación se muestra el tiempo real dedicado mediante un cronograma (*Fig. 2.3)*. El eje horizontal está asociado con el marco del tiempo y, el vertical, con cada una de las iteraciones. Además, también se resaltan las desviaciones respecto a la planificación inicial.

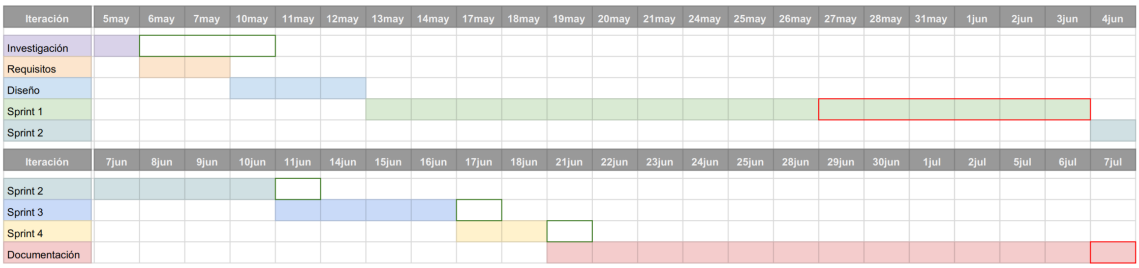


Figura 2.3: Diagrama de Gantt resultante.

3 | Tecnologías y lenguajes

En este capítulo se detallan las tecnologías y lenguajes de programación utilizados durante la realización del trabajo, así como la integración de productos de terceros, cuyo desarrollo no pertenece al autor de este documento.

Ninguna de las tecnologías aquí mencionadas es vinculante al proyecto, pudiendo reemplazarse cada una de ellas por cualquiera de sus alternativas. Todas las elecciones de esta lista se han realizado valorando los conocimientos del autor, así como la comodidad durante el desarrollo.

3.1. Golang

`https://golang.org/`

Golang (o *Go*) es un lenguaje de programación enfocado en la concurrencia, desarrollado por *Google*. Está inspirado en la sintaxis de *C*, siendo a su vez un lenguaje que cuenta con un compilador y gestor de dependencias propio. Es dinámico como *Python*, pero destaca al ofrecer un rendimiento similar a *C* o *C++*.

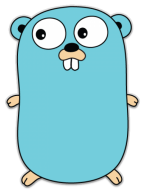


Figura 3.1: Mascota de Golang. Gopher.

Será el lenguaje de programación principal del programa, encargado de levantar el servidor, manejar las peticiones *HTTP* y servir los archivos estáticos.

La decisión se ha llevado a cabo debido a que ofrece grandes herramientas de trabajo que permiten la ejecución de tareas concurrentes, así como las denominadas *go tools*, enfocadas en proporcionar al programador la capacidad de formatear, evaluar y documentar el código, entre otras cosas. También cuenta con un compilador multiplataforma, independiente del sistema operativo y arquitectura anfitrión, con capacidad de detección de condiciones de carrera (o *race conditions*), además de un gestor de paquetes integrado (similar a *pip* en *Python* o *Gem* en *Ruby*).

3.2. Vue.js

https://vuejs.org/

Vue (pronunciado */vju:/*, como *view*) es un *framework* para construir interfaces de usuario. Se enfoca principalmente en la capa de visualización, convirtiéndose así en una elección ideal para el desarrollo de aplicaciones web del tipo *Single-Page*. Además, ofrece herramientas para utilizarlo o integrarlo fácilmente con otras librerías o proyectos existentes.

Figura 3.2: Logo de Vue.

En este proyecto, será el encargado de manejar la interfaz de usuario. En un principio se planteó el desarrollo usando *React*[1], pero al final se decidió utilizar *Vue* solamente con la intención de aprender a usarlo.

3.3. Yarn

https://yarnpkg.com/

Yarn es un instalador de módulos *JavaScript* y gestor de dependencias desarrollado por *Facebook*, en colaboración con otras organizaciones (como *Google*). Es muy rápido y muy fácil de usar, debido a que fue concebido teniendo la seguridad y el rendimiento como objetivos prioritarios.

Figura 3.3: Logo de Yarn.

Su integración en este proyecto ofrece comodidad al desarrollar. Se puede sustituir por *npm* (su alternativa directa), o prescindir completamente del gestor de dependencias utilizando copias locales del software de terceros requerido.

3.4. GNU Make

https://www.gnu.org/software/make/

Make es una herramienta pensada en dirigir la compilación o generación automática de un proyecto. Ofrece una sintaxis propia enfocada en definir tareas con las que ejecutar los comandos necesarios para llevar a cabo el proceso de compilación.

Figura 3.4: Logo de GNU.

En este proyecto, proporcionará una manera automatizada de generar el programa realizando todas las tareas de compilación necesarias.

[1] *React* es una biblioteca *Javascript* de código abierto diseñada para crear interfaces de usuario. https://reactjs.org/

3.5. Docker

https://docker.com/

Docker es un proyecto enfocado en automatizar el despliegue de aplicaciones dentro de contenedores. Proporcionan una capa adicional de abstracción y permite la automatización de la virtualización de aplicaciones en múltiples sistemas operativos.

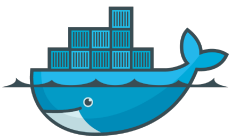


Figura 3.5: Logo de Docker.

Su uso está reservado para poder realizar también la distribución de la aplicación final en forma de contenedor a través de un fichero *Dockerfile*. Además, proporciona la opción de orquestar un entorno completo de desarrollo mediante *docker-compose*.

3.6. Swagger

https://swagger.io/

Swagger es un conjunto de herramientas de software de código abierto para diseñar, construir, documentar, y utilizar servicios web *RESTful*. Consta de una sintaxis basada en *JSON* o *YAML* para generar documentación automatizada tanto de código como de casos de prueba.

Figura 3.6: Logo de Swagger.

En este proyecto, proporcionará la documentación de las llamadas relativas a la *API*, así como sus requisitos, códigos de respuesta, el modelo de los parámetros y ejemplos de uso.

La documentación generada no será incluida en la versión compilada debido a que se consideraría una vulnerabilidad del tipo *Information Disclosure*. Además, al ser un *software* de terceros, podría causar nuevos vectores de ataques contra la aplicación.

3.7. LaTeX

https://latex-project.org/

"LaTeX *es un sistema de composición tipográfica de alta calidad. Incluye funciones diseñadas para la producción de documentación técnica y científica. Es el estándar de facto para la comunicación y publicación de documentos científicos, además de que está desarrollado bajo la ideología de software libre." –Wikipedia*

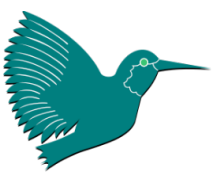


Figura 3.7: Logo de LaTeX.

Es el sistema con el que está generado este documento. Su elección se debe a su gran capacidad de personalización de artículos mediante código, similar a un lenguaje de programación.

3.8. Otras tecnologías

A continuación, se describen otras tecnologías y herramientas secundarias para la realización de este proyecto:

Git: Control de versiones pensando en la eficiencia, la confiabilidad y la compatibilidad. Es un estándar para la gestión de repositorios. `https://git-scm.com/`

GitHub: Web enfocada a alojar proyectos utilizando el sistema de control de versiones *Git*. `https://github.com/`

Visual Studio Code: Editor de código fuente. Ofrece soporte para la depuración, control de versiones, resaltado de sintaxis, finalización inteligente de código y refactorización. `https://code.visualstudio.com/`

UPX: Empaquetador de ejecutables, portable y de alto rendimiento. Además, incluye funciones de protector de software. `https://upx.github.io/`

VMware Workstation Pro: Hipervisor alojado para la gestión de máquinas virtuales. `https://www.vmware.com/products/workstation-pro.html`

Trello: Tablón virtual versátil e intuitivo usado para cualquier tipo de tarea que requiera organizar información. `https://trello.com`

MockFlow: Herramienta de *wireframing*. Permite crear *wireframes* y *mockups* en la nube. `https://mockflow.com`

Google Sheets: Programa de hoja de cálculo de *Google* basado en la web. `https://docs.google.com/spreadsheets`

Como mención especial, se ha utilizado la librería **ReteJS** (`https://rete.js.org/`) para manejar el flujo de datos. Las limitaciones de la misma, como pueden ser los eventos personalizados, controles sobre el estado del editor o su correcta integración con *VueJS*, han sido resueltas mediante *plugins* creados por el autor de este documento.

4 | Análisis de requisitos

En este capítulo se resumen los requisitos descritos en el apartado 2.3.2, agrupándolos por funcionalidad y categoría, utilizando un lenguaje natural.

4.1. Requisitos funcionales

El sistema debe ser capaz de realizar auditorías de seguridad

Actualmente existen múltiples metodologías enfocadas a la realización de auditorías de ciberseguridad, como pueden ser *OWASP WSTG* [10] o *MITRE ATT&CK* [1]. Dichas metodologías definen una serie de controles o pruebas que deben ser llevadas a cabo por un auditor para garantizar la seguridad de los activos.

La aplicación debe poder realizar tareas con el fin de garantizar el seguimiento de los controles más habituales.

Las historias de usuario relativas a este requisito son: **HU02**, **HU04**, **HU09**, **HU13** y **HU14**.

El usuario podrá definir diferentes flujos de ejecución

Es importante que las tareas se realicen en un orden concreto, proporcionando al usuario los mecanismos para controlarlo, incluyendo la habilidad de establecer condiciones o de definir varios flujos de datos simultáneos.

También debe ser capaz emitir eventos que inicien el flujo del programa, así como visualizar el estado actual del mismo.

Las historias de usuario relativas a este requisito son: **HU03**, **HU06**, **HU07** y **HU08**.

El sistema podrá extender su la funcionalidad mediante nuevos controles

Debido a que el mundo de la ciberseguridad está en constante cambio, es necesario que el sistema sea capaz de extenderse a través de la creación de nuevos controles sin la necesidad de cambiar el código base.

Los nodos deberán estar definidos en función de una plantilla estándar que sepa interpretar la aplicación. Ésta última debe constar con un control de versiones para evitar problemas de compatibilidad.

Las historias de usuario relativas a este requisito son: **HU05** y **HU10**.

4.2. Requisitos no funcionales

El sistema debe ser intuitivo

Debe tener una interfaz amigable, para que un nuevo usuario no tarde más de un día en aprender a utilizar la aplicación.

Las historias de usuario relativas a este requisito son: **HU01**, **HU04**, **HU15**, **HU16** y **HU17**.

El programa debe poder distribuirse fácimente

Es necesario que el programa sea fácil de descargar y utilizar en cualquier sistema con el fin de poder ser adoptado en entornos educativos.

Las historias de usuario relativas a este requisito son: **HU11** y **HU12**.

El sistema debe contar con mecanismos de seguridad

Aunque se trate de una aplicación pensada en un uso limitado (donde el usuario la ejecute solamente cuando sea necesario), es posible valorar la idea de desplegarla sobre un servidor, dando así soporte a múltiples usuarios simultáneamente.

Esto, unido a que el sistema realiza interacciones con servicios externos, requiere que se implementen ciertas medidas de seguridad que garanticen la integridad del sistema anfitrión.

La historia de usuario relativa a este requisito es: **HU18**.

5 | Arquitectura

En este capítulo se detalla la arquitectura seguida, la cual define de manera abstracta las interfaces y los componentes que llevan a cabo alguna tarea de computación, así como la comunicación entre ellos.

