Escuela
Politécnica
Superior

Diseño e Implementación de un Analizador de Vulnerabilidades

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:
Cristian García Romero

Tutor/es:
Francisco José Mora Gimeno

Junio 2020

Universitat d'Alacant
Universidad de Alicante

Agradecimientos

Han habido momentos duros en estos últimos cinco años de grado. Momentos en los que incluso la posibilidad de abandonar asomaba, pero tú has sido mi gran apoyo. Por ello, Rossmery, quiero agradecerte tu apoyo incondicional, no solo en estos últimos años, sino desde que nos conocemos, ya que han sido los mejores años de mi vida. Muchas gracias por tenerme esa paciencia infinita, es admirable. Este trabajo va para ti, mi vida.

Mis padres son quienes más me han soportado y apoyado a dosis iguales. Muchas gracias, pues siempre habéis estado ahí para ayudarme en todo lo que he necesitado y más. Os quiero.

Me considero una persona introvertida. Aún así, han habido compañeros en estos últimos años con los que he compartido experiencias y nos hemos apoyado mutuamente. Muchas gracias. Sois maravillosos y maravillosas.

Brian, Brinxer, desde que nos conocimos te convertiste en mi mejor amigo, y aunque hayan habido momentos en los que he estado más ausente, siempre estaré ahí. Gracias por tu apoyo y amistad. *Troll bot first*.

Antes de llegar a la Universidad de Alicante, estuvo el instituto, el cual ha sido, sin duda, la peor experiencia de mi vida por diferentes factores. Esto consiguió hacer que valorara como un tesoro mi paso a la universidad. Aún y todo, me gustaría destacar la gran labor de algunos de los profesores que en su momento me ayudaron y no abandonaron aún con la mala actitud que mostré. Gracias Bea, Pascual, Jaume, Alicia, Vivi, Maties y Joaquín por vuestra excelente labor. Por supuesto, una mención muy especial a Antonio Coloma, mi profesor de informática, quien empezó mi instrucción en el mundo de la informática. Gracias, Antonio, pues no habría podido llegar hasta aquí sin tu ayuda y apoyo.

En mi paso por la Universidad de Alicante me he encontrado, en su gran mayoría, con excelentes docentes en el Grado en Ingeniería Informática. Me gustaría agradecerles su gran labor docente, pues nos permite a los estudiantes obtener una formación de gran calidad gracias a sus esfuerzos. Algunos de estos grandes docentes son Jerónimo Mora Pascual, Patricia Compañ Rosique, Eva Gómez Ballester, José Ángel Berná Galiano, Antonio Ferrández Rodríguez, Cristina Pomares Puig, Antonio Jimeno Morenilla, Francisco Mora Lizán, María Isabel Alfonso Galipienso, Pilar Arques Corrales, Francisco José Gallego Durán, Juan Ramón Rico Juan, Francisco Moreno Seco y José Vicente Berná Martínez. Un agradecimiento especial a Francisco José Mora Gimeno, tutor del presente trabajo y gran docente, pues no solo a mí, sino también a muchos otros compañeros ha inspirado y dirigido hacia el campo de la ciberseguridad a través de una clara motivación mostrada en sus clases que ha sabido transmitir de manera excelente. La ayuda brindada para poder superar ciertas dificultades que se han presentado conforme se desarrollaba este trabajo es inestimada.

"Mi objetivo principal al comprometer sistemas era
la curiosidad intelectual, la seducción de la aventura."

— Kevin Mitnick

"No tengas lástima de los muertos, Harry.
Ten pena de los vivos, y, sobre todo, de aquellos que viven sin amor."

— Albus Percival Wulfric Brian Dumbledore

Resumen

En el presente documento se detalla el desarrollo de BOA, un analizador de vulnerabilidades de propósito general. Las partes relevantes del analizador se detallan en los capítulo 6 y 7, donde se comenta el diseño e implementación, respectivamente, del analizador. Por último, se detallan los resultados obtenidos en el capítulo 8 y las conclusiones en el capítulo 9.

El diseño que se ha obtenido ha permitido dotar al analizador de una gran flexibilidad y funcionalidad. Partiendo de unos objetivos iniciales, se describen unos objetivos de diseño que conducen a una arquitectura modular. El resultado obtenido permite a un usuario adaptar el analizador de vulnerabilidades BOA a las necesidades del mismo.

Partiendo del diseño, la implementación que se obtiene es la materialización de los objetivos. Algo a destacar de la solución es la sencillez con la que se puede hacer uso del analizador, pues con una configuración mínima, se puede empezar a utilizar. El usuario tiene la libertad de extender el comportamiento base y de implementar sus propios módulos con el fin de detectar ciertas vulnerabilidades sobre diferentes ficheros de código con independencia del lenguaje de programación, pues a través de la interoperabilidad con otros analizadores, se obtiene toda la información necesaria.

Índice

Índice de figuras

Capítulo 1

Introducción

En sus inicios, la informática no se planteaba como una ciencia donde la seguridad, o ciberseguridad como es llamada en su ámbito, fuera algo necesario. Debido a esto, los primeros sistemas eran vulnerables, y la problemática de ello se hizo notoria con el gran primer incidente que ocasionó pérdidas monetarias millonarias en los inicios de la red: el gusano Morris. El 2 de Noviembre de 1988 empezó la catástrofe donde se estima que el gusano Morris infectó 6000 de los 60000 equipos que había la red en esos momentos, ocasionando una pérdida estimada de 98 millones estadounidenses. Lo que se dedicaba a hacer este *malware* (programa informático con fines maliciosos) era replicarse y propagarse a sí mismo (por esta característica es llamado ”gusano”) a través de la red, y una vez detectaba que había llegado a otra máquina, debía de parar, pero no lo hacía sino que continuaba autorreplicándose hasta agotar los recursos de las máquinas y dejarlas inservibles. Aunque catastrófico, fue una lección que hizo notar la falta de ciberseguridad y la necesidad de la misma [1].

