UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Titulo: Swarming - Implementación para la ejecución inteligente de ataques de fuerza bruta
Autores: Gaston Traberg
Director: Lía Hebe Molinari, Paula Venosa
Codirector:
Asesor profesional:
Carrera: Licenciatura en Informática

Resumen

La Fuerza Bruta es una modalidad de ataque mediante la cual se intenta obtener acceso no autorizado realizando intentos reiterados y sistemáticos de login sobre un servicio de red, a través de la utilización de credenciales potencialmente válidas.
Este tipo de ataques se realiza muy rara vez durante una auditoría de seguridad, debido a los problemas que presentan las herramientas que hasta hoy lo implementan.
El presente trabajo propone el desarrollo de la aplicación Swarming, la cual apunta a contemplar en mejor medida tecnologías como HTTP, realizar un aprovechamiento de la información existente en el contexto de la auditoría e integran todos los datos recabados para maximizar los resultados obtenidos.

Palabras Claves

Fuerza Bruta, Cracking, Crawling, Python, JSON, HTTP, Asincronismo, Auditoría de Redes

Conclusiones

Se realizó la implementación de una aplicación capaz de llevar a cabo ataques de Fuerza Bruta, para los cuales se aprovecha la información que el contexto ofrece. A partir de los análisis realizados sobre otras aplicaciones, se pudieron reconocer problemas y carencias que fueron tomadas para ser solucionadas e incorporadas a la serie de funcionalidades que la aplicación posee.

Trabajos Realizados

- *Análisis de aplicaciones de Fuerza Bruta*
  - *Ventajas y Desventajas*
- *Desarrollo del núcleo de la aplicación.*
  - *Gestión dinámica del número de procesos*
  - *Arquitectura modularizada*
  - *Trabajo concentrado en la Base de Datos*
- *Implementación de la unidad HTTP*
  - *Detección de aplicaciones Web*

Trabajos Futuros

Algunas de las características que se presentan como trabajo a futuro, existen desde el comienzo de la aplicación mientras que otras fueron sugeridas durante el desarrollo de la misma.

1. *Soporte de otros protocolos*
2. *Reconocimiento de Dispositivos de Red*
3. *Reconocimiento de Aplicaciones Web*
4. *Generación de diccionarios*
5. *Incorporación de Scrapers*
6. *Agregado de Claves Públicas como credenciales*

Fecha de la presentación: Mayo/2015

Universidad Nacional de La Plata

Facultad de Informática

Swarming
Implementación para la ejecución inteligente de ataques de fuerza bruta

Gaston Traberg

Directoras

Lía Hebe Molinari

Paula Venosa

Índice

1. Introducción

La ***Fuerza Bruta*** [25] es una modalidad de ataque mediante la cual se intenta obtener acceso no autorizado realizando intentos reiterados y sistemáticos de *login* sobre un servicio de red, a través de la utilización de credenciales potencialmente válidas.
Las credenciales utilizadas durante la realización de este tipo de ataques, suelen provenir de conjuntos de credenciales que existen por defecto para ciertos dispositivos o aplicaciones, de credenciales comúnmente utilizadas por los usuarios o de credenciales que fueron comprometidas previamente a través de otros tipos de ataque [9]. En el contexto de las auditorías de seguridad, mediante la fuerza bruta se detectan todos o la mayoría de los casos en los cuales las credenciales utilizadas para el acceso a un servicio presentan alguna debilidad, las cuales potencialmente permitirían a un atacante acceder de forma no autorizada, al sistema de información que está siendo auditado.

En base a lo anterior es que la presente tesis propone el desarrollo de la aplicación ***Swarming*** [24], la cual surge como una herramienta que encara de forma moderna, la implementación de los ataques de fuerza bruta, buscando en el proceso contemplar vacíos dejados por otros programas e incorporando a su vez tecnologías para las que existen escasos o ningún soporte.

1.1. Motivación

La utilización de ataques de fuerza bruta en el ámbito de las auditorías de sistemas de información, habitualmente presentan dos situaciones adversas:

- En cuanto a la **modalidad**, se realizan de una manera que desaprovecha mucha de la información expuesta por los servicios que la red posee, información que revela mucho sobre cuales son las conveniencias en la realización del ataque.

- En cuanto a la **dificultad**, presentan ciertas dificultades al momento de configurar el mismo, lo cual sucede en servicios que ofrecen recursos de gran complejidad en su utilización, como es el caso de *HTTP*.

Como consecuencia de esto, la fuerza bruta durante una auditoría es poco practicada o realizada sin aprovechar su potencial, ya sea por cuestiones de tiempo o por cuestiones de complejidad, dejando así fuera del análisis importantes vulnerabilidades en los servicios que son objeto de auditoría.

En la actualidad existen tres herramientas que sobresalen al momento de hablar de fuerza bruta sobre redes: ***Hydra*** [22], ***Medusa*** [13] y ***Ncrack*** [16], de las cuales se expondrán ventajas y limitaciones que constituyen la base del desarrollo planteado.

Hydra es una de las aplicaciones de fuerza bruta más antiguas que existen en el contexto de la seguridad. Desarrollada por el Alemán *van Hauser* en 2001, *Hydra* es la primera opción para muchos al momento de hablar de herramientas de fuerza bruta sobre redes. Esta aplicación es activamente mantenida, habiendo sido actualizada por última vez el 12 Mayo del 2014. Entre las ventajas que se pueden destacar de *Hydra* se encuentran el gran número de protocolos que soporta como así también el número de plataformas sobre las que se indica que funciona. Por otro lado *Hydra* posee algunas características que no la hacen muy popular al momento de utilizar o de extender en funcionalidad, entre las que se destacan:

- Suele suceder que la aplicación se corrompe durante su ejecución u omite resultados producto de los errores que se generan[1]. Si bien algunos de estos problemas se fueron solucionando con el tiempo, otros aún persisten.

- Al estar implementada en *C* y con un código muy desprolijo, es difícil tener un entendimiento claro de lo que ocurre dentro de la aplicación, dificultando mucho la realización de extensiones.

[1] http://security.stackexchange.com/questions/41913/linux-hydra-issue-random-correct-password-success

Bajo estas mismas observaciones y más, es que miembros del grupo *Foofus.net* realizaron el desarrollo de *Medusa*, subsanando de esta manera la mayoría de los problemas que *Hydra* presenta. *Medusa* posee un código fuente mucho más organizado y prolijo, además de otras características que lo hacen una interesante opción, pero a diferencia de *Hydra*, *Medusa* tiene soporte sobre un número de protocolos algo menor.

Como tercera opción existe *nCrack* que es un proyecto nacido en el *"Google Summer of Code"* de 2009 pero que aún no es considerado como un programa terminado ni por los propios autores. No es una opción muy competitiva contra las dos primeras, en principio porque no ofrece nada nuevo y en segundo lugar porque se desempeña de muy mala manera en todos los aspectos en los que se los puede comparar con respecto a *Hydra* y *Medusa*.

A su favor podemos decir que estas herramientas demostraron en conjunto poseer un buen número de protocolos en su soporte, buena velocidad en la realización del ataque y una relativa simplicidad en su uso [1][2]. Sin embargo, más allá de estas características, lo que hace que estas herramientas sean la primer opción al momento de hablar de fuerza bruta sobre redes, es la inexistencia de alternativas.
Realizando simples casos de uso de estas aplicaciones sobre redes y servicios comúnmente encontrados hoy en día, queda rápidamente en evidencia los muchos problemas que éstas presentan a lo largo de su utilización, entre los que podemos listar:

- De existir un *Firewall* o un *IPS* como ***iptables*** [23] o ***fail2ban*** [10], los servicios solo se pueden acceder de a intervalos de tiempo regulares o un cierto número de veces por cantidad de tiempo. Esto genera situaciones que no siempre son bien manejadas por las herramientas, provocando que estas ignoren algunas credenciales o que las mismas sean rechazadas, dejando en desconocimiento si dichas credenciales son válidas o no.
- Existen protocolos (*HTTP* por ejemplo) que pueden presentar una

gran complejidad en los procesos de *login*, desde *Cookies* hasta ***Tokens CSRF*** [26], además de otras. Si bien algunas de estas características están contempladas por algunas de las herramientas antes mencionadas, lo están de manera parcial o simplemente no lo están, haciendo que *HTTP* sea un ejemplo de protocolo muy excluido, aunque sea este extremadamente utilizado hoy en día.

- Siguiendo con *HTTP*, a diferencia de otros servicios, en la actualidad este tiene un rol extremadamente destacado dentro de una red, en muchos de los aspectos posibles. Se ha vuelto un servicio extremadamente complejo, y aparejado a esa complejidad creció el número de potenciales problemas de seguridad existentes. Todas las herramientas mencionadas se limitan a auditar solo lo explícitamente indicado por su ejecutor, generando que la auditoría de un servicio con estas características sea una tarea imposible, debido a la enorme cantidad de contenido a analizar antes de poder generar una ejecución válida de las mismas.

- La velocidad con la que se realiza un número de intentos de *login* no es relevante al momento de un ataque. Este objetivo parece ser el buscado en principio por las aplicaciones listadas, no siendo el caso de la propuesta. Si algo deja como experiencia la auditoría de redes y sus servicios es que en la mayoría de los casos, las credenciales que uno busca están al alcance de la mano o suelen ser de fácil inferencia, quedando la velocidad de ataque rezagada a un segundo plano ya que el problema radica en la obtención de las credenciales y no en la velocidad con la que se las utiliza.

Así como las anteriormente expuestas, existen infinidad de características más que hacen de estas tres herramientas algo poco práctico al momento de hablar de procesos de fuerza bruta, más cuando se trata de auditorías de redes grandes.

1.2. Objetivos

Partiendo de todos los problemas anteriormente analizados como así también de ideas propias y sugeridas, es que se elaboró una lista de requerimientos que sirvieron como base para el diseño de la aplicación solución, entre los cuales encontramos:

- Realizar procesos de fuerza bruta (obviamente), también conocidos como ***Cracking***.
- Realizar un recorrido automático del servicio auditado, trabajo conocido como ***Crawling***.
- Crecer dinámicamente en el número de procesos que realizan los trabajos.
- Reconocer aplicaciones, dispositivos y cualquier característica que signifique un ahorro de trabajo en la auditoría.
- Iniciar de manera automática procesos de *Crawling* sobre recursos a los que se tuvo acceso, repitiendo el ciclo todas las veces que sea posible.
- Brindar una interfaz de usuario simple, basada en una *API* que permita la interacción de otras herramienta con la aplicación, de manera sencilla.
- Seguir un modelo en el cual cada módulo encargado de procesar un protocolo en particular, funcione de la manera más aislada posible, permitiendo la carga y descarga de los mismos además de una fácil y flexible implementación.

De esta manera se establece un escenario que a futuro facilitará la tarea de agregar nuevas funciones, como protocolos a soportar y otras.
Si bien es imprescindible la implementación de la mayor cantidad de protocolos posible para un aprovechamiento máximo de la capacidad de la aplicación, en esta primer instancia del desarrollo se apuntó a *HTTP* como primer protocolo a contemplar. Esta decisión se justifica porque *HTTP* es uno de

los protocolos con peor soporte por parte de las aplicaciones antes mencionadas, además de ser el protocolo que más problemas puede presentar debido a su complejidad y al gran número de diferentes situaciones existentes en su utilización.

1.3. Solución y Desarrollo

El ***Swarming*** nace como una aplicación cuyo objetivo principal es el desarrollar automáticamente todas las actividades posibles en torno a los procesos de fuerza bruta, buscando con esto aprovechar al máximo toda la información obtenida del contexto, sin que se requiera intervención humana.

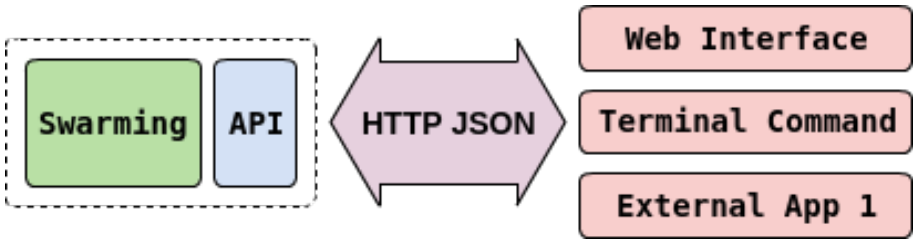

Figura 1: Modelo del *Swarming*

Para lograr esto, se siguió un modelo que presenta una única interfaz a través de la cual no solo el usuario controla la aplicación, sino que también permite que otras aplicaciones puedan acceder a los datos y controles, logrando de esta manera una fácil extensión en el uso del *Swarming* por otros programas. La *figura 1* muestra una representación general de como el único medio de acceso, ya sea desde una interfaz de usuario o desde una aplicación externa, es a través de la *API* del programa, la cual es accedida a través del envío de mensajes en formato ***JSON*** [27].

Para la implementación, si bien existe un espectro interesante de lenguajes, se optó por la utilización del lenguaje de programación ***Python*** [12] en su versión 3, el cual presenta muchas cualidades dentro de las que podemos destacar:

- *Lenguaje Interpretado.* Esto al momento de desarrollar representa una ventaja enorme, ya que las pruebas pueden realizarse inmediatamente

después de que los cambios fueron persistidos. Si bien el hecho de que sea un lenguaje interpretado no ayuda demasiado en lo relacionado a *performance*, existe en *Python* la posibilidad de realizar módulos en algún lenguaje compilado (*C++* por ejemplo), los cuales pueden ser importados y utilizados desde *Python*, característica que será aprovechada para dar soporte a protocolos como ***MySQL***, ***SSH***, y otros.

- *Soporte nativo de Unicode.* Hoy en día, la capacidad de soportar la codificación *Unicode* en las comunicaciones es más un requerimiento que una opción. Es por esto que se optó por *Python 3* para la implementación del *Swarming*, ya que *Python* en sus versiones anteriores, como la 2.7, no soporta *Unicode* de manera simple.

- *Gran Número de Librerías.* Algo que siempre caracterizó a *Python* es el hecho de que posee un espectro muy interesante de librerías para muchas actividades. Desde *ORMs* para el modelado de bases de datos siguiendo una orientación a objetos, pasando por *Frameworks* para el desarrollo de interfaces *Web* y utilización de *JSON*, hasta módulos para el manejo de protocolos como *HTTP* de forma muy simple pero no por eso menos potente.

Si bien al momento de escoger el lenguaje de programación, todo lo anterior fue decisivo, también existieron otras cuestiones no tan técnicas, como la facilidad que el autor tenía sobre el lenguaje o el hecho de que a la fecha *Python* es uno de los lenguajes más populares para el desarrollo de aplicaciones, como así también ampliamente soportado en gran número de plataformas.

Todo lo antes mencionado pretende aclarar a muy grandes rasgos, algunas de las motivaciones para la selección de determinadas tecnologías, además de dar una perspectiva muy abstracta del modelo de la aplicación. En los capítulos siguientes se expondrá de manera más detallada el uso del *Swarming* para luego seguir hasta el final del documento, con una inmersión paulatinamente más profunda en su funcionamiento.

2. Funcionamiento de la aplicación

A continuación se expone y explica un caso de uso a través del cual el *Swarming* se muestra superando situaciones que, con las herramientas existentes sólo podrían llevarse a cabo mediante varias etapas, debiendo estas ser gestionadas por una persona. Con esto, se pretende dar una primera visión de la aplicación en cuanto a su funcionamiento y sus capacidades, aspectos sobre los que se basan las explicaciones de los capítulos siguientes.

2.1. Contexto

Para analizar el funcionamiento del *Swarming*, supongamos el siguiente caso de uso.

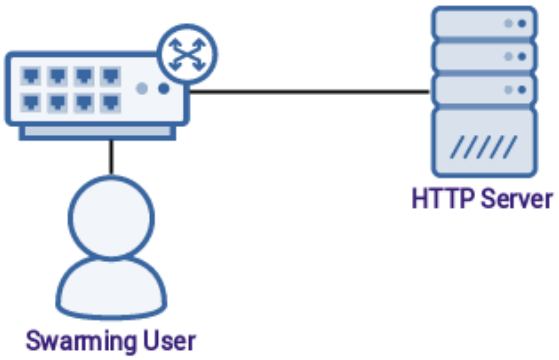

Figura 2: Topología del Caso de Uso.

El usuario se encuentra frente a una topología constituida por un *Switch* y un *Servidor*, el cual brinda un servicio *HTTP* del que no se tiene información. Nuestro objetivo es lograr el mayor nivel de acceso posible sobre dicho servicio, utilizando como único elemento adicional, un conjunto de credenciales que poseemos previamente.

2.2. Ejecución

Para iniciar la auditoría del servicio, el usuario a través de la ventana de diálogo ***New Task***, realizará el agregado de una nueva tarea la cual se define mediante un recurso en formato *URL* y por su estado inicial, concretando la creación de la tarea al presionar el botón *Add*. En la *figura 3* se observa la ventana de diálogo donde se distingue en rojo la entrada que espera la *URL* del recurso a procesar, y en azul la entrada que especifica el estado inicial de la tarea.