> *"La arquitectura de software se selecciona y diseña con base en objetivos (requisitos) y restricciones. Los objetivos son aquellos prefijados para el sistema de información, [...] como el mantenimiento, la auditoría, flexibilidad e interacción con otros sistemas de información. Las restricciones son aquellas limitaciones [...] derivadas de las tecnologías disponibles para implementar sistemas de información."* – Wikipedia.

Existen varios tipos de arquitecturas seguidas durante el desarrollo del proyecto. A continuación se detalla cada una de ellas y se explican los beneficios que otorgan al producto final.

5.1. Arquitectura en *pipeline*

La arquitectura basada en filtros (o *pipeline*) debe su nombre al uso de tuberías para procesar información. Una **tubería** es una cola de mensajes, donde un mensaje puede ser cualquier cosa. Un filtro es un proceso, hilo u otro componente que lee perpetuamente los mensajes de una tubería de entrada, uno a la vez, procesa cada mensaje y luego escribe el resultado en una tubería de salida [11].

A pesar de no ser una de las arquitecturas más populares, su uso es frecuente en el mundo de la programación. Como ejemplos, se pueden destacar los patrones que siguen el diseño *map-filter-reduce* para tratamiento de listas, o el intérprete de comandos propio de cualquier sistema operativo, en el que se enlazan tareas por medio de tuberías (o *pipes*).

Es la arquitectura base de este proyecto, debido a que es uno de los principios de la programación visual y, en concreto, la programación basada en flujos de datos. Esta arquitectura es apreciable de forma gráfica mediante el editor de nodos.

5.2. Arquitectura en pizarra

La arquitectura en pizarra [14] es un marco de trabajo de sistemas que representa un enfoque general para la resolución de problemas basados en conocimiento.

Su concepto es similar a la arquitectura anterior (Apartado 5.1). Consta de múltiples elementos funcionales, donde cada uno está especializados en una tarea concreta. Estos elementos son conocidos como **agentes** (en este caso, los nodos). Además, existe un elemento de control denominado **pizarra** (el editor de nodos). Todos los agentes cooperan para alcanzar una meta común, si bien sus objetivos individuales no suelen estar aparentemente coordinados.

Cabe destacar que no es la arquitectura que mejor detalla al sistema, dado que el comportamiento básico de cualquier agente consiste en examinar la pizarra, realizar su tarea y escribir sus conclusiones en la misma pizarra. En el caso concreto de la aplicación, los agentes interactúan directamente entre sí, aunque pueden obtener el estado de otros agentes mediante la pizarra. Esto permite, a su vez, obtener una traza de las operaciones realizadas durante el proceso de resolución.

Aún así, sí que define correctamente varios aspectos del sistema: la pizarra facilita la conexión, coordinando a los distintos agentes, y los resultados generados por los agentes deben responder a un lenguaje y semántica común.

5.3. Arquitectura orientada a servicios

La arquitectura orientada a servicios [4] o *SOA* (del inglés *Service Oriented Architecture*) es un tipo de arquitectura que se basa en la creación de representaciones lógicas de ciertas actividades y que tiene un resultado de negocio específico (por ejemplo: comprobar el crédito de un cliente, obtener datos de clima, consolidar reportes de perforación, *etc.*). Los **servicios**, a su vez, se definen como funciones sin estado, auto-contenidas, que aceptan una o más llamadas y devuelve una o más respuestas mediante una interfaz bien definida.

En la aplicación, es la arquitectura que sigue la **API**, la cual estará formada por rutas y *middlewares* que acturán como servicios.

6 | Diseño del *software*

En este capítulo se detallan los diferentes componentes de la aplicación (diseño estático) y cómo interactúan entre ellos (diseño dinámico).

6.1. Diagrama de componentes

Se han agrupado los componentes en dos paquetes principales (*Fig. 6.1)*: ***User interface***, relativo a los componentes diseñados mediante lenguajes de *frontend*, y ***Server***, referente a aquellos procesos realizados en segundo plano y que tienen su base en lenguajes de *backend*.

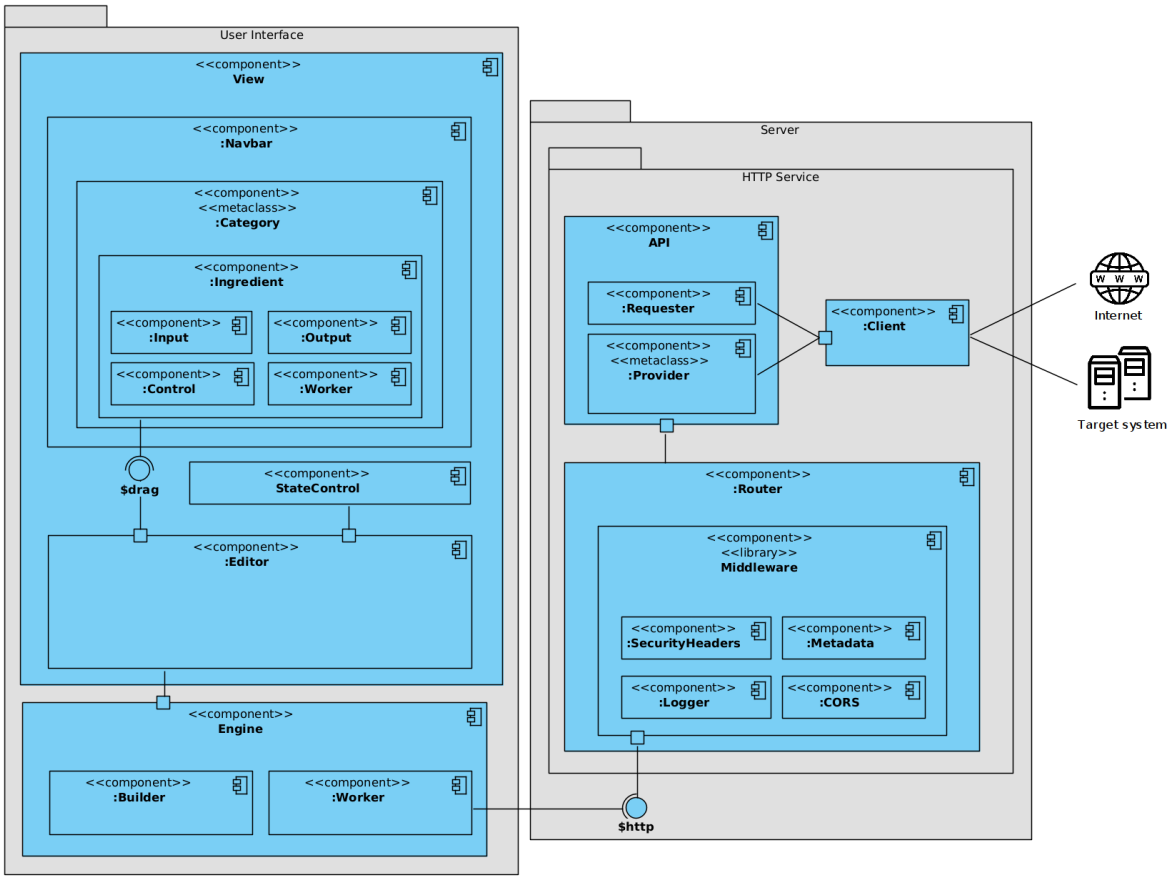


Figura 6.1: Diagrama de componentes.

Se ha dividido a su vez el paquete *Server*, creando una nueva agrupación llamada ***HTTP Server***, formado por los componentes que tienen relación con el procolo *HTTP*. Esto tiene su origen en que no se descarta la idea de implementar nuevos servicios en un futuro (como *FTP* o *SMTP*).

A continuación, se ofrece una pequeña descripción de la funcionalidad de cada componente.

Los componentes pertenecientes a la interfaz de usuario son:

– **View**: Encargado de renderizar la vista *HTML* que compone la aplicación. Está formado, a su vez, por otras vistas.

– **Navbar**: Semi-vista del contenido lateral. Ofrece la opción de seleccionar y filtrar los *Ingredients*.

– **Category**: Elemento contenedor de los diferentes *Ingredients* agrupados según su funcionalidad.

– **Ingredient**: Uno de los dos elementos más importantes de la interfaz. Es la abstracción de cada una las tareas propias de una auditoría de ciberseguridad.

Está formado por componentes de entrada (o ***Inputs***), de salida (***Outputs***), parámetros de configuración (***Controls***) y un componente puramente funcional encargado de realizar las operaciones de cómputo (***Worker***).

– **Editor**: El otro elemento más importante de la interfaz. Es el encargado de realizar las conexiones y renderizar los *Ingredients* que lo forman[1].

– **StateControl**: Elemento encargado de manejar el estado general del *Editor* mediante acciones de actualización, traslación, escalado, captura o carga.

– **Engine**: Se encarga de construir y registrar los *Ingredientes* usando la especificación de la tarea correspondiente (***Builder***). También es el encargado de realizar las comprobaciones necesarias y manejar de forma asíncrona las tareas (***Worker***).

Por otro lado, los componentes pertenecientes al servidor son:

– **Router**: Componente dedicado al enrutamiento de peticiones *HTTP*.

– **API**: Componente encargado de realizar peticiones y procesar los datos relativos a servicios y sistemas externos. Ofrece comunicación directa con los equipos auditados (***Requester***) y obtención de información a través de fuentes abiertas (***Provider***).

[1]Para añadir un nuevo componente del tipo *Ingredient* al *Editor*, solamente basta con arrastrarlo. Los datos se transfieren a través del evento ***drag***.

– **Client**: Es el encargado de manejar las conexiones que requiere la *API* y de definir las directrices por defecto de las peticiones.

– **Middleware**: Librería de componentes que tiene como objetivo modificar la respuesta *HTTP* antes de que ésta sea servida.

Proporciona cabeceras de seguridad (***SecurityHeaders***), de información (***Metadata***), controla el intercambio de recursos de origen cruzado (***CORS***) y registrar un seguimiento de las peticiones (***Logger***).

6.2. Diagrama de paquetes

En esta sección se detalla la distribución por paquetes de la aplicación (*Fig. 6.2)*. En el caso concreto de la interfaz de usuario, dicha distribución se ha omitido debido a que todos los elementos están contenidos por un mismo paquete, el cual hace referencia a la única vista existente, que a su vez está formado por el resto de vistas parciales.

Por otro lado, para lo referente al servidor, se ha seguido el modelo de paquetes que define *Golang*[13], donde cada paquete o módulo no es más que un directorio que contiene una serie de archivos de código fuente. Además, tienen asociados las características relativas a un elemento o funcionalidad en concreto. Estos paquetes pueden, a su vez, contener otros paquetes. La dependencia entre uno o más módulos nunca puede ser cíclica.

