UNIVERSIDAD AUTONOMA
DE MADRID

Escuela Politécnica Superior

20
21

Trabajo fin de grado

Análisis automatizado de metadatos expuestos en ficheros públicos

Antonio Solana Vera

Escuela Politécnica Superior
Universidad Autónoma de Madrid
C\Francisco Tomás y Valiente nº 11

Campus Internacional excelencia UAM CSIC+

www.uam.es

UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Análisis automatizado de metadatos expuestos en ficheros públicos

Autor: Antonio Solana Vera
Tutor: Jaime Lopez Sánchez
Ponente: Fernando Díez Rubio

junio 2021

Todos los derechos reservados.
Queda prohibida, salvo excepción prevista en la Ley, cualquier
forma de reproducción, distribución comunicación pública
y transformación de esta obra sin contar con la autorización
de los titulares de la propiedad intelectual.
La infracción de los derechos mencionados puede ser constitutiva de
delito contra la propiedad intelectual *(arts. 270 y sgts. del Código Penal).*

DERECHOS RESERVADOS
© 3 de Noviembre de 2017 por UNIVERSIDAD AUTÓNOMA DE MADRID
Francisco Tomás y Valiente, n° 1
Madrid, 28049
Spain

Antonio Solana Vera
Análisis automatizado de metadatos expuestos en ficheros públicos

Antonio Solana Vera
C\ Francisco Tomás y Valiente Nº 11

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

A mi familia y a mis amigos

Resumen

Los metadatos son “datos que nos dan información sobre otros datos”. Por ejemplo: la fecha de edición de un archivo, el autor de un documento o el programa usado para editar una fotografía. La mayor parte del software que se ejecuta en nuestros ordenadores guarda esta información sobre nuestros archivos sin interacción por parte de los usuarios. Esto no tiene por que suponer un problema ya que la mayoría es inofensiva (fecha de guardado, dimensiones de una foto, licencia de un archivo, etc) pero debemos tener cuidado de eliminar la que no lo es (geolocalización, nombres, comentarios privados, etc).

Aunque la gran mayoría de las redes sociales y servicios de subida de ficheros ya se ocupan de eliminar los metadatos de forma automática, muchas empresas más pequeñas no lo hacen. Y aunque lo intentaran, actualmente no existe una herramienta o librería estándar que facilite esta tarea.

El propósito de este trabajo es el desarrollo de un sistema que sirva para extraer estos metadatos de una página web, filtrar los que sean potencialmente peligrosos y generar informes con toda la información relevante.

Palabras clave

extracción de metadatos, seguridad de la información, anonimidad, manipulación de datos, araña

Abstract

Metadata is “data that gives us information about other data”. For example: the last time a file was modified, the author of a document, or the program used to edit an image. The vast majority of software that is executed in our computers saves this information about our files without requiring any user interaction. This should not be a problem given that most of it is harmless (creation date, image resolution, source licenses, etc) but we should be careful to remove the parts that are not (geolocation, names, private comments, etc).

Even though most social networks and file upload services already take care of eliminating metadata automatically, many smaller companies do not. And even if they tried, currently there does not exist a tool or standard library that facilitates this task.

The purpose of this project is the development of a system that extracts all metadata from a website, filters potentially dangerous parts and generates reports with all the relevant information.

Keywords

metadata extraction, information security, anonimity, data handling, scrapper

Índice

1 Introducción **1**

1.1 Fases de realización del proyecto 1

1.2 Estructura del documento 2

2 Estado del arte **3**

2.1 Foca 3

2.2 National Library of New Zealand's Metadata Extraction Tool 4

2.3 Harvard's File Information Tool Set 5

3 Desarrollo de la solución **7**

3.1 Análisis de requisitos 7

3.1.1 Requisitos funcionales 7

3.1.2 Requisitos no funcionales 8

3.1.3 No requisitos 8

3.2 Elixir, Erlang y OTP 8

3.2.1 Erlang 8

3.2.2 OTP 9

3.2.3 Elixir 9

3.3 Introducción a la arquitectura propuesta 10

3.4 Creación de una araña 12

3.4.1 Parseo de HTML y limpieza de URLs 12

3.5 Extracción de metadatos 13

3.5.1 Libextractor 13

3.5.2 Filtros 14

3.5.3 Base de datos 15

3.6 Interfaz web 15

4 Resultados **17**

4.1 Núcleo 17

4.2 API 18

4.3 Interfaz Web 19

4.4 Base de datos 20

4.5 Informes 24

5 Conclusiones y trabajo futuro **27**

Bibliografía 29

Definiciones 31

Apéndices 33

A Instalar Krptkn 35

Listas

Lista de figuras

2.1 Pantalla principal de FOCA 4

3.1 Sintaxis de Erlang y sintaxis de Elixir 10
3.2 Arquitectura general de Krptkn 11

4.1 Estructura de Krptkn 17
4.2 Metadatos mostrados en la API 18
4.3 Página principal de Krptkn 19
4.4 Panel de administración durante ejecución 21
4.5 Panel de administración antiguo 22
4.6 Diagrama EDR de la base de datos 23
4.7 Metadatos almacenados en la base de datos 23
4.8 URLs visitadas en la base de datos 24
4.9 Portada del informe generado por Krptkn 25
4.10 Ejemplo de informe 26

1 Introducción

Aunque la mayoría de los metadatos son inofensivos en circunstancias normales, cuando existe un adversario que tiene intención de atacar nuestro sistema informático, le pueden dar una ventaja inusual. Por ejemplo: metadatos en nuestra página web pública que revelan que nuestro diseñador gráfico utiliza una versión de Adobe Photoshop vulnerable a ataques remotos. O el filtrado de datos que geolocalizan una fotografía, conocidos comúnmente como geotags. Este último puede ser especialmente peligroso [1] en caso de que se hagan públicos estos datos ya que nos permitiría conocer rutas frecuentes o lugar de residencia de una persona cualquiera. Además, la tarea de eliminar estos metadatos es responsabilidad casi exclusiva de la plataforma: muchos teléfonos todavía guardan los geotags sin haber obtenido el consentimiento explícito del usuario antes [2].

El objetivo de este trabajo es el desarrollo de una herramienta, a la que se le ha dado el nombre "Krptkn", que facilite el descubrimiento de metadatos antes de que estos puedan suponer un problema de seguridad. Se utilizará el lenguaje de programación Elixir que se introduce junto con otros conceptos relevantes en la sección 3.2.

1.1. Fases de realización del proyecto

Para el desarrollo del trabajo, se han seguido las etapas estándar del ciclo de vida del software [3]. Estas son las siguientes:

- **Planteamiento del problema**: se formalizan los requisitos y se decide cuáles de estos son necesarios o centrales al proyecto y cuáles son opcionales. Esto se determinará basándonos tanto en el objetivo del proyecto como en el tiempo de desarrollo y la dificultad que supondría cada uno de los requisitos.
- **Estudio de soluciones existentes**: investigación de herramientas que solucionan el problema completa o parcialmente.
- **Diseño de la solución**: durante esta fase se decide el conjunto de software que se va a usar para construir la herramienta o stack. Se tienen en cuenta los requisitos decididos durante el planteamiento para elegir lenguaje, base de datos, plataforma, etc.
- **Desarrollo**: creación de la solución propuesta. Esta fase incluye también pruebas y revisiones.

A lo largo del trabajo, el diseño se ha mantenido muy estable, lo que ha aumentado mucho la

velocidad de trabajo. Se han evitado reescrituras de grandes bloques de código y ha sido fácil mantener una imagen global del sistema en todo momento.

1.2. Estructura del documento

Hemos intentado que la estructura de esta memoria siga, de forma cercana, el mismo orden que las fases de desarrollo de la solución. Cada una de las fases vistas en la sección 1.1 se corresponde con un capítulo de esta memoria de la siguiente manera:

- **Introducción** - Capítulo 1 (Planteamiento del problema).
- **Estado del arte** - Capítulo 2 (Estudio de soluciones existentes).
- **Análisis de requisitos** - Capítulo 3, sección 3.1 (Diseño de la solución).
- **Desarrollo de la solución** - Capítulo 3, sección 3.3 y en adelante (Desarrollo).

Además la memoria consta de dos capítulos más: **Resultados** (capítulo 4) y **Conclusiones y trabajo futuro** (capítulo 5). En estos exploraremos la solución final, sus defectos y posibles mejoras que se pueden aplicar.

2 Estado del arte

En esta sección analizaremos tres herramientas que tienen en común la manipulación de metadatos. Veremos las características principales que las hacen destacar y también lo que las hace inservibles para nuestro caso de uso. Este análisis nos será útil para valorar los requisitos de nuestra solución.

2.1. Foca

FOCA o Fingerprinting Organizations with Collected Archives [4] [5] cumple una tarea muy similar a la de Krptkn. Descarga archivos, extrae los metadatos (haciendo uso de una libreria interna en lugar de una externa como Krptkn) y los presenta al usuario.