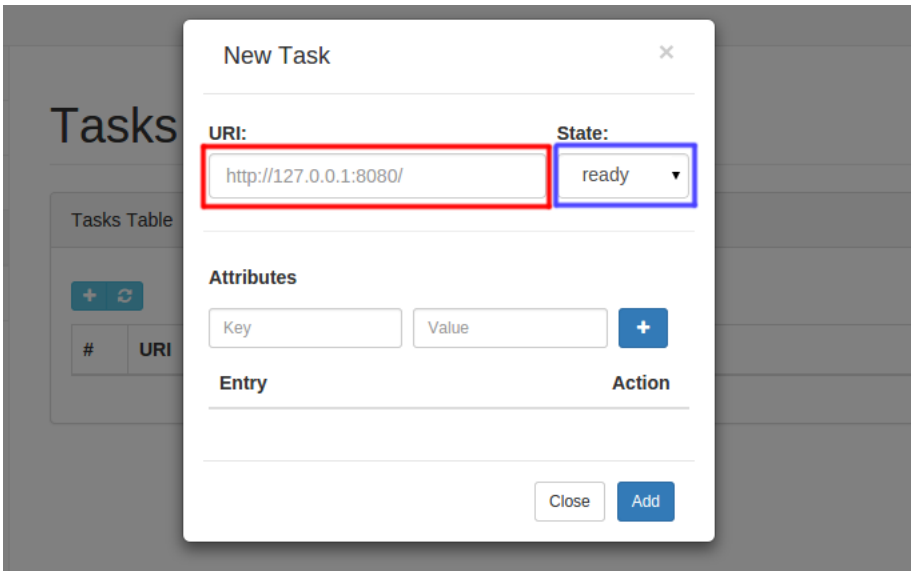

Figura 3: Diálogo para agregar Tareas

Una vez han sido completados estos pasos y agregada la tarea, la misma se podrá ver incluida en la tabla de tareas presentada por el panel ***Task***, el cual lista todas las tareas existentes dentro de la aplicación, tal y como se observa en la *figura 4*.

A partir de los valores indicados en la tabla bajo los títulos ***Stage*** (etapa) y ***State*** (estado), podemos definir la situación en la cual una tarea en particular se encuentra. El valor *initial* en el campo *Stage* es el valor por defecto para todas las tareas recién creadas y representa la etapa en la cual

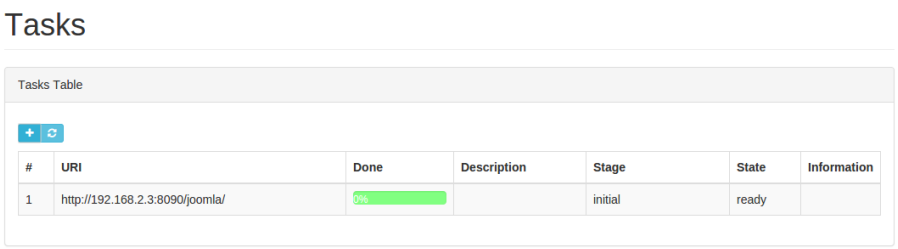

Figura 4: Panel *Task*. Etapa *initial*

la ***Unidad*** responsable del procesamiento de la tarea, controla que la misma esté completa y el servicio que referencia, disponible, para luego efectuar la transición de la misma a su etapa siguiente.

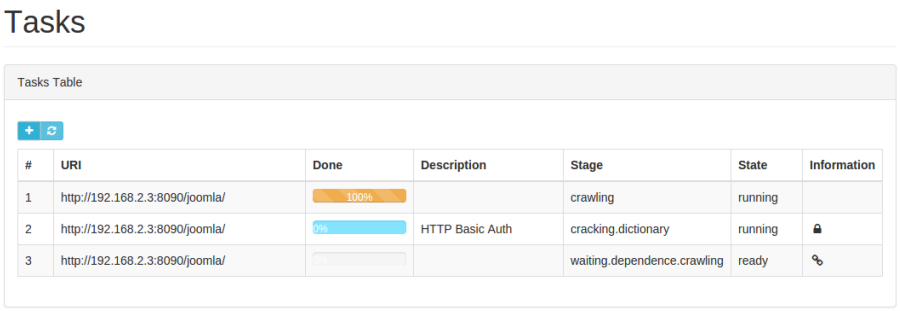

Figura 5: Etapa *Crawling* y creación de nuevas tareas.

Siguiendo con el ejemplo, una vez la tarea se puso en marcha (*State* cambió a *running*) y los controles sobre la misma fueron correctos, se realiza la transición a su próxima etapa, la cual para esta tarea en particular corresponde ***Crawling***. Esto se manifiesta, como muestra la *figura 5*, con el cambio del valor *Stage*, de *initial* a *crawling* y con el cambio de color de la barra bajo el título ***Done***, de verde a naranja, dejando en claro el proceso que la tarea está llevando a cabo en el momento. Todo los procesos comentados de transición entre etapas como así también entre estados, son procesos que se llevan a cabo de forma automática, siendo la creación de la tarea el punto en el que se inicia el tratamiento de la misma.
Por otro lado, la *figura 5* presenta dos nuevas tareas en etapas diferen-

tes a la primera, una en la etapa *cracking.dictionary* y otra en la etapa *waiting.dependence.crawling*. Para comprender mejor el significado de estas nuevas dos etapas, se presenta en la *figura 6* el esquema que representa el conocimiento que el *Swarming* tiene hasta el momento, sobre el contexto de la auditoría.
Habiendo nuestra tarea original derivado en un proceso de *Crawling* sobre el servidor *HTTP*, el *Swarming* pudo detectar durante su recorrido del árbol de directorios, la existencia de una restricción de acceso por medio de una autenticación ***HTTP Basic***, la cual se observa en la *figura 5* como la entrada #2 de la tabla y en la *figura 6*, como el punto nro 2. Esta nueva tarea que fue generada por el *Crawler*, se encuentra en la etapa *cracking.dictionary* desde un comienzo, indicando a través de esta, que se encuentra realizando un proceso de *Cracking* con ***Usuarios*** y ***Passwords*** (*.dictionary*) como credenciales.

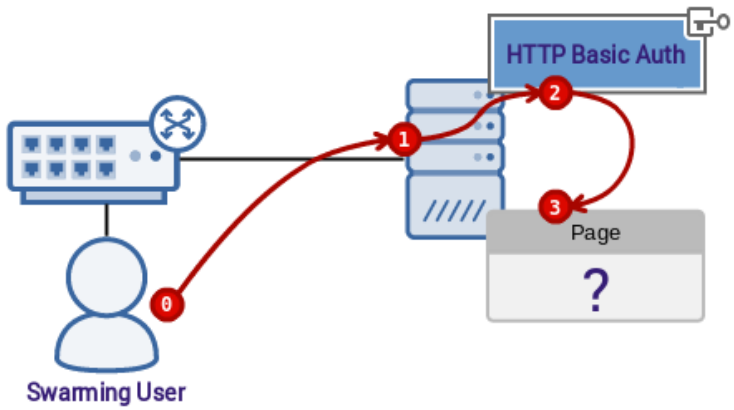

Figura 6: Conocimiento luego del *Crawling*.

En la *figura 6*, el paso indicado con el número 3, representa el elemento oculto detrás de la autenticación *HTTP Basic*, representado en la tabla de la *figura 5* por la entrada con el mismo número. La etapa de esta 3er tarea es *waiting.dependence.crawling*, lo cual significa que el *Swarming* debe esperar por la tarea de la que depende (*waiting.dependence*) antes de pa-

sar a su nueva etapa que es *Crawling* (*.crawling*). En la entrada #2 de la *figura 5* y en la #3 de la misma figura, se pueden observar dos símbolos: Un candado cerrado y una cadena. El candado nos indica que esa tarea se encuentra bloqueada, sin haber podido aún conseguir credenciales válidas. Por otro lado, la cadena nos indica que esa tarea depende de otra, o lo que es lo mismo, requiere que otra tarea logre resultados antes de ella poder iniciar su trabajo. En nuestro ejemplo, la tarea #3 depende de la tarea #2, ya que sin las credenciales que la #2 genera, la #3 no podrá avanzar.

Una vez la tarea #2 logra algún resultado, su condición de bloqueo cambia automáticamente y todas las tareas dependientes de ellas (la #3 en nuestro caso), dejan de esperar y pasan a la siguiente etapa que les corresponde. En la *figura 7* se observan estos cambios en la aplicación.

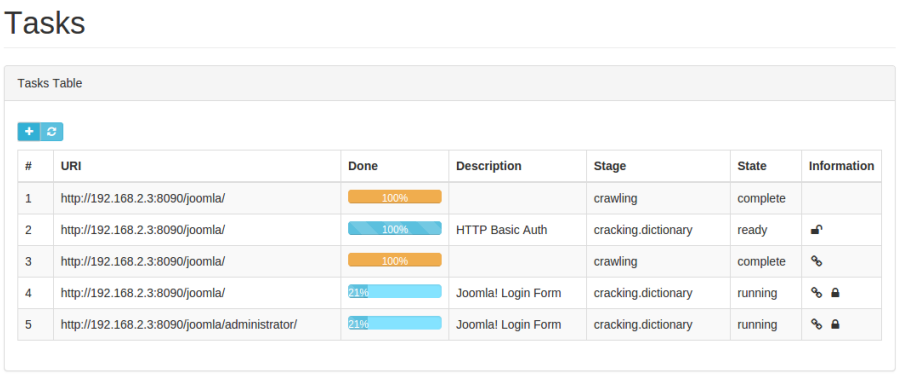

Figura 7: Desbloqueo de la tarea en espera.

Habiendo la tarea #2 obtenido resultados y la tarea #3 cambiado de etapa a *crawling*, esta última no se ve ya más limitada por la autenticación *HTTP Basic* e inicia su proceso de descubrir que hay más allá. Durante este nuevo proceso de *Crawling* es que se identifica la existencia de una aplicación ***Joomla!***, la cual el *Swarming* no sólo reconoce sino que sabe como explotar, generando nuevas tareas de *Cracking* sobre los formularios que *Joomla!* posee. Una vez comprobada la existencia de estos formularios, la tarea #3 dispara dos nuevas tareas, la #4 y la #5 las cuales al igual que

la tarea #3, dependen de la #2 (necesitan las credenciales de *HTTP Basic* para avanzar), lo que ya no representa ningún obstáculo debido a que las mismas ya se conocen. Con estas últimas dos tareas, finaliza el proceso de descubrimiento sobre el servicio *HTTP*, quedando las tres tareas de *Cracking* a la expectativa de nuevos diccionarios los cuales utilizar.

2.3. Resultado

Se pudo observar a través del ejemplo presentado, el grado de automatización que el *Swarming* ofrece, no solo en el descubrimiento de recursos a atacar, sino también en la generación de nuevas tareas a desarrollar sobre el servicio, con el fin de lograr el máximo nivel de acceso posible sobre el mismo.

Cada etapa analizada a lo largo de la ejecución del *Swarming*, de haberse ejecutado a través de las aplicaciones como *Hydra* o *Medusa*, hubiese significado la repetida ejecución y análisis de los resultados de las mismas de forma manual, desperdiciando así todo el tiempo que la buena performance de estas herramientas podría llegar a ofrecer.

3. Modelo general

En una primera aproximación, la idea general detrás del *Swarming* es la de una aplicación que corre como un proceso desatendido y cuya única forma de control es a través de mensajes enviados a una interfaz. Esto le permite a herramientas externas el acceso a la información sin demasiados problemas, facilitando la implementación de las mismas y de la aplicación. Internamente el *Swarming* se encuentra constituido por ***Unidades***, las cuales existen como componentes que podrían ejecutarse de forma autónoma en procesos diferentes, motivo por el cual el *Swarming* implementa la comunicación entre ellas mediante *Pasaje de Mensajes*.
La utilización de *Procesos* como mecanismo de paralelización de las tareas surge del análisis de dos posibles políticas:

- Paralelizar utilizando ***Procesos***, como se hizo.
- Paralelizar utilizando ***Thread***.

Si bien la utilización de *Threads* presenta una ventaja clara que es el espacio de direcciones compartido entre ellos (lo que facilita mucho el intercambio de información y sincronización entre las componentes), no ofrece garantías de aprovechamiento real de los múltiples núcleos que un sistema puede tener. Es por esto que, para asegurar la concurrencia sin importar la plataforma sobre la que se realice la ejecución, se optó por la utilización de procesos como mecanismo de multitarea, delegando la utilización de *Thread* a las unidades que los requieran.

Para la *encapsulación de la información* que se enviará a través del pasaje de mensajes entre las Unidades, *Python* ofrece de forma nativa la estructura de datos diccionario (tipo de dato *dict*) que sigue el mismo formato que la notación de objetos en *JavaScript* conocida como *JSON* (*JavaScript Object Notation*). Esta notación es soportada en *Python* a partir del módulo llamado *json* el cual permite la conversión de estructuras *dict* a cadenas *JSON* y viceversa. De manera paralela, analizando la complejidad temporal

de las funciones que se pueden aplicar sobre los diccionarios [7], queda en evidencia otro de los motivos por los cuales esta estructura de datos es la más adecuada para la implementación de la encapsulación de datos.

De la idea de utilizar *Procesos* y *Pasaje de Mensajes* para la ejecución y comunicación de las componentes del *Swarming*, surge la decisión de basar la implementación en el módulo *multiprocessing* de *Python*, el cual posee mecanismos de ejecución de funciones/métodos en el contexto de un nuevo *Proceso*, aportando a su vez un medio de comunicación entre éstos conocido como ***Queue***, el cual permite el *pasaje de mensajes* de un proceso a otro de forma bidireccional.

En base a todo lo anterior se puede establecer un primer diagrama que expone de modo general la arquitectura que el *Swarming* presenta, tanto en la manera de escalar hacia la multitarea, como en la comunicación de sus componentes y con ella, como se observa en la *figura 8*.

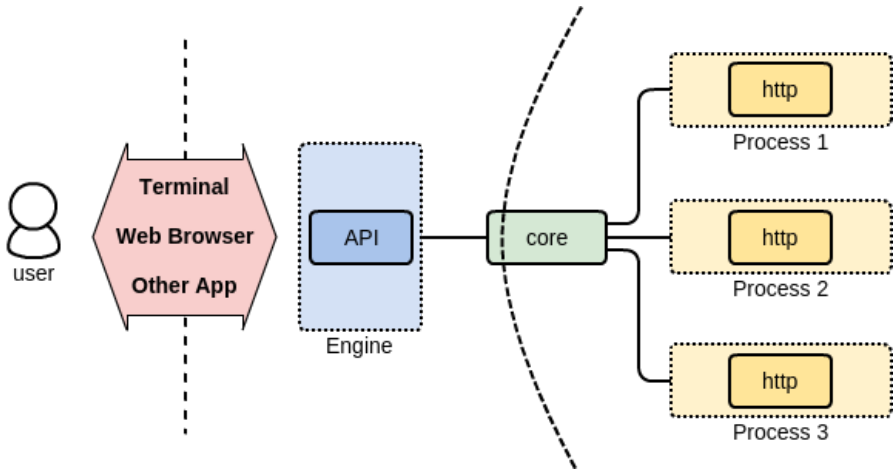

Figura 8: *Swarming* - Modelo Interno Abstracto

En la *figura 8* podemos observar delimitadas tres zonas por las líneas punteadas. Partiendo desde la izquierda, el usuario a través del navegador, de una terminal o de otra aplicación, controla el *Swarming*, el cual ofrece

una única interfaz (La *API*). La *Unidad* ***Engine***, la cual contiene la *API*, es una unidad especial de la aplicación, de la cual solo existe una instancia de ejecución en todo el programa. Las unidades que implementen soporte para los diferentes protocolos (*HTTP* en este caso), existen comúnmente como instancias propias de cada proceso dedicado al consumo de tareas (*Process N* en la figura), pudiendo existir muchos de estos a lo largo de una ejecución. Por último, la *Unidad* ***Core*** se encarga de intercomunicar todas las unidades existentes en la aplicación, además de llevar a cabo algunas tareas propias como unidad.

3.1. Componentes

Para comenzar a conocer en detalle los aspectos internos del funcionamiento del *Swarming*, se presenta en la *figura 9* un diagrama el cual plasma la arquitectura del mismo con un muy reducido grado de abstracción, la cual servirá de base para el entendimiento de cada uno de los componentes y mecanismos existentes dentro de la aplicación.

Analizando a primera vista la imagen, podemos distinguir que la misma se encuentra compuesta por cuatro cuadros de los cuales tres son de un mismo color. Cada uno de estos cuadros representa una *Capa* o ***Layer*** dentro de la aplicación, las cuales constituyen el mecanismo que utiliza el *Swarming* para la implementación de la multitarea. Las *Layers*, individualmente, no son otra cosa que un proceso el cual define su rol en función de quien lo controla. En la *figura 9*, las *Layers* 1, 2 y 3 (las coloreadas en amarillo), se encuentran controladas por las ***Unidades Executor*** las cuales son las responsables de recibir los mensajes destinados a la capa y realizar la ejecución de los mismos sobre las unidades que estos tengan como destino. La *Layer* 0 es la primera *Layer* en ser creada por el *Swarming* y se encuentra controlada por la ***Unidad Engine***, la cual se encarga de tres tareas principales

- La generación del trabajo a realizar a partir del conocimiento existente, llevado a cabo por la componente ***Tasker***.