La distribución de paquetes es la siguiente:

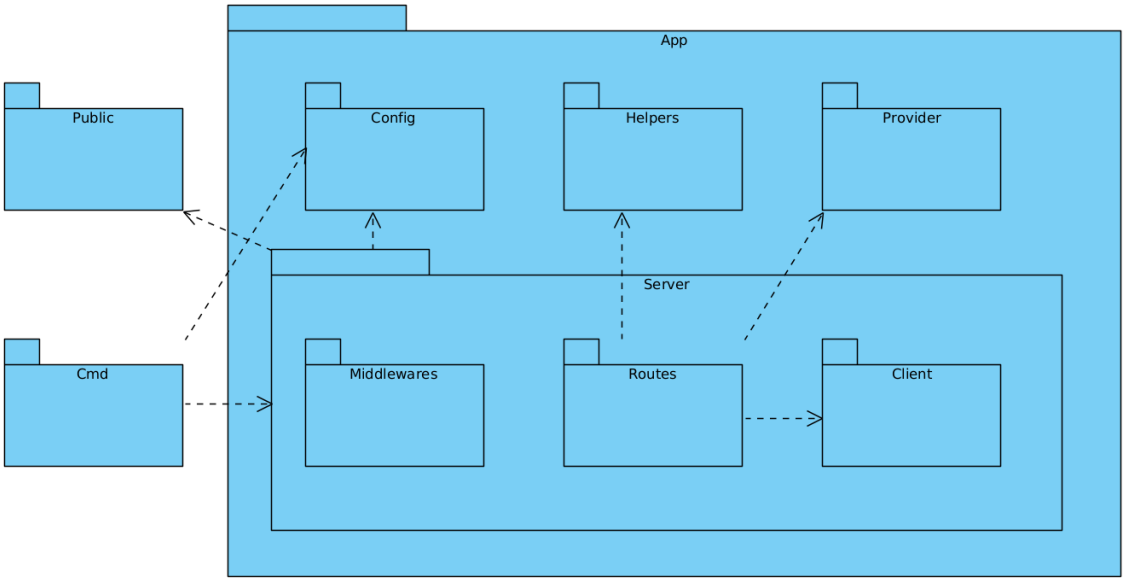


Figura 6.2: Diagrama de paquetes.

Concretamente, cada paquete se puede definir como:

– **Public**: Archivos estáticos propios de la interfaz gráfica.

– **Cmd**: Punto de entrada al ejecutar la aplicación.

– **App**: Núcleo del programa.

– **Config**: Parámetros de configuración necesarios para personalizar la ejecución.

– **Helpers**: Métodos auxiliares para el tratamiento de datos.

– **Provider**: Proveedores de información a través de fuentes abiertas.

– **Server**: Archivos relativos con el servidor *HTTP* local. Contiene el enrutador.

– **Middlewares**: Acciones auxiliares para el procesamiento de peticiones *HTTP*.

– **Routes**: Rutas estipuladas por el enrutador. Contiene los servicios de la *API*.

– **Client**: Gestor de conexiones. Define las directrices por defecto de las peticiones.

6.3. Diagrama de clases

Una propiedad del modelo de paquetes que define *Golang* [13] (introducido en el apartado 6.2) es la habilidad de cada paquete para ejercer como clase a pesar de no seguir un patrón relativo a la *Programación Orientada a Objetos*. No existe el concepto de herencia y la visibilidad de los atributos y métodos se especifica capitalizando su nombre: si el identificador comienza por una letra mayúscula, la visibilidad es pública; en el caso contrario, es privada.

El diagrama de clases correspondiente quedaría de la siguiente manera (*Fig. 6.3)*:

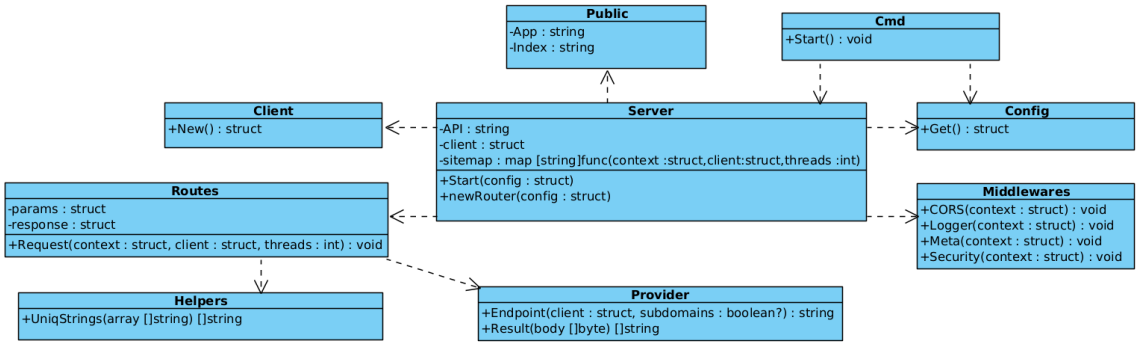


Figura 6.3: Diagrama de clases del servidor.

Los atributos y métodos que aparecen en el diagrama son:

Client		
Propiedad	**Nombre**	**Descripción**
Método	New	Devuelve un nuevo cliente con los parámetros por defecto

Tabla 6.1: Clase *Client*.

Cmd		
Propiedad	**Nombre**	**Descripción**
Método	Start	Llama al servidor *HTTP* usando la configuración dada

Tabla 6.2: Clase *Cmd*.

Config		
Propiedad	**Nombre**	**Descripción**
Método	Get	Obtiene los parámetros de configuración

Tabla 6.3: Clase *Config*.

Helpers		
Propiedad	**Nombre**	**Descripción**
Método	UniqString	Elimina los duplicados existentes en un *array* de texto

Tabla 6.4: Clase *Helpers*.

Middlewares		
Propiedad	**Nombre**	**Descripción**
Método	CORS	Permite el intercambio de recursos de origen cruzado
Método	Logger	Registra las peticiones *HTTP* y sus respuestas
Método	Meta	Añade cabeceras con *metainformación* a las respuestas
Método	Security	Añade cabeceras de seguridad a las respuestas

Tabla 6.5: Clase *Middlewares*.

Provider		
Propiedad	**Nombre**	**Descripción**
Método	Endpoint	Devuelve la dirección de la fuente abierta
Método	Result	Procesa la respuesta perteneciente a una fuente abierta

Tabla 6.6: Clase *Provider*.

Public		
Propiedad	**Nombre**	**Descripción**
Atributo	App	Código *JavaScript* relativo a la interfaz
Atributo	Index	Plantilla *HTML* relativa a la interfaz

Tabla 6.7: Clase *Public*.

Routes		
Propiedad	**Nombre**	**Descripción**
Atributo	params	Parámetros *HTTP* requeridos
Atributo	response	Parámetros *HTTP* devueltos
Método	Request	Realiza una petición *HTTP*

Tabla 6.8: Clase *Routes*.

Server		
Propiedad	**Nombre**	**Descripción**
Atributo	API	Ruta y versión de la *API*
Atributo	client	Cliente *HTTP* usado en las conexiones
Atributo	sitemap	Parejas de rutas y servicios de la *API*
Método	Start	Levanta el servidor *HTTP* usando la configuración dada
Método	newRouter	Devuelve un nuevo enrutador

Tabla 6.9: Clase *Server*.

Cabe destacar que tanto *Provider* como *Routes* se tratan de *super clases*, similares a una interfaz, que especifican los métodos y atributos propios de sus clases derivadas. Su diseño se debe a la futura creación y soporte de *plugins*.

Por otro lado, para el desarrollo de la interfaz también se ha usado un formato basado en clases que se puede representar mediante el siguiente diagrama (*Fig. 6.4)*:

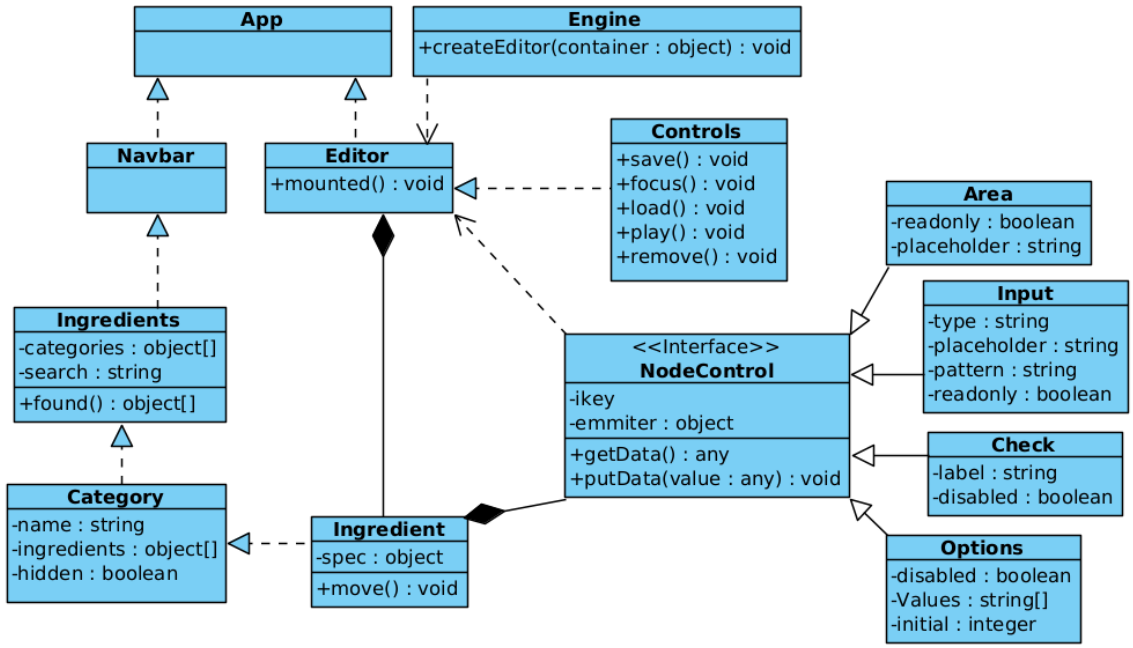


Figura 6.4: Diagrama de clases de la interfaz.

Los atributos y métodos que aparecen en el diagrama son:

Engine		
Propiedad	**Nombre**	**Descripción**
Método	createEditor	Crea, inicializa y agrega el editor a la estructura *HTML*

Tabla 6.10: Clase *Engine*.

Editor		
Propiedad	**Nombre**	**Descripción**
Método	mount	Evento realizado tras montar la vista en la aplicación

Tabla 6.11: Clase *Editor*.

Ingredients		
Propiedad	**Nombre**	**Descripción**
Atributo	categories	Categorias relativas a los ingredientes
Atributo	search	Término usado durante la búsqueda de ingredientes
Método	found	Ingredientes filtrados durante la búsqueda

Tabla 6.12: Clase *Ingredients*.

Category		
Propiedad	**Nombre**	**Descripción**
Atributo	name	Nombre de la categoría
Atributo	ingredients	Ingredientes pertenecientes a la categoría
Método	hidden	Define el estado relativo a su visualización

Tabla 6.13: Clase *Category*.

Ingredient		
Propiedad	**Nombre**	**Descripción**
Atributo	spec	Nombre de la categoría
Método	move	Genera la información requerida por el evento *drag*

Tabla 6.14: Clase *Ingredient*.

Controls		
Propiedad	**Nombre**	**Descripción**
Método	save	Guarda el estado actual del editor en un archivo *.recipe*
Método	focus	Hace zoom en los nodos del editor
Método	load	Carga el estado del editor a través de un archivo *.recipe*
Método	play	Inicia el flujo de datos
Método	remove	Limpia el editor de nodos

Tabla 6.15: Clase *Controls*.

NodeControl		
Propiedad	**Nombre**	**Descripción**
Atributo	ikey	Identificador del control
Atributo	emitter	Gestor de eventos
Método	getData	Obtiene el valor del estado interno
Método	putData	Cambia el valor del estado interno

Tabla 6.16: Clase *NodeControl*.