La principal diferencia entre Krptkn y FOCA es que mientras que la primera tiene una araña interna que se encarga de analizar la web objetivo en profundidad, la segunda utiliza los resultados de buscadores como Google o DuckDuckGo.

FOCA tiene múltiples métodos para analizar un objetivo aparte del uso de motores de búsqueda:

- **Búsqueda a través de DNS**. Se encuentran nuevos dominios e IPs a través de peticiones a servidores DNS. También se hacen uso de registros registro PTR en caso de que tengamos IPs que queramos escanear.
- **Búsqueda recursiva**. Cada IP nueva que se encuentra, se añade como nuevo objetivo y se analiza.
- **Fuerza bruta**. Se usan diccionarios para probar nombres comunes contra el servidor DNS.
- **Predicción de DNS**. Se detectan patrones de nombres para probar nuevas peticiones o “queries” contra el servidor DNS.
- **Robtex**. Un servicio externo que se usa para encontrar nuevos dominios asociados con el objetivo.

A pesar de tener unas características tan atractivas, FOCA solo puede ser ejecutado en Windows y la base de datos que utiliza es SQL Server 2014. Esta falta de flexibilidad hace que quede relegado a un uso más esporádico. La herramienta es muy útil para equipos de auditoría de seguridad buscando hacer un análisis en profundidad de una empresa, pero no tanto para su uso en forma de servicio (para el que un sistema derivado de Linux es más común [6]).

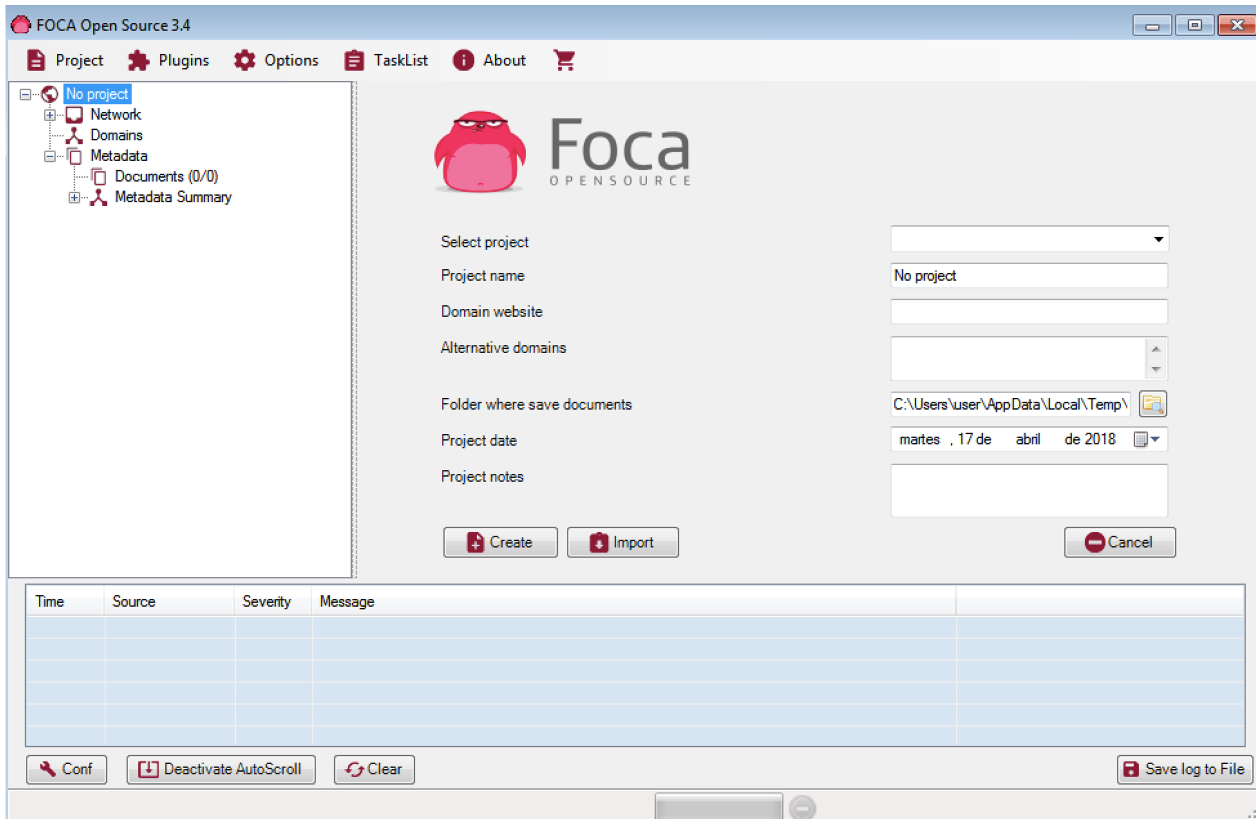


Figura 2.1: Pantalla principal de FOCA. Fuente: Wikipedia (CC BY-SA 4.0)

2.2. National Library of New Zealand's Metadata Extraction Tool

La "Metadata Extraction Tool" de la Biblioteca Nacional de Nueva Zelanda tiene como objetivo la extracción de metadatos de documentos de texto. No tiene funciones de análisis web ni crawling.

El propósito principal de esta herramienta era su uso interno en la Biblioteca Nacional, extrayendo metadatos de los archivos allí almacenados y preservándolos. Por eso no actúa como competidor directo de Krptkn pero sigue siendo importante analizar las funciones de extracción, normalización y almacenamiento.

Las características más notorias de esta herramienta son las siguientes:

- **Soporte tanto para Windows como para Linux**.
- **Alto rendimiento para la extracción**. La herramienta analiza solo una pequeña parte de los archivos dados.
- **Normalización de los metadatos a XML**. XML es un lenguaje de almacenamiento de datos muy flexible que facilita el procesamiento y la búsqueda de datos.
- **Soporte para muchos formatos**. El principal atractivo es la cantidad de archivos de oficina que puede analizar: MS Word, Word Perfect, Open Office, MS Works, MS Excel, MS PowerPoint y PDF. Estos son relativamente comunes y al ser para uso interno suelen contener muchos metadatos reveladores.

- **Interfaz de usuario y librería**. Esto es ideal para nuestro caso de uso: una interfaz para visualizar resultados y una librería para un análisis no supervisado en forma de servicio.

El principal inconveniente de esta herramienta es su relativa antigüedad. La última versión es de 2014 y muchos de los formatos soportados no se usan en las empresas modernas (Word Perfect por ejemplo). Al ser un programa desarrollado exclusivamente con el propósito de clasificar libros de la Biblioteca Nacional de Nueva Zelanda, no tienen incentivo para continuar actualizándolo si actualmente les funciona bien.

2.3. Harvard's File Information Tool Set

FITS o File Information Tool Set [7], es una herramienta desarrollada por la Universidad de Harvard que agrega una docena de librerías todas dedicadas a la extracción de metadatos.

Algunas características de FITS:

- **Gran cantidad de archivos soportados**. Ya que agrega muchas herramientas distintas, la suma de todos los archivos soportados está en los miles.
- **Normalización**. La mayoría de librerías que utiliza están escritas en Java y usan XML para normalizar los datos extraídos.
- **La herramienta se actualiza una o dos veces al año**. Las actualizaciones no añaden nueva funcionalidad, solo modifican la versión de las librerías que usa.
- **Puede ser utilizada en Windows, Mac o Linux**.
- **Interfaz de usuario y librería**. La interfaz no es gráfica, funciona a través de la terminal. La librería pone a disposición sus funciones a través de una interfaz web (API).

En general la herramienta es útil y proporciona una gran cantidad de metadatos sobre muchos formatos distintos. La falta de una librería podría ser ignorada en favor de un despliegue conjunto de Krptkn y FITS, dejando la parte de análisis web, generación de informes, etc en Krptkn y la parte de extracción en FITS, comunicándose ambos a través de la API de FITS.

Por otro lado, la estructura de esta herramienta (agregador de software) tiene algunos inconvenientes como un mayor coste de mantenimiento o una falta de integración que puede ralentizar el proceso de manipulación de los metadatos.

3

Desarrollo de la solución

A lo largo de este capítulo veremos las distintas partes de Krptkn. Hablaremos primero sobre los requisitos que se han propuesto, luego veremos la arquitectura general y por último estudiaremos en profundidad el diseño de cada una de las partes. Estas se explicarán en el mismo orden en que fluye la información por nuestro sistema.

3.1. Análisis de requisitos

Se dividen los requisitos del sistema en funcionales (funciones que debe cumplir) y no funcionales (características generales usadas para juzgar la solución).

Además se ha añadido una sección para comentar los no-requisitos. Estos son atributos en los que no se empleará tiempo de desarrollo por diversas razones que serán explicadas en esa sección.