- El manejo del conocimiento que la aplicación obtiene a lo largo de la ejecución, gestionado por el componente ***Knowledge***.
- La presentación de una interfaz al usuario, que permite el control y acceso a la información, aspecto manejado por la componente ***WebUI***.

Cada uno de estos tres componentes, posee una instancia de la clase ***ORM*** la cual es responsable de generar la abstracción del modelo de datos a objetos, además de controlar el acceso a la base de datos ***SQLite3***.

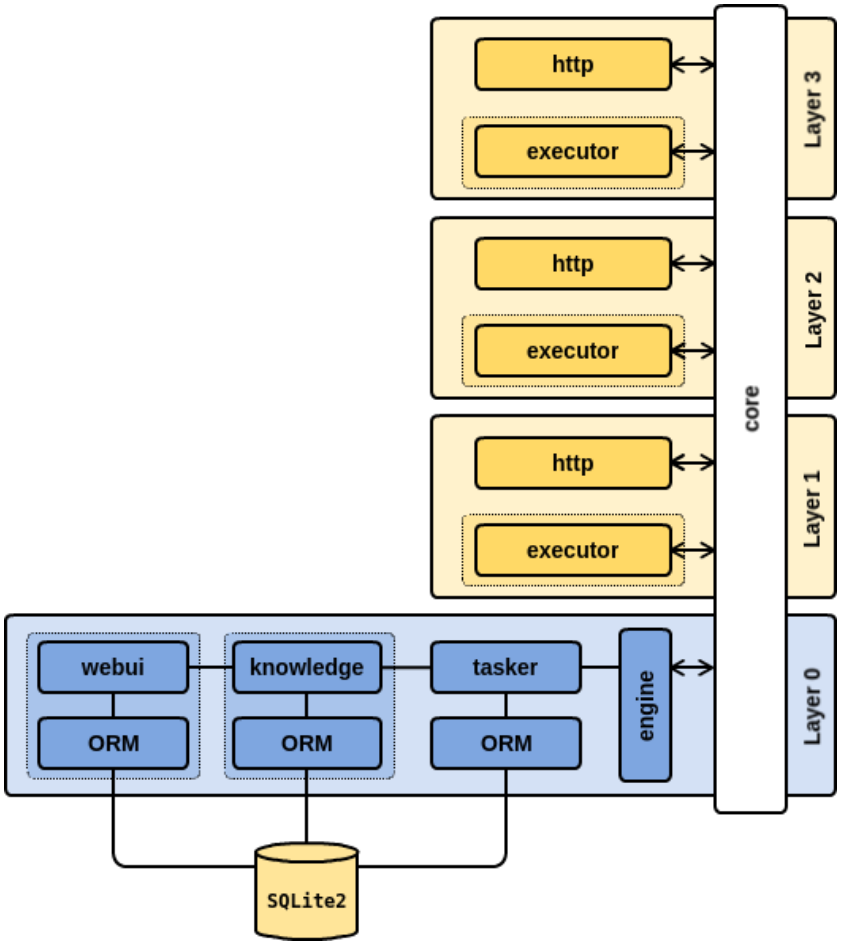

Figura 9: *Swarming* - Modelo Interno

Los cuadros punteados que rodean a algunas componentes y unidades

en las diferentes capas, representan *threads* creados por la unidad que se encuentra contenida dentro de este. Por ejemplo, todos los *Executors* no sólo administran el proceso perteneciente a la *Layer* en la que se encuentran, sino que además poseen un *thread* extra cuya función es atender los mensajes recibidos, sin que esto interrumpa su funcionamiento.
Algo que se puede notar en el diagrama es el hecho de que la unidad ***Core*** no se encuentra contenido en ningún proceso, lo cual es así porque *Core* es la única unidad que es común a todas y que no posee una ejecución propia, funcionando entre otras formas, como medio de comunicación entre las diferentes capas.
Por último, las unidades ***HTTP***, como así también cualquier otra unidad que gestione un protocolo, son unidades que existen como instancias sólo dentro de las capas superiores (por encima de la *Layer 0*), motivo por el cual se las conoce dentro del *Swarming* como ***Light Units*** (*Unidades Livianas*). Un concepto ligado a esto último es el de las ***Heavy Units*** (*Unidades Pesadas*) las cuales poseen su propia *Layer* y existen en la *Layer 0* o por debajo de la misma. El único ejemplo de este tipo de unidades en la presente versión del *Swarming* es la *Unidad Engine*, aunque se pretenden hacer futuras expansiones que agregues mas de estas unidades.

Existen otros componentes que integran algunas de las diferentes unidades mencionadas que no fueron expuestos en la presentación, los cuales serán abordados más adelante en los apartados correspondientes a cada una de ellas.

4. Comunicación

La comunicación entre las diferentes unidades del *Swarming* ocurre mediante el *Pasaje de Mensajes* el cual se encuentra implementado a partir de la clase *Queue* del módulo *multiprocessing* y del tipo de encapsulamiento de datos *dict*, ambos elementos nativos de *Python*. La comunicación es llevada a cabo por medio del envío de diccionarios de un extremo de la cola de mensajes al otro, a través de la conversión de los mismos a cadenas con formato *JSON*. Este formato garantiza el fácil intercambio de información con otros procesos que puedan existir, permitiendo al *Swarming* concebir una política de fácil incorporación de componentes a futuro.

4.1. Asincronismo

Los mensajes *asincrónicos* constituyen la política de comunicación por defecto seguida por el *Swarming*. Al realizarse el envío de un mensaje dentro de la aplicación, se desconoce el tiempo que pueda su receptor requerir para su procesamiento, ya que la variedad de tareas posibles a realizarse va desde las más simples y rápidas, como son los accesos a la base de datos, hasta las más complejas y lentas, como es el ataque a un recurso remoto. Por otro lado las diferentes unidades que componen la aplicación, operan de manera muy aislada las unas de las otras, siendo necesario en muy raros casos, conocer la respuesta de un mensaje inmediatamente después de su envío. Todos estos aspectos y alguno otros, son el motivo por el cual el *asincronismo* se tomó como política a seguir para el intercambio de información dentro de la aplicación.

Para poder distinguir cuál de las respuestas recibidas corresponde a un mensaje determinado, en el *Swarming* se implementó el concepto de ***Channel*** (*Canal*). Al momento de enviarse un mensaje que requiere un cambio de contexto (pasar de un proceso a otro, por ejemplo), la unidad que lo recibe generara un numero de canal único que identificará la comunicación, devolviendo dicho valor a quien realizó el envío en un principio. Una vez

la unidad que recibió el mensaje finaliza con la tarea de procesarlo, genera un nuevo mensaje el cual contiene la respuesta al mensaje original como así también, el canal que se entregó en respuesta al mismo. En la unidad receptora (aquella que generó el mensaje en un comienzo), los mensajes recibidos como respuesta son almacenados e indexados a través del número de canal con el cual arribaron, para luego poder ser accedidos en el momento en que se los requiera.
Para entender mejor este mecanismo, debemos analizar las diferentes partes que lo constituyen, siendo los ***mensajes*** el primer elemento que debemos entender.

4.2. Los Mensajes

Dentro del *Swarming* todas las unidades poseen una única interfaz para la transmisión de mensajes de unas a otras, la cual se compone por un único método llamado ***dispatch(message)***. Codificada en el mensaje recibido como parámetro, se encuentra la información necesaria para iniciar la ejecución del comando que el mensaje representa, como así también lo necesario para dar respuesta al mismo. Todos los mensajes que circulan a lo largo del *Swarming* contienen la misma información, estando estos constituidos de la siguiente manera

- ***dst:*** Indica la unidad a la cual va dirigido el mensaje, a través del nombre de la misma.
- ***src:*** Contiene el nombre de la unidad que dio origen al mensaje, siendo esta a quien debe ir dirigida la respuesta que se genere en base al mismo.
- ***cmd:*** El comando a ejecutar sobre la unidad destino. Este campo a través de la cadena de caracteres que referencia, indica que comando de los registrados por la unidad destino, se desea ejecutar.
- ***params:*** Es un diccionario anidado dentro del mensaje, que contiene todos los valores requeridos para la ejecución del comando indicado

por *cmd*. El único factor a tener en cuenta en cuanto a los valores indexados por este diccionario, es que los mismos deben poder ser convertidos a *JSON*, de otro modo su transmisión a otras unidades sería imposible.

- ***heavy:*** Este campo existe para orientar a algunas unidades sobre cómo procesar el mensaje que lo contiene. En general este campo indica si el comando del mensaje requerirá de mucho tiempo para su procesamiento (valor a *True*) o si el procesamiento del mismo será prácticamente inmediato (valor a *False*). Si ocurre que este campo se omite, se asume que su valor es *False*.

- ***channel:*** Indica el canal a través del cual se espera la respuesta resultado del mensaje enviado. Este campo está constituido por un número entero positivo de *32bits* generado aleatoriamente, cuyo único objetivo es distinguir unívocamente el envío y recepción de un mensaje particular entre dos unidades.

- ***layer:*** El número entero que este campo contiene, indica la capa a la cual el mensaje corresponde. Este valor es agregado al mensaje de manera automática por la unidad *Core* durante el envío del mismo y utilizado principalmente por la misma unidad para los procesos de enrutamiento.

La dinámica de interacción entre mensajes involucra la ejecución habitual de distintas rutinas para llevar a cabo la comunicación en forma adecuada. Es por este motivo que el *Swarming* incluye el módulo conocido como ***Message*** para la manipulación de mensajes, el cual permite realizar la asignación de un canal, controlar los valores y desarrollar una respuesta en base al mensaje, sistematizando la interacción.

5. Las Unidades

Las ***Unidades*** son aquellos elementos que constituyen al *Swarming* y que a través del pasaje de mensajes con otras *Unidades*, desarrollan un rol definido dentro de la aplicación. Las *Unidades* poseen como rasgo distintivo la implementación de la interfaz *dispatch()* la cual sin embargo, no es característica suficiente para la definición de una unidad como tal. Todos los elementos y comportamientos que un módulo debe poseer para ser considerado una unidad, son proporcionados por el *Swarming* a través de la clase abstracta ***Unit***, de la cual estos deberán heredar para establecer las bases de su funcionamiento como unidades.

5.1. La clase abstracta Unit

La clase abstracta *Unit* existe como contenedor de todas las características que hacen de un módulo que hereda de ella, una unidad del *Swarming*. Si bien existen muchos comportamientos de los cuales *Unit* solo se encarga de indicar si el mismo se encuentra presente o no, en los módulos que la incluyen, existen otros de los cuales *Unit* realiza la implementación casi total, dentro de los que tenemos:

- Definición de la variable ***protocols*** de la unidad, la cual provee a través de un diccionario de los nombres de los protocolos soportados por la unidad, además del puerto por defecto de cada uno de ellos. Por ejemplo, en el caso de la unidad *HTTP*, esta variable contiene el valor **{"http":80, "https":443}**

- **add_cmd_handler()** es el método utilizado por las unidades para registrar la rutina encargado de manejar determinado comando recibido a través de un mensaje.

- **get_response()** permite consultar por el arribo de una respuesta a partir de un canal indicado, pudiendo establecer si se desea bloquear o no la llamada hasta que el arribo se concrete.

- **set_knowledge()** y **get_knowledge()** son métodos que existen con el fin de facilitar el acceso a la componente *Knowledge* de la unidad *Engine* (el almacenamiento y recuperación de datos de la Base de datos). A partir de estos dos métodos, las diferentes unidades realizarán procesos como por ejemplo, la creación de nuevas tareas, o la actualización de su información, los cuales implican crear nuevas entras en la base de conocimiento o modificar algunas ya existentes.

- Dentro del *Swarming* existen dos comandos por defecto en todas las unidades, los cuales son ***halt*** y ***response***. El primero es el comando recibido cuando la unidad debe iniciar su detención, mientras que el segundo es el comando ejecutado durante la recepción de una respuesta a otro mensaje enviado.

- Por último tenemos el método ***dispatch()*** y los métodos ***forward()*** y ***digest()***, los cuales en conjunto se encargan de la recepción, enrutamiento y procesamiento de los mensajes a lo largo del *Swarming*.

Para poder avanzar en el entendimiento de otros mecanismos existentes dentro de la aplicación, primero debemos conocer más en detalle el funcionamiento de algunos de los elementos antes listados, de las *Unidades*.

5.1.1. Las Respuestas

Ante el envío de un mensaje desde una unidad, el receptor canalizará la respuesta mediante un comando response. Este comando no requiere enviar una respuesta cuando la unidad receptora lo recibe. Al momento de recibirse un mensaje *response*, el manejo que se realiza del mismo consiste en el agregado de este a un diccionario de la clase *Unit* que contiene todas las respuestas indexadas por canal. Cuando la unidad receptora del mensaje *response* desea conocer si una determinada respuesta existe dentro del conjunto de respuestas recibidas, realiza un llamado al método *get_response()* indicando el canal sobre el cual la respuesta debería haber arribado.
En la *figura 10* podemos observar el proceso a través del cual una respuesta es recibida y procesada por una unidad. La línea punteada divide la imagen

en los dos *Threads* que interactúan en el proceso. Del lado izquierdo de la misma, tenemos al *Thread* que realiza la recepción de la respuesta, el cual una vez logra garantizar su acceso exclusivo al contenedor de respuestas, agrega la misma indexándola por su número de canal. Una vez la respuesta se encuentra contenida en *responses*, el *Thread 1* notifica a todos los demás *Threads* que pueden estar esperando por respuestas, que una ha arribado, para luego liberar el acceso exclusivo que posee. Del otro lado de la línea punteada, el *Thread 2* inicia una llamada a *get_response()* la cual intenta obtener acceso exclusivo al diccionario *responses* para consultar por la existencia de una respuesta en particular. Si la respuesta existe, se la extrae del diccionario y se la devuelve a quien realizó la llamada, mientras que en caso de no existir, se espera la siguiente notificación de arribo antes de volver a realizar la consulta, repitiendo este ciclo las veces que sean necesarias hasta obtener la respuesta requerida.

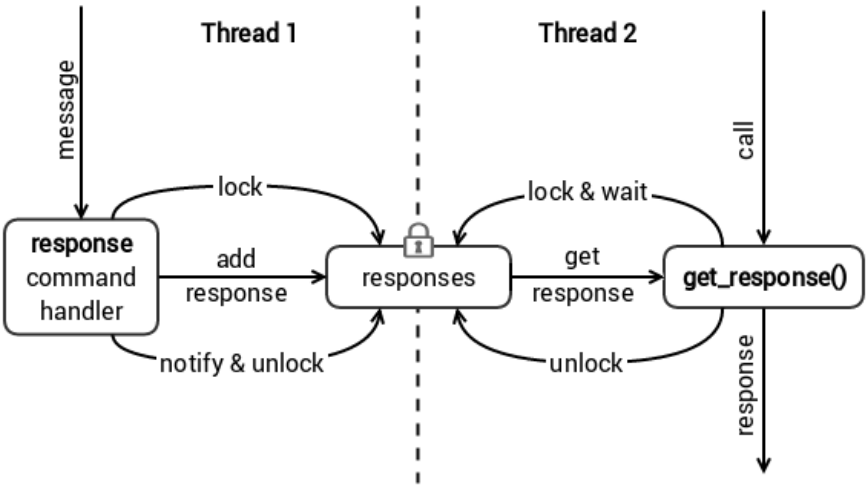

Figura 10: Comando *response*

Este proceso, como cualquier mensaje transmitido dentro del *Swarming*, involucra a la interfaz *dispatch()* para la transmisión, con la salvedad de que al final del procesamiento del mismo ningún otro mensaje es generado.

5.1.2. La interfaz *dispatch()*

El proceso de recepción de un mensaje por parte de una unidad, se inicia en el método *dispatch()* de la misma. Este método se encarga de recepcionar y en algunos casos analizar parcialmente, el mensaje con el objetivo de conocer si el mismo debe ser procesado por la unidad o reenviado.
La implementación por defecto del método *dispatch()* que existe en la clase *Unit*, básicamente controla si el mensaje esta destinado a la unidad que lo recibe o no. En caso de estarlo, pasa a ser analizado por el método *digest()* de la misma, el cual en su implementación por defecto, busca ejecutar el comando indicado en el mensaje. En caso de que el mensaje no vaya dirigido a la unidad, la implementación por defecto del método *forward()* es la encargada de tratarlo, la cual no realiza otra cosa que volver a enviar el mensaje a la interfaz *dispatch()* de la *Unidad Core*.
Esta rutina de tratamiento del mensaje es principalmente conservada por las unidades livianas que solo realizan trabajos de auditoría, mientras que otras unidades, como *Executor*, *Core* o *Engine*, poseen re-implementaciones de algunos o todos los métodos que componen este circuito de recepción de los mensajes.
Siguiendo con las imágenes, la *figura 11* presenta el circuito de procesamiento de un mensaje comenzando desde su arribo a *dispatch()* hasta su finalización en *digest()* o *forward()*. Los tres recuadro que simbolizan las diferentes fases por las que se puede mover el mensaje recibido, serán los componentes que se modificarán a lo largo de algunas unidades para realizar tratamientos diferentes de los mensajes durante su recepción.