6.4. Diagramas de secuencia

Las interacciones entre las diferentes clases se entiende mejor al esquematizarlas mediante diagramas dinámicos, como el diagrama de secuencias. Por otra parte, la ejecución de un conjunto de tareas mediante el uso de la aplicación, como puede ser una auditoría de seguridad, ya es en sí mismo un diagrama de flujo, por la naturaleza propia de un editor gráfico basado en nodos.

La figura 6.5 muestra el flujo de datos al hacer *click* sobre el botón de ejecución. El editor manda el evento al motor, el cual es el encargado de llamar a todos los nodos por orden. Los nodos, a su vez, tienen la capacidad de obtener del editor el estado de los datos y, si es preciso, hacer llamadas al servidor para que interactuar con servicios externos. Después de calcular el resultado en función de los parámetros obtenidos, comunica su respuesta y emite una señal al motor para que se ejecuten los siguientes nodos. Esta tarea finaliza cuando todos los nodos han sido procesados.

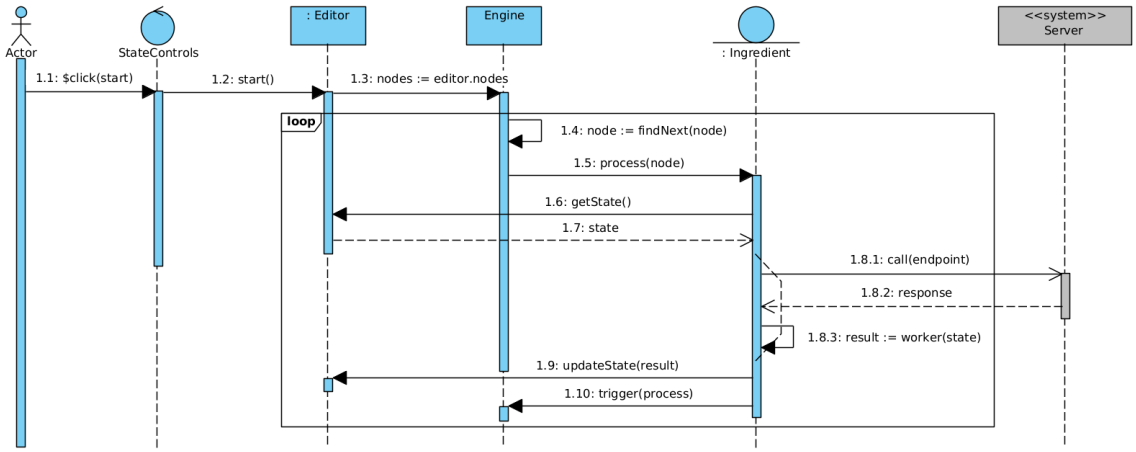


Figura 6.5: Diagrama de secuencia de la interfaz.

Por otra parte, cuando se hace una llamada al servidor (*Fig. 6.6)*, el enrutador comprueba que la petición esté correctamente formada. Después, pasa el contexto de la petición a los *middlewares* para que se encarguen de aplicar las cabeceras necesarias (de seguridad, de información y de intercambio de recursos). Una vez el contexto vuelve a estar en manos del enrutador, comprueba la petición y la manda al *Requester* (si se necesita interactuar directamente con el objetivo) o a uno de los *Providers* (si se requiere de información disponible en fuentes abiertas[2]). Antes de que la respuesta llege de nuevo al enrutador, el *middleware* de *logging* registra la petición junto al estado de la respuesta. Por último, se devuelve la respuesta al nodo que la pidió.

[2]Las fuentes abiertas muchas veces contienen datos duplicados, por lo que se ha de limpiar la respuesta

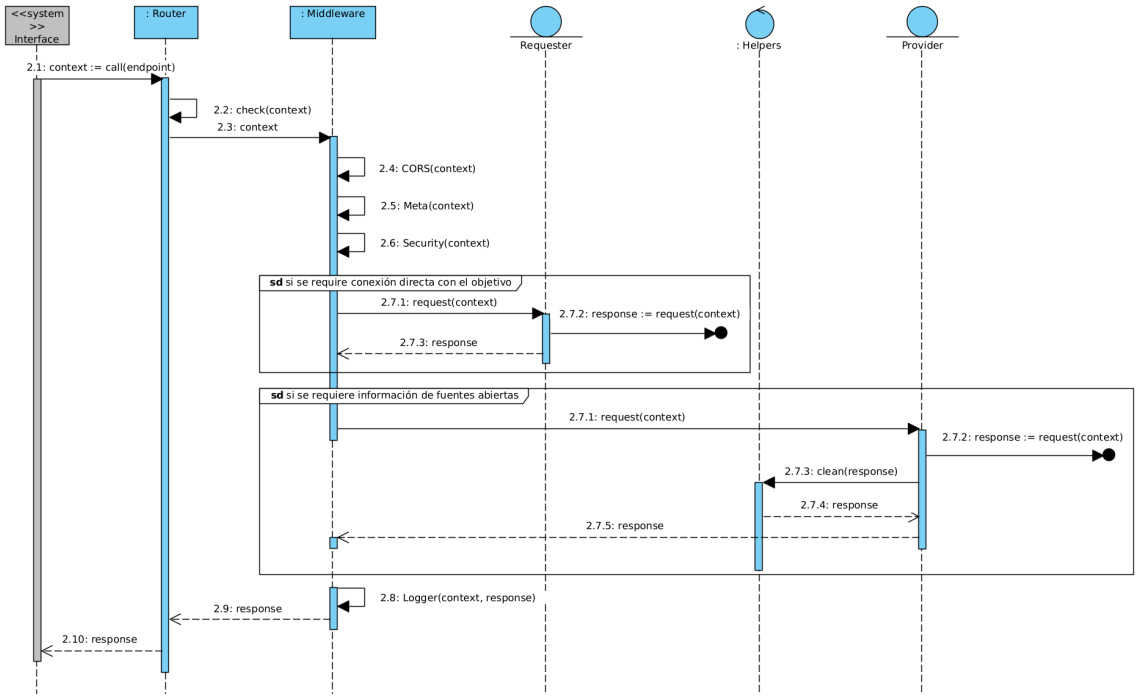


Figura 6.6: Diagrama de secuencia del servidor.

6.5. Diseño de la interfaz gráfica

A continuación, se muestra un boceto de la interfaz (*Fig. 6.7)*, que constará de una única página dinámica. Contiene un *banner* con el nombre de la aplicación, el editor (controlado por botones en la parte inferior de la interfaz) y una barra lateral con un buscador y los diferentes nodos agrupados por categorías.

Los nodos podrán añadirse al editor e interconectarse entre sí por medio del evento *drag* (arrastrando el cursor mientras se presiona).

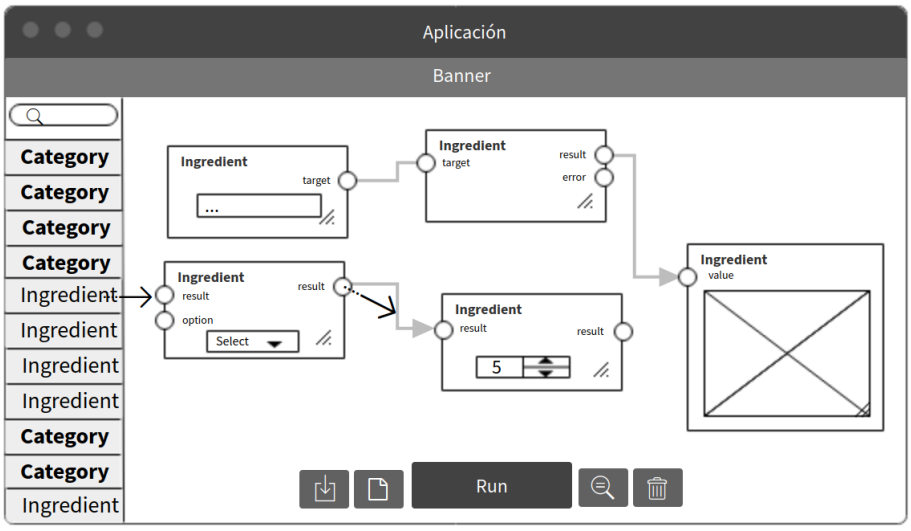


Figura 6.7: Diseño de la interfaz de usuario.

7 | Gestión de los datos e información

En este capítulo se detalla cómo se gestiona la información que maneja la aplicación. El tratamiento de estos datos se realiza de dos maneras diferentes, dependiendo de si son parte del flujo de datos perteneciente a los nodos del editor o a la *API*.

7.1. Información de los nodos

Los nodos tienen tres vías principales (las entradas, las salidas y los controles) (*Fig. 7.1)* y una secundaria (llamadas a la *API*) por las que manejan datos. Las **entradas** son la información que recibe de otros nodos; **los controles**, la información que condiciona el comportamiento del nodo y **las salidas**, la información que transmite en relación a sus entradas y/o controles. Un nodo puede estar formado por cualquier combinación de entradas, salidas y controles.

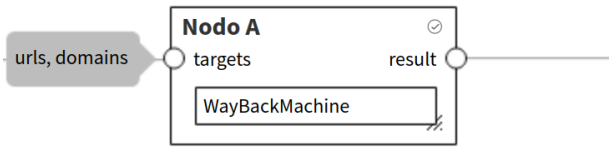


Figura 7.1: Representación de un nodo.

Las entradas y las salidas se visualizan a modo de conectores (de izquierda a derecha, respectivamente) Las salidas pueden conectarse a las entradas mediante cables y todos los datos que se transmiten están sujetos a un modelo *tipado*, similar a los tipos de variables en los lenguajes de programación tradicionales. Dos conectores con tipos de datos incompatibles no se pueden unir entre sí.

Dichos tipos de datos se generan automáticamente a la par que los nodos, los cuales son especificados mediante un archivo con formato *YAML*[1] *(Fig. 7.2)*. Los tipos de datos pueden comprobarse estacionando el cursor encima del conector en el que se esté interesado. Además, existe un tipo de dato, ***any***, que no tiene restricciones de conexión.

Respecto a la ejecución, los nodos reportan su estado mediante un icono en la parte superior derecha.

[1] *YAML Ain't Markup Language.* `https://yaml.org/`

Por otra parte, actualmente existen cuatro estilos de controles diferentes que se pueden implementar en un nodo. Los controles no son más que la abstracción de los elementos propios de formularios *HTML*: ***Area***, ***Check***, ***Input*** y ***Options.***

```yaml
# ------------------------------------------------------
- name: WSTG-INFO-02.1
  category: OWASP WSTG
  aliases: [owasp, fingerprint, banner, web server]
  description: Fingerprint Web Server
  inputs:
    - name: response(s)
      type: responses
  outputs:
    - name: url(s)
      type: urls
    - name: banner(s)
      type: text
  code: |
    const urls = [];
    const banners = [];
    const criticalHeaders = ["Server", "X-Powered-By"];
    inputs["response(s)"].forEach((response) => {
```

Figura 7.2: Especificación de un nodo.

Por último, tal como se puede observar en la figura 7.2, los nodos procesan los datos mediante código *JavaScript*. Este código consta de cuatro variables globales principales a la hora de manejar los datos que son: *inputs* (datos de entrada), *outputs* (datos de salida), *node.data* (parámetros de los controles) y *call* (llamadas a la *API* usando *fetch*). Para más información, consultar en el Anexo B.