3.1.1. Requisitos funcionales

Estos son los requisitos funcionales de Krptkn:

- **Crawling de páginas web**. Es necesario el desarrollo de un módulo que se encargue de visitar la web objetivo y que descargue todos los archivos que encuentre en ésta.
- **Extración de metadatos**. De los archivos descargados, tenemos que extraer todos los metadatos y normalizarlos a un formato que pueda ser analizado.
- **Análisis de metadatos**. Una vez obtenidos los metadatos, podemos proceder a filtrar aquellos que nos parezcan peligrosos.
- **Persistencia en base de datos**. Los metadatos obtenidos en la fase de extracción han de ser guardados para su posterior uso en los informes.
- **Generación de informes**. El formato debe permitir la edición de los informes. Se busca esta característica para poder añadir comentarios en caso de haber revisado los metadatos manualmente y querer señalar algo sobre estos.

3.1.2. Requisitos no funcionales

El sistema a desarrollar deberá tener las siguientes características:

- **Modular y extensible**. Los distintos componentes de los que estará formado el sistema se podrán reemplazar sin cambios externos.
- **Estabilidad**. El sistema deberá poder correr durante largos periodos de tiempo, recuperarse de errores menores y reiniciarse en caso de que no se pueda recuperar.
- **Escalable**. En caso de ser necesario expandir la potencia de la solución, el escalado horizontal debe ser lo más simple posible. Gracias a Elixir y su librería estándar (OTP) tanto la creación de clusters como el escalado en una sola máquina resulta muy simple.
- **Facilidad de uso**. Es importante que el sistema sea fácilmente manejado por una persona que no esté familiarizada con este software.

3.1.3. No requisitos

Es tan importante conocer los requisitos del proyecto como conocer los no-requisitos. Por ello, se enumeran ahora atributos que serán evitados en el desarrollo de la solución:

- **Seguridad**. El sistema no está diseñado para ser expuesto a Internet y por tanto no se empleará tiempo en auditar su seguridad ni endurecerlo contra ataques externos.
- **Alto rendimiento**. Aunque no se pretende que la solución sea lenta, cuando se trata con servidores externos (no preparados para aguantar mucho estrés) el cuello de botella no son los procesos internos sino el servidor al que estamos haciendo las peticiones.

3.2. Elixir, Erlang y OTP

Antes de comenzar con la explicación del desarrollo del sistema, ofreceremos una breve introducción a los lenguajes de programación Erlang y Elixir así como al entorno que ambos utilizan (OTP).

3.2.1. Erlang

Erlang es un lenguaje de programación que fue inventado por la compañía de telecomunicaciones Ericsson en 1998 para su uso interno [8]. Debido a los requisitos que tiene una compañía de teléfonos, Erlang fue diseñado para satisfacer las siguientes características:

- Concurrencia
- Resistencia a fallos
- Alta disponibilidad

Erlang se ejecuta en una máquina virtual llamada “BEAM” que permite cosas tan impresionantes

como cambiar el código de una aplicación mientras se está ejecutando o tener miles de procesos ejecutándose en paralelo [9] [10].

3.2.2. OTP

OTP o Open Telecom Platform es el conjunto de herramientas que se utilizan junto con Erlang para aumentar sus capacidades. A pesar de su nombre, esta librería se usa en proyectos de todo tipo, no solo en empresas de telecomunicaciones. Los principales componentes de OTP son:

- El intérprete y el compilador de Erlang que ejecutan el código.
- Un protocolo de comunicación para los nodos del sistema.
- Una herramienta para el análisis estático de código llamada Dialyzer.
- Una base de datos distribuida llamada Mnesia.
- Muchos comportamientos genéricos que nos sirven como plantilla para ahorrar código.

Entre los componentes más utilizados en este trabajo están el comportamiento “gen_server” y el comportamiento “supervisor”. El primero nos permite crear procesos que reciben peticiones y responden por medio de colas de mensajes (permitiendo que nuestro código sea asíncrono por defecto) y el segundo nos permite supervisar todo nuestro sistema para que, en caso de que algo falle, se reinicie la parte defectuosa sin afectar al resto del programa.

También es importante conocer el componente Erlang Term Storage o ETS [11] que nos permite guardar valores en memoria con una clave que los identifica. Estos elementos se almacenan en tablas en las que los podemos buscar, filtrar y extraer de forma rápida y simple.

3.2.3. Elixir

Elixir es un lenguaje de programación que se ejecuta en la máquina virtual de Erlang y hace uso de todas las herramientas de OTP pero con una sintaxis y un entorno de programación más modernos.

Además de “gen_server” y “supervisor”, también hacemos uso de un comportamiento exclusivo de Elixir: “gen_stage”. Este nos permite construir nuestro programa como una linea de producción en la que vamos procesando y refinando los datos que entran por un lado hasta que tenemos exactamente lo que necesitamos en el otro. Además, en nuestra linea de producción, la presión viene del final. Esto permite que nuestro sistema no se sobrecargue cuando nos llegan demasiados datos.

Erlang

```
-module(hello_module).
-export([some_fun/0, some_fun/1]).

% A "Hello world" function
some_fun() ->
  io:format('~s~n', ['Hello world!']).

% This one works only with lists
some_fun(List) when is_list(List) ->
  io:format('~s~n', List).

% Non-exported functions are private
priv() ->
  secret_info.
```

Elixir

```
defmodule HelloModule do
  # A "Hello world" function
  def some_fun do
    IO.puts "Hello world!"
  end

  # This one works only with lists
  def some_fun(list) when is_list(list) do
    IO.inspect list
  end

  # A private function
  defp priv do
    :secret_info
  end
end
```

Figura 3.1: Sintáxis de Erlang y sintaxis de Elixir

3.3. Introducción a la arquitectura propuesta

La idea inicial de cómo debería funcionar el software no ha cambiado desde las primeras iteraciones. A continuación se muestra en forma de pseudocódigo el núcleo del sistema:

```
# Datos de salida
urls_no_visitadas = []
urls_visitadas = []
metadatos = []

# Datos de entrada
urls_no_visitadas.push(input_usuario())

# Funcion principal
for url in urls_no_visitadas:
    # Descargamos los datos de la URL
    archivo = descargar(url)

    # Si es HTML, extraemos URLs del archivo
    if is_html(archivo):
        nuevas_urls = extraer_urls(archivo)
        urls_no_visitadas.push(nuevas_urls)
    # Si no es HTML, extraemos metadatos
    else:
        nuevos_metadatos = extraer_metadatos(archivo)
```

```
metadatos.push(nuevos_metadatos)
```

Aunque el núcleo del sistema es ese, necesitamos estructuras de datos más complejas para poder cumplir los requisitos propuestos. En el diagrama 3.2 queda plasmado el verdadero aspecto de nuestra aplicación. En azul podemos ver los servicios externos:

- Web Objetivo. Página de la que queremos extraer los metadatos para analizarlos.
- Libextractor. Librería que usamos para la extracción de metadatos. Se verá en la sección 3.5.1.
- Base de datos. Servicio que usaremos para almacenar los datos extraídos. Corresponde con la sección 3.5.3.

Vemos también dos cilindros en la parte superior del diagrama. Ambos representan dos tablas ETS:

- **Cola de URLs**. Almacena las URLs que todavía no hemos visitado.
- **URLs visitadas**. Almacena todas las URLs que hemos visitado. Se utiliza para evitar descargar múltiples veces una misma página.

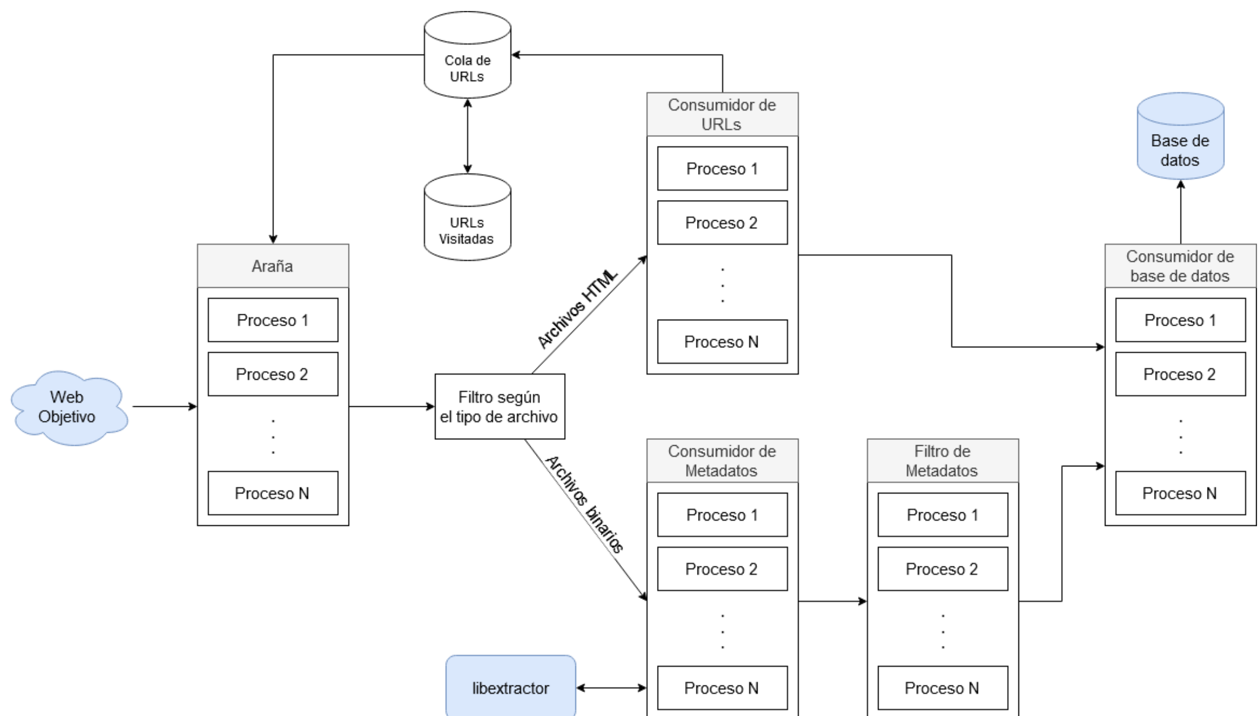


Figura 3.2: Vista de águila de la arquitectura de Krptkn

El resto de elementos son procesos que se comunican mediante colas de mensajes y realizan las siguientes tareas (de izquierda a derecha en el diagrama):

1.– **Araña**. Descarga datos de la Web Objetivo.