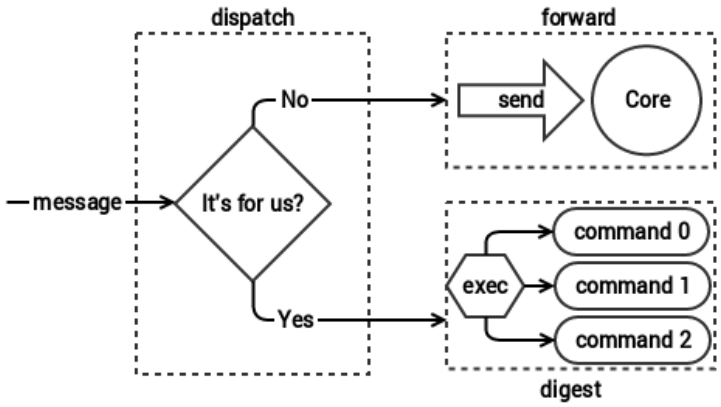

Figura 11: Circuito de *dispatch()* por defecto

Es importante dejar en claro la estructura que el manejo del mensaje posee, ya que en las secciones siguientes se expondrán circuitos más complejos de tratamiento del mensaje, que se encuentran basados sobre este.

5.2. El módulo Messenger

El envío de mensajes entre unidades que se encuentran en procesos diferentes involucra por un lado el pasaje de mensajes entre los mismos y por el otro, un mecanismo de consumo de los mensajes recibidos que no interrumpa la normal ejecución de la unidad. Tomando esto como base surge la implementación del módulo ***Messenger***, el cual está compuesto por una cola de mensajes *Queue* sobre la cual se colocan los mensajes recibidos, y por un *Thread* el cual toma dichos mensajes y los analiza, enviando estos a los métodos *forward()* o *digest()* de la unidad receptora, según corresponda. A través de la sobrecarga del método *dispatch()*, las unidades que poseen el módulo *Messenger* interceptan el envío de mensajes que se realiza sobre ellas, encolando estos y devolviendo en la respuesta inmediata, el canal correspondiente a la comunicación. Para el procesamiento de los mensajes que son obtenidos en el otro extremo de la cola, *Messenger* de forma autónoma se encarga de tomar los mismos y siguiendo el análisis realizado por *dispatch()* en su implementación por defecto, evalúa si el mensaje debe ser

enviado al método *forward()* o *digest()* de la unidad, volviendo a repetir el ciclo una vez se finaliza.

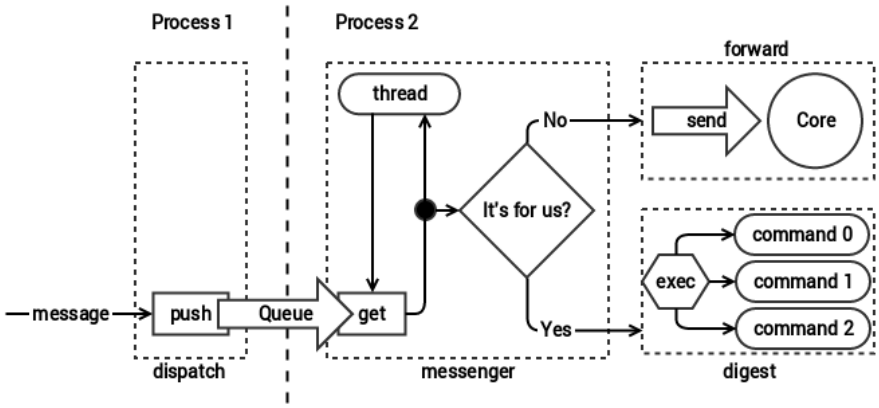

Figura 12: *Messenger*

De esta manera, durante la ejecución de la unidad existente en el proceso 2, puede ocurrir que el envío de un mensaje desde el proceso 1 sea iniciado sin que esto bloquee a ninguno de los dos. Con el mensaje ya agrega al buffer de *Queue*, la instancia de *Messenger* existente en el proceso 2, a través de su *Thread* realizará la extracción del próximo mensaje a tratar, redirigiendo el mismo o ejecutandolo sobre la unidad sin que esto interfiera con la normal actividad que el proceso 2 pueda estar realizando. De esta manera, el módulo *Messenger* le otorga a la unidad que lo posee, la recepción desatendida de mensajes desde otras unidades que se encuentran en otros procesos, dando cierre así a todos los requerimientos restantes de la comunicación.

6. Unidades Core

Una de las unidades más importantes del *Swarming* es la unidad ***Core***, la cual no solo nuclea e intercomunica todas las unidades de las que mantiene referencia, sino que también es quien inicia toda la ejecución de la aplicación en un comienzo. Esta característica que posee *Core* de conocer todas las unidades de la aplicación, es lo que llevó a que sea la unidad responsable de implementar entre otras cosas, los siguientes requerimientos

- Manejar la lógica de redireccionamiento de los mensajes, que se encarga del envío de los mismos dentro de la capa y hacia las demás capas.
- Gestionar la ejecución de mensajes dirigidos a *Core*, la cual es dependiente de la capa sobre la que se realiza.
- Implementar los mecanismos encargados de la gestión de las unidades (carga y descarga) que existen dentro de la capa.

El desenvolvimiento que la unidad *Core* tiene en todas sus actividades, está directamente ligado a la capa en la que se encuentra. La capa 0 (*Layer 0*) es especial respecto a las actividades que *Core* realiza, ya que es la única capa que existe de forma obligatoria. La instancia de *Core* existente en la capa 0 es la única que conoce a todos los *Executors* dentro del *Swarming*, por lo que sólo a través de esta se puede realizar la distribución homogénea del trabajo entre las mismas. Cuando ocurre que un mensaje recibido por *Core* debe ser reenviado a una unidad que no reconoce, de ocurrir en la Capa 0, este será asignado a una nueva capa de forma aleatoria, generándose un mensaje de error en el caso de que lo mismo ocurra en una capa diferente. Otro comportamiento que distingue al *Core* de la capa 0 de los demás, es el hecho de que solo este sabe como manejar los mensajes que van dirigidos a instancias de *Core* de otras capas. Cuando un mensaje es recibido en la capa 0 con *Core* como destino, es el campo *"layer"* del mensaje el que indicara sobre cual de todas las capas se lo debe ejecutar, asumiendo que la capa es la 0 en caso de que el campo *"layer"* no exista. Fuera de estos

aspectos, todas las instancias de *Core* se comportan de la misma manera, enviando a la unidad destino dentro de su capa, el mensaje recibido. Estos comportamientos se encuentran implementados en los métodos *forward()* y *digest()* de *Core*, representándose una abstracción de los mismos en la *figura 13*.

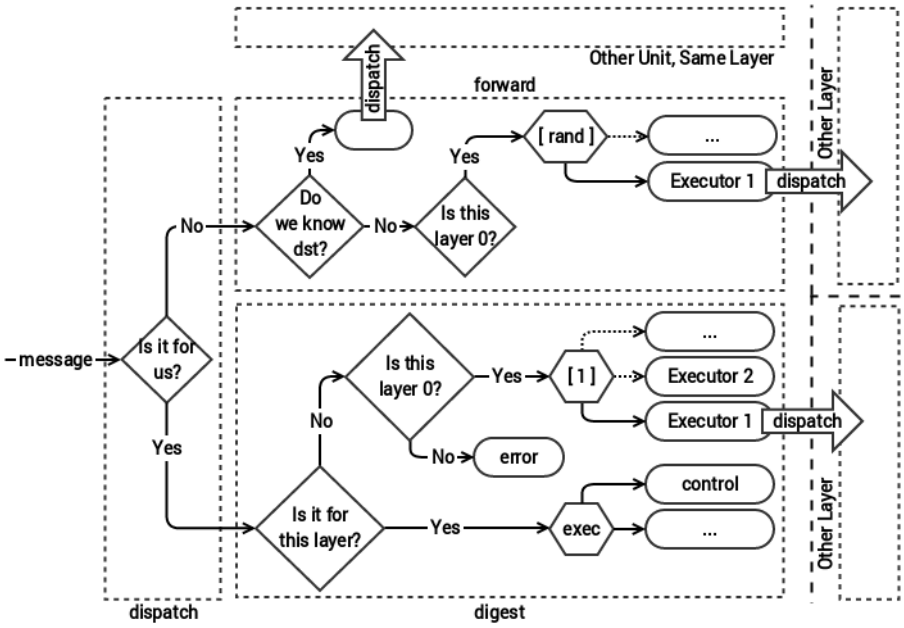

Figura 13: Método *forward* y *digest* de *Core*

El último aspecto desarrollado por *Core* es la gestión de las unidades dentro de la capa, la cual consiste entre otras cosas, en la creación de las unidades que se desea que la capa posea. Esto es llevado a cabo a través del comando ***control*** que *Core* implementa y que básicamente consta de tres rutinas: ***load***, ***drop*** y ***reload***.

- La rutina *load* crea y pone en funcionamiento la unidad indicada por el parámetro *'unit'*, dentro de la capa. La creación de la unidad está basada en el método de clase ***build*** del cual se utiliza su implementación por defecto para las *unidades livianas* (*HTTP*, *SSH*, etc), pero que es sobrecargado por las demás (*Executor* y *Engine* por ejemplo)

para llevar la tarea a cabo.

- *drop* es la rutina ejecutada para realizar la baja de una unidad dentro de una capa, siendo el nombre de la unidad el único parámetro.
- *reload* es una rutina que existe con el único motivo de evitar tener que realizar la baja y el alta de una unidad a través del envío de los mensajes *drop* y *load*. Esta rutina es útil si existe algún cambio en la implementación de una unidad que requiere ser reflejado sobre la ejecución del *Swarming*, sin que esto implique detener al mismo.

Estas rutinas son parte importante del mecanismo multitarea del *Swarming*, ya que a través de ellas es que se realiza la carga de las unidades *Executors*, las cuales crean las nuevas capas.

7. Unidad Executor

Las unidades encargadas del trabajo de *Cracking* y *Crawling* dentro del *Swarming* (las *Light Units* o *Unidades Livianas*), son unidades que son invocadas de forma externa durante el proceso de recepción de un mensaje. La ***Unidad Executor*** es la encargada de proporcionar dicha ejecución a las unidades livianas que existen dentro de su capa, siendo además esta unidad el único punto de acceso que la capa posee.

Supongamos que la unidad *Engine* desea enviar un mensaje a una unidad liviana (por ejemplo *HTTP*). Para esto puede indicar la capa a la cual desea dirigir el mensaje o dejar que su instancia de *Core* seleccione una de forma aleatoria. En cualquiera de los dos casos, cuando el mensaje llega a la unidad *Core* de *Engine* (*Core* de la capa 0), esta toma el *Executor* correspondiente a la capa seleccionada y realiza el *dispatch* sobre la misma, retornando a *Engine* la respuesta inmediata. Una vez la unidad *Executor* de la capa seleccionada recibe el mensaje, se encarga de clasificar este según lo considere *simple* de procesar o *complejo*, lo cual se establece a partir de la presencia del *flag* ***"heavy"*** dentro del mensaje. Si el mensaje es considerado simple de procesar, este es ejecutado de forma inmediata sobre la unidad destino, mientras que si se lo considera complejo, será vuelto a encolar en una cola especial de mensajes la cual es consumida de forma paralela. Esto permite que, en caso de ocurrir el arribo de un mensaje que requiere atención inmediata por parte de la unidad que se encuentra realizando un trabajo (por ejemplo un mensaje urgente como *halt*), este pueda llegar a ser ejecutado en un tiempo razonable, sin tener que esperar la finalización de la ejecución en curso. La *figura 14* ilustra el ejemplo.

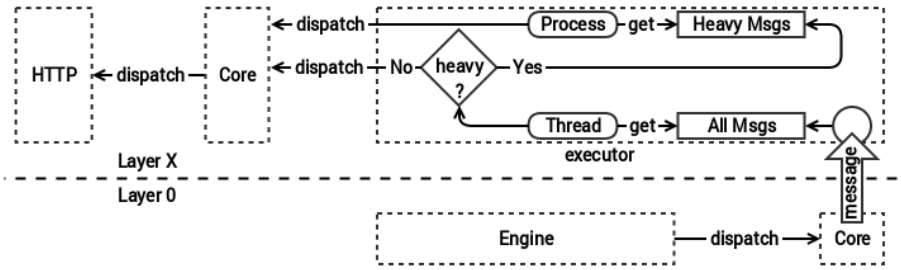

Figura 14: *Light Units* y *Executor*

Como ocurre con todas las unidades que reciben mensajes de otras capas, la unidad *Executor* posee un *Messenger* el cual se encarga de la recepción y consumo de los mismos. A través de la sobrecarga del método *forward()*, *Executor* modifica el comportamiento por defecto en la redirección de los mensajes, siendo este el punto en el que se controla si los mismos serán ejecutados en el momento o vueltos a encolar para ser procesados de otra manera. Esta cola paralela de mensajes, dedicada a los mensajes *complejos*, recibe un procesamiento muy similar al que *Messenger* realiza sobre los mensajes, con la diferencia de que el hilo de ejecución que la consume, es el hilo principal del proceso perteneciente al *Executor*. Cuando un *Executor* es creado, el mismo es lanzado sobre un nuevo proceso el cual representa la capa sobre la que se encuentra. Una vez el proceso crea el *Messenger* y da contexto a la capa, la ejecución se vuelca a la atención de esta cola de mensajes *complejos*, la cual permanece a la espera de los mensajes que el *Messenger* pueda enviarle. Para ilustrar mejor este proceso, la *figura 15* muestra la implementación del *Executor* y como a través de la sobrecarga del método *forward()*, el *Messenger* es capaz de clasificar el trabajo.

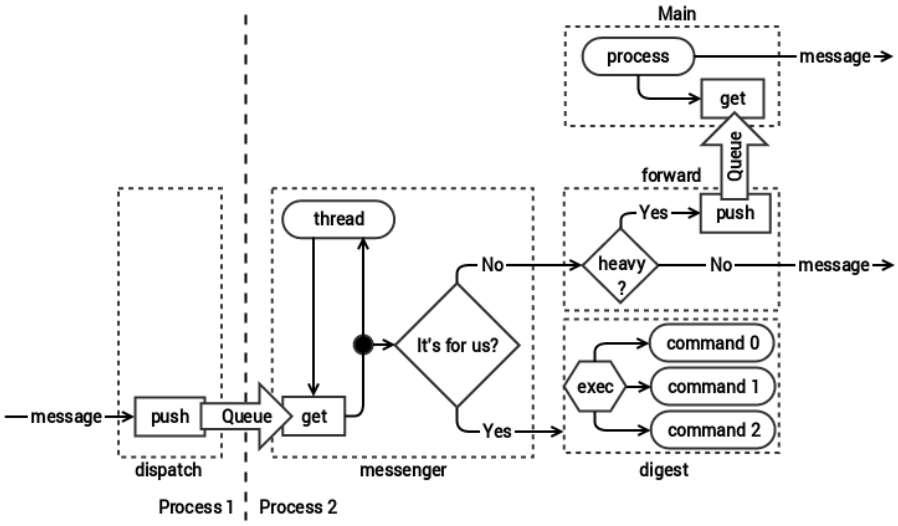

Figura 15: Manejo de mensajes por *Executor*

Con las capas administrando a partir de *Executor* la recepción, clasificación y ejecución de los mensajes a enviar a las unidades livianas, sólo resta conocer la unidad encargada de la generación de dichos mensajes.

8. Unidad Engine

El manejo de la información obtenida a lo largo de la ejecución de la aplicación, el análisis para la realización del trabajo pendiente y la presentación de una *API* que nos permita gestionar el sistema, son las responsabilidades que definen a la unidad ***Engine***. Esta unidad posee una base de datos como elemento principal en su constitución, la cual es accedida por todos los componentes que implementan su funcionalidad: ***Knowledge***, ***Tasker*** y ***WebUI***. El componente *Knowledge* es el encargado de realizar el almacenamiento y obtención de la información que otras unidades pueden querer realizar sobre la base de datos, la componente *Tasker* es la encargada de analizar dicha información para la inferencia y actualización de las tareas que se deben ejecutar, y *WebUI* es quien brinda al usuario una *API* simple que le permita realizar la gestión de la aplicación, como así también tener acceso a la información conocida por la misma.
Antes de iniciar un análisis más profundo de la implementación de cada una de las componentes, vamos a explicar el funcionamiento y los aspectos relacionados a una subcomponente conocida como ***ORM***, la cual fue observada por primera vez en el modelo de la *Figura 9* y que tiene como función el modelado de los datos y abstracción del accesos a la base de datos.