Además, para evitar bucles infinitos de procesamiento o ejecuciones innecesarias, el motor que maneja el flujo de datos comprueba si los *inputs* o controles han cambiado desde la última ejecución, las entradas requeridas están conectadas y si los datos que se transmiten por ellas son válidos, evitando evaluar el código si alguna de estas condiciones no se cumple.

7.2. Información de la *API*

Respecto a los datos que maneja la *API* (documentado en profundidad en el Anexo B), viajan a través del protocolo *HTTP*. Al ser un servicio, se pueden realizar peticiones sin necesidad de usar el editor, lo cual permite integrar la *API* en flujos de trabajos externos.

Tal como se explica en el apartado 6.3, cuando llega una petición a la *API*, lo primero que hace es comprobar que tenga el formato correcto. Entre otras cosas, busca dentro de la petición la cabecera personalizada "*X-Powered-By: Masterchef!*" y exige que la petición sea realizada usando el verbo *POST*. Posteriormente, procesa los datos en el cuerpo de la petición y envía una respuesta, ambos en formato *JSON*.

8 | Pruebas realizadas

En este capítulo se detallan algunas de las pruebas llevadas a cabo durante la realización del proyecto, con el objetivo de asegurar el funcionamiento del programa, sus posibles aplicaciones en entornos reales y la correcta implementación de los requisitos descritos en el apartado 4.

8.1. Pruebas de disponibilidad e integridad

El proyecto debe contar con la capacidad de garantizar que tanto el sistema como los datos van a estar disponibles al usuario en todo momento.

Con el fin de evidenciar este aspecto, se han realizado **pruebas de estrés** y **de rendimiento**.

Pruebas de estrés y rendimiento

Consisten en medir la capacidad del sistema en función a la carga de trabajo y sus tiempos de respuesta. Este tipo de pruebas **dependen de las especificaciones del sistema operativo anfitrión**, sus procesos y la calidad de la conexión a Internet[1].

Respecto al rendimiento, se ha realizado un *benchmarking* mediante el uso de la herramienta ***go-wrk***[2]. A pesar de que *Golang* cuenta con herramientas nativas de *profiling*, no ofrece un buen soporte para realizar este tipo de pruebas sobre servicios *HTTP*.

Se han realizado tres pruebas de estrés en las que se pide a la ***API*** que interaccione con una, dos y cuatrocientas páginas web reales, respectivamente. La capacidad y los tiempos de respuesta medidos durante 10 segundos son los mostrados en las figuras 8.1 y 8.2. La última prueba no ha pasado el control debido a que una única petición a cuatrocientas *URLs* simultáneas tarda aproximadamente medio minuto (*Figs. 8.3 y 8.4*).

[1] Pruebas realizadas sobre Ubuntu 20.04 con 8Gb de RAM, 2Gb de *swap*, procesador Intel Core-i7 8th Gen y conexión por wifi (72Mbps descarga/89Mbps subida).

[2] `https://github.com/tsliwowicz/go-wrk`

```
Masterchef on  main [!?] on  v20.10.7
➜ res; go-wrk -c 80 -H "X-Powered-By: Masterchef\!" -H "Content-Type: applic
ation/json" -M POST -body '{"urls":["http://example.com/"], "method": "GET"}
' http://localhost:7767/api/v1/request

--------------- RESPUESTA ---------------
Running 10s test @ http://localhost:7767/api/v1/request
  80 goroutine(s) running concurrently
3886 requests in 10.107165676s, 10.76MB read
Requests/sec:           384.48
Transfer/sec:           1.06MB
Avg Req Time:           208.073405ms
Fastest Request:        193.958265ms
Slowest Request:        320.690863ms
Number of Errors:       0
```

Figura 8.1: *Benchmark*: 3886 peticiones a 1 *URL*. 208ms/respuesta. 0 errores.

```
Masterchef on  main [!?] on  v20.10.7
➜ res; go-wrk -c 80 -H "X-Powered-By: Masterchef\!" -H "Content-Type: applicat
ion/json" -M POST -body '{"urls":["http://example.com/","https://example.com"]
, "method": "GET"}' http://localhost:7767/api/v1/request

--------------- RESPUESTA ---------------
Running 10s test @ http://localhost:7767/api/v1/request
  80 goroutine(s) running concurrently
1918 requests in 10.274370414s, 9.30MB read
Requests/sec:           186.68
Transfer/sec:           926.42KB
Avg Req Time:           428.545168ms
Fastest Request:        396.529887ms
Slowest Request:        691.016893ms
Number of Errors:       0
```

Figura 8.2: *Benchmark*: 1918 peticiones a 2 *URLs* simultáneas. 428ms/respuesta. 0 errores.

```
Masterchef/backend on  main [!?] via  v1.16.5 took 14s

➜ go run main.go
 ** Server started! **
http://localhost:7767/
signal: killed


Masterchef/backend on  main [!?] via  v1.16.5 took 38s

➜
```

```
Masterchef on  main [!?] on  v20.10.7
➜ res; go-wrk -c 80 -H "X-Powered-By: Masterchef\!" -H "Co
ntent-Type: application/json" -M POST -body @test/performa
nce/400webs.json http://localhost:7767/api/v1/request

--------------- RESPUESTA ---------------
Running 10s test @ http://localhost:7767/api/v1/request
  80 goroutine(s) running concurrently
redirect?
An error occured doing request Post "http://localhost:7767
/api/v1/request": net/http: timeout awaiting response head
ers
```

Figura 8.3: *Benchmark*: Peticiones a 400 *URLs* simultáneas. Prueba de estrés fallida.

```
Masterchef/backend on  main [!?] via  v1.16.5 took 4m
 23s
➜ go run main.go
 ** Server started! **
http://localhost:7767/
2021-07-01T19:42:10 | 127.0.0.1:52040/tcp | [200] POST:
/api/v1/request
```

```
Masterchef on  main [!?] on  v20.10.7
➜ curl -SsLo /dev/null http://localhost:7767/api/v1/reque
st -H "content-type:application/json" -H "x-powered-by: M
asterchef\!" -d @test/performance/400webs.json

Masterchef on  main [!?] on  v20.10.7 took 32s
➜
```

Figura 8.4: *Test*: Petición a 400 *URLs* simultáneas. 32s/respuesta. 0 errores.

8.2. Pruebas sobre la funcionalidad

Para comprobar la correcta funcionalidad del sistema, se han ejecutado varios tipos de flujos de datos. Estas pruebas garantizan que se cumplen los requisitos funcionales descritos en el apartado 4.1.

La figura 8.5 muestra la capacidad de poder crear conjuntos de tareas independientes entre sí, así como la posibilidad de definir múltiples objetivos.

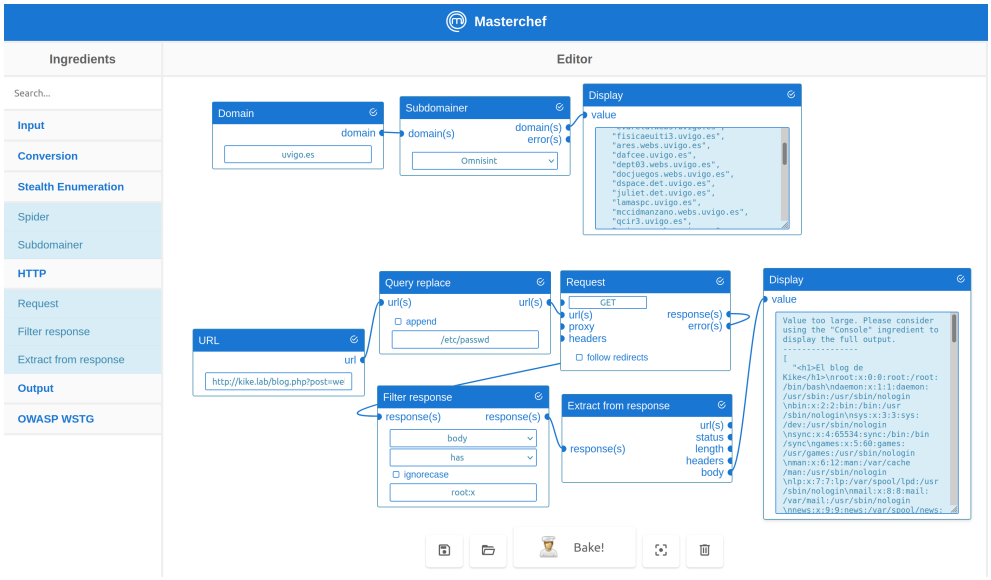


Figura 8.5: Flujos independientes: Reconocimiento (arriba) y vulnerabilidad (abajo).

Además, en la figura 8.6 se muestra cómo es posible realizar controles propios de una auditoría (ver Anexo C) a partir de un activo.

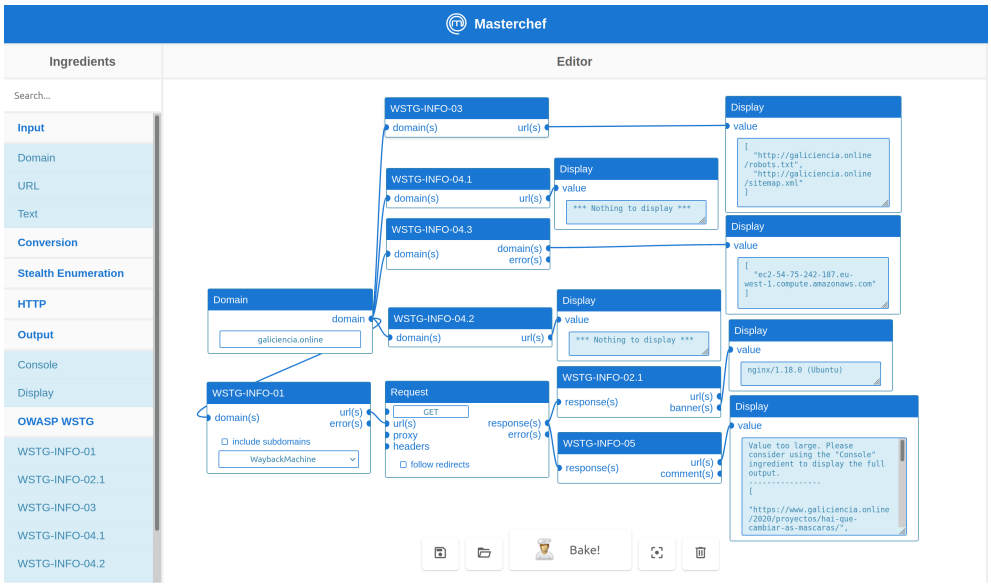


Figura 8.6: Ejemplo de controles pertenecientes a *OWASP WSTG-INFO*.

8.3. Pruebas de seguridad

Por último, se han realizado pruebas de seguridad para garantizar la ausencia de vectores de ataques que puedan poner en peligro el sistema.

Relativo al editor de nodos, las pruebas constan de la correcta gestión de errores, como bucles infinitos, que podrían provocar una sobrecarga de la *CPU* (*Fig. 8.7*).