2.– **Filtro por tipos**. Comprueba si los archivos descargados por la araña son de tipo HTML. Si lo son, los manda al **Consumidor de URLs**. Si no, los manda al **Consumidor de Metadatos**.

3.– **Consumidor de URLs**. Se encarga de extraer URLs de los archivos HTML que recibe. Todas las URLs que recibe las inserta en la **Cola de URLs** y las envía al **Consumidor de base de datos**.

4.– **Consumidor de Metadatos**. Extrae los metadatos de los archivos que recibe y los remite al **Filtro de**

Metadatos.

5.– **Filtro de Metadatos**. Comprueba si los metadatos recibidos son peligrosos o no y los pasa al **consumidor de base de datos**.

6.– **Consumidor de base de datos**. Guarda en las tablas correspondientes de la Base de datos toda la información que recibe.

3.4. Creación de una araña

Como ya hemos visto, en el núcleo de la aplicación se encuentra la araña, que se encarga de explorar la página web en profundidad. El funcionamiento básico de la araña consiste en extraer una nueva URL de la tabla de URLs (Cola de URLs en la parte superior de la figura 3.2), hacer una petición GET al servidor para descargar el archivo y mandar el resultado al filtro que se ocupará de redirigir el contenido al módulo de procesamiento adecuado.

Existen dos casos en los que se pueden producir errores (siempre internos y controlados; no causan una parada del sistema). En caso de que no exista ninguna URL en la tabla, se producirá un timeout y se reintentará hasta que alguien inserte una nueva URL a ésta. También puede ocurrir que el servidor no responda a tiempo a la petición GET. Esto es ignorado y se tratará de la misma forma que el problema anterior.

La librería que se usa para las peticiones HTTP es HTTPoison, inspirada por otra librería llamada HTTPotion y usando como base Hackney. La decisión de usar HTTPoison se basa en que es fácil de usar, es madura y muy estable, usa SSL [1] por defecto y no nos va a crear un cuello de botella. Aunque hay evidencia de que Hackney tiene problemas de escalado para aplicaciones de mucho tráfico [12] [13], hemos optado por ignorarlos, ya que el cuello de botella de Krptkn es la parte de análisis de metadatos.

3.4.1. Parseo de HTML y limpieza de URLs

La acción de “parsear” un archivo implica un análisis formal de un lenguaje para identificar sus componentes y las relaciones entre estos. Una vez terminado el análisis podemos crear un árbol de parseo que nos permite trabajar con los datos del lenguaje de forma estructurada y rápida.

En Krptkn tenemos un módulo de parseo de HTML que utiliza la librería Floki [2] para facilitar la búsqueda de las distintas partes de la web. Floki ofrece tres parsers distintos:

- **mochiweb_html**: Escrito en Erlang, sin dependencias externas, lento.

[1] Capa de seguridad que encripta las comunicaciones web.
[2] `https://github.com/philss/floki`

- **fast_html**: Usa lexbor [3], escrito en C, muy rápido.
- **html5ever**: Usa html5ever [4], escrito en Rust, muy rápido también.

Se ha optado por el uso de fast_html porque ha sido el que menos fallos ha dado al probarlo con páginas web complejas. También es importante mencionar que la velocidad no era un problema en ningún caso, al usar muchos hilos paralelos, los tres cumplen de sobra el requisito de velocidad.

Una vez hemos parseado el HTML se buscan los nodos que contienen una propiedad “href” o “src”, ya que son los que más a menudo contienen URLs interesantes. Una vez obtenidos estos nodos, se pasa a la limpieza de las URLs.

Esta limpieza es heurística y pasa por varias etapas:

1.– Se comprueba si la URL pertenece al dominio objetivo. Si estamos analizando “example.org” la URL “https://code.jquery.com/jquery-3.6.0.js” se descarta.

2.– Agregamos o eliminamos las barras oblicuas al final de la URL. Por ejemplo “http://example.org/blog” se convierte en “http://example.org/blog/” y “http://example.org/index.html” se queda igual.

3.– HTML permite usar links del estilo “/index.html” cuando en realidad la URL completa sería “http://example.org/index.html". Otra de las funciones de la heurística se encarga de completar este tipo de URLs.

4.– Por último se eliminan los links que usan otros protcolos como FTP, SSH, IPFS, etc. Por ejemplo, descartamos la URL “mailto:help@example.org".

Una vez se ha hecho esto, se eliminan los duplicados y los que nos quedan se insertan en la cola de URLs. La cola de URLs se ocupa de comprobar si ya hemos visitado el link que le hemos pasado.

3.5. Extracción de metadatos

La etapa más importante del pipeline de datos de Krptkn es el tratamiento de metadatos. En esta se extraen, filtran e insertan los metadatos encontrados en los archivos de la página web objetivo. En la siguientes secciones analizaremos estas tres partes y cómo cada una de ellas transforma los metadatos.

3.5.1. Libextractor

De las distintas alternativas posibles para la extracción de metadatos se decidió usar *libextractor* por múltiples razones:

- **Lenguaje**: Al ser una librería escrita en C, se tiene la seguridad de que la máquina virtual de Erlang o BEAM

[3] https://github.com/lexbor/lexbor
[4] https://github.com/servo/html5ever

(sobre la que se ejecuta nuestro código) va a ser compatible. También es tranquilizador saber que, si fuera necesario, es posible modificar el código fuente de forma relativamente fácil y rápida.

- **Extensiones o plugins**: *Libextractor* está pensado para ser expandido con nuevos formatos. Cada formato que puede leer esta aislado en un plugin externo que se encarga de cargar el archivo y devolver los metadatos encontrados. Esto nos permite expandir las capacidades de Krptkn o quitar plugins que no sean necesarios o estén anticuados.
- **Procesos aislados**: Una característica importante del funcionamiento de *libextractor* es que las llamadas a los plugins externos usan la llamada al sistema “fork”. Esto nos asegura que lo peor que puede ocurrir cuando intentamos extraer metadatos de un archivo corrupto o maligno es que ese proceso muera y no nos devuelva metadatos.

Aunque estas características son muy importantes, *libextractor* tiene también múltiples inconvenientes:

- **Lentitud**: A pesar de estar escrito en C, la carga de los plugins, la creación del fork y la copia del archivo para analizar puede ser lenta en comparación con la araña. Debido a esto, la velocidad de exploración se ve limitada. En la versión final de Krptkn, no se limita la velocidad de la spider pero cuando acaba la exploración hay que esperar durante unos minutos para que todos los archivos descargados terminen de ser analizados.
- **Paralelismo**: Una de las principales ventajas de Elixir y OTP es su capacidad para paralelizar el código de forma muy simple. El estar usando una librería nativa que hace uso de forks y lee librerías compartidas (.so, .a, .o) hace imposible aprovechar el procesamiento en paralelo que sí se usa en el resto de módulos de Krptkn.

Estos problemas se podrían mitigar a la vez que mantenemos las ventajas haciendo uso de una librería de extracción de metadatos escrita en Elixir. Esto nos permitiría hacer uso de tantos procesos paralelos como fuera necesario, acelerando mucho la ejecución.

En las fases iniciales del proyecto se creó un módulo para leer metadatos de imágenes PNG y se concluyó que aunque la creación del parser había sido bastante rápida (entre dos y tres horas), desarrollar los módulos necesarios para igualar a *libextractor* (PDF, Word, MP3, OGG, AVI, DEB, etc.) iba a ser imposible sin un equipo de desarrolladores más amplio o más tiempo.

En conclusión, lo más adecuado ahora mismo es usar *libextractor* y si fuera posible, ir creando parsers en Elixir para acelerar la fase de extracción. Ambos pueden ser usados a la vez gracias a la arquitectura de Krptkn por lo que el desarrollo de uno para archivos que sabemos que son muy comunes (como por ejemplo PDF o JPG) podría acelerar mucho el sistema sin requerir demasiado tiempo.