8.1. Subcomponente ORM

La subcomponente ***ORM*** (por *Object-Relational Mapping*) es un módulo de la unidad *Engine* el cual permite realizar una abstracción de los datos de la base de datos, como así también del acceso a la misma. El módulo *ORM* sigue un patrón *Singleton* el cual busca que todos los componentes utilicen el mismo canal de comunicación con la base de datos, encargándose *ORM* de generar la exclusión mutua entre ellos. Otro aspecto importante del módulo es la posibilidad de convertir las entradas obtenidas como resultado de una consulta, en objetos con formato *JSON*, lo cual facilita la inserción de los mismos de manera casi inmediata en el circuito de mensajes del *Swarming*. Por cada tabla existente en el modelo de datos, *ORM* posee

una clase que define y da funcionalidad a la misma, definiéndose en estas la manera en la que una entrada de la tabla debe ser convertida a un objeto *JSON* y viceversa.
Para facilitar el acceso a la información por parte de las componentes que utilicen *ORM*, el módulo implementa una interfaz que posee dos métodos, ***set*** y ***get***. Ambos métodos requieren como únicos parámetros el nombre de la tabla sobre la que operar y los datos sobre los cuales basar la consulta. En el caso de *get*, los datos suministrados serán utilizados para identificar las entradas que serán devueltas como resultado, mientras que para *set*, se intenta reconocer una entrada existente la cual modificar con la información suministrada (por ejemplo a partir del *id* indicado) o se crea una nueva. Estos métodos utilizados mayormente por la componente *Knowledge* son expuestos de mejor manera durante su explicación.

8.1.1. Modelo de Datos

El modelo de datos pretende establecer la forma que se le da a la información al momento de ser persistida, independientemente del manejador de base de datos que sea utilizado. En la *figura 16* podemos ver representado el modelo de datos utilizado por el *Swarming*, con todas sus entidades y relaciones entre ellas.

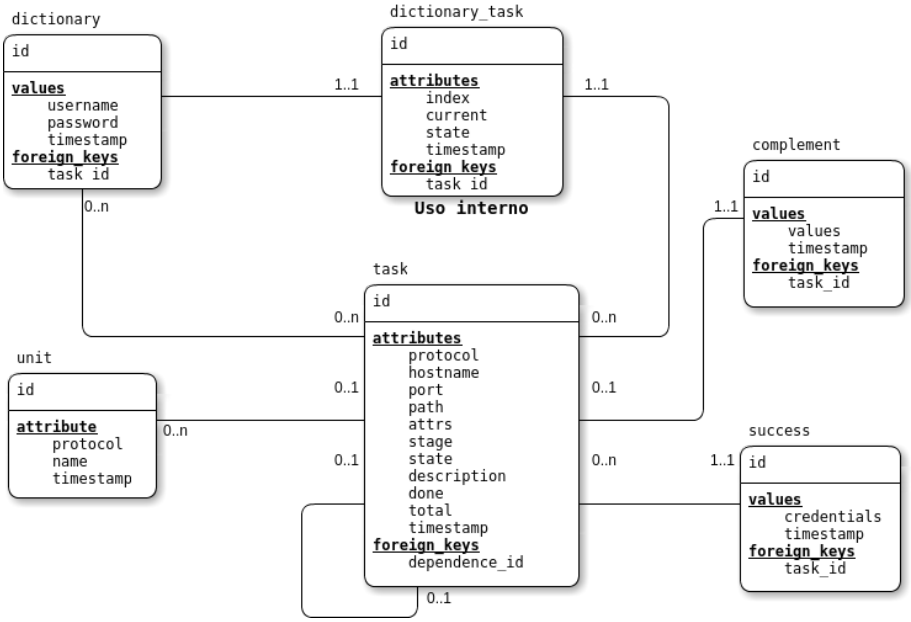

Figura 16: *Modelo Relacional* de datos de *Tasker*

Si bien en muchos casos, a partir de los nombres de las tablas o de sus atributos, podemos inferir el rol de las mismas, no es esto siempre así, por lo cual a continuación detallamos la función de cada tabla como a así también una descripción de sus atributos, en los casos en los que sea requerido.

- ***Unit*** (*unidad*) se encarga de mantener un registro de que *unidad* procesa que *protocolo* a través del nombre de la unidad y del protocolo, los cuales se almacenan en los atributos *name* y *protocol* respectivamente. Por ejemplo, durante el procesamiento del comando *register* por parte de la unidad *HTTP*, se realiza el registro de los protocolos que esta administra, los cuales son enviados a *Engine* para la creación de una entrada por cada uno de ellos, en la tabla *Unit*.

- ***Task*** (*tarea*) es la entidad central dentro del modelo, ya que todas las demás giran en torno a ella. Todo lo que el *Swarming* realiza se encuentra de una u otra forma ligado a una tarea, motivo por el cual esta es una de las entidades más requeridas. Cada entrada dentro de

esta entidad representa un proceso que ya fue hecho o que aún resta por hacer del *Swarming*, siendo sus diferentes atributos los responsables de definir las características del mismo.

- Los atributos ***protocol***, ***hostname***, ***port***, ***path*** y ***attrs*** definen el *Recurso* sobre el cual se ejecuta la tarea, de los cuales los primeros cuatro hacen referencia a los mismos componentes que constituyen una *URL* con la forma

  ```
  <protocol>://<hostname>:<port><path>
  ```

  El atributo ***attrs*** es más complejo y funciona como contenedor de características propias del recurso que son sólo significativas para este, como es el caso del método a utilizar por *HTTP* (*POST* o *GET* por ejemplo) o el nombre de la base de datos para una unidad como *MySQL*.
- Los atributos ***stage*** y ***state*** definen la *etapa* en la cual la tarea se encuentra en su ejecución y el *estado* de la misma, respectivamente. Los diferentes valores que estos campos pueden adquirir como así también una definición más completa de los mismos, se presentará en la exposición del *Componente Tasker*.
- ***description*** es una cadena que contiene una descripción corta de lo que el recurso es, la cual suele ser completada durante la etapa *initial* o cuando una tarea da origen a otra.
- ***done*** y ***total*** son números que indican la cantidad de trabajo realizado y la cantidad de trabajo total a realizar por la tarea. Estos valores son actualizados periódicamente y son utilizados para calcular el porcentaje de trabajo restante antes de terminar con la tarea.

- ***success*** es la tabla en la cual las credenciales que resultaron exitosas en un proceso de *Cracking*, son almacenadas, existiendo las mismas en formato *JSON* dentro del campo ***credentials***.

- ***complement*** se encarga de almacenar en su atributo ***values*** una cadenas en formato *JSON* conteniendo todo lo que una unidad debe saber para poder tener acceso al recurso de *login* al que la entrada pertenece. Por ejemplo, cuando una unidad que realiza un proceso de *Cracking* sobre un recurso logra obtener acceso, la misma agrega en una entrada de *complement* todo lo que las unidades que de ella dependen, deben saber para poder acceder de la misma manera. Esta información no siempre es tan simple como conocer las credenciales para el acceso, motivo por el cual la tabla *success* no es utilizada con este fin.

- La tabla ***dictionary*** está compuesta por los campos ***username*** y ***password*** en los cuales se almacenan las credenciales a ser utilizadas por los *Crackers*. En los casos en los cuales se busca agregar al diccionario una entrada consistente únicamente en un *nombre de usuario* o el *password*, lo que se hace es introducir la misma dejando el atributo restante en *NULL* de manera tal que para el *Swarming* la misma se encuentre constituida solo por un elemento.

- Por último, la tabla ***dictionary_task*** es una tabla de uso interno al *Swarming* (no visible desde ninguna unidad que no esté en *Engine*) cuyo objetivo es establecer qué entradas de *dictionary* ya fueron consumidas por una tarea de *Cracking* o están en proceso de serlo. El atributo ***state*** indica el estado en el que se encuentra el consumo de un conjunto de entradas del diccionario, mientras que ***current*** e ***index*** se utilizan para definir dicho conjunto. Los detalles de su funcionamiento son vistos durante la descripción de la *Componente Tasker*.

Todo el modelo de datos presentado constituye la estructura que la información posee dentro del *Swarming*, utilizándose para la implementación y gestión de la misma, un mecanismo de abstracción conocido como *Mapeo Objeto-Relacional*.

8.1.2. SQLAlchemy y SQLite3

Una librería de abstracción ***ORM*** (*Object-Relational Mapper*) implementa un mecanismo a través del cual se realiza la persistencia de información existente en una instancia de clase, a una tabla de una base de datos relacional, sin la necesidad de realizar consultas *SQL* por parte del usuario. Cuando se desea crear una entidad del modelo de datos, como por ejemplo *task*, se crea una clase *ORM* la cual a través de mecanismos dependientes de la herramienta que lo implementa, genera una relación entre la clase y su tabla correspondiente, reflejando a partir de ese momento todas las instancias de la clase como entradas de dicha tabla. Las instancias creadas a partir de esta clase poseerán entonces un tiempo de vida que trasciende al de la instancia en sí, ya que los datos de la misma existirán en la base de datos después de su destrucción, con la posibilidad de volver a crear la instancia a partir de ellos. Otra de las cualidades que este modelo posee, es la capacidad de utilizar herramientas propias de la programación orientada a objetos en las clases creadas, como la *herencia* y la *sobrecarga*, permitiendo dar comportamientos particulares a las clases y extender su funcionalidad. La herramienta utilizada por el *Swarming* para realizar la implementación del *ORM* dentro de la unidad *Engine* es ***SQLAlchemy*** [20], la cual no sólo es potente y simple en su uso, sino que también es una de las más utilizadas en el contexto de los usuarios *Python* [19].

El ***DBMS*** (*DataBase Management System*) por otro lado, es la base de datos propiamente dicha, existiendo infinidad de estas hoy en día, que implementen el *Modelo Relacional*. ***SQLite3*** [21] es el *DBMS* seleccionado para ser utilizado junto con *SQLAlchemy* ya que es extremadamente rápido, muy portable entre plataformas y posee la característica de utilizar como objeto de persistencia los archivos. Esta característica de *SQLite* de almacenar los datos en un único archivo, permitirá al usuario la administración de los mismos de manera sencilla, dándole mucha flexibilidad sobre el manejo de la información de la ejecución.

A través de estas dos herramientas el subcomponente *ORM* implementa la persistencia del conocimiento que la unidad *Engine* maneja, manteniendo un nivel de abstracción que permite el cambio de la base de datos sin que esto genere modificaciones profundas en el módulo. Durante el proceso de selección del mecanismo de persistencia, existieron otras opciones que, por como funciona el *Swarming*, brindaban algunas ventajas que el modelo relacional no ofrecía. El manejador de bases de datos conocido como ***MongoDB*** [14] es un *DBMS no-relacional*, lo que significa que la información que almacena no precisa tener una estructura predefinida como sí ocurre con las bases de datos relacionales. *MongoDB* brindaba la posibilidad de almacenar los datos recibidos en formato *JSON* de forma directa ya que es este el formato que la base de datos utiliza. El inconveniente que *MongoDB* plantea es su falta de portabilidad ya que para hacer uso de la misma el cliente debe primero instalarlo, lo que no sólo es incómodo sino que también injustificado respecto a las ventajas que plantea.

8.2. Componente Knowledge

Durante la ejecución del *Swarming*, la información generada por las diferentes unidades, es almacenada y consultada a través de los comandos que el *Componente* ***Knowledge*** implementa en *Engine*, los cuales son ***set*** y ***get***. Tanto el comando *set* como el comando *get* poseen parámetros relativamente similares, radicando la diferencia únicamente en como estos son utilizados. Ambos comandos reciben una lista de diccionarios que representan operaciones a realizar sobre la base de datos, siguiendo los mismos la forma ***"tabla":{valores}*** en cada una de sus entradas. Por ejemplo, una lista de operaciones enviada por una unidad a *Engine* podría tener la forma

```
[{"task":{...}, "success":{...}}, {"task":{...}}]
```

donde podemos observar dos operaciones: La primera realizando dos consultas, una a *"task"* y otra a *"success"*, y la segunda solo realizando una a *"task"*. De esta manera, la unidad que lo requiera puede realizar pedidos complejos de información sin necesidad de efectuar el envío de muchos mensajes. Cada una de las consultas que se puedan realizar, representa un acceso

a los métodos *get* y *set* del módulo *ORM*, siendo estos quienes realizan la real implementación de las mismas. En la implementación del método *get* de *ORM*, la búsqueda se suele realizar normalmente bajo el criterio de los campos *id* y *timestamp*, aunque se permite efectuar búsquedas bajo otros criterios. El campo *timestamp* es un valor que existe en todos los datos almacenados dentro de *Engine*, el cual indica la última vez que el dato fue modificado. A través de este campo es posible conocer los elementos que sufrieron modificaciones desde la última vez que se accedió a la misma, obteniendo de esta manera solo la diferencia entre los datos nuevos y los datos viejos. Este campo a diferencia de los demás, durante su consulta es analizado no por medio del operador de *igualdad* sino a través del operador **“>”**, contemplando de esta manera un conjunto de valores. El campo *id* en caso de especificarse en la consulta, ignora lo indicado por *timestamp*, devolviendo en su resultado la entrada de la tabla indicada por su valor. Al realizarse una consulta sobre elementos compuestos (por ejemplo success que posee un *id* del *task* al que pertenece), los datos obtenidos contendrán de forma anidada la información que es necesario conocer de los elementos de los cuales dependen, devolviendo a quien realizó la consulta, una versión más completa del elemento, que no requiera resolver todas sus dependencias. Supongamos que deseamos obtener todas las entradas de *success* cuyo *timestamp* sea mayor a 0 (o lo que es lo mismo, desde el comienzo de la ejecución). Para esto, la unidad que las requiere enviará dentro de la lista de consultas del comando **“get”** algo como **{“success”:{“timestamp”:0}}**, lo cual devolverá en el mensaje de respuesta a la unidad, datos como los siguientes

```
{"success":{"id":0, "credentials":"...", "task":{"id":3,...}}},
{"success":{"id":1, "credentials":"...", "task":{"id":1,...}}},
  ...
```

De esta manera, quien realizó la consulta a *success*, no solo obtendrá la información de cuáles fueron las credenciales que tuvieron éxito, sino también, la información (por cada credencial) de la tarea sobre la cual se obtuvo

acceso.

El método *set* de *ORM* por otro lado, sigue una estructura similar a *get* en los parámetros que recibe, cambiando en este caso la manera en la cual se identifica la entrada referenciada. La actualización ocurre cuando entre los valores de la consulta se encuentra establecido el *id*, el cual referencia un elemento en particular, haciendo uso del resto de la información suministrada para actualizar el mismo. De no existir el campo *id*, lo siguiente que el método *set* intenta es averiguar si ya existe una entrada en la base de datos que corresponda con un elemento exactamente igual al que se desea crear, el cual en caso de existir es obtenido para luego devolver su *id*, sin realizar sobre este ninguna modificación. Por último, si ocurre que el elemento no existe, se crea una instancia de este utilizando los datos suministrados, se almacena la misma y una vez se le asigna un *id* dentro de la base de datos, este es devuelto como ocurre en los demás casos. Para aclarar lo anterior, a continuación se presenta un ejemplo en el cual se realizan tres operaciones diferentes de *set* sobre una misma entrada, ilustrando así los tres posibles casos antes descriptos.

- Supongamos que en un primer momento se realiza el envío de información de una nueva tarea, que no corresponde con ninguna entrada existente.
  - Enviamos la siguiente operación a *set*

```
{"task":{
    "protocol":"http",
    "hostname":"127.0.0.1",
    "port":80,
    "path":"/index.php"}}
```

  - Donde obtenemos el ID 6 como resultado.

```
{"id":6}
```

- A continuación, modificamos el atributo *port* de la tarea estableciendo como *id* el valor obtenido en la operación anterior.
  - La operación enviada seria

```
{"task":{
    "id":6,
    "port":8080}}
```

  - Obteniendo el mismo ID como respuesta, pero sabemos que el puerto cambió al valor *8080*.

```
{"id":6}
```

- Por último, volveremos a intentar crear una tarea con las mismas características que las poseídas por la entrada con ID 6, obteniendo como resultado la tarea creada en un comienzo y no una nueva.
  - Enviamos la información de una nueva tarea a crear

```
{"task":{
    "protocol":"http",
    "hostname":"127.0.0.1",
    "port":8080,
    "path":"/index.php"}}
```

  - No obtenemos un nuevo ID (nueva entrada), sino que obtenemos el de la tarea ya creada con esas características

```
{"id":6}
```

De esta manera es como entonces, a través de los comandos *get* y *set* implementados por *Knowledge*, las diferentes unidades pueden gestionar parte de la información almacenada en *Engine*, existiendo aún la necesidad de realizar inferencias automáticas a partir de la misma.

8.3. Componente Tasker

La *Componente* ***Tasker*** es aquella encargada de llevar adelante el avance de las unidades a través de sus diferentes etapas además de la asignación de trabajo a las mismas. La componente *Tasker* esta constituida por una batería de rutinas que son ejecutadas de forma periódica, las cuales por un lado se encargan de analizar las respuestas recibidas como resultado de los mensajes enviados en ciclos anteriores, mientras que por otro, llevan a cabo la generación de nuevos mensajes con nuevos trabajos. Todas las rutinas ejecutadas por *Tasker* están basadas en el análisis de la etapa en la cual una tarea se encuentra, siendo las diferentes rutinas las encargadas de realizar los cambios necesarios sobre la tarea, transiciones a otras etapas o cualquier otro comportamiento relacionado a la misma.
La etapa de una tarea, indicada a través del atributo *stage*, se encuentra constituida por una única cadena la cual puede ser descompuesta en subcadenas separadas por ".". La manera en la que este atributo se analiza es de izquierda a derecha existiendo como primer elemento siempre el nombre de la etapa y luego los parámetros de la misma. Este formato se asume como globalmente conocido dentro del *Swarming* por todas las unidades que lo componen, ya que el tratamiento del atributo *stage* no está limitado a la unidad *Engine* sino también a las unidades que consumen la tarea.
Con la idea de conocer mejor las diferentes etapas en las cuales una tarea se puede encontrar, como así también sus parámetros, a continuación describimos cada una de estas.