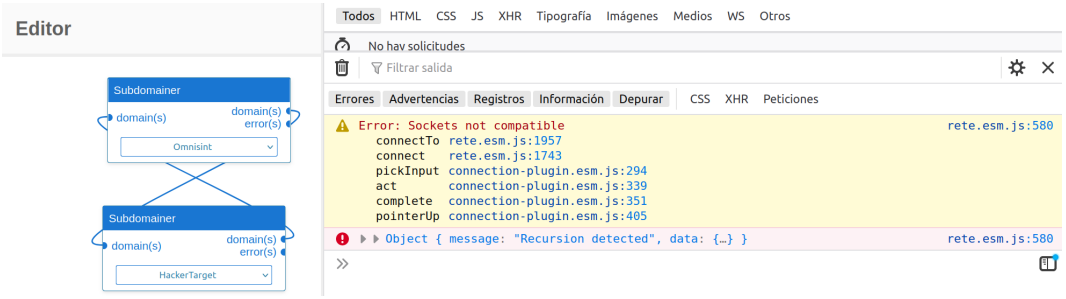


Figura 8.7: Manejo de errores: Conexiones incompatibles y bucles infinitos.

Por otro lado, se han implementado medidas de seguridad para evitar los ataques web más comunes, como *XSS* o *Clickjacking*, mediante el uso de etiquetas *span*[3] y cabeceras de seguridad (*Fig. 8.8*).

Figura 8.8: Cabeceras de seguridad y pruebas contra *XSS*.

Por último, el servidor se interrumpe cuando detecta peticiones más grandes de lo normal contra la *API* (*Fig. 8.3*). Esto no se trata de una medida de seguridad, sino de una gestión de errores temporal que tendrá que ser reemplazada por medidas reales, como la segmentación de las peticiones o la implementación de listas negras.

[3]Las etiquetas *span* no permiten la renderización de código *HTML*

9 | Epílogo

En este último capítulo se resumen las principales aportaciones de este proyecto según los objetivos descritos en el apartado 1.3, así como las conclusiones (tanto técnicas como personales) y las vías de trabajo futuro.

9.1. Principales aportaciones

Este trabajo aporta, principalmente, una nueva herramienta de código abierto. Dicha herramienta es capaz de automatizar las auditorías de seguridad, ofreciendo una gestión visual.

Con esta acción se pretende, entre otras cosas, desmitificar el mundo del *hacking*. Tanto los docentes, como aquellas personas que sientan interés por el tema, pueden obtener una manera sencilla de iniciarse en la materia. La aplicación es fácil de descargar y ejecutar, contando a su vez con una interfaz amigable e intuitiva.

Por otro lado, se ha conseguido aportar una nueva aplicación con la que reemplazar muchas de las herramientas existentes en el mercado. Esto consigue reducir la cantidad de soluciones *software* que necesita un *pentester* para llevar a cabo auditorías de ciberseguridad.

Por último, y aunque actualmente no sea una aportación real, se espera que se genere una nueva comunidad en torno al proyecto, la cual contribuya creando nuevos *ingredientes* y *recetas* que permitan realizar tareas cada vez más complejas.

9.2. Conclusiones

En este punto, el autor del documento expondrá sus pensamientos en lo referente a la realización del proyecto, tanto desde una perspectiva técnica como desde un punto de vista personal.

9.2.1. Conclusiones técnicas

A pesar de no ser un práctica habitual por parte de los estudiantes, es increíble lo que el uso de metodologías (como pueden ser las descritas en el apartado 2.1) son capaces de aportar a un proyecto. Como conclusiones técnicas, se destaca varios puntos:

La aplicación de los conocimientos obtenidos durante la carrera, como pueden ser el análisis y el diseño de la aplicación, han conseguido minimizar la cantidad de errores que se producen normalmente durante la etapa de desarrollo.

También es importante mencionar que gracias a las bases adquiridas, se ha conseguido hacer uso de herramientas y tecnologías que se ajustan a los requisitos y motivaciones del proyecto. Un ejemplo de esto son lenguajes de programación concurrentes con los que no se trabaja en la Universidad o la implementación de arquitecturas nunca antes vistas por el alumno.

Algunos de los conocimientos adquiridos al realizar este trabajo, y que servirán de base para el futuro, son:

- Aprender a implementar paradigmas de programación basadas en flujos de datos.

- El descubrimiento de nuevos *frameworks* que permitan un mejor diseño.

- LaTeX como estándar para la maquetación de documentos académico-científicos.

9.2.2. Conclusiones personales

En un primer momento, la idea de desarrollar un programa similar a nuevo lenguaje de programación visual era bastante imponente. No sólo se aleja del tipo de proyectos que concebimos habitualmente, si no que la documentación relativa a la materia es bastante escasa y/o incompleta. Esto, unido al capricho de desarrollar la interfaz utilizando un *framework* con el que nunca antes se había trabajado, suponía un reto ideal que encajaba perfectamente con la ideología de un Trabajo de Fin de Grado.

Ahora, tras haber finalizado el proyecto y habiendo conseguido desarrollar una base sólida sobre la que hacer crecer la idea, el autor de este documento es consciente de que no existen ideas imposibles. El error suele estar solamente en un mal enfoque o una falta de motivación real para llevarlas a cabo.

Por otro lado, es muy importante no olvidarse de que los grandes logros nunca se consiguen de manera individual. *La informática no sólo se basa en relacionarse con ordenadores.* Al igual que en muchos otros aspectos de la vida, también es necesario que existan personas que te apoyen, aconsejen, critiquen y motiven para seguir adelante, pudiendo ofrecer cada vez mejores resultados[1].

[1]Gracias a mi tutor y a mi círculo de personas cercanas por hacérmelo tener siempre presente.

9.3. Vías de trabajo futuro

Durante el desarrollo del trabajo, se ha conseguido desarrollar las bases de la aplicación tal y como estaba previsto. Aún así, el *software* presentado se trata de una versión inicial, sobre el cual se irán añadiendo nuevas mejoras. En este apartado se numeran algunas de ellas, a corto plazo.

9.3.1. Creación de nuevas *recetas* e *ingredientes*

Actualmente la aplicación consta de los nodos básicos para demostrar la utilidad y eficacia de la misma. Es necesario que la aplicación siga creciendo y se mantenga actualizada. Para ello, una de las tareas a realizar es la creación de nuevos nodos, que extiendan su funcionalidad, así como crear un repositorio de plantillas en el cual poder compartir pruebas y metodologías usando la aplicación.

Esta tarea irá desarrollándose de manera proporcional al interés que sienta la comunidad, tal y como se comenta en el último párrafo del apartado 9.1.

9.3.2. *Refactorización* del código

Debido a que el proyecto fue desarrollado teniendo en cuenta una fecha límite, su implementación no es la más correcta. Es necesario revisar el código fuente para mejorar los mecanismos de seguridad y el rendimiento de sus componentes, entre otras cosas.

9.3.3. Creación de pruebas automatizadas

En el apartado 2.6.2 se detalla el motivo por el cual el código no consta de prueba unitarias que aseguren su correcto funcionamiento. La creación de pruebas automatizadas es una de las partes fundamentales en el desarrollo de *software* sostenible. Es por ello, que se pretende implementarlas en el futuro más próximo.

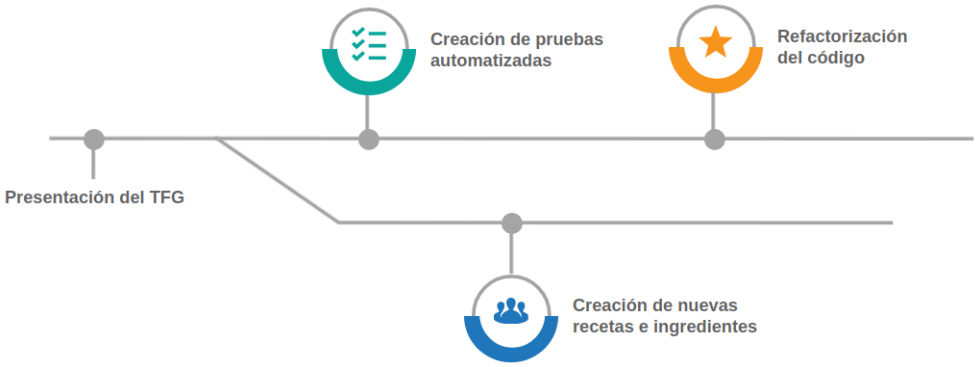


Figura 9.1: Metas a corto plazo.

A | Bibliografía

[1] *¿Qué es ATT&CK de MITRE y cuál es su utilidad?* URL: https://www.anomali.com/es/resources/what-mitre-attck-is-and-how-it-is-useful.

[2] C. *Cybersecurity Talent Crunch To Create 3.5 Million Unfilled Jobs Globally By 2021*. Inglés americano. 5 de ago. de 2020. URL: https://cybersecurityventures.com/jobs/.

[3] C. Cimpanu. *RDP and VPN use skyrocketed since coronavirus onset*. Inglés. 30 de mar. de 2020. URL: https://www.zdnet.com/article/rdp-and-vpn-use-skyrocketed-since-coronavirus-onset/.

[4] Colaboradores de Wikipedia". *Arquitectura orientada a servicios*. 22 de jun. de 2021. URL: https://es.wikipedia.org/wiki/Arquitectura_orientada_a_servicios.

[5] Colaboradores de Wikipedia". *Kanban*. 6 de jun. de 2021. URL: https://es.wikipedia.org/wiki/Kanban.

[6] L. Columbus. *Cybersecurity Spending To Reach $123B In 2020*. Inglés. 10 de ago. de 2020. URL: https://www.forbes.com/sites/louiscolumbus/2020/08/09/cybersecurity-spending-to-reach-123b-in-2020/.

[7] *Coronavirus is now possibly the largest-ever security threat - here's how we may be able to tackle it | Perspectives | Reed Smith LLP*. Inglés. URL: https://www.reedsmith.com/en/perspectives/2020/03/coronavirus-is-now-possibly-the-largest-ever-security-threat.

[8] Jefatura de Estado". *Ley Orgánica 10/1995, de 23 de noviembre, del Código Penal*. 24 de nov. de 1995. URL: https://www.boe.es/buscar/pdf/1995/BOE-A-1995-25444-consolidado.pdf.

[9] Jefatura de Estado". *Ley Orgánica 5/2010, de 22 de junio, por la que se modifica la Ley Orgánica 10/1995, de 23 de noviembre, del Código Penal*. 23 de jun. de 2010. URL: https://www.boe.es/buscar/pdf/2010/BOE-A-2010-9953-consolidado.pdf.

[10] OWASP". *OWASP Web Security Testing Guide*. Inglés. URL: https://owasp.org/www-project-web-security-testing-guide/.

[11] San José State University". *Pipeline Architecture*. Inglés. URL: http://www.cs.sjsu.edu/~pearce/modules/patterns/distArch/pipeline.htm.

[12] B.J. Tidy. *Google blocking 18m coronavirus scam emails every day*. british. 17 de abr. de 2020. URL: https://www.bbc.com/news/technology-52319093.

[13] Unknown. *Go Modules Reference - The Go Programming Language*. Inglés. URL: https://golang.org/ref/mod.

[14] Wikipedia contributors". *Blackboard system*. Inglés. 22 de mayo de 2021. URL: https://en.wikipedia.org/wiki/Blackboard_system.

B | Manual de usuario

En este anexo se describen los aspectos básicos de la aplicación, como la estructura de los directorios, la guía de instalación, la creación de *ingredientes* y el uso básico de la misma.

El código de la aplicación está disponible *online* en la siguiente dirección web:

`https://github.com/cosasdepuma/Masterchef`

B.1. Estructura de los directorios

Los archivos y directorios más relevantes del proyecto se encuentran estructurados siguiendo el esquema mostrado a continuación.