Desde el gran primer incidente relacionado con la seguridad informática, la cantidad de problemas relacionados con equipos informáticos no ha parado de aumentar como podemos observar en la figura 1.1. Estos incidentes ocasionan no solo pérdidas multimillonarias, sino que también perjudican a empresas y usuarios, ya que es un hecho que hoy en día dependemos de la informática porque está en nuestro día a día, ya sea por trabajo, ocio, comunicación o cualquier otro motivo. Los sistemas informáticos son relevantes, y los incidentes que dejan inoperativos dichos sistemas, son perjudiciales.

Los incidentes informáticos ocurren debido a las **vulnerabilidades**, ya que en los incidentes se hace uso de estas para alcanzar los fines maliciosos que se persigan, por lo que será necesario descubrir las vulnerabilidades para reducir el número de incidentes. Una vulnerabilidad *software* puede ser definida como una instancia de un error en la especificación, desarrollo o configuración de modo que su explotación puede llevar a la violación de las políticas de seguridad definidas para dicho *software* [2]. Con el fin de poner en perspectiva cuales son las vulnerabilidades más recurrentes, en la tabla 1.1 se pueden observar aquellas que han sido reportadas con más frecuencia en los últimos años.

Hay herramientas que intentan buscar vulnerabilidades con el fin de conseguir que los programas informáticos sean más fiables, lo cual no implica que un sistema sea seguro. Al fin y al cabo, la seguridad total no existe, pero podemos tender a ella conforme más aumentemos su fiabilidad. Para alcanzar este fin, hay que ser metódicos y ser conscientes de la realidad, y esa realidad respecto a la ciberseguridad es que es muy difícil para una persona analizar sistemas grandes y complejos. Por este motivo se crearon estos analizadores,

estas herramientas, que nos ayudan a centrarnos en aquellas partes de los sistemas que, por algún motivo, hacen que se detecte alguna anomalía que puede llevarnos a encontrar alguna vulnerabilidad, y si detectamos dicha vulnerabilidad, podremos evitar incidentes de cara al futuro. Algunas de estas herramientas podrían ser OpenVAS[1] o Intruder[2]. Estas herramientas utilizan un conjunto de técnicas, más o menos formales, con el fin de poder analizar los sistemas y encontrar indicios de lo que podría ser una potencial vulnerabilidad.

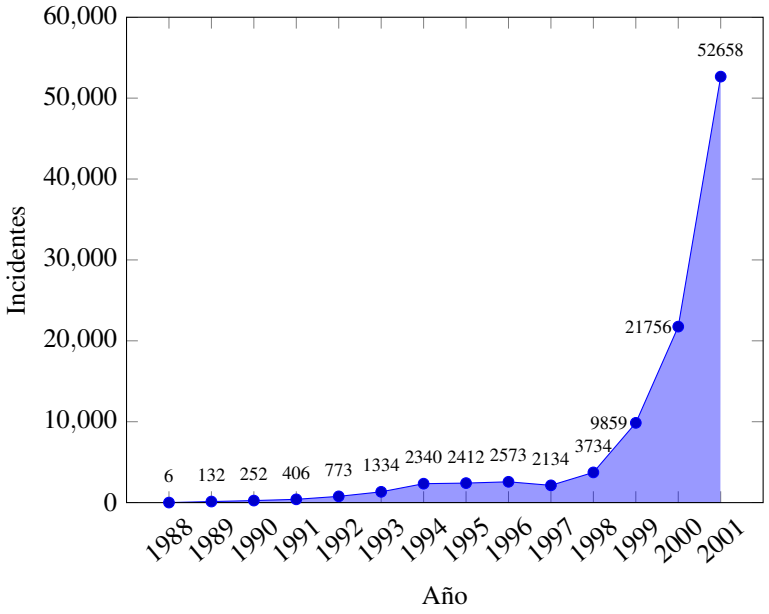


Figura 1.1: Incidentes reportados entre 1988 y 2001 [3]

Vulnerabilidad	Casos Reportados
Buffer Overflow	6756 (18.1 %)
Cross-Site Scripting (XSS)	6220 (16.7 %)
Permisos, privilegios y accesos	4661 (12.5 %)
SQL Injection	3828 (10.3 %)
Validación de Entradas	3763 (10.1 %)
...	...
Total	37325

Tabla 1.1: Vulnerabilidades reportadas entre 2008 y 2016 [4]

Una posible manera de automatizar la detección de vulnerabilidades es el uso de analizadores de vulnerabilidades, y, aunque no está exenta de problemas, ayuda en la labor del experto en ciberseguridad a realizar la tarea de la búsqueda de vulnerabilidades de una manera mucho más sencilla debido a que los grandes sistemas pueden alcanzar a tener millones de líneas de código fácilmente, lo cual puede llegar a ser inmanejable incluso para un equipo de personas, pero si de alguna manera tenemos una forma de saber dónde mirar, la tarea puede pasar de inmanejable a factible e incluso sencilla, todo dependerá de la calidad del analizador. Dicha calidad dependerá de buscar un equilibrio en el funcionamiento de dichos analizadores, ya que si reportan una gran porción del sistema bajo análisis, lo más seguro es que esté fallando y reportando falsos positivos (i.e. resultados tomados como positivos que en realidad no lo son), y eso puede llevarnos al problema de partida:

[1] Sitio oficial de OpenVAS: `https://www.openvas.org/`
[2] Sitio oficial de Intruder: `https://www.intruder.io/`

muchos reportes nos lleva a analizar gran porción del sistema, lo cual puede ser inmanejable dependiendo del tamaño del mismo. Dentro del equilibrio, tenemos que intentar minimizar a su vez los falsos negativos (i.e. resultados no reportados y que en realidad deberían de reportarse), ya que tener una gran cantidad de los mismos nos llevaría a una herramienta que no nos sirve en la tarea de facilitar la automatización del descubrimiento de vulnerabilidades. Es un equilibrio delicado y que no siempre es fácil conseguir pero que hay que alcanzar en mayor o menor medida para tener una herramienta de calidad y que sea de utilidad.

Debido a todo lo anteriormente expuesto, en el presente trabajo se pretende realizar el diseño e implementación de un analizador de vulnerabilidades con el fin de que facilite la labor de identificación de dichas vulnerabilidades. El resultado es **BOA** (*Buffer Overflow Anihilator*), un analizador de vulnerabilidades de propósito general basado en reglas.