- ***initial:*** En el momento en que una tarea es creada, esta es la etapa establecida por defecto. Cuando una unidad recibe una tarea que se encuentra en la etapa *initial*, la misma debe realizar en ese momento todo lo necesario para conocer si la tarea es procesable, para luego definir la transición a la nueva etapa. Por ejemplo, durante el procesamiento de una tarea con protocolo *HTTP*, la unidad que la recibe, revisara si la misma referencia un recurso realmente disponible además de controlar si el mismo no cambia luego de su utilización (un *redirect* por ejemplo), los cuales son aspectos que la unidad debe poder

manejar.

- ***cracking:*** Una tarea en estado *cracking* es una tarea que posee en su información todo lo necesario para utilizar un recurso de manera correcta en un proceso de autenticación. Por lo general las tareas que se encuentran en esta etapa nacen a raíz de otra tarea que reconoce su existencia, como son las tareas en etapas de *crawling* o *initial*, siendo raro que una tarea nazca por un medio externo, en este estado. La etapa *cracking* posee como único atributo, el tipo de credenciales que se utiliza para realizar el proceso. Por ejemplo, para el caso de *usuarios* y *passwords* como credenciales, las cuales se encuentran en *dictionary*, el campo *stage* contendrá entonces *cracking.dictionary* como valor.
  Las tareas que se encuentran en esta etapa, por como funciona, jamás arriba al estado de *complete* ya que el *Swarming* nunca sabe cuando pueden aparecer nuevas credenciales que deban ser procesadas.

- ***crawling:*** Tanto la obtención de información como la búsqueda de nuevos recursos sobre los cuales hacer fuerza bruta, es el trabajo llevado a cabo durante el *crawling*. A diferencia de las tareas que se encuentran en la etapa de *cracking*, las de *crawling* se *auto-administran* a través de las acciones que toman las unidades que las procesan. Esta etapa apunta a un análisis del recurso, lo que puede generar nuevos recursos sobre los cuales seguir haciendo *crawling* (con fines de paralelización) o iniciar una tarea de *cracking*, arribando al estado de *complete* una vez haya agotado la información a extraer. Esta etapa no posee parámetros por lo que se la encuentra en el atributo *stage* únicamente como *crawling*.

- ***waiting:*** Esta es una de las etapas más ricas en cuanto a funcionalidad ya que plantea muchas posibles formas de ejecución. Hasta el momento existen dos posibles eventos por los cuales *waiting* puede encontrarse esperando. Por un lado existe ***dependence***, el cual indica que la tarea de la que se depende debe satisfacerse para poder continuar, mientras que por el otro tenemos a *time*, el cual establece la cantidad de segun-

dos que deben pasar antes de transitar a otro estado. Ambas etapas contienen al final de su expresión la etapa a la cual se pasará una vez el requerimiento que especifican sea satisfecho. Para visualizar mejor lo anterior, tenemos

- *dependence:*

  `waiting.dependence.<nueva_etapa>`

- *time:*

  `waiting.dependence.<nueva_etapa>`

donde <nueva_etapa> es la etapa siguiente, expresada de la misma manera a como se la expresa en *stage*.

Durante el proceso de *crawling* es frecuente que ocurra que algunos caminos puedan verse bloqueados por una autenticación que debe ser realizada antes de poder continuar. En estos casos, el proceso no se detiene sino que crea una nueva tarea de *crawling* sobre dicho camino, la cual se encuentra condicionada a que la tarea de *cracking* responsable de dicha autenticación, encuentre las credenciales necesarias para avanzar, pudiendo así la tarea original continuar con su proceso. Una vez la tarea que realiza el trabajo de *cracking* consigue credenciales válidas, la condición por la cual el nuevo *crawler* se encuentra bloqueado se ve satisfecha, permitiendo que el mismo continúe su camino a través del uso de las credenciales aportadas por su tarea hermana. Esto es lo que ocurre en el ejemplo explicado en el *capítulo 1* y que permite que el *Swarming* realice anidamientos para poder llevar la auditoría a lo más profundo del sistema.

Otro proceso llevado a cabo por la componente *Tasker* que vale la pena que sea explicado, es la manera a partir de la cual se gestionan los conjuntos de credenciales que serán enviados a las unidades que estén ejecutando alguna tarea de *cracking*. Lo que la componente busca lograr es construir

conjuntos de tres listas constituidas por *usuarios*, *passwords* y *tuplas* de forma separada, siendo las tuplas las entradas de *dictionary* que poseen ambos valores (*usuario* y *password*). El proceso seguido por la componente *Tasker* para la formación de dichos conjuntos consiste en la selección de una entrada que llamaremos ***current***, a partir de la cual se toman todas las entradas que se encuentren por debajo de ella, las cuales irán siendo recorridas por la variables ***index***, que junto a *current* buscarán formar el par *usuario*, *password*. Por ejemplo, tomando en cuenta los valores expuestos en la *tabla 1*, si asumimos que la entrada #4 es current, las entradas con las cuales se formarán pares serán la #2 y la #3, obteniendo como conjunto

```
{"usernames":["admin"], "passwords":["1234", "12345"]}
```

Este proceso se repite haciendo avanzar *current* sobre la lista de entradas, uniendo los conjuntos generados para así reducir el espacio que los mismos ocupan.

ID	**username**	**password**
01	root	
02		1234
03		12345
04	admin	
05	default	default
06		111111
07		123456

Tabla 1: Ejemplo de *dictionary*

Cuando una de las entradas a las que *current* apunta es un par completo (como ocurre con la entrada #5 de la tabla), la misma se agrega al grupo bajo el nombre ***"pairs"***, haciendo avanzar *current* a la siguiente entrada, sin modificar el valor de *index*. El factor que determina cuando el conjunto total es lo suficientemente grande para ser enviado a la tarea de *cracking*, es el número de pares que se puede generar a partir del mismo, lo cual se determina a través de

Usuarios * Passwords + Tuplas > N

donde *N* es un valor estático, interno a *Tasker*, que en futuras implementaciones se pretende calcular tomando como referencia la velocidad de consumo de las credenciales que la unidad posea.

Para llevar un registro de los conjuntos creados, el *Swarming* posee la tabla ***dictionary_task***, la cual es gestionada por *Tasker* sin que otra unidad o componente intervenga en ella. Dicha tabla almacena los valores de *current* e *index*, además de ***state***, el cual indica el estado en el que se encuentra el proceso de consumo del conjunto de credenciales que referencia. El valor de *index* es almacenado ya que puede ocurrir que el conjunto de credenciales se encuentre limitado antes de que *index* termine de recorrer todos los valores por debajo de *current*, lo cual genera que durante la generación de un nuevo conjunto de credenciales a consumir, se deba retomar el valor de *index* desde donde fue por última vez utilizado.

Esta estrategia de generación de grupos de credenciales a enviar a las diferentes unidades que estén procesando una tarea de *Cracking*, permite la incorporación dinámica de nuevas credenciales a la tabla *dictionary*, sin que esto implique reiniciar el proceso, obtener pares repetidos o algún otro tipo de problema sobre el consumo de las mismas.

8.4. Componente WebUI

La *Componente* ***WebUI*** (*Web User Interface*) es la encargada de brindar el servicio *HTTP* a través del cual se tiene acceso a la ***API*** del sistema, sirviendo el mismo de forma paralela una ***interfaz web*** la cual permite operar el *Swarming*. Dicha *interfaz web* es un elemento que no pertenece a la *API*, pero que por restricciones de seguridad existentes en los navegadores [17] debe ser servida desde el mismo lugar para poder funcionar.
La componente se encuentra constituida por las clases ***WebUI*** y ***UIApi*** las cuales se encargan de implementar la totalidad del comportamiento que la misma debe poseer. La clase *WebUI* lleva a cabo la inicialización del servicio

HTTP en un nuevo *Thread*, además de las configuraciones que el mismo requiere. La clase *UIApi*, por otro lado, implementa todas las funcionalidades que se encuentran en la *API*, siendo esta la única clase que hace uso del módulo *ORM* dentro de esta componente.
Todos los comportamientos relacionados al servicio *HTTP*, son desarrollados a través de la utilización del *framework web* ***CherryPy*** [6] de *Python*, el cual de forma muy simple, permite crear un servidor *HTTP* estrechamente integrado al funcionamiento del resto de la aplicación. Para la atención de requerimientos que deban ser procesados de forma dinámico, *CherryPy* ofrece un mecanismo el cual establece una relación entre un *path* y un objeto, reflejando todos los métodos indicados del objeto, como recursos dentro de dicho *path*. De esta manera es que se estableció que los recursos estáticos correspondientes a la *interfaz web*, debían ser servidos bajo el nombre ***/ui/***, mientras que la clase *UIApi* atendería de forma dinámica, todos los requerimientos que fuesen realizados con ***/api/*** como raíz.
Con el objetivo de facilitar aún más el procesamiento de los requerimientos, *CherryPy* ofrece la posibilidad de trabajar los datos recibidos y enviados, directamente en formato *JSON*, lo cual permite una mejor integración con la información ya existente en el *Swarming*.

8.4.1. La API

La interfaz que la *API* ofrece hasta momento en el que se escribió el presente documento, es extremadamente simple, consistiendo la misma de solo dos métodos: ***get*** y ***set***. Estos métodos funcionan casi de la misma manera en como funcionan los comandos de igual nombre de la componente *Knowledge* con la salvedad de que *get* incorpora algo más de ayuda para que el usuario limite el número de resultados.
El método *get* de la interfaz incorpora dos parámetros que es posible indicarle, los cuales son ***limit*** y ***offset***. A partir de estos, el método realiza una restricción sobre el conjunto de resultados obtenidos de la consulta realizada por el usuario, en el que toma como primer elemento el indicado con el índice *offset*, y a partir del cual selecciona la cantidad de elementos indicada por

limit. Por ejemplo, si el usuario especificó que *offset* es 20 y *limit* es 10 para una consulta hecha sobre la tabla *task*, entonces lo que se devolverá serán los 10 primeros elementos a partir del elemento número 20 obtenido de la tabla *task*.

8.4.2. La Interfaz Web

Toda la complejidad de la que carece la *API* radica en la *interfaz web*, recayendo en el navegador del usuario todo el computo que se debe realizar. Esta interfaz se encuentra constituida únicamente por elementos *JavaScript*, *HTML* y *CSS* a partir de los cuales se obtiene la información de la *API*, se la procesa y se la presenta al usuario de una manera clara y prolija. Para la implementación del estilo de la interfaz se utilizó el *framework* ***Bootstrap*** [5] mientras que para el manejo del *HTML* y acceso a la *API* se hizo uso de ***jQuery*** [11], las cuales facilitan y reducen en gran medida la cantidad de código requerido.
La presentación de la información se realiza a través de tres paneles los cuales son *Success*, *Dictionary* y *Tasks*.

Success es el encargado de presentar las credenciales que tuvieron éxito en el intento de acceder a un determinado recurso.

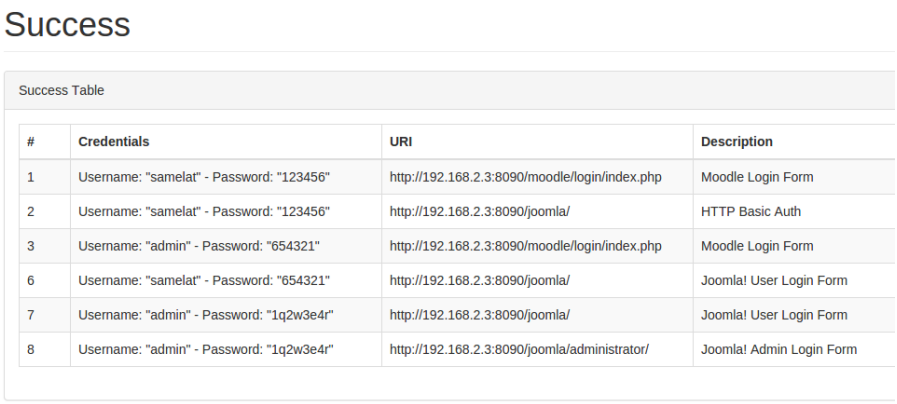

Figura 17: Panel *Success*

De izquierda a derecha, se observa el ID de la entrada, las credenciales exitosas, el recurso en formato *URI* sobre el cual lo fueron y la descripción de la misma.

Dictionary presenta todas las credenciales existentes dentro del *Swarming* que serán utilizadas por las tareas que realicen procesos de *Cracking*. El mismo, presente en la *figura 18*, está compuesto por el ID de la entrada, el nombre de *usuario*, el *password* y el ID de la tarea al cual la entrada pertenece. Las cruces negras presentes en las entradas, representan la ausencia del elemento, indicando que dicha entrada solo se encuentra compuesta por el que está presente. Por ejemplo, en la imagen que muestra el panel, la entrada #1 solo consiste en el nombre de usuario ***samelat***, mientras que la entrada #7 solo lo esta por el *password* ***123123***. La columna bajo el nombre *Task* en la imagen, se encuentra vacía ya que las mismas son de uso global, o lo que es lo mismo, son entradas que deben ser utilizadas sobre todas las tareas que existen en el *Swarming*. De especificarse un ID en dicho campo, la entrada solo sería utilizada sobre la tarea que corresponde.

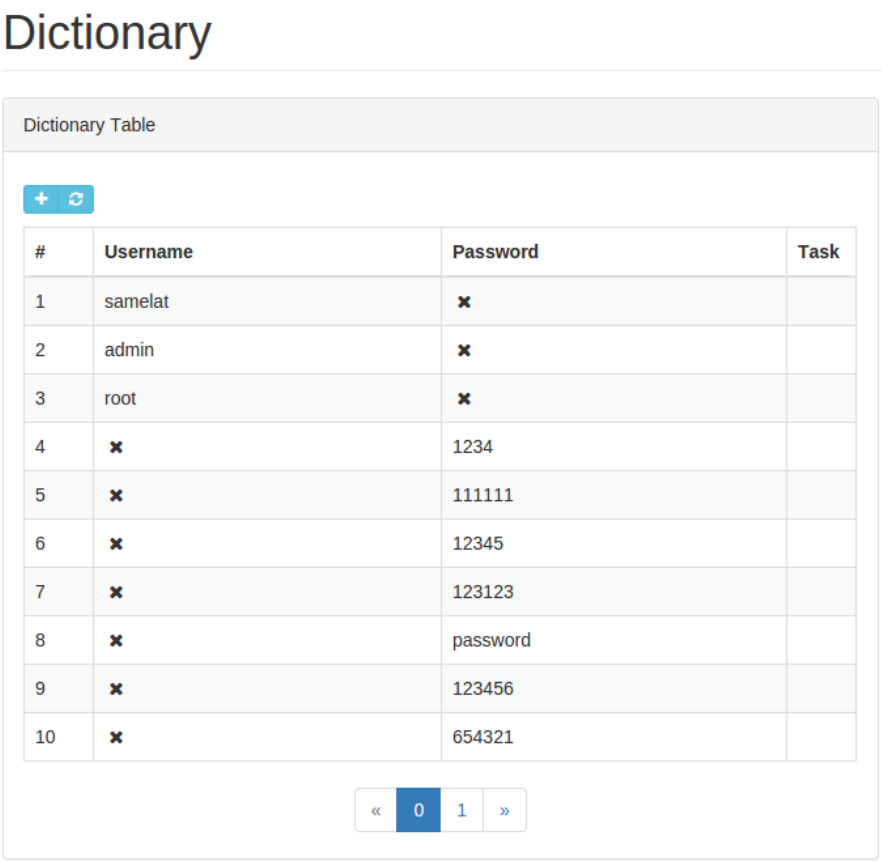

Figura 18: Panel *Dictionary*

Task, el cual ya fue presentado al comienzo del documento durante la explicación del caso de uso, presenta todas las tareas en curso, dando además, detalles de su evolución, estado, etapa en la que se encuentran y dependencia con otras tareas (ver *figuras 4, 5 y 7*). La interfaz consta de más elementos en su constitución los cuales no es necesario conocer para operar la aplicación, sino que existen con el único fin de facilitar el acceso a la información que suele ser más relevante de conocer durante la auditoría.

9. Unidades Livianas

Bajo este nombre es que se conoce a todas las unidades que pueden existir en una capa del *Swarming* y que no tienen capacidad de ejecución por sí mismas. Estas suelen ser las responsables de realizar el trabajo de procesar las tareas en sus diferentes etapas, llevando a cabo por ende los procesos de *Cracking* y *Crawling* entre otros.
Las unidades que implementan este comportamiento, heredan de la clase ***LightUnit*** la cual define los requerimientos básicos que la unidad debe desarrollar, además de algunas facilidades para procesos típicamente realizados por este tipo de unidades. Los métodos a destacar dentro de los implementados por *LightUnit* son

- ***build*** el cual es el método ***Factory*** [4] por defecto de estas unidades, el cual crea y configura la misma como corresponde.