B.1.1. Directorio raíz

El **directorio raíz** cuenta con tres carpetas:

.github/: Archivos pertenecientes a la documentación de *GitHub*.
backend/: Código fuente del servidor.
frontend/: Código fuente de la interfaz.

También cuenta con los siguientes ficheros:

.gitignore: Archivos ignorados por *GitHub*.
.dockerignore: Archivos ignorados por *Docker*.
Makefile: Parámetros de auto-compilación.
Dockerfile: Archivo para compilar y desplegar la aplicación usando contenedores.
docker-compose.yml: Fichero para orquestar contenedores.
version: Versión actual del programa.
swagger.yml: Documentación de la *API*.
README.md: Documentación del repositorio.
LICENSE: Licencia del repositorio.

B.1.2. Directorio *frontend*

.
La carpeta *frontend* cuenta con los siguientes archivos:

.browserslistrc: Navegadores soportados por la interfaz.
babel.config.js: Archivo de configuración de *Babel*.
compile.js: *Script* para añadir el *frontend* en ficheros *.go*.
package.json: Archivo de configuración del *frontend*.
patches.js: *Script* para corregir errores en las dependencias.
vue.config.js: Archivo de configuración de *Vue*.
yarn.lock: Versiones e integridad de las dependencias de desarrollo.

Respecto a las carpetas, existen dos dentro del directorio *frontend*: **public**/ (con los archivos estáticos, como la plantilla *HTML*) y **src**/, con el código fuente relativo al núcleo de la interfaz. Esta última está estructurada de la siguiente manera:

App.vue: Vista principal de la aplicación.
main.js: Entrada de la aplicación, encargada de montar la vista.
assets/: Archivos auxiliares. Contiene las especificaciones de los nodos.
components/: Componentes visuales de la interfaz.
ingredients/: Archivos relativos a la interpretación de los nodos.
scripts/: Otros ficheros de *Javascript*. Contiene la lógica del **editor**/, formado a su vez por **plugins**/ y **components**/ (tipos de controles). También provee de funciones auxiliares (**helpers**/), iconos (**icons**/) y validaciones (**validator**/).

B.1.3. Directorio *backend*

.
La carpeta *backend* contiene los siguientes directorios y archivos:

go.mod: Dependencias de desarrollo.
go.sum: Versiones e integridad de las dependencias de desarrollo.
main.go: Punto de entrada de la aplicación si se ejecuta mediante un binario.
cmd/: Punto de entrada de la aplicación si se instala con *go get*.
public/: Archivo auto-generados por el *frontend*.
app/: Código fuente relativo al núcleo del servidor.

La carpeta **app**/ está estructurada de la siguiente forma:

config/: Parámetros de configuración por defecto y variables de entorno.
helpers/: Funciones auxiliares.
providers/: Código para la obtención de información mediante fuentes abiertas.
server/: Archivos relativos al servidor *HTTP*.

La carpeta **server**/ consta, a su vez, de los archivos **server.go** (servicio *HTTP*) y **router.go** (enrutador). También cuenta con los directorios **client**/ (cliente *HTTP*), **middlewares**/ (funciones auxiliares del enrutador) y **routes**/ (rutas de la *API*).

B.2. Compilación

En entornos **nix*, la compilación se puede llevar a cabo por medio del comando **make**, gracias a las reglas descritas en el archivo *Makefile*.

Por el contrario, si se desea realizar una compilación manual o no se dispone de las herramientas necesarias para ejecutarla de manera automatizada, se puede realizar el siguiente procedimiento:

—— Interfaz
yarn –cwd frontend/ install
yarn –cwd frontend/ run compile
—— Servidor
cd backend/
go build main.go
—— Opcional
upx -o masterchef main

B.3. Instalación

La aplicación está disponible de modo portable. Se puede descargar directamente desde el apartado *Releases* del repositorio *online*. Su tamaño aproximado es de 1,9M para la versión 0.1.0.

Si se desea instalar, es posible realizarlo mediante el comando[1]:

go get -u github.com/cosasdepuma/Masterchef

B.4. Requisitos mínimos

La aplicación no cuenta con requisitos mínimos. Es posible compilarla para cualquier sistema operativo y arquitectura. Tampoco depende de ficheros o programas de terceros.

En el caso de ejecutar tareas de recopilación de información o que tengan como objetivo un activo externo, puede ser necesaria una conexión a Internet.

Nota: Actualmente el editor de nodos no funciona correctamente en dispositivos móviles o *tablets*.

[1]Es necesario tener *Golang 1.16* instalado en el sistema

B.5. Uso de la aplicación

A continuación, se detalla el uso de la aplicación. Se divide en tres partes: ejecución, llamadas a la *API* y creación de nuevos nodos.

B.5.1. Ejecución

Para iniciar el servidor, solamente basta con lanzar el programa (*Fig. B.1a)*. Si se ejecuta desde una terminal, se pueden configurar los parámetros mediante las variables de entorno **MC_ADDR** y **MC_THREADS** (*Fig. B.1b)*:

(a) Ejecución estándar. (b) Ejecución personalizada.

Figura B.1: Ejecución de la aplicación.

También es posible levantar el servicio usando *Docker* gracias al archivo *Dockerfile*.

Una vez ejecutado, será necesario acceder a la dirección definida con el fin de visualizar la interfaz gráfica (*Fig. B.2)*. La *URL* por defecto es `http://127.0.0.1:7767/`.

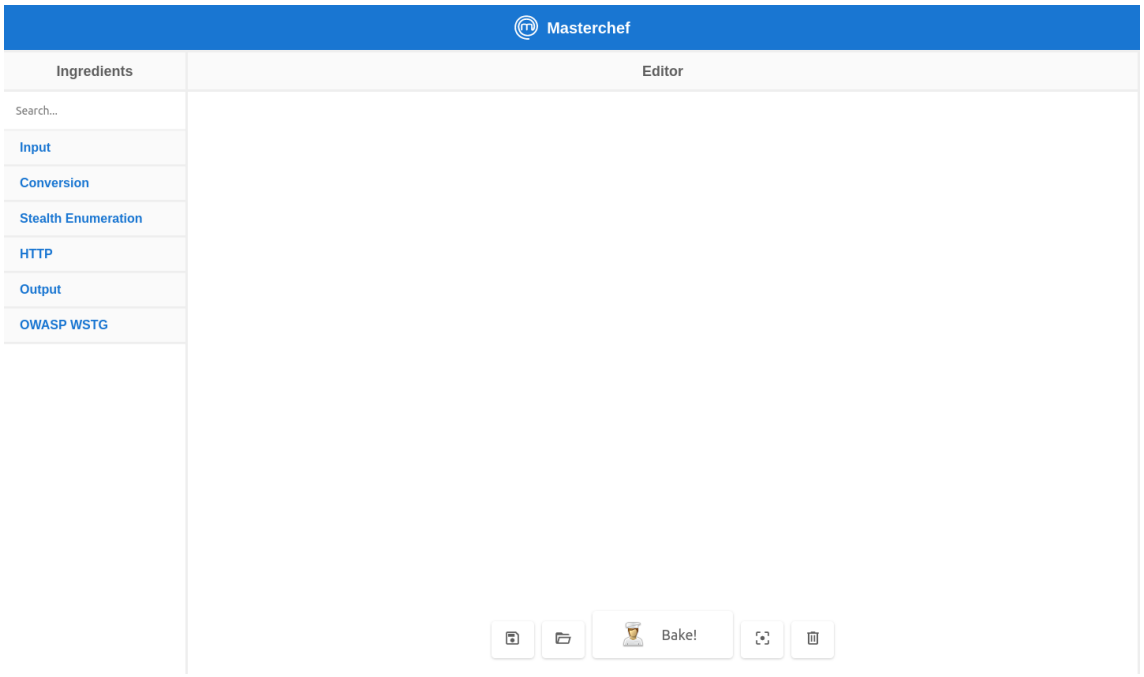


Figura B.2: Interfaz de la aplicación.

El panel lateral izquierdo cuenta con los nodos disponibles (denominados *ingredientes*). Están agrupados por categorías. Se puede desplegar u ocultar una categoría pulsando

sobre ella. También contiene un buscador que permite filtrar los nodos por palabras clave. Es posible obtener la descripción de un ingrediente depositando el cursor encima del mismo (*Fig. B.3)*.

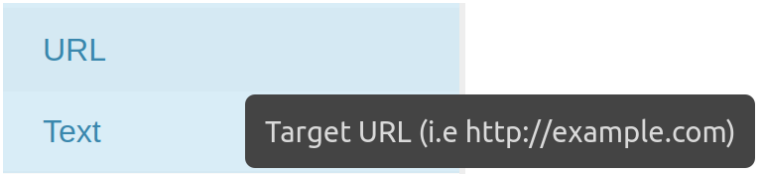


Figura B.3: Descripción del ingrediente.

Los ingredientes pueden ser añadidos al editor arrastrándolos sobre él. Éstos constan de tres partes fundamentales (*Fig. B.4)*: los datos de entrada (a la izquierda), los datos de salida (a la derecha) y los controles (en el centro). Cada dato lleva asociado un tipo específico que limita su conexión con otros nodos. Para comprobar el tipo de conexión, basta con mantener el cursor encima del conector.

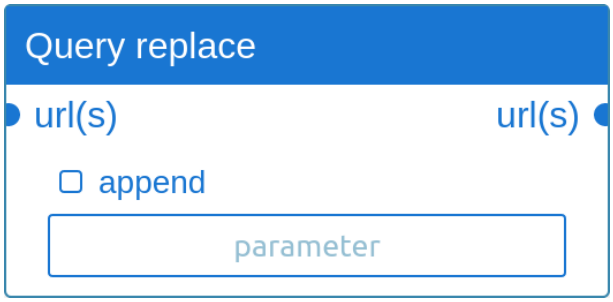


Figura B.4: Estructura de un ingrediente.

La unión de dos o más nodos puede efectuarse arrastrando un conector de salida a uno de entrada del mismo tipo, o viceversa. Para eliminar una conexión, es posible hacerlo añadiendo una nueva o arrastrando a un espacio vacío el conector de entrada. Además, es posible eliminar un nodo pulsando la tecla *Suprimir*.

El editor puede controlarse mediante los botones situados en la parte inferior de la interfaz. Sus funcionalidades son, de izquierda a derecha: guardar estado actual del editor en una receta, cargar una receta desde un archivo, ejecutar el programa, centrar la vista en los nodos existentes y limpiar el editor de nodos.

Cuando se ejecutan los ingredientes, es posible conocer su estado mediante un icono situado en la parte superior derecha de los mismos. Un *tick* significa que el nodo se ha ejecutado correctamente; una cruz, que ha habido algún fallo (revisar la consola) y la ausencia de iconos indica que aún no ha sido procesado.

Un ejemplo de receta para buscar o comprobar vulnerabilidades puede ser la mostrada en la figura B.5.