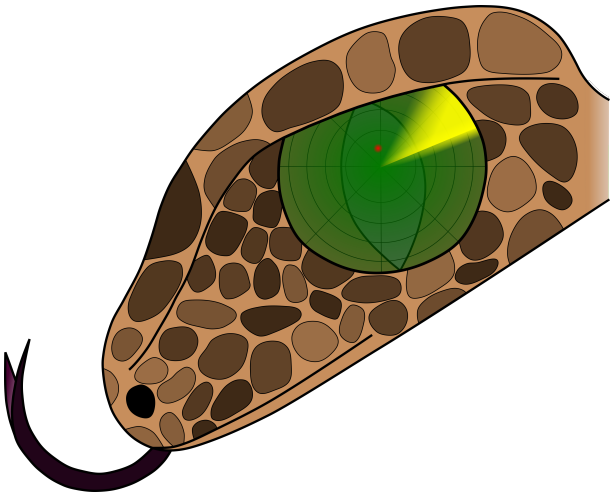


Figura 1.2: Icono de BOA

Capítulo 2

Motivación y Objetivos

De entre todas las palabras que podemos pensar a la hora de hablar de la informática, una de ellas es **automatización**. La automatización es algo que la informática hace de manera genial siempre que no se trate de procesos demasiado complejos que requieran de intervención humana, aunque de esos procesos ya está encargándose la Inteligencia Artificial (e.g. diagnósticos médicos [5, 6]), a veces incluso mejorando los resultados de los especialistas [7]. Lo normal es que a nadie le guste realizar tareas repetitivas, ya que no es algo que a los humanos nos parezca atractivo, e incluso puede llegar a afectar a nuestra salud mental [8], y debido justamente a esto hacemos uso de la automatización para evitar realizar tareas repetitivas, tediosas, monótonas, que poco o nada aportan por el hecho de que una persona sea quien las realice. Un ejemplo de ello podemos encontrarlo en la figura 2.1.

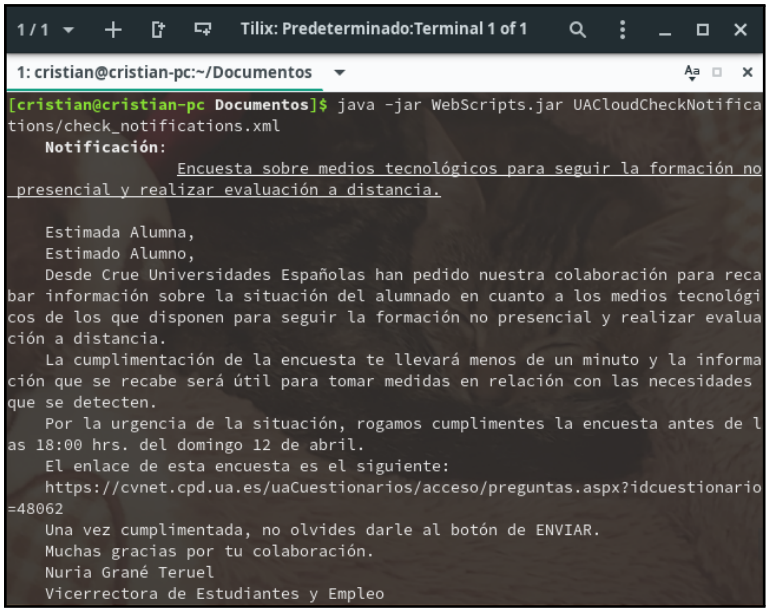


Figura 2.1: Automatización en la obtención de notificaciones de UACloud con WebScript[1]

Por otro lado, la ciberseguridad tiene las siguientes características que, en lo personal, hace que me guste tanto este campo: requiere de un gran conocimiento técnico, es apasionante y, en muchas ocasiones, hay

[1] Repositorio de WebScript: `https://github.com/cgr71ii/WebScript/`

que hacer uso de la creatividad (esta última característica es lo que hace tan complejo que se esté aplicando la inteligencia artificial en este campo). Junto a la ciberseguridad, la automatización es el otro gran área que podría decir que me gusta (quizá sea porque se dice que los informáticos somos vagos, y no soy la excepción), y sea la razón de que haya escogido la tarea de crear un analizador de vulnerabilidades, ya que este realiza la tarea de buscar vulnerabilidades de manera automática uniendo ambas áreas.

Por lo general, a día de hoy el trabajo de la búsqueda de vulnerabilidades sigue siendo en gran parte artesanal aunque existan analizadores que obtengan muy buenos resultados, y ya sea porque no se utilizan o porque no detectan todas las vulnerabilidades que nos gustaría, en muchas ocasiones se detectan las vulnerabilidades cuando alguien las explota. Otra posible razón puede ser que la base de conocimiento de estos analizadores se queda obsoleta y no detecte nuevas vulnerabilidades o variantes de las conocidas. Con tal de evitar estas situaciones, nuestro analizador de vulnerabilidades busca cumplir los siguientes objetivos:

- Automatizar el análisis de vulnerabilidades: el propósito es evitar analizar todo un sistema de manera manual, lo cual podría llegar a ser inviable. Es el propósito principal de este trabajo.
- Diseñar una arquitectura modular: el propósito de una arquitectura modular es poder ampliar la base de conocimiento de las diferentes vulnerabilidades y poder tener un analizador que pueda adaptarse a las necesidades actuales y futuras de la ciberseguridad principalmente, pero también nos permite poder evaluar diferentes técnicas dentro de la misma herramienta, aislar los errores, concatenar los resultados de un módulo con los de otros, personalización, etc.
- Permitir la personalización: el propósito es dar la posibilidad, a quien lo necesite, de personalizar el analizador a las necesidades del experto, usuario u organización. Las necesidades de un individuo a otro pueden ser muy cambiantes, y lo que para alguien es suficiente, para otra persona puede no serlo.
- Facilitar la ampliación: el propósito es, aprovechando el diseño de una arquitectura modular, que la base de conocimiento puede ser fácilmente ampliable sin necesidad de realizar un procedimiento laborioso.
- Implementar una funcionalidad mínima: el propósito es que el analizador sea de utilidad cuando vaya a utilizarse y no sea una herramienta en la que haya que hacer un trabajo inicial laborioso para poder empezar a ser útil.
- Admitir diferentes perfiles de configuración: el propósito es intentar adaptar el comportamiento a las necesidades sin necesidad de realizar modificaciones en el sistema, pero esto dependerá del grado de libertad que los diferentes módulos ofrezcan en su configuración. El permitir cambiar de un perfil de configuración a otro puede hacer que un módulo sea de utilidad en muchos ámbitos y no sea necesaria su modificación cuando las necesidades son mínimas respecto al comportamiento original por defecto.
- Devolver una descripción sencilla y concisa de los resultados: el propósito es diferenciarse de aquellas herramientas que sus resultados son, como poco, crípticos. Los resultados son una de las partes esenciales del analizador y que tienen que ser de ayuda para el experto o usuario que lo esté utilizando.
- Documentar el desarrollo: el propósito es facilitar la comprensión del analizador a quien quiera ampliar alguna funcionalidad, modificar su diseño o curiosear. Entender el sistema para el que se quiere añadir un nuevo módulo puede ser esencial según las necesidades.
- Evaluar el funcionamiento: el propósito es evaluar el correcto funcionamiento de la mínima funcionalidad de los módulos que el analizador implementa.

Capítulo 3

Metodología

Algo esencial a discutir para la elaboración de cualquier estudio y/o proyecto es la metodología a seguir. En el caso de proyectos de ingeniería, la metodología a seguir suele ser la ingenieril, la cual no es única y hay muchas variantes definidas debido a que dicha metodología suele adaptarse a las necesidades de los proyectos. En la figura 3.1, podemos ver una metodología ingenieril genérica que intenta ser una traducción del método científico.

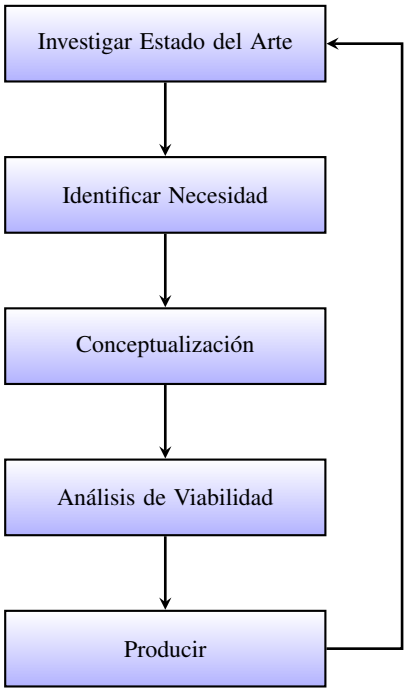


Figura 3.1: Metodología ingenieril genérica (asociación con el método científico) [9]

En el caso del presente trabajo, hemos seguido una metodología ingenieril, la cual se muestra en la figura 3.2. La metodología empleada se adapta a una metodología iterativa e incremental donde construimos versiones del producto y vamos ampliando secciones concretas mediante iteraciones, no todo a la vez. Hemos empleado esta metodología porque se adapta a nuestras necesidades, las cuales eran conseguir un desarrollo rápido y que fuera creciendo y evolucionando conforme se añadían las funcionalidades necesarias respecto a los objetivos, los cuales se especificaron en el capítulo Motivación y Objetivos.

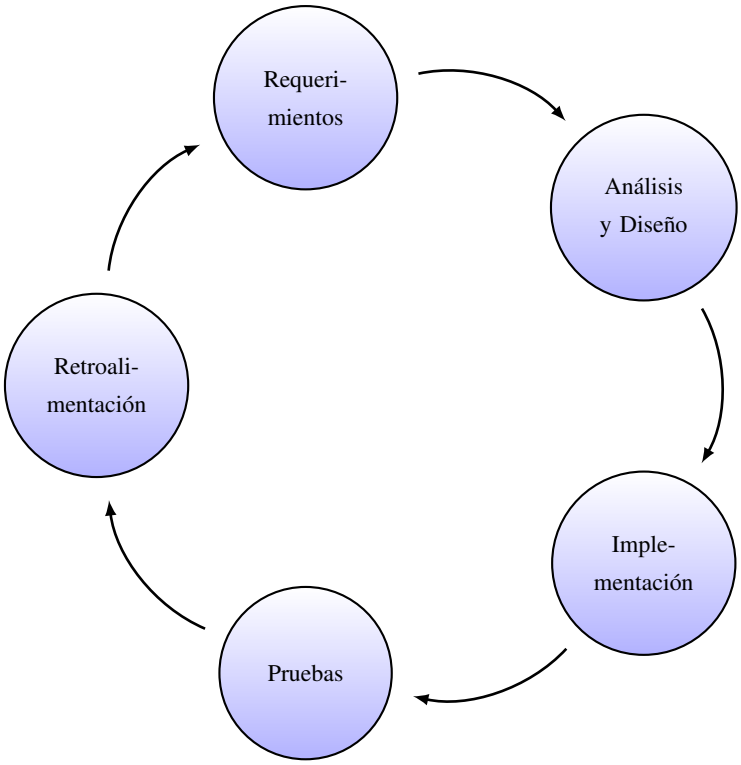


Figura 3.2: Metodología aplicada en BOA

Al utilizar una metodología iterativa e incremental, se nos ha hecho muy fácil utilizar diferentes versiones para el proyecto, donde cada versión es una versión completa del analizador donde diferentes partes del mismo se ha mejorado. El desarrollo de cada una de estas versiones se puede ver como una iteración. En la figura 3.3 tenemos un ejemplo de cómo funciona una metodología iterativa e incremental.

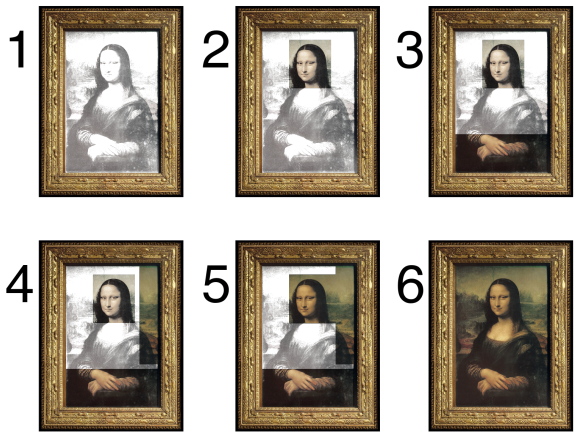


Figura 3.3: Analogía de metodología iterativa e incremental

A continuación, explicamos los elementos de la figura 3.2:

- Requerimientos: de acuerdo a los objetivos definidos para el analizador, obtenemos unos requerimientos que tienen que estar presentes en el diseño. Si además tenemos retroalimentación, la utilizamos junto a los objetivos para refinar los requerimientos.
- Análisis y Diseño: a partir de los requerimientos, los analizamos y realizamos un diseño que cumpla los

requerimientos de la mejor manera posible.

- Implementación: partiendo del diseño, realizamos la implementación del mismo.
- Pruebas: una vez implementado el diseño, realizamos pruebas de las partes finalizadas o que tengan una funcionalidad mínima.
- Retroalimentación: una vez tenemos los resultados de las pruebas, sabemos si el diseño inicial que se realizó cubre las necesidades mínimas y necesarias para cumplir con los objetivos. En la medida que se cumplan los objetivos, la retroalimentación será positiva o negativa y será necesario replantearse la solución actual, lo cual llevaría a volver a diseñar si fuera necesario.

Capítulo 4

Estado del Arte

4.1 Introducción

Este trabajo intenta cubrir dos aspectos: las vulnerabilidades *software* y el diseño e implementación de un analizador. Tanto para las vulnerabilidades como para desarrollar el analizador, necesitamos conocer los elementos principales. En el caso de las vulnerabilidades, necesitamos conocer cuales son las más explotadas y entender cómo se explotan. Para el analizador necesitamos conocer cuales son las principales técnicas que se emplean para detectar vulnerabilidades. En este capítulo vamos a cubrir exactamente lo descrito: principales vulnerabilidades que hoy en día son explotadas y técnicas empleadas para su detección.

4.2 Vulnerabilidades

Las vulnerabilidades *software* nos han acompañado desde los inicios de la informática, y en los últimos tiempos la preocupación por las mismas ha ido en aumento en todos los ámbitos debido a los daños materiales, económicos e incluso a la reputación que pueden ocasionar. Los ciberdelincuentes y profesionales de la ciberseguridad (entiéndase que hablamos de aquellos que, éticamente, se dedican a la ciberseguridad y no buscan causar perjuicios en base a sus conocimientos) estudian las vulnerabilidades con el fin de, explotarlas en el caso de los ciberdelincuentas, y darles una solución, mitigar los daños o prevenir una posible incidencia en el caso de los profesionales de la ciberseguridad.

El día en el que no hayan vulnerabilidades de las que preocuparnos será el día en el que no hayan sistemas relevantes, porque mientras hayan objetivos jugosos para los cibercriminales, habrá alguno que utilice el ingenio con el fin de poder atacarlo de alguna forma. Las vulnerabilidades siempre van a estar ahí porque es imposible deshacernos de ellas, ya que, normalmente, son el efecto negativo de algún beneficio. Y es que vulnerabilidades tan dañinas como puede ser un *buffer overflow*, es justamente lo que a unos cuantos lenguajes (e.g. C, C++) les da una ventaja de eficiencia frente a otros lenguajes de programación.

Con el fin de poder entender cómo funcionan algunas de las vulnerabilidades más explotadas hoy en día, y entender bien a qué nos enfrentamos a la hora de realizar un analizador de vulnerabilidades, vamos a estudiar las siguientes vulnerabilidades:

- *Buffer Overflow*: una de las vulnerabilidades más explotadas de todos los tiempos y una de la más temidas debido a su gran potencial dañino. Ha sido principalmente explotada en servicios ofrecidos por los sistemas operativos, y en los últimos tiempos ha sido la razón de grandes dolores de cabeza en sistemas tan inmensos como el núcleo del sistema GNU/Linux.

- *Cross-Site Scripting* (XSS): vulnerabilidad común en entornos web. Muy temida debido a que permite, dependiendo de las condiciones, hacer cambios permanentes en el comportamiento de un sitio web. Han habido casos de *bug hunters* (profesionales que se dedican a la búsqueda de vulnerabilidades) que han encontrado este tipo de vulnerabilidad en sitios web y se les ha recompensado con cantidades económicas nada despreciables, como el caso de Facebook en Julio de 2015, en el que el investigador en ciberseguridad apodado *fin1te* encontró una vulnerabilidad XSS en el sitio y lo reportó, y recibió la cantidad de $7500.

- *SQL Injection*: vulnerabilidad muy común en entornos web. Permite obtener y/o modificar información de la base de datos, y debido a que la base de datos es el activo más importante debido a que contiene toda la información del sistema, puede comprometerlo en su totalidad. El alcance de esta vulnerabilidad no es solo el acceso a información, lo cual dependerá del contexto del sistema, sino que también puede permitir traspasar la autenticación y/o autorización del sitio web, entre otras posibles acciones. Dependerá del contexto de la aplicación web el alcance de la vulnerabilidad.

El motivo de estudiar estas vulnerabilidades y no otras es debido a que son de las más conocidas y comunes hoy en día. En el caso de las vulnerabilidades XSS y *SQL Injection* están presentes en el OWASP[1], en su documento *top 10*[2], con los puestos séptimo y primero, respectivamente, en los datos del año 2017[3] (se trata de la última actualización en los momentos de escribir estas palabras debido a que los datos se actualizan cada 3 años). Por otro lado, vamos a hablar de la vulnerabilidad *buffer overflow* porque es la más conocida en términos de sistemas operativos y que más se ha explotado a lo largo de los años. La vulnerabilidad *buffer overflow* será la que comentaremos en profundad debido a que, posteriormente, la utilizaremos para realizar las pruebas e implementación de módulos concretos sobre esta vulnerabilidad, por lo que nos conviene conocerla en profundidad. Las otras vulnerabilidades se comentarán más superficialmente.

4.2.1 *Buffer Overflow*

La vulnerabilidad *buffer overflow* (también se la conoce por *buffer overrun*, *stack overrun* o *stack smashing*, entre otras denominaciones) ha sido una de las más explotadas y peligrosas en la historia de la ciberseguridad. Los efectos de una explotación exitosa de esta vulnerabilidad, como mínimo, causan una denegación de servicio, y en la mayoría de ocasiones se puede llegar a ejecutar código arbitrario, lo cual podría llegar a alcanzar el control total de la máquina. En esta sección vamos a ver qué es esta vulnerabilidad, cómo se explota, unas cuantas variantes de la técnica original y casos recientemente descubiertos. Para ello, vamos a utilizar de ejemplo el sistema operativo GNU/Linux y la arquitectura x86 de 32 bits por ser el escenario más estudiado y sencillo de entender, aunque la extensión a 64 bits sigue permitiendo la explotación de la mayor parte de vulnerabilidades, pero la complejidad aumenta.

[1] Sitio oficial del Open Web Application Security Project (OWASP): `https://owasp.org/`

[2] Top 10 del OWASP: `https://owasp.org/www-project-top-ten/`

[3] Top 10 del OWASP publicado en 2017: `https://github.com/OWASP/Top10/blob/master/2017/OWASP%20Top%2010-2017%20(en).pdf`

Diseño de la Memoria en Sistemas GNU/Linux

Los sistemas operativos tienen que hacer una gestión de la memoria, la cual es un recurso esencial y que tiene que estar muy optimizado para evitar que el rendimiento del sistema sea pésimo. La memoria que el sistema operativo gestiona es la memoria física, y cuando nos referimos a la memoria física, nos estamos refiriendo a la memoria RAM. Cuando un sistema se inicia, tiene que utilizar esta memoria para almacenar todo lo que necesita con el fin de poder llevar a cabo su funcionamiento normal, y para ello, tiene que saber dónde están localizados ciertos elementos. Dicha localización es gestionada internamente por el núcleo, por lo que no tenemos que preocuparnos por ello.

Por otro lado, los procesos necesitan también saber dónde están localizados ciertos elementos, y desde que tenemos sistemas operativos que ejecutan varios procesos al mismo tiempo, no podemos permitir que un proceso pueda acceder a la memoria de otro proceso, y para ello, cada proceso se ejecuta de manera aislada en un *sandbox*. La idea de encerrar a cada proceso trae consigo beneficios e inconvenientes, ya que por un lado conseguimos seguridad pero por otro aumentamos la complejidad de la gestión de los mismos. Lo más importante a tener en cuenta, es que ahora cada proceso tendrá un direccionamiento propio, el cual se llama direccionamiento virtual (actúa sobre la memoria virtual) y que el sistema operativo gestiona su asociación con la memoria física a través de las tablas de páginas. La clasificación más grande que podemos hacer de la memoria virtual es la separación entre el espacio de núcleo y espacio de usuario, el cual podemos ver en la figura 4.1 en el caso de los sistemas GNU/Linux, y es que el núcleo también tiene que estar bajo las mismas normas del direccionamiento virtual para que los procesos puedan acceder a ciertas partes del núcleo como pueden ser las librerías (e.g. libc.so).

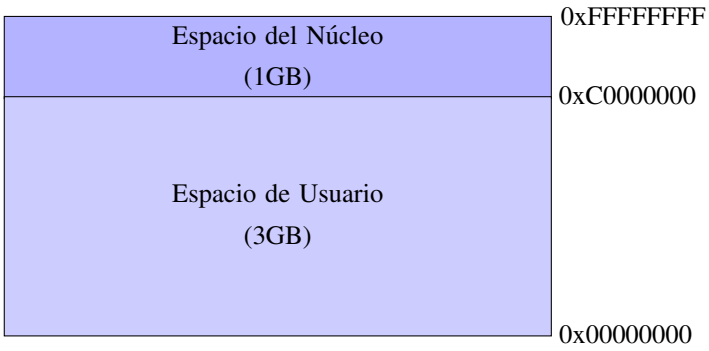


Figura 4.1: Memoria virtual en sistemas GNU/Linux con arquitectura x86 de 32 bits (general)

Los sistemas GNU/Linux con arquitectura x86 de 32 bits hacen una separación de 1GB para el espacio del núcleo y 3GB para el espacio del usuario (direccionamiento máximo de 4GB para arquitecturas de 32 bits con el byte como elemento mínimo: $2^{32}B = 4294967296B = 4294967296B * \frac{1KB}{1024B} * \frac{1MB}{1024KB} * \frac{1GB}{1024MB} = 4GB$). Que el núcleo reserve 1GB para sí mismo no significa que todo el espacio esté siendo ocupado por sí mismo, sino que se reserva espacio para poder asignar direcciones a otros objetos como librerías. Los 3GB restantes son parte del espacio de usuario, el cual es utilizado por los procesos y, cada vez que el núcleo cambie de la ejecución de un proceso a otro, el direccionamiento virtual será diferente debido a que el direccionamiento virtual es dependiente del proceso que acceda al mismo, por lo que si un proceso X accede a la dirección 0x08046000 no estará accediendo a la misma dirección física que si lo hiciera el proceso Y.

La parte del espacio de usuario es el direccionamiento virtual que los procesos utilizarán. En este espacio, se utiliza una estructura común, que en el caso de los sistemas GNU/Linux con arquitectura x86 de 32

bits, podemos observar en la figura 4.2.

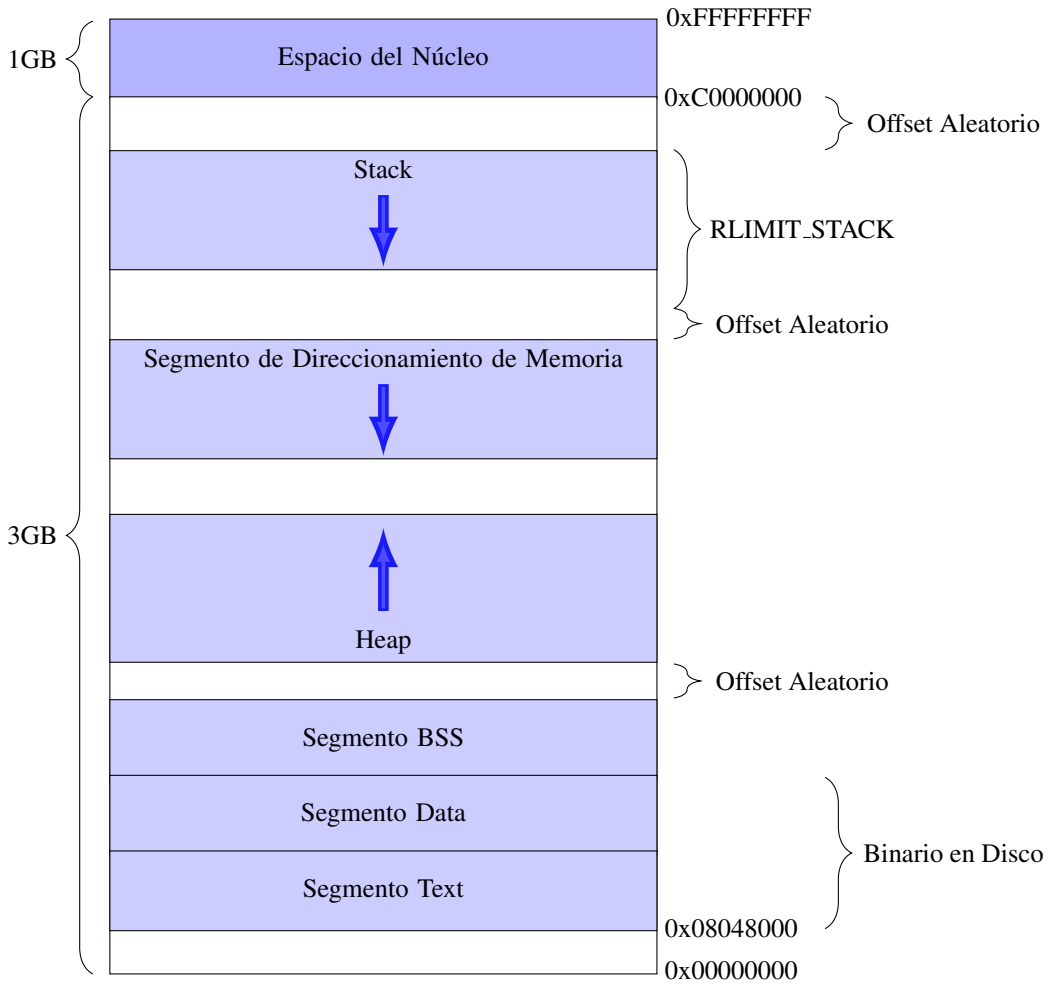


Figura 4.2: Memoria virtual en sistemas GNU/Linux con arquitectura x86 de 32 bits (diseño flexible)

Los sistemas GNU/Linux emplean el formato ELF para sus ficheros binarios, los cuales tienen toda la información necesaria para poder ejecutarse. Este fichero está formado por diferentes secciones, las cuales están referenciadas por una serie de cabeceras, y una vez se carga el binario en memoria, los segmentos en los que se carga el binario son los segmentos *data* y *text* como se puede ver en la figura 4.2. Los aspectos más importantes del espacio de usuario son los siguientes:

- *Stack* o pila: el *stack* es una estructura de datos fundamental para la ejecución de los procesos. En esta estructura de datos, la cual utiliza una política LIFO (*Last-In, First-Out*), en la que se almacenan todos los datos locales de las funciones como sus parámetros, variables locales y los datos necesarios para ejecutar las llamadas a funciones. La pila es la estructura de datos gracias a la cual es posible la recursión (i.e. función que se llama a sí misma). El tamaño máximo que puede alcanzar este segmento está definido por RLIMIT_STACK, y crece hacia las direcciones bajas (i.e. aumenta su tamaño hacia abajo en la memoria).
- Segmento de Direccionamiento de Memoria (*Memory Mapping Segment*): direccionamiento de ficheros (e.g. librerías dinámicas) y direccionamientos anónimos (i.e. direccionamientos que no corresponden a ningún fichero, sino que sirve para direccionar datos como puede ser una solicitud de memoria dinámica al *heap*, y si esta solicitud sobrepasa el valor de MMAP_THRESHOLD, en lugar de almacenarse en el

heap, se almacenará en este segmento). Crece hacia las direcciones bajas (i.e. aumenta su tamaño hacia abajo en la memoria).

- *Heap* o montículo: el *heap* es la estructura de datos que almacena los datos que se reservan de manera dinámica (i.e. no se puede calcular el tamaño de la solicitud de memoria en tiempo de compilación). Crece hacia las direcciones altas (i.e. aumenta su tamaño hacia arriba en la memoria).
- Segmento BSS: almacena las variables globales y las variables estáticas no inicializadas, las cuales se inicializa a cero en esta región.
- Segmento *Data*: almacena las variables globales y las variables estáticas inicializadas. Este segmento forma parte del binario y contiene diferentes áreas definidas (e.g. *data*, *ordata*).
- Segmento *Text*: almacena las instrucciones que se van a ejecutar (i.e. la lógica del programa). Este segmento forma parte del binario.

Respecto a los múltiples *offset* que podemos ver en la figura 4.2, se trata de desplazamientos aleatorios que se añaden al *stack*, al segmento de direccionamiento de memoria y al *heap* para aumentar la seguridad aleatorizando el direccionamiento y así consiguiendo que no sea determinista y no se pueda acceder a las secciones concretas, lo cual es necesario para explotar con éxito ciertas vulnerabilidades (esta medida de protección se conoce como ASLR (*Address Space Layout Randomization*)). Esto se aplica en el diseño flexible, el cual se aplica si se conoce un tamaño máximo para el *stack*, tamaño máximo el cual se conoce a través de RLIMIT_STACK, pero si no está definido o se quita el límite o se asigna el valor 1 a la variable *sysctl_legacy_va_layout* (a través de sysctl() o del fichero /proc/sys/vm/legacy_va_layout), entonces se utilizará el diseño clásico que se puede observar en la figura 4.3.

La diferencia entre el diseño flexible (figura 4.2) y clásico (figura 4.3) están reflejados en la tabla 4.1.

Segmento	Diseño Clásico	Diseño Flexible
Stack	Empieza en la dirección 0xC0000000 (si ASLR no está activado) y crece hacia las direcciones bajas de la memoria	
Segmento de Direccionamiento de Memoria	Empieza en la dirección 0x40000000 y crece hacia las direcciones altas de la memoria	Empieza en una dirección cercana al final del tamaño máximo del *stack* (RLIMIT_STACK) y crece hacia las direcciones bajas de la memoria
Heap	Empieza después del segmento BSS (si ASLR no está activado) y crece hacia las direcciones altas de la memoria	
Segmento BSS	Empieza después del segmento *Data*	
Segmento *Data*	Empieza después del segmento *Text*	
Segmento *Text*	Empieza en la dirección 0x08048000	

Tabla 4.1: Diseño clásico *vs* flexible de la memoria en sistemas GNU/Linux con arquitectura x86 de 32 bits

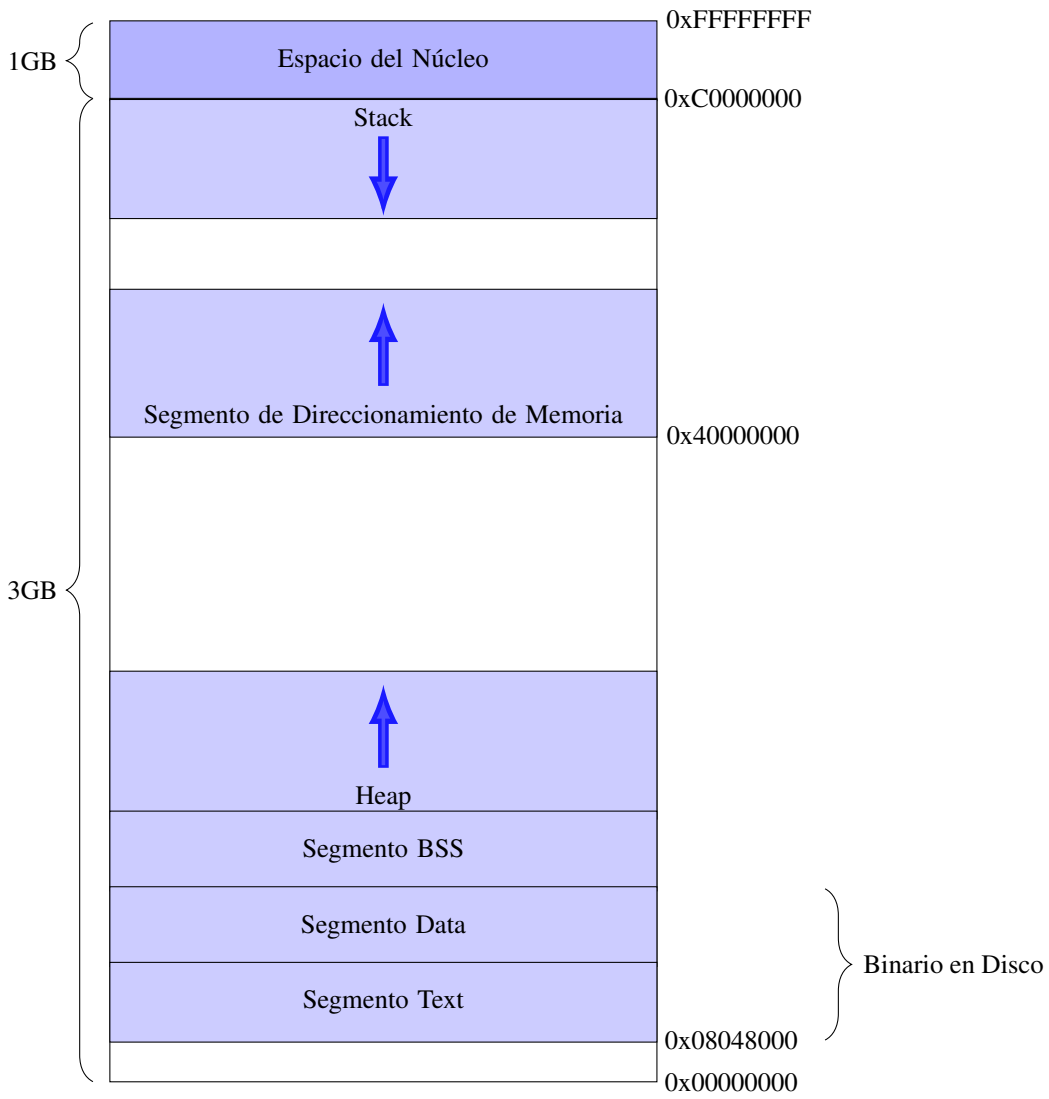


Figura 4.3: Memoria virtual en sistemas GNU/Linux con arquitectura x86 de 32 bits (diseño clásico)

Una de las partes más importantes del espacio de usuario es el *stack*. En esta estructura de datos es donde tenemos que prestar especial atención porque es donde, entre otras cosas (e.g. variables de entorno cargadas con el proceso), se introducen los datos necesarios cuando se hacen llamadas a funciones. Cada vez que se hace una llamada a una función, en el *stack* se almacenan los parámetros de la función, la dirección de retorno para cuando se termine la ejecución de la llamada, sus variables locales, etc. En la figura 4.4 tenemos una representación del *stack* que representa los datos que están almacenados y los registros que son relevantes en la arquitectura x86.

De la estructura del *stack*, que se puede observar en la figura 4.4, destacamos el proceso que se realiza a la hora de llamar a una función. Cada vez que se llama a una función, se tiene que almacenar la información necesaria para que, cuando se termine la ejecución de la misma, se pueda reanudar la ejecución por donde iba antes de realizar la llamada a la función. Además, también se necesita alguna forma de hacer referencia a los argumentos y variables locales de la función, ya que están almacenados de manera relativa.