- ***prepare*** es proporcionado por *LightUnit* a la unidad que de él hereda, para la realización de las configuraciones que la misma pueda requerir durante la recepción de un comando *consume*.

- ***success*** existe con el objetivo de facilitar el registro de credenciales que tuvieron éxito en un proceso de *Cracking*, enviando a *Engine* no solo las credenciales sino también el *complemento* requerido para que las tareas dependientes sean desbloqueadas.

- ***consume*** es uno de los dos comandos registrados por defecto para estas unidades. El mismo implementa un manejo por defecto estableciendo en el proceso algunas variables de contexto que facilitan a la unidad el procesamiento y ejecución del comando. Esta ejecución es llevada a cabo a través de alguno de los métodos relacionados a *consume*, los cuales la unidad registra para los diferentes valores de *stage* que soporta. Por ejemplo, si la unidad que hereda de *LightUnit* desea manejar los comandos *consume* con un valor de *stage "crawling"* a traves del metodo *self.my_crawling_handler()*, entonces lo que la uni-

dad hace es relacionar en el diccionario *stages* de *LightUnit*, la entrada *"crawling"* con el método a ejecutar como sigue

```
self.stages["crawling"] = self.my_crawling_handler
```

dejando así establecido el método que será llamado cuando el comando *consume* sea recibido con ese tipo de *stage*.

- ***register***, al igual que consume, es una implementación por defecto pero del comando *register*, el cual es llamado por *Engine* cuando una unidad que jamás fue cargada, es iniciada en alguna capa. Este comando como única tarea realiza el envío de un mensaje a *Engine* indicando el nombre de la unidad y los protocolos, información que es almacenada bajo la tabla *Unit* del modelo de datos.

Otra clase que no posee un rol tan protagónico dentro de los procesos de una unidad liviana como *LightUnit*, pero que igual mencionaremos brevemente, es la clase ***Dictionary***, la cual es responsable de realizar la mezcla de todas las credenciales recibidas por una unidad que debe procesar una tarea de *Cracking*. Cuando una unidad recibe una tarea la cual representa un proceso de *Cracking*, el mensaje además de la tarea, contiene los conjuntos de credenciales que debe utilizar, los cuales como se mencionó en la descripción de *Engine*, consisten de tres listas conteniendo *usernames*, *passwords* y *pares*. Si bien los *pares* no requieren nada para ser utilizados, los *usernames* y los *passwords* deben ser mezclados los unos con los otros, obteniendo de estos todos los posibles pares de valores a controlar. La utilización de *Dictionary* se inicia con la instanciación de la clase a partir de las tres listas de credenciales recibidas como parámetros. Una vez la instancia fue creada, lo único que resta es iterar sobre el valor devuelto por su método ***pairs()***, de la siguiente manera

```
dictionary = Dictinary(usernames=..., passwords=..., pairs=...)
for username, password in dictionary.pairs():
    ...
```

De esta forma, dentro del *for* se realizará la autenticación utilizando el par de credenciales del momento, finalizando la iteración cuando las credenciales se hayan consumido por completo.

Todos estos métodos y clases son la base de la implementación de unidades más complejas como *HTTP*, las cuales involucran más desarrollo que debe ser analizado unidad por unidad.

9.1. Unidad HTTP

La unidad ***HTTP*** es probablemente la unidad liviana más compleja a ser desarrollada en el *Swarming*. La misma intenta contemplar todos los casos de *Cracking* y *Crawling* que pueden darse bajo *HTTP* y *HTTPS*, no radicando la complejidad únicamente en estos protocolos sino en las diferentes tecnologías que sobre los mismos pueden viajar. Para la implementación de la unidad se utilizaron diferentes módulos *Python* dentro de los que se destacan ***requests*** y ***BeautifulSoup***. *requests* [18] es un módulo que permite la realización de requerimientos *HTTP* de manera muy configurable y simple, reduciendo de forma significativa el código requerido para el manejo de situaciones como los errores. Es en base al método *request* de *requests* que la unidad modela los requerimientos, estando estos representados por un diccionario donde cada entrada indica el parámetro del método y su valor. De esta manera, a través del desempaquetado de un diccionario como argumentos de un método [8], los requerimientos son utilizados sobre *request* sin que exista entonces la necesidad de crear un contenedor de datos que modele los mismos. *BeautifulSoup* [3] permite realizar de forma muy abstracta la manipulación del contenido *HTML* recibido en las respuestas a requerimientos, permitiendo que la unidad busque ciertos *tags*, información o cualquier dato requerido, de forma simple. Para el procesamiento de tareas, la unidad *HTTP* registra los métodos ***http_initial_stage***, ***http_crawling_stage*** y ***http_cracking_stage*** bajo los valores de stage *"initial"*, *"crawling"* y *"cracking.dictionary"* respectivamente.

9.1.1. Las tareas *initial*

El proceso de atención de las tareas por *http_initial_stage* se inicia con la realización de un requerimiento utilizando el recurso que la tarea posee. El objetivo de este es establecer si el servicio que se intenta alcanzar, se encuentra en funcionamiento y si el mismo realiza algún cambio sobre el recurso. Es frecuente que servidores que atienden requerimientos *HTTP* realicen una redirección a otro servicio que utiliza *HTTPS*, con el objetivo de hacer utilizar al usuario un medio seguro de acceso. Este tipo de redirecciones es analizada durante esta etapa permitiendo que el recurso de la tarea *"mute"* a *HTTPS* si ese es el único cambio detectado sobre el recurso, alcanzando la etapa de error en cualquier otro caso. Una vez esta unidad realiza dichos controles de forma satisfactoria, modifica la tarea para que la misma pase a la siguiente etapa correspondiente, siendo esta *"crawling"* para el caso de *HTTP*.

9.1.2. Las tareas *crawling*

Las tareas de *Crawling* poseen un tratamiento más complejo, estando el mismo constituido por una serie de pasos que se repiten de forma cíclica sobre un conjunto de requerimientos *HTTP* que se deben realizar. El proceso comienza cuando una de estas tareas arriba a la unidad *HTTP*, la cual es atendida por el método *http_crawling_stage* quien delega el tratamiento de la tarea de forma casi inmediata, al método ***crawl*** de la clase ***Crawler***. Dentro de este método se inicia una serie de ciclos formados por dos etapas, las cuales se definen en torno al elemento ***Container*** y a los módulos ***Spider***. El *Container* es un objeto que se encarga del almacenamiento de los requerimientos que aún restan ser procesados, mientras que los *Spiders* son los encargados de la extracción de la información existente en las respuestas a dichos requerimientos. A través de la *figura 19* se intenta dar una perspectiva un poco más clara de esta idea.

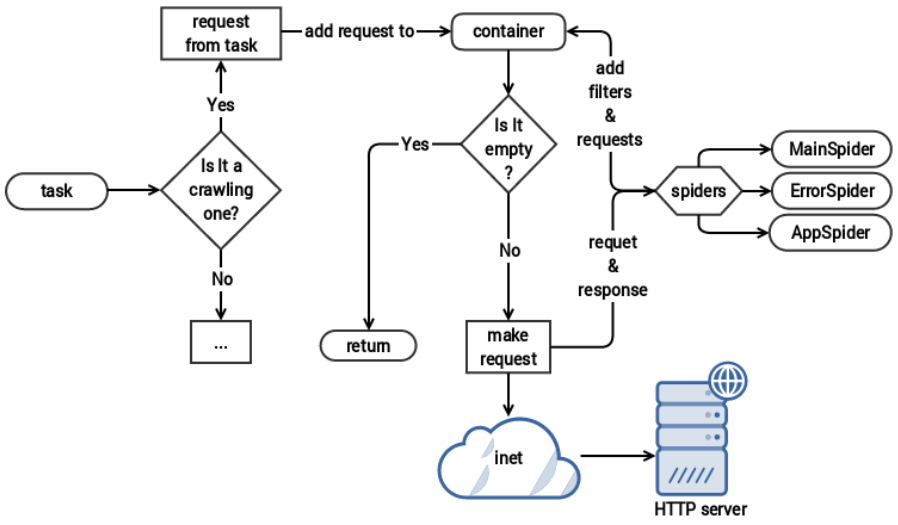

Figura 19: Clase *Crawler* de *HTTP*

Container es una clase que implementa tres mecanismos de control básicos sobre los requerimientos que almacena. Cuando la instancia de *Container* se crea, a la misma se le indica la *URL* que será la base de los requerimientos a realizar, eliminando cualquier requerimiento que se agregue y que no esté contenido en el dominio de la misma. Por ejemplo, si iniciamos el objeto para que trabaje sobre la *URL https://www.unlp.edu.ar/* e intentamos agregar la *URL http://cespi.unlp.edu.ar/index.php*, la misma sera ignorada ya que no existe como un recurso dentro de la primera. Otra restricción que la componente realiza es el filtrado a partir de expresiones regulares agregadas por el usuario. Estas permiten indicarle a *Container*, por ejemplo, que determinada ruta no debe ser ya más analizada, lo cual genera que una expresión como *"/my_path/.*"* elimine todos los requerimientos que *Container* tiene para realizar dentro de *"/my_path/"*. Esta propiedad es frecuentemente utilizada durante la detección de aplicaciones web, ya que permite ignorar todos los recursos correspondientes a la misma.

El tercer control que *Container* realiza consiste en la corroboración de que el requerimiento que se intenta agregar, no exista ya dentro del conjunto de requerimientos a realizar o en el conjunto de requerimientos realizados,

evitando de esta manera el envío de requerimientos innecesarios. Con el número de requerimientos a realizar, restringido, los resultados obtenidos a partir del envío de los mismos se encuentran disponibles para ser analizados por los *Spiders*.

Los *Spiders* son el motor en el proceso de *Crawling*, ya que de ellos nacen nuevos *requerimientos* por realizar, nuevos *filtros* por aplicar y *credenciales* que se puedan inferir. Cuando un requerimiento es obtenido del *Container* y enviado, tanto su respuesta como el requerimiento en sí, son entregados por la clase *Crawler* a cada uno de los *Spiders* que la misma conoce. Hasta el momento, los *Spiders* implementados son los siguientes

- ***MainSpider:*** Se encarga de la extracción de las *URLs* o rutas que puedan encontrarse en tags *HTML*, como ser <a>, <meta>, <form>, etc. A partir de los datos que de allí se obtienen, el *Spider* genera una serie de requerimientos con método *"get"* por defecto, retornando a la clase *Clawler* los mismos para ser luego agregados a la instancia de *Container* existente.

- ***ErrorSpider:*** Es el *Spider* encargado de los procesos relacionados al tratamiento de errores ocurridos durante la realización de un requerimiento. En este *Spider* podemos encontrar los pasos seguidos al detectarse, por ejemplo, el *error 401* en una respuesta, el cual indica que no estamos autorizados a acceder al recurso (*Unauthorized*). Para este error, el *Spider* inicia una rutina dentro de la cual se generan dos mensajes, el primero para la creación de un nuevo proceso de *Cracking* sobre la autenticación encontrada y el segundo para la creación de un nuevo proceso de *Crawling* sobre la ruta del recurso previamente solicitado. El tercer elemento creado es un filtro aplicado a la misma ruta sobre la que se creó el nuevo *Crawling*, el cual tiene por objetivo evitar que el actual proceso intente realizar más requerimientos dentro de dicha ruta. Algo a tener en cuenta de estos tres pasos detallados es que el primero de ellos debe ser realizado y esperado, ya que el proceso de *Crawling* (el segundo mensaje enviado) depende del primero, y para

establecer esa dependencia debemos conocer el ID que se le asignó a la primer tarea, el cual es devuelto en la respuesta que *Engine* envía a este primer mensaje.

- ***AppSpider:*** Este es un *Spider* que se encuentra en constante proceso de evolución ya que a través del mismo el *Swarming* realiza la detección de aplicaciones *web* que existen en un servidor remoto. En la raíz del *Swarming* se encuentra el directorio */json/apps/*, dentro del cual podemos encontrar todos los archivos *JSON* que modelan cada una de las aplicaciones que el programa es capaz de detectar. Para realizar el análisis del formato seguido por los mismos, a continuación se expone el contenido del archivo ***joomla.json***, el cual servirá como base para la realización de la explicación.

```
{
    "resources":[
        {"description":"Joomla! Login Form",

         "condition":{"tag":{"name":"meta",
                             "attrs":{"name":"generator"}},
                     "attr":"content",
                     "regex":"^Joomla! "},

         "path":{"tag":{"name":"script",
                        "attrs":{"type":"text/javascript"}},
                 "attr":"src",
                 "regex":"(.+)media\\/system\\/js\\/"},

         "seeds":["administrator/"]
        }
    ]
}
```

Cada archivo *JSON* esta compuesto por un objeto *diccionario* el cual

posee como única clave a *"resources"* la cual indexa una lista de objetos que modelan los recursos característicos de la aplicación. Cada uno de estos objetos a su vez, se encuentra constituido por una serie de campos los cuales son:

description proporciona una descripción corta de lo que el recurso es. Esta es la información que se establece en el campo *description* de *task* cuando una tarea es creada por este *Spider*.

attempts (el cual no es utilizado por *joomla.json* y no es obligatorio) representa el número de veces que una sesión es utilizada antes de ser reiniciada (eliminadas las *Cookies*). Esta opción existe debido a que algunas aplicaciones web (*Moodle* por ejemplo) permiten que el usuario realice no más de N intentos fallidos de *login* antes de pedir el reinicio de la sesión, debiéndose reiniciar la misma antes de volver a intentar una autenticación.

condition es un objeto que modela un *tag HTML* el cual debe existir para determinar que el recurso corresponde a la aplicación que se intenta detectar. El elemento buscado se encuentra constituido por el nombre del *tag* y por una serie de atributos, todo valores que se incluyen dentro del elemento ***"tag"***. Las claves ***"attr"*** y ***"regex"***, establecen sobre qué atributo (el indicado por *"attr"*) debe contemplarse la expresión regular contenida en *"regex"*. Por ejemplo, en el caso de *joomla.json*, para que *"condition"* se cumpla, el contenido *HTML* obtenido de la respuesta al requerimiento debe poseer un *tag* <*meta*> con el atributo *name* establecido a *"generator"*, y el atributo *content* poseer un valor que coincida con la expresión regular *"^Joomla! "*. Entonces un ejemplo de un *tag* que indicaría que estamos en presencia de un *Joomla!* seria:

```
<meta name="generator" content="Joomla! Open Sour..." />
```

path está compuesto y funciona de la misma manera a como lo hace *condition*, con la diferencia de que la expresión regular intenta obtener como resultado la raíz donde se encuentra la aplicación, siendo este el objetivo con el cual *path* existe.

seeds es una lista de recursos que se utilizarán, una vez se conozca la raíz de la aplicación, para generar nuevos recursos los cuales se deducen del hecho de que nos encontramos frente a la aplicación detectada. Por ejemplo, en el caso de *Joomla!*, al saber que es esa la aplicación en la que nos encontramos, inmediatamente sabemos que es probable que exista el panel de *administrador* en el *path "administrator/"* a partir de la raíz de la aplicación, lo que en muchos casos nos brindará acceso a otro recurso *web* sobre el cual realizar un proceso de *Cracking*.

Durante el inicio de *AppSpider*, todos los archivo *.json* son cargados por la clase, formando así una batería de modelos sobre los que se intenta establecer una correspondencia con la información obtenida por el *Spider*, desarrollándose de esta manera el proceso de detección de las aplicaciones.

Al final de la ejecución de cada *Spider*, todos los requerimientos y filtros generados, son agregados al *Container*, mientras que todas las credenciales obtenidas son enviadas a *Engine*, repitiéndose este proceso hasta el agotamiento de los recursos de los que *Container* dispone.

9.1.3. Las tareas *cracking*

Sobre el protocolo *HTTP* existen varias maneras a través de las cuales se puede desarrollar un proceso de autenticación, siendo el método del requerimiento (*GET*, *POST*, etc) y los *headers* del mismo, los medios más popularmente utilizados en su realización.
Para la atención de las tareas de *Cracking*, la unidad *HTTP* proporciona soporte de la autenticación ***Basic*** y a aquellas basadas en el método ***POST***,

quedando las demás relegadas a futuras implementaciones. El desarrollo de estos mecanismos es llevado a cabo a través de las clases ***Post*** y ***BasicAuth*** de la unidad, las cuales son instanciadas y ejecutadas durante la recepción de una tarea de *Cracking*. Para conocer qué mecanismo de autenticación es el que se debe utilizar en el procesamiento de una tarea en particular, la unidad analiza el valor de ***auth_scheme*** contenido en los atributos de la tarea, el cual es establecido durante su creación a los valores de *"post"* o *"basic"* según corresponda.