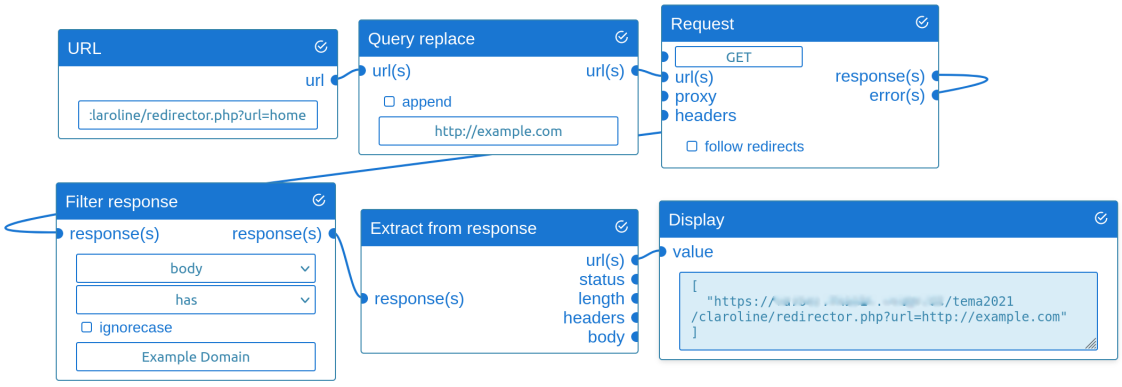


Figura B.5: Receta: Vulnerabilidad *OpenRedirect*.

B.5.2. Llamadas a la *API*

Al tratarse de un servicio web, es posible realizar llamadas a la *API* sin necesidad de hacer uso de la interfaz. Esta opción está enfocada en los usuario más avanzados que requieran automatizar su flujo de trabajo mediante *scripts*.

Para ello, es necesario realizar una petición *HTTP* usando el método *POST*. Además, la petición debe incluir la cabecera "*X-Powered-By: Masterchef!*" y enviar la consulta en formato *JSON*.

Las peticiones actualmente soportadas son[2]:

/api/v1/request

Realiza peticiones *HTTP*

Parámetro	**Tipo**	**Descripción**
urls	array[url]	Listado de *URLs* objetivo
method	string	Verbo *HTTP*
proxy	url	Proxy por el cual enviar las consultas
headers	map[string]string	Cabeceras *HTTP*.

Tabla B.1: *Endpoint* /request

/api/v1/lookup/dns

Realiza consultas inversas de *DNS*

Parámetro	**Tipo**	**Descripción**
ips	array[ip]	Listado de *IPs*

Tabla B.2: *Endpoint* /lookup/dns

[2]Los *endpoints* se encuentran detallados en profundidad en el archivo *swagger.yml*

/api/v1/lookup/ips Realiza consultas de *IP*		
Parámetro	**Tipo**	**Descripción**
domains	array[domain]	Listado de dominios

Tabla B.3: *Endpoint* /lookup/ips

/api/v1/stealth/spider Obtiene un listado de activos usando fuentes abiertas		
Parámetro	**Tipo**	**Descripción**
domains	array[domain]	Listado de dominios
provider	string	Proveedor de búsqueda
includeSubdomains	boolean	Incluir subdominios

Tabla B.4: *Endpoint* /stealth/spider

/api/v1/stealth/subdomainer Obtiene un listado de subdominios usando fuentes abiertas		
Parámetro	**Tipo**	**Descripción**
domains	array[domain]	Listado de dominios
provider	string	Proveedor de búsqueda

Tabla B.5: *Endpoint* /stealth/subdomainer

Los servicios disponibles a través de la *API* devuelven el código de estado **404** si no se ha realizado una petición correcta (error en la ruta, método o cabeceras); **406**, si no se incluyen los parámetros necesarios; o **200**, si la consulta se ha realizado correctamente.

La figura B.6 muestra un ejemplo de petición a través de comandos.

```
Masterchef on  main [!] on  v20.10.7
➜ curl -s -H "X-Powered-By: Masterchef\!" -H "Content-Type: application/json" localhost:7767/api/v1/stealth/subdomain
er -d '{"domains": ["uvigo.gal"], "provider": "hackertarget"}' | jq
{
  "domains": [
    "uvigo.gal"
  ],
  "subdomains": [
    "hosting.uvigo.gal",
    "celmi.uvigo.gal",
    "lab6.munics.det.uvigo.gal",
    "dpv.uvigo.gal",
    "autodiscover.alumnado.uvigo.gal",
    "talga.webs.uvigo.gal",
```

Figura B.6: Llamada al *subdomainer* a través de *curl*.

B.5.3. Especificación de nuevos *ingredientes*

Los tipos de ingredientes están especificados en el archivo */frontend/src/assets/ingredients.yml*. Actualmente no es posible modificar los ingredientes en soluciones precompiladas. Las especificaciones siguen un formato *YAML*.

Para crear un nuevo tipo de ingrediente, es necesario definir los siguientes valores. Los campos opcionales se marcan con una interrogación (?):

- **id**? ... Identificador único (sin uso, por el momento).
- **name** ... Nombre del ingrediente.
- **category** ... Categoria a la que pertenece.
- **aliases** ... Etiquetas de búsqueda.
- **description** ... Descripción de la funcionalidad del ingrediente.
- **outputs**? ... Conexiones de salida.
  - **name** ... Nombre de la conexión.
  - **type** ... Tipo de la conexión.
- **inputs**? ... Conexiones de entrada.
  - **name** ... Nombre de la conexión.
  - **type** ... Tipo de la conexión.
  - **control**? ... Formulario para la modificación de valores por defecto.
    - **type** ... Tipo de control.
    - **options**? ... Valores de configuración del control.
- **controls**? ... Formularios con los valores de configuración.
  - **name** ... Nombre del control.
  - **type** ... Tipo de control.
  - **options**? ... Valores de configuración del control.
- **code** ... Código *JavaScript* para procesar los datos.

El campo **code** cuenta con una serie de variables globales que permiten el acceso a los datos del nodo: **inputs** (datos de entrada), **outputs** (datos de salida), **node.data** (valor de los controles) y **call** (llamadas a la *API*).

Un ejemplo de especificación de un ingrediente es el mostrado en la figura 7.2.

C | OWASP – Web Security Testing Guide

En este anexo, se resumen algunas de las categorías habituales durante un *pentesting* según la metodología *OWASP*. En concreto, se hace referencia a la *Web Security Testing Guide v4.2*. El documento original completo se puede encontrar en la siguiente dirección web: `https://owasp.org/www-project-web-security-testing-guide/v42/`.

C.1. Contextualización

> *"El Proyecto de Pruebas OWASP ha estado en desarrollo durante muchos años. El objetivo del proyecto es ayudar a la gente a entender el qué, el por qué, el cuándo, el dónde y el cómo de las pruebas de las aplicaciones web. El proyecto ha proporcionado un marco de pruebas completo, no sólo una simple lista de comprobación o una prescripción de cuestiones que deben abordarse. Los lectores pueden utilizar este marco como plantilla para construir sus propios programas de pruebas o para calificar los procesos de otras personas. La Guía de Pruebas describe con detalle tanto el marco general de pruebas como las técnicas necesarias para ponerlo en práctica." – OWASP*

Hay que entender que las pruebas de seguridad nunca serán una ciencia exacta. No es posible llegar a definir una lista concreta con todos los posibles problemas que deben intentar solucionarse.

El objetivo de la *OWASP WSTG* es intentar recopilar el mayor número de pruebas, explicar las técnicas necesarias y mantener la guía actualizada. El método de pruebas de seguridad de aplicaciones web de ***OWASP* se basa en el enfoque de caja negra**, es decir, el auditor no sabe nada o tiene muy poca información sobre la aplicación que va a probar.

C.2. Categorías

Las pruebas contenidas en la guía se agrupan por escenarios. Cada escenario tiene un identificador único, en el formato **WSTG-*categoría-control***. La categoría es una cadena de 4 caracteres en mayúsculas que identifica el tipo de prueba o debilidad, y el control es un número del 01 al 99. Por ejemplo: *WSTG-INFO-03* es la tercera prueba de la categoría *Recopilación de Información*.

La versión 4.2, usada como referencia durante el proyecto, consta de un total de noventa y siete controles, algunos de ellos con múltiples pruebas. Estos controles están agrupados dentro de doce categorías diferentes.

La aplicación presentada como resultado del Trabajo de Fin de Grado descrito en este documento ofrece algunos *ingredientes* ya definidos con los que poder realizar parte de los controles.

Las pruebas cubiertas mediante estos nodos son los siguientes (*Tab. C.1)*.

Ingredientes de la categoría *OWASP*	
Identificador	**Descripción**
WSTG-INFO-01	Reconocimiento con motores de búsqueda
WSTG-INFO-02 (1)	*Fingerprint* del servidor web
WSTG-INFO-03	Revisión de los archivos de *metainformación*
WSTG-INFO-04 (1)	Enumeración de aplicaciones web en diferentes *URLs*
WSTG-INFO-04 (2)	Enumeración de aplicaciones web en diferentes *puertos*
WSTG-INFO-04 (3)	Enumeración de aplicaciones web usando *hosts* virtuales
WSTG-INFO-05	Revisión del contenido de la página web

Tabla C.1: Ingredientes de la categoría *OWASP*

Un ejemplo del uso de estos nodos mediante la aplicación es el mostrado en la figura (*Fig. C.1)*.

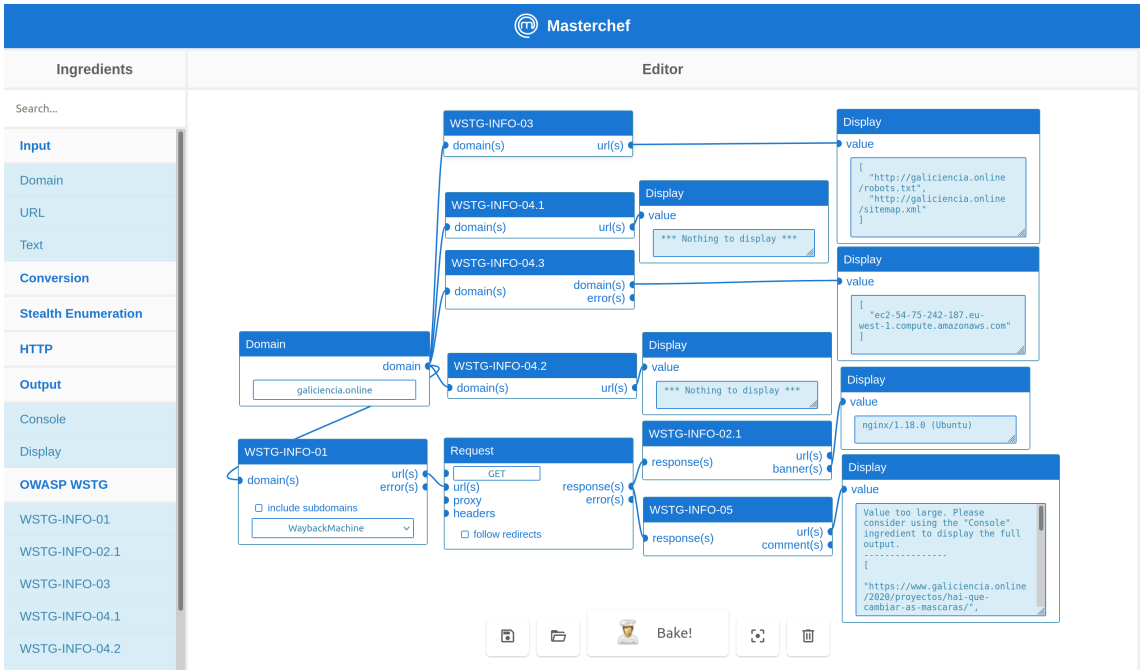


Figura C.1: Controles *INFO* de *OWASP WSGT*.