3.5.2. Filtros

Aunque en iteraciones iniciales del proyecto se había planteado la posibilidad de usar aprendizaje automático para la clasificación y filtrado de metadatos, se ha optado por una solución menos compleja: Regex.

Regex es un lenguaje que permite escribir “expresiones regulares” que encuentran patrones en

un texto dado. A través de unas reglas de regex muy simples, se extraen correos electrónicos y se comprueba si el texto de los metadatos contiene o no palabras clave que son indicativas de que los metadatos contienen información sensible.

Los datos filtrados no son descartados. Los datos que pueden ser peligrosos se marcan como tal y se insertan en la base de datos para un posterior análisis estadístico y para la creación de los informes pertinentes.

3.5.3. Base de datos

La decisión principal con respecto a bases de datos que se toma en el desarrollo de este proyecto es sobre el uso de SQL o NoSQL. Ambos son lenguajes para extraer información de una base de datos. Las bases de datos SQL los almacenan en tablas con relaciones entre ellas mientras que las NoSQL utilizan pares de clave-valor. SQL es por tanto mejor para datos estructurados y NoSQL para datos con formas flexibles [14]. También hay diferencias en el escalado de estos dos sistemas pero para nuestro caso de uso no será necesario plantearse formas de optimizar la velocidad el sistema.

Debido a que los metadatos de diferentes formatos tienen estructuras muy distintas, se suelen representar como pares de clave valor como `"Size": "480x360"`. Esto encaja muy bien con bases de datos NoSQL como MongoDB. A pesar de esto, se ha decidido usar PostgreSQL y el formato JSON (similar a XML pero más simple). Podríamos haber usado una columna de texto pero el formato JSON de PostgreSQL nos permite verificar que el campo ha sido creado correctamente. También tenemos la ventaja de poder cambiar de JSON a JSONb en cualquier momento. JSONb (json binario) nos daría una mayor velocidad de procesamiento a cambio de perder velocidad de inserción.

Y por último, debido a que SQL es la opción por defecto en la mayoría de casos y cuenta con una gran cantidad de soporte para todo tipo de problemas, usar PostgreSQL es la decisión más segura.

3.6. Interfaz web

Una vez la funcionalidad básica del proyecto estaba terminada, iniciamos el desarrollo de una interfaz web para facilitar su uso. También ha resultado extremadamente útil para encontrar problemas en la aplicación durante su desarrollo.

La decisión de usar un framework web en lugar de una interfaz de terminal o una interfaz de usuario nativa se debe a que Krptkn está pensado para ser ejecutado en un servidor sin interfaz gráfica y para este tipo de despliegues lo más adecuado es una interfaz web a la que puedas acceder desde un ordenador en la misma red local. También se podría configurar para su uso a través de internet aunque el software no esta pensado para aceptar múltiples usuarios a la vez y podría suponer un problema de seguridad.

El framework web que utiliza Krptkn es Phoenix. Éste es el estándar para desarrollo de aplicaciones web en Elixir y por lo tanto la opción con más documentación disponible. Aunque en un principio se consideró la idea de usar la variante LiveView de Phoenix que hace uso de websockets para actualizar la interfaz instantáneamente sin necesitar código Javascipt, finalmente decidimos usar Phoenix junto con una API JSON para actualizar los datos de la interfaz. Esto nos da más flexibilidad en el futuro para añadir nuevas funciones y aumenta mucho la velocidad de desarrollo de la parte web (en la cual no se debería invertir más tiempo del necesario).

4 Resultados

Como se puede observar en la figura 4.1, Krptkn consta de tres capas. Estas son etapas incrementales e independientes y han permitido un desarrollo en el cual desde el principio se podían probar todas las funciones de Krptkn. En las siguientes secciones serán explicadas de dentro a fuera.

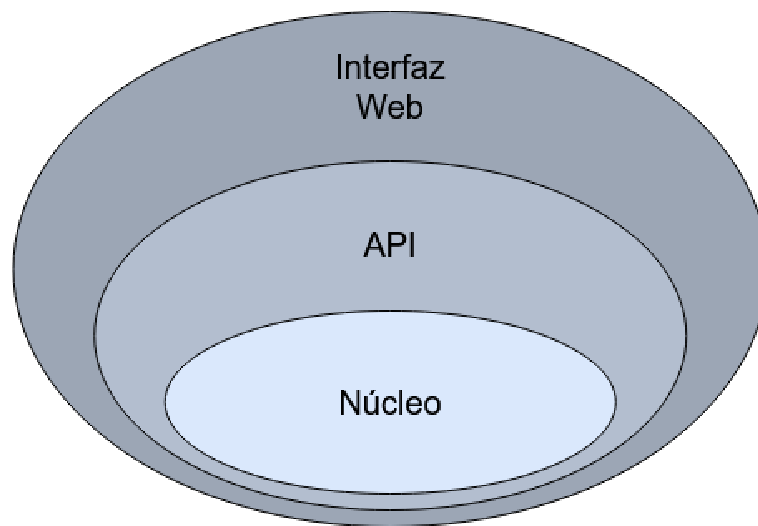


Figura 4.1: Estructura de Krptkn

4.1. Núcleo

El núcleo de la aplicación consiste en todas las partes explicadas a lo largo del capítulo 3. A continuación se puede ver un ejemplo de uso de las funciones de Krptkn sacado de uno de los módulos de la aplicación:

```
def manual_start(initial_url) do
    # Creamos un objeto URI con la URL
    initial_uri = URI.parse(initial_url)

    # Creamos una lista de URLs que contiene la proporcionada
```

```
    # y otras generadas por Krptkn
    initial_urls = [initial_url | Krptkn.Prelaunch.dictionary(initial_uri)]

    # Cada URL la metemos en la cola
    for url <- initial_urls do
        Krptkn.UrlQueue.push(url)
    end
end
```

Se puede encontrar la documentación (sin el código fuente) en `https://solanav.github.io/krptkn_docs/Krptkn.html`. Ésta contiene información sobre los parámetros de todas las funciones del núcleo y ejemplos de uso para las más importantes.

4.2. API

La API, o “Application Programming Interface”, es una interfaz para la comunicación entre la página web y el núcleo de la aplicación a través de HTTP. Se ha utilizado el framework Phoenix para su desarrollo y las respuestas se dan en formato JSON por su amplia compatibilidad. En la figura 4.2 se puede observar un ejemplo de respuesta tras unos minutos analizando una web. La petición se realiza a una URL como “http://localhost:4000/state/last_metadata”.


```
▼ 0:
    comment:                "File written by Adobe Photoshop¨ 4.0"
▼ 1:
    comment:                "File written by Adobe Photoshop¨ 4.0"
▼ 2:
    comment:                "File written by Adobe Photoshop¨ 4.0"
▼ 3:
  ▼ comment:                "[c] 2004 [illegible] & [illegible]"
▼ 4:
    produced by software:   "Adobe ImageReady"
▼ 5:
    keywords:               "2012-08-15T14:09:49+01:00"
▼ 6:
    keywords:               "2012-09-18T23:49:36+01:00"
    produced by software:   "www.inkscape.org"
```


Figura 4.2: Metadatos mostrados en la API

La API proporciona acceso a muchos más datos sobre el estado interno de la aplicación. Algunos

ejemplos son:

- CPU, memoria y procesos de la máquina virtual de Erlang.
- Estadísticas sobre el número de metadatos o de URLs.
- Últimos metadatos que se han encontrado.
- Últimas URLs visitadas.

Estos endpoints se usan a través de la interfaz como veremos más adelante. La excepción son los de CPU, memoria y procesos. Estos últimos fueron utilizados durante la fase de depurado del proyecto para mejorar el rendimiento y el uso de memoria de Krptkn pero no en su versión final.

4.3. Interfaz Web

La interfaz gráfica consta de una sola página HTML y usa Javascript para hacer llamadas a la API (descrita en la sección 4.2).

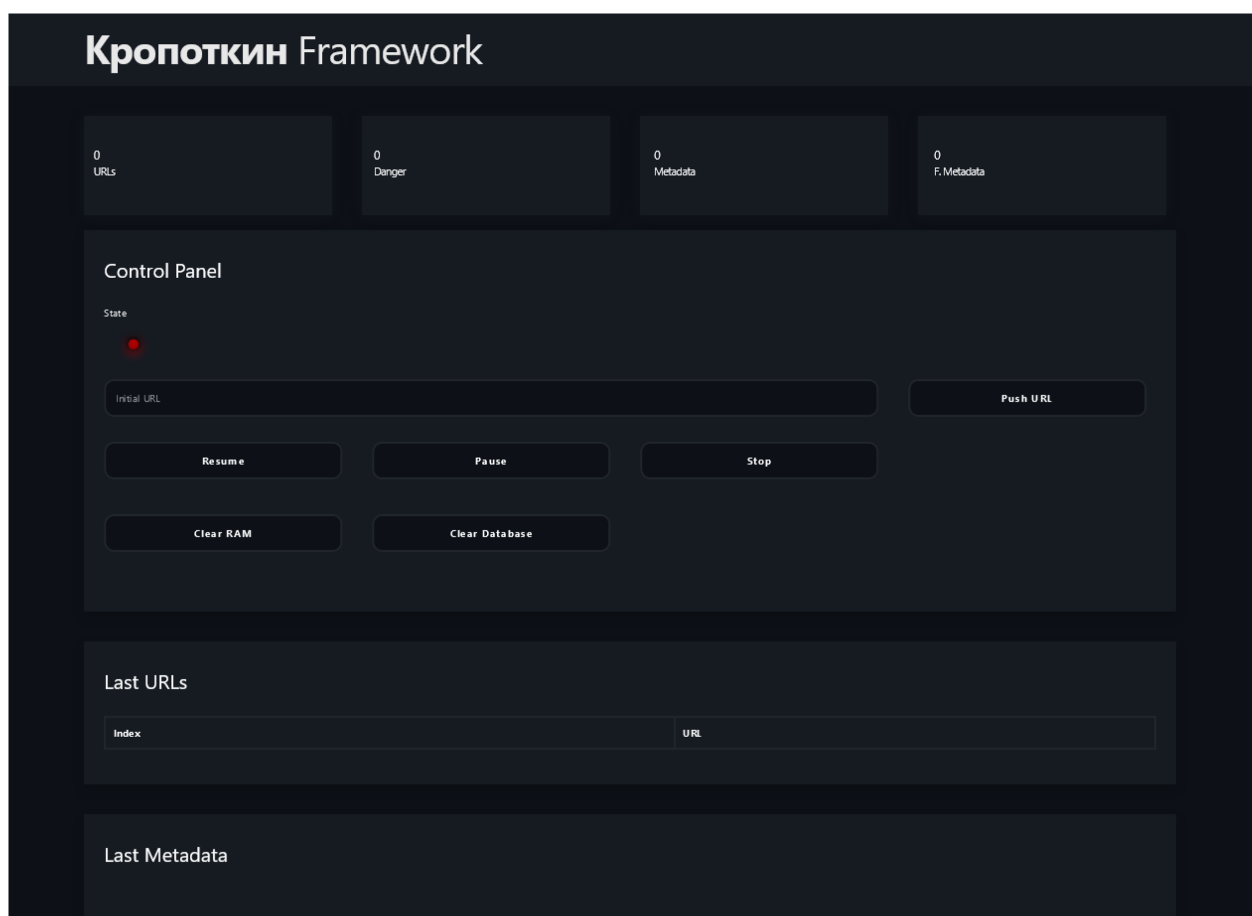


Figura 4.3: Página principal de Krptkn (Panel de administración)

Como se puede observar en la figura 4.3 en la parte superior de la página tenemos cuatro contadores básicos.

- **“URLs”**. Número de URLs visitadas.
- **“Danger”**. Número de metadatos peligrosos encontrados.
- **“Metadata”**. Número de metadatos detectados por la librería interna (libextractor por defecto).
- **“F. Metadata”**. Número de metadatos filtrados. Este filtrado elimina elementos vacíos o irrelevantes.

En la sección “Control Panel” se encuentran todas las funciones que necesita un administrador. Una pequeña bombilla con la etiqueta “State” nos indica si el programa está ejecutándose (verde), pausado (amarillo) o parado (rojo). Justo debajo tenemos un cuadro para insertar una URL en la cola. Los botones “Resume”, “Pause” y “Stop” nos permiten continuar, pausar o parar el sistema respectivamente. “Clear RAM” y “Clear Database” permiten resetear el estado interno de Krptkn y limpiar la base de datos si fuera necesario (por haber generado ya los informes relevantes por ejemplo).

En la figura 4.4 se ve la página de administración con las dos últimas secciones (“Last URLs” y “Last Metadata”) mostrando información. Ambas guardan solo una fracción de los datos: los suficientes para que el administrador pueda asegurarse del funcionamiento correcto de la aplicación.

En esta última (figura 4.5 se muestra una visualización (no presente en la versión final de Krptkn) que mostraba datos sobre la CPU, la memoria y los procesos ejecutándose en la máquina virtual de Erlang. Aunque la API correspondiente sigue presente para su uso futuro, esta parte de la interfaz desapareció después de la fase de depurado debido a unos problemas con la biblioteca que se usaba para dibujar los gráficos. En su lugar se usa ahora una herramienta de depurado desarrollada en Erlang llamada “observer” [1]. Esta se puede dejar activada solamente cuando haga falta y muestra los mismos datos que la de Krptkn.

4.4. Base de datos

La base de datos en la que se guarda la información es un PostgreSQL [2]. La estructura es muy simple. Consta de dos tablas: “metadata” (figura 4.7) y “urls” (figura 4.8).

Ambas tablas comparten los siguientes campos:

- **“id”**. Campo que identifica de forma única cada entrada.
- **“session”**. Nombre que identifica la sesión actual. Resulta muy útil para filtrar resultados en caso de almacenar varios objetivos a la vez en la base de datos.
- **“url”**. Lugar en la que se encontró un metadato (tabla metadata) o la URL visitada (tabla urls).
- **“type”**. Indica el tipo MIME [3] de la fila.
- **“inserted_at”**. Fecha en la que se insertó la entrada. No cambia nunca.
- **“updated_at”**. Fecha de la última actualización de la entrada. Podría cambiar, pero en la versión actual no lo

[1] https://erlang.org/doc/man/observer.html
[2] https://www.postgresql.org/
[3] https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types

Figura 4.4: Panel de administración durante la ejecución de un análisis

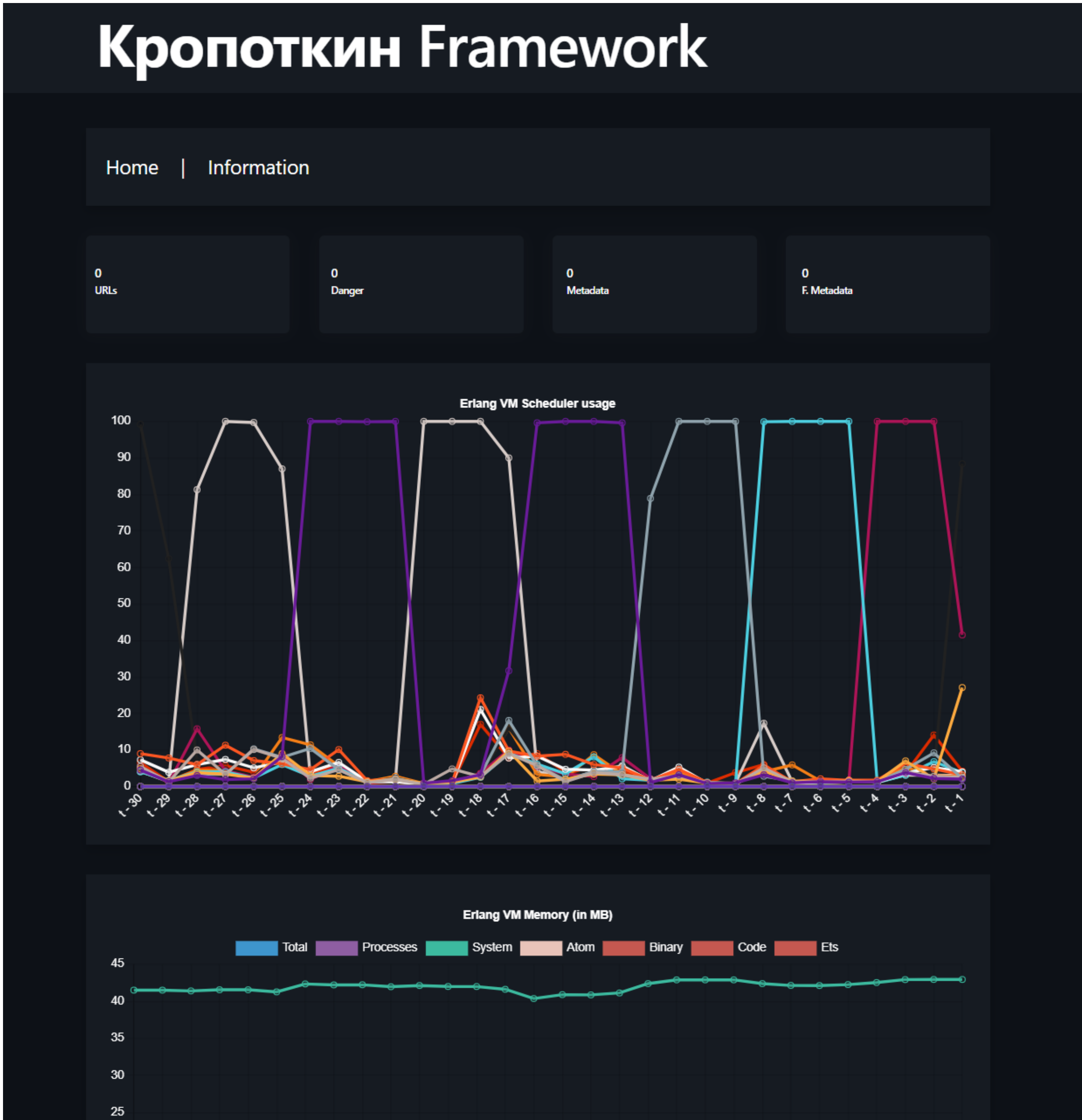


Figura 4.5: Panel de administración antiguo

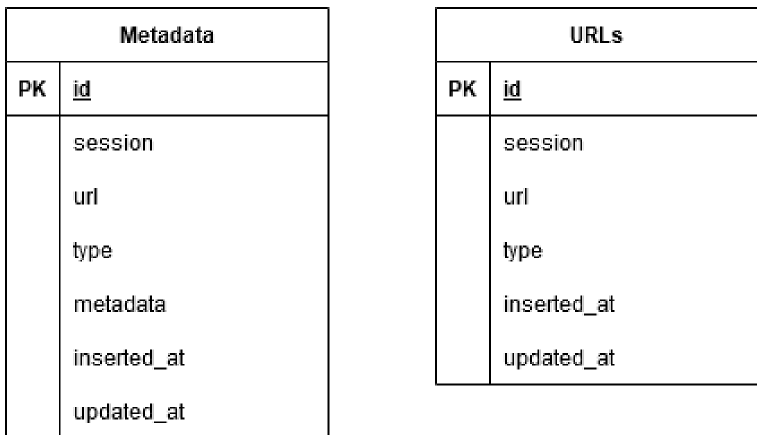


Figura 4.6: Diagrama EDR de la base de datos

id	session	url	type	metadata	inserted_at	updated_at
2,391	real_[illegible]	https://[illegible]	audio/ogg	{"genre": "Protesta", "title": "Guantanamero", "arti...	2021-04-26 15:33:44	2021-04-26 15:33:44
2,392	real_[illegible]	https://[illegible]	image/jpeg	{"comment": "LEAD Technologies Inc. V1.01"}	2021-04-26 15:33:57	2021-04-26 15:33:57
2,393	real_[illegible]	https://[illegible]	image/jpeg	{"comment": "LEAD Technologies Inc. V1.01"}	2021-04-26 15:33:58	2021-04-26 15:33:58
2,394	real_[illegible]	https://[illegible]	image/jpeg	{"comment": "binary comment", "camera make": "...	2021-04-26 15:33:59	2021-04-26 15:33:59
2,395	real_[illegible]	https://[illegible]	image/jpeg	{"comment": "binary comment", "camera make": "...	2021-04-26 15:34:01	2021-04-26 15:34:01
2,396	real_[illegible]	https://[illegible]	audio/ogg	{"vendor": "Xiph.Org libVorbis I 20101101 (Schauf...	2021-04-26 15:34:07	2021-04-26 15:34:07
2,397	real_[illegible]	https://[illegible]	image/png	{"produced by software": "www.inkscape.org"}	2021-04-26 15:34:10	2021-04-26 15:34:10
2,398	real_[illegible]	https://[illegible]	image/png	{"produced by software": "www.inkscape.org"}	2021-04-26 15:34:12	2021-04-26 15:34:12
2,399	real_[illegible]	https://[illegible]	image/png	{"comment": "Created with GIMP", "modification d...	2021-04-26 15:34:12	2021-04-26 15:34:12
2,400	real_[illegible]	https://[illegible]	image/png	{"title": "The Saint Button", "copyright": "Copyrigh...	2021-04-26 15:34:13	2021-04-26 15:34:13
2,401	real_[illegible]	https://[illegible]	image/png	{"produced by software": "www.inkscape.org"}	2021-04-26 15:34:18	2021-04-26 15:34:18
2,402	real_[illegible]	https://[illegible]	image/png	{"produced by software": "Adobe ImageReady"}	2021-04-26 15:34:19	2021-04-26 15:34:19
2,403	real_[illegible]	https://[illegible]	image/jpeg	{"created by software": "Adobe Photoshop CS Wi...	2021-04-26 15:34:21	2021-04-26 15:34:21
2,404	real_[illegible]	https://[illegible]	image/jpeg	{"comment": "[c] 2004[illegible] ...	2021-04-26 15:34:22	2021-04-26 15:34:22
2,405	real_[illegible]	https://[illegible]	image/jpeg	{"comment": "[c] 2004[illegible] ...	2021-04-26 15:34:23	2021-04-26 15:34:23
2,406	real_[illegible]	https://[illegible]	image/jpeg	{"comment": "[c] 2004[illegible] ...	2021-04-26 15:34:23	2021-04-26 15:34:23
2,407	real_[illegible]	https://[illegible]	image/jpeg	{"comment": "File written by Adobe PhotoshopÂ¨...	2021-04-26 15:34:24	2021-04-26 15:34:24
2,408	real_[illegible]	https://[illegible]	image/jpeg	{"comment": "File written by Adobe PhotoshopÂ¨...	2021-04-26 15:34:25	2021-04-26 15:34:25
2,409	real_[illegible]	https://[illegible]	image/jpeg	{"comment": "File written by Adobe PhotoshopÂ¨...	2021-04-26 15:34:25	2021-04-26 15:34:25
2,410	real_[illegible]	https://[illegible]	image/png	{"keywords": "2016-02-03T18:03:04-05:00", "mo...	2021-04-26 15:34:26	2021-04-26 15:34:26
2,411	real_[illegible]	https://[illegible]	image/png	{"keywords": "2012-09-18T23:49:36+01:00", "pro...	2021-04-26 15:34:29	2021-04-26 15:34:29
2,412	real_[illegible]	https://[illegible]	image/png	{"keywords": "2012-08-15T14:09:49+01:00"}	2021-04-26 15:34:29	2021-04-26 15:34:29

Figura 4.7: Metadatos almacenados en la base de datos

id	session	url	type	inserted_at	updated_at
24,500	real_	https://	text/html	2021-04-26 15:33:42	2021-04-26 15:33:42
24,501	real_	https://	text/html	2021-04-26 15:33:42	2021-04-26 15:33:42
24,502	real_	https://	text/html	2021-04-26 15:33:44	2021-04-26 15:33:44
24,503	real_	https://	text/html	2021-04-26 15:33:49	2021-04-26 15:33:49
24,504	real_	https://	text/html	2021-04-26 15:33:49	2021-04-26 15:33:49
24,505	real_	https://	text/html	2021-04-26 15:33:49	2021-04-26 15:33:49
24,506	real_	https://	text/html	2021-04-26 15:33:49	2021-04-26 15:33:49
24,507	real_	https://	text/html	2021-04-26 15:33:49	2021-04-26 15:33:49
24,508	real_	https://	text/html	2021-04-26 15:33:49	2021-04-26 15:33:49
24,509	real_	https://	text/html	2021-04-26 15:33:50	2021-04-26 15:33:50
24,510	real_	https://	text/html	2021-04-26 15:33:50	2021-04-26 15:33:50
24,511	real_	https://	text/html	2021-04-26 15:33:50	2021-04-26 15:33:50
24,512	real_	https://	text/html	2021-04-26 15:33:50	2021-04-26 15:33:50
24,513	real_	https://	text/html	2021-04-26 15:33:50	2021-04-26 15:33:50
24,514	real_	https://	text/html	2021-04-26 15:33:50	2021-04-26 15:33:50
24,515	real_	https://	text/html	2021-04-26 15:33:50	2021-04-26 15:33:50
24,516	real_	https://	text/html	2021-04-26 15:33:50	2021-04-26 15:33:50
24,517	real_	https://	text/html	2021-04-26 15:33:50	2021-04-26 15:33:50
24,518	real_	https://	text/html	2021-04-26 15:33:50	2021-04-26 15:33:50

Figura 4.8: URLs visitadas en la base de datos

hace.

La única columna que no es compartida es “metadata”. En ella se guarda en formato JSON la información encontrada de cada uno de los archivos almacenados.

Es importante saber que los archivos descargados se descartan una vez se han extraído los metadatos: no los estamos guardando en la base de datos. Guardar los archivos completos nos podría suponer un problema importante en cuanto a espacio.

4.5. Informes

Los informes generados comienzan con una portada (ver figura 4.9) que contiene la información más relevante. Está pensada para ser suficiente en caso de no tener tiempo para leer el documento completo.

Además del identificador de la sesión, se muestran el dominio principal, la URL desde la que se empezó a analizar la web, el nivel de peligro estimado y la fecha de inicio y fin del análisis.

El nivel de peligro se calcula dividiendo el número total de metadatos detectados como peligrosos entre el número total de archivos con metadatos que se ha encontrado.

El resto del informe contiene gráficos y tablas que muestran con más detalle cuáles han sido los metadatos encontrados, las URLs más visitadas y los tipos de archivo más frecuentes en la web objetivo. Se puede ver un ejemplo de estas secciones en la figura 4.10.

Nombre de la sesión: real_

Dominio base: .org

URL de entrada: https://

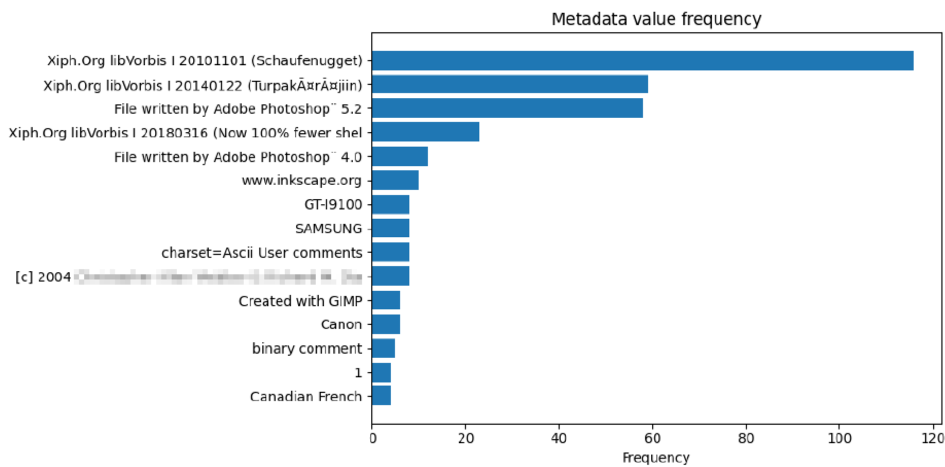
Nivel de peligro: 82.14%

Primera entrada BD: 2021-04-26 15:33:42

Última entrada BD: 2021-04-26 15:39:53

Figura 4.9: Portada del informe generado por Krptkn

5 METADATA VALUES FREQUENCY



Valor de metadatos	Frecuencia
Xiph.Org libVorbis I 20101101 (Schaufenugget)	116
Xiph.Org libVorbis I 20140122 (TurpakÃ¤rÃ¤jiin)	59
File written by Adobe Photoshop¨ 5.2	58
Xiph.Org libVorbis I 20180316 (Now 100% fewer shells)	23
File written by Adobe Photoshop¨ 4.0	12
www.inkscape.org	10
[illegible]	8
charset=Ascii User comments	8
SAMSUNG	8
GT-I9100	8

Figura 4.10: Ejemplo de un gráfico en el informe generado por Krptkn

5 Conclusiones y trabajo futuro

Debido a la enorme extensión que puede adquirir un sistema que realiza las tareas propuestas, es difícil determinar cuándo está terminado. A pesar de esto, se han cumplido tanto las tareas propuestas para un producto mínimo viable (araña, extracción de metadatos, creación de informes, etc.) como los extras (la interfaz web, el panel de control, el diseño, etc.). Por estas razones, consideramos satisfactorio el resultado final.

Aún siendo el resultado bueno, quedan múltiples áreas en las que se puede trabajar para tener un producto realmente profesional:

- **Añadir la descarga de informes a la interfaz web**. Ahora mismo se encuentra en un programa externo que aunque fácil de automatizar, tiene que ser llamado desde una terminal.
- **Añadir un sistema automatizado de despliegue**. Un pipeline en el que se pudieran integrar el testeo y despliegue a través de un repositorio Git sería enormemente útil para darle un uso profesional a la herramienta. Ahora mismo solo se cuenta con una descripción del entorno en “Nix”, un lenguaje que permite definir los requisitos para desplegar el programa.
- **Soporte para distintas bases de datos**. El poder utilizar bases de datos locales como SQLite sería de especial utilidad para un despliegue y una integración continua (CI/CD).
- **Soporte para múltiples usuarios en la interfaz web**. Gracias a Elixir/Phoenix, esto no debería ser muy costoso de desarrollar y tener varios usuarios en cada despliegue podría ser muy importante en ciertos casos de uso (por ejemplo para tener varios administradores). Actualmente, si dos usuarios se conectan a la vez, ambos tendrán permiso de escritura y lectura por lo que si no están coordinados se pisarán el trabajo el uno al otro.
- **Mejorar la extracción de metadatos**. Todavía se pueden aplicar las múltiples mejoras discutidas en la subsección 3.5.1.

Bibliografía

[1] K. Drakonakis, P. Ilia, S. Ioannidis, and J. Polakis, “Please forget where i was last summer: The privacy risks of public location (meta) data,” *arXiv preprint arXiv:1901.00897*, 2019.

[2] G. McKenzie and K. Janowicz, “Coerced geographic information: The not-so-voluntary side of user-generated geo-content,” in *Eighth international conference on geographic information science*, 2014.

[3] V. T. Rajlich and K. H. Bennett, “A staged model for the software life cycle,” *Computer*, vol. 33, no. 7, pp. 66–71, 2000.

[4] C. Alonso, E. Rando, F. Oca, and A. Guzmán, “Disclosing private information from metadata, hidden info and lost data,” *Black Hat Europe 2009 Media Archives*, 2009.

[5] W. Mincewicz, “Metadata–a silent privacy killer,” *Studia Politologiczne*, vol. 54, 2020.

[6] W3Techs, “Usage statistics of web servers,” 2020. [Online; accessed 19-June-2021].

[7] S. McEwen and R. Stern, “Fits-the file information tool set,” Georgia Institute of Technology, 2009.

[8] J. Armstrong, “The development of erlang,” in *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pp. 196–203, 1997.

[9] H. Sutter, “The free lunch is over: A fundamental turn toward concurrency in software,” *Dr. Dobb’s journal*, vol. 30, no. 3, pp. 202–210, 2005.

[10] Sintes, Tony, “Four for the ages: Answers on threads, class.forname(), multiple values, and shallow copying,” 2001. [Online; accessed 19-June-2021].

[11] S. L. Fritchie, “A study of erlang ets table implementations and performance,” in *Proceedings of the 2003 ACM SIGPLAN workshop on Erlang*, pp. 43–55, 2003.

[12] Gainetdinov, Damir, “Gun. the powerful erlang http client,” 2019. [Online; accessed 19-June-2021].

[13] Koutmos, Alex, “The state of elixir http clients,” 2020. [Online; accessed 19-June-2021].

[14] Vogels, Werner, “Amazon dynamodb – a fast and scalable nosql database service designed for internet scale applications,” 2012. [Online; accessed 19-June-2021].

Definiciones

araña Spider, araña o crawler es el nombre dado a un tipo de software que visita páginas webs de forma automática.

DNS La función principal de un servidor DNS es transformar nombres de dominio (google.com) en IPs.

endpoint Uno de los extremos de un canal de comunicación de una API. En el caso de una API web, un endpoint tiene forma de URL.

fork Llamada del sistema usada en sistemas POSIX (como Linux o BSD) que permite a un usuario crear un proceso hijo independiente del actual.

HTTP Protocolo de comunicación para la web.

IP Dirección numérica que identifica a los ordenadores que se conectan a internet.

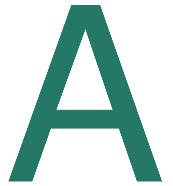
libreria Conjunto de código empaquetado para su reutilización en múltiples proyectos de software. Una librería puede permitir, por ejemplo, descargar imágenes. Al ser una librería, cualquiera que necesite descargar imágenes puede usarla en lugar de escribir de nuevo el código.

parser Software que se encarga de analizar una cadena de símbolos de acuerdo con una gramática predefinida.

registro PTR Registro que contiene información sobre el nombre asociado a una cierta IP. Esta información esta almacenada en un servidor DNSbibliographyconfig.

URL El “Uniform Resource Locator” normalmente abreviado simplemente a URL, es el identificador de la localización de un recurso web. Por ejemplo: “http://www.example.com/index.html” nos indica que en la página web “www.example.com”, podemos encontrar el archivo “index.html”.

Apéndices



Instalar Krptkn

Para la instalación de Krptkn será necesario instalar Erlang y OTP, Elixir, Libextractor y Phoenix. Esta instalación ha sido diseñada para Debian 10 buster.

Para instalar Erlang:

```
wget https://packages.erlang-solutions.com/erlang-solutions_2.0_all.deb
sudo dpkg -i erlang-solutions_2.0_all.deb
sudo apt-get update
sudo apt-get install esl-erlang erlang-dev erlang-parsetools erlang-tools
```

A continuación instalamos Elixir:

```
sudo apt-get install elixir
```

Una vez tenemos Elixir y Erlang podemos descargar Krptkn:

```
git clone https://github.com/solanav/krptkn && cd krptkn
```

Desde dentro del directorio en que hemos descargado Krptkn instalamos las dependencias de Phoenix y compilamos los assets para la web:

```
sudo apt install inotify-tools nodejs npm
cd assets
npm install
cd ..
```

Descargamos las dependencias de libextractor y compilamos el código:

```
sudo apt install build-essential libextractor-dev
mix compile
```

Y podemos ejecutar el servidor web:

```
mix phx.server
```

UAM

UNIVERSIDAD AUTONOMA

DE MADRID