9.1.3.1. Basic Auth

La autenticación *Basic* es uno de los mecanismos de autenticación más simples que ofrece *HTTP*, consistiendo éste únicamente en el envío de las credenciales dentro de los *headers* de cada uno de los requerimientos que se haga, permitiendo el servidor el acceso al recurso, si las mismas son correctas. Supongamos que deseamos acceder al recurso *"/admin/index.html"* de un servidor *HTTP*, el cual se encuentra protegido por una autenticación *Basic*. Al realizar el requerimiento del recurso sin especificar credenciales, el servidor nos responderá con un mensaje de *Error 401 Not Authorized*, el cual nos indica que no estamos autorizados a acceder al mismo. Entre los *headers* de la respuesta que obtuvimos, podremos observar una entrada que nos indica el mecanismo de autenticación que debemos utilizar

```
WWW-Authenticate: Basic realm="My Server"}
```

el cual en este caso es *Basic*, pero que en otro podría haberse tratarse de ***Digest*** o ***NTLM***. Entonces, para poder realizar el requerimiento de forma correcta y que el servidor nos autorice el acceso al recurso, primero debemos conocer las credenciales que debemos utilizar para el mismo, las cuales asumiremos como *"samelat"* para el *nombre de usuario* y *"kawabonga"* para el *password*. Una vez conocemos las mismas, debemos agregar a nuestro requerimiento el mismo *header* que utilizamos para conocer el mecanismo de autenticación, componiendo este de la siguiente manera

1. Concatenamos el *usuario* y el *password* por medio de `":"`.

2. Codificamos la cadena del paso 1 utilizando para ello la variación de la codificación *Base64*, ***RFC2045***, la cual es muy similar.

3. Agregamos `"Basic "` al comienzo del valor obtenido del paso 2.

Utilizando nuestras credenciales sobre los pasos anteriores obtendremos como resultado primero la cadena `"samelat:kawabonga"` y luego la cadena codificada `"c2FtZWxhdDprYXdhYm9uZ2E="`, para finalizar con el header

```
WWW-Authenticate: Basic c2FtZWxhdDprYXdhYm9uZ2E=
```

el cual al ser agregado al requerimiento que hicimos en primer lugar, nos habilita el acceso, obteniendo *Error 200 OK* en la respuesta al mismo (además del contenido del recurso).

Toda esta implementación de *Basic* se encuentra ya soportada por el módulo *requests*, a través del uso de su parámetro ***auth***, al cual se le especifica una tupla que contiene el *nombre de usuario* y el *password* como sus elementos. La clase *BasicAuth*, encargada del procesamiento de las tareas de *Cracking* con esta autenticación, realiza el consumo de las credenciales agregado las mismas a los requerimientos por medio del parámetro *auth*, informando de estas a *Engine* únicamente si el código de estado de la respuesta es *200 (OK)*.

9.1.3.2. POST

Para introducirnos en el funcionamiento de la clase *Post*, analizaremos algunos conceptos a través de ejemplos sobre los cuales, no sólo se explica la mecánica de la autenticación, sino también el modelo seguido en la implementación.
La autenticación por *POST* consiste en la elevación de los privilegios de una sesión mantenida con el servidor *HTTP*, a partir de un requerimiento cuyos datos contienen las credenciales requeridas. La sesión se inicia con el servidor durante el envío del primer requerimiento, del cual se recibe dentro

de su respuesta, la *Cookie* que distinguirá unívocamente nuestra comunicación con el. Si bien dicha sesión nos diferencia de otros clientes, la misma no implica privilegios, siendo estos recién adquiridos cuando logramos que en el servidor se correlacione nuestra sesión con un usuario existente. Para llevar a cabo la obtención de dichos privilegios, el servidor proporciona un formulario requiriendo las credenciales necesarias para el proceso de autenticación, las cuales una vez establecidas, son codificadas y enviadas. Dicho formulario en formato *HTML*, se encuentra constituido por un *tag* ***<form>*** que contiene al menos dos entradas ***<input>***, las cuales se caracterizan por ser de tipo ***text*** una y de tipo ***password*** la otra. La entrada de tipo *text* es la encargada de contener el nombre de usuario que se debe ingresar, mientras que la de tipo *password*, contendrá su clave. Estas características tenidas en cuenta para la identificación de un *formulario de login* dentro de un documento *HTML*, son las mismas que se utilizaron en la implementación del módulo ***html***, el cual describiremos en breve. Una vez los datos se ingresaron y el envío del formulario fue iniciado, el navegador toma cada una de las entradas del mismo y extra los valores de sus atributos ***name*** y ***value***, codificando estos por medio del símbolo `"="` para cada par de valores, y uniendo al final todos los pares a través del símbolo `"&"`. Por ejemplo, supongamos que el siguiente es el formulario presentado por un servidor para la autenticación de un usuario:

```
<form action="login.php" method="POST">
    <input type="text"     name="username" >Username</input>
    <input type="password" name="password" >Password</input>
    <input type="hidden"   name="csrf"     value="7b69...51a3"/>
    <input type="submit"   name="login"    value="true"/>
</form>
```

En el mismo se observan entradas para el *nombre de usuario*, el *password*, un *token* de *CSRF* y por último el botón de envío de la información, las cuales son todas entradas utilizadas en el requerimiento. Entonces, asumiendo que las credenciales utilizadas sobre este *login* son las mismas que

las utilizadas en el ejemplo de la autenticación *Basic*, el envío codificará los datos de la siguiente manera para su requerimiento

```
username=samelat&password=kawabonga&csrf=7b69...51a3&login=true
```

Si bien este proceso se expone sobre un formulario *HTML*, el mismo puede verse implementado de igual manera sobre diferentes tecnologías como ***Flash*** o ***JavaScript***, las cuales en general[2] sólo varían en la forma de obtención de las credenciales, generando al final datos igualmente codificados a los datos expuestos.
Habiendo el servidor recibido el requerimiento y controlado que las credenciales son válidas, este asigna entonces los privilegios que la sesión debe poseer en base al usuario utilizado, finalizando así el proceso de autenticación.

A partir de todo este proceso descripto, surge como necesidad para el desarrollo de la clase *Post*, la elaboración de una estructura de datos que contenga no sólo la información necesaria para la autenticación, sino también la forma de tratar la misma, pudiendo indicar como codificarla, que entradas utilizar para las credenciales, etc. Dicha estructura es conocida por la unidad *HTTP* como ***form***, y es a través de esta que, por ejemplo, una tarea de *Crawling* informa a una de *Cracking* como desarrollar su proceso, almacenándose la misma dentro de los atributos de la tarea. Para conocer los diferentes elementos de *form*, vamos a representar el formulario del ejemplo expuesto anteriormente a través de esta estructura.

[2] https://www.google.com/search?q=JSON+authentication

```
{
    "index":0,
    "usr_field":"username",
    "pwd_field":"password",
    "fields":{
        "username":"",
        "password":"",
        "login":"true",
        "csrf":"7b69...51a3"
    }
}
```

En la misma se observan cuatro entradas principales, las cuales son ***index***, ***usr_field***, ***pwd_field*** y ***fields***. La entrada *fields* es aquel que contiene todos los elementos del formulario que deben ser codificados para ser enviados como datos del requerimiento, encontrándose en esta entrada las *claves* y *valores* de los mismos. Para identificar dentro de este conjunto, que entradas corresponden al *nombre de usuario* y al *password*, es que existen las entradas *usr_field* y *pwd_field*, conteniendo cada una de estas el nombre de la clave correspondiente. Analizando lo ocurrido en el ejemplo podemos ver que, las entradas de *fields* correspondientes a las credenciales son *"username"* y *"password"*, ya que estas son las indicadas por los valores *usr_field* y *pwd_field*. Estando estas tres entradas presentes, tenemos la información necesaria para el desarrollo del requerimiento, siendo estas claves totalmente dependientes entre sí. La entrada *index*, por otro lado, referencia el índice del formulario de *login*, dentro del conjunto de formularios de *login* que se puedan extraer de la página analizada. El objetivo de esta entrada es evitarle al usuario la necesidad de especificar todos los elementos del formulario, existiendo como restricción de este mecanismo el que solo funciona para páginas *HTML*. Es normal dentro del *Swarming*, que una unidad que crea una tarea de *Cracking* que incluye una referencia a un formulario, solo indique su índice, encargándose la clase *Post* de inferir los valores de *usr_field*, *pwd_field* y *fields* restantes, a partir del formulario indicado. El problema

surge cuando el *login* que se desea representar no existe en forma de *HTML* sino a través de otra tecnología como *Flash*. En este caso el usuario debe indicar a partir de los demás campos de *form* los valores que se deben utilizar, sin poder especificar un índice de formulario ya que el mismo no existe. Para estas situaciones, el usuario dispone de herramientas que permiten analizar los requerimientos hechos por un navegador web, a partir de los cuales se pueden extraer los valores que deben ser utilizados.

El módulo encargado de la extracción de los formularios de *login* a partir del código *HTML* que se le proporciona, es el módulo *html*, el cual por medio de su método ***get_login_forms*** realiza una devolución del conjunto de todos los formularios de *login* que fueron identificados a partir de los criterios expuestos anteriormente. Es así entonces como el método *Post*, siempre que se le indique un valor por medio de *index* en la estructura *form*, podrá realizar la utilización del formulario para la obtención de la información que requiere.

10. Conclusión

Si bien la aplicación aún no presenta todas las capacidades que se pretenden, plantea un escenario el cual no reviste dificultad para la extensibilidad, dejando establecidos los elementos básicos para la fácil incorporación de nuevas características o funcionalidades. Las limitaciones que la aplicación posee se encuentran fuertemente ligadas al tiempo que fue requerido para lograr obtener una arquitectura que presentase capacidad de escalar sobre todos los requerimientos esperados (paralelismo, buena comunicación entre unidades, soporte dinámico de unidades, etc), impidiendo que se realice un avance más sustancial en materia de soporte de protocolos. Más allá de estos inconvenientes, la aplicación logra sus objetivos al implementar de forma completa todos los mecanismos básicos, delegando como trabajo a futuro el desarrollo de las componentes restantes.

10.1. El núcleo de la aplicación

Varios elementos debían ser implementados antes de poder dar inicio al desarrollo de las unidades encargadas de llevar a cabo los ataques. A través de la implementación de las unidades críticas del *Swarming* (*Engine*, *Core* y *Executor*), se logró que el mismo tuviese el comportamiento esperado en cuanto a ejecución de tareas en paralelo, intercambio de información y generación de trabajo a desarrollar, restando a partir de la finalización de esta etapa, únicamente el desarrollo de las unidades que harían uso de estas características.

10.2. Las unidades Livianas

La incorporación de unidades livianas es el factor que seguirá en crecimiento de ahora en mas sobre la aplicación, siendo este el aspecto que más marcará el grado de utilidad de la misma. La unidad *HTTP* logró desarrollar todas las actividades pretendidas, siendo de todas manera esta una unidad que seguramente permanezca en constante crecimiento debido a su complejidad.

11. Trabajo a Futuro

En lo que sigue se presentan características que se pretende implementar a futuro sobre el *Swarming*, algunas habiendo sido contempladas como objetivos desde el comienzo de la aplicación y otras que se pensaron o fueron sugeridas durante el desarrollo de la misma.

11.1. Soporte de otros protocolos

Es imperante que la aplicación soporte otros protocolos además de *HTTP*, no solo porque este es el objetivo principal de la aplicación sino también porque existen otras funcionalidades que dependen de este aspecto. Por ejemplo, el reconocimiento de dispositivos de red, está estrechamente ligado a los protocolos que la aplicación soporta, impactando de forma directa el número de protocolos sobre el grado de reconocimiento que la aplicación pueda hacer de los diferentes elementos que existen en el contexto.

11.2. Reconocimiento de Dispositivos de red

El reconocimiento de dispositivos de red, como el soporte de otros protocolos, es un objetivo que existía como una de las ideas principales de la aplicación. La idea detrás de este proceso es generar *usuarios* y *passwords* por defecto para los *teléfonos*, *routers*, *switches* y demás dispositivos de red que la aplicación pueda llegar a reconocer dentro de la red auditada.

11.3. Reconocimiento de Aplicaciones Web

El reconocimiento de aplicaciones de web es un aspecto que se debe seguir expandiendo de la unidad *HTTP*, ya que esta unidad es una de las principales innovaciones que la aplicación presenta y como tal, debería ser esta la unidad mejor soportada de todo el conjunto.

11.4. Generación de diccionarios

En muchas ocasiones ocurre que se agotan las credenciales a ser probadas sobre los servicios, siendo el único recurso que resta probar, la generación de diccionarios por parte del atacante, en particular de *passwords*. Existen en *internet* herramientas como *mutator* [15] las cuales realizan mutaciones sobre una palabra particular, permitiendo realizar un último intento poco sofisticado, antes de dar por finalizada la auditoría.

11.5. Incorporación de Scrapers

Los ***Scrapers*** son un nuevo tipo de tarea que se pretende implementar a futuro con la intención de asumir el rol que hoy tienen en parte asignados los *crawlers*, que es el de recolectar información. La idea del *Scraper* es la de una tarea que, habiéndose conseguido acceso a un sistema, conoce la manera correcta de extraer información útil del mismo. Por ejemplo, si se consigue acceso a una Base de datos, un *scraper* debería ser capaz de reconocer tablas de *usuarios* y *passwords* y extraer la información de las mismas, ocurriendo algo similar para los casos de accesos logrados a sistemas de archivos. Esto permitirá que el *Swarming* no solo aproveche la información que se puede obtener de forma inmediata del contexto, sino que también podrá extraer información a partir de los privilegios que se logren durante la auditoría.

11.6. Agregado de Claves Públicas como credenciales

Las claves públicas son otro mecanismo muy comúnmente utilizado en la autenticación de algunos protocolos, siendo el caso más interesante en este contexto, el del protocolo *SSH*. Una vez la aplicación posea soporte para este protocolo, un buen paso a dar sería el realizar la implementación del soporte a la autenticación por credenciales, siendo una potencial fuente de estas los *scrapers*.

Referencias

[1] *Análisis comparativo hecho por la gente de Hydra.* URL: https://www.thc.org/thc-hydra/network_password_cracker_comparison.html.

[2] *Análisis comparativo hecho por la gente de Medusa.* URL: http://foofus.net/goons/jmk/medusa/medusa-compare.html.

[3] *Beautiful Soup - Library for pulling data out of HTML and XML files.* URL: http://www.crummy.com/software/BeautifulSoup/.

[4] Judith Bishop. «Factory Method Pattern». En: *C# 3.0 Design Patterns*. Cap. 5, pág. 110.

[5] *Bootstrap - Framework para el desarrollo Web.* URL: http://getbootstrap.com/.

[6] *CherryPy - A Minimalist Python Web Framework.* URL: http://www.cherrypy.org/.

[7] *Complejidad temporal de las estructuras de datos en Python.* URL: https://wiki.python.org/moin/TimeComplexity.

[8] *Desempaquetado de diccionarios en Python.* URL: https://docs.python.org/2/tutorial/controlflow.html#unpacking-argument-lists.

[9] Nitesh Dhanjani, Billy Rios y Brett Hardin. «Hacking: The Next Generation». En: *Brute-Forcing Your Way In*. Cap. 3, pág. 74.

[10] *Fail2ban HomePage.* URL: http://www.fail2ban.org/wiki/index.php/Main_Page.

[11] *jQuery - Librería JavaScript.* URL: https://jquery.com/.

[12] Mark Lutz. *Learning Python*. 5.ª ed.

[13] *Medusa - Parallel Network Login Auditor.* URL: http://foofus.net/goons/jmk/medusa/medusa.html.

[14] *MongoDB - DBMS no-relacional.* URL: http://www.mongodb.org/.

[15] *Mutator - wordlist mutator with hormones*. URL: https://bitbucket.org/alone/mutator.

[16] *Ncrack - high-speed network authentication cracking tool*. URL: http://nmap.org/ncrack/.

[17] Shelley Powers. «The Same-Origin Security Policy». En: *Learning JavaScript: Add Sparkle and Life to Your Web Pages*. Cap. 10, pág. 222.

[18] *Requests - HTTP for Humans*. URL: http://docs.python-requests.org/en/latest/.

[19] *SQLAlchemy - The most widely used stand-alone ORM*. URL: http://www.fullstackpython.com/databases.html.

[20] *SQLAlchemy - The Python SQL Toolkit and Object Relational Mapper*. URL: http://www.sqlalchemy.org/.

[21] *SQLite3 - DBMS SQL basado en archivos*. URL: http://www.sqlite.org/.

[22] The Hacker's Choice Team. *THC-Hydra*. URL: https://www.thc.org/thc-hydra/.

[23] *The netfilter.org "iptables" project*. URL: http://www.netfilter.org/projects/iptables/index.html.

[24] Gaston Traberg. *Repositorio GIT de codigo del Swarming*. URL: https://github.com/samelat/swarming/.

[25] Andrew Whitaker y Daniel P. Newman. «Penetration Testing and Network Defense». En: *Brute Force Attacks*. Cap. 9, pág. 209.

[26] Hanqing Wu y Liz Zhao. «Cross-Site Request Forgery». En: *Web Security: A WhiteHat Perspective*. Cap. 4, pág. 123.

[27] Nicholas C. Zakas. «Ajax - Data Formats - JSON». En: *High Performance JavaScript*. Cap. 7, pág. 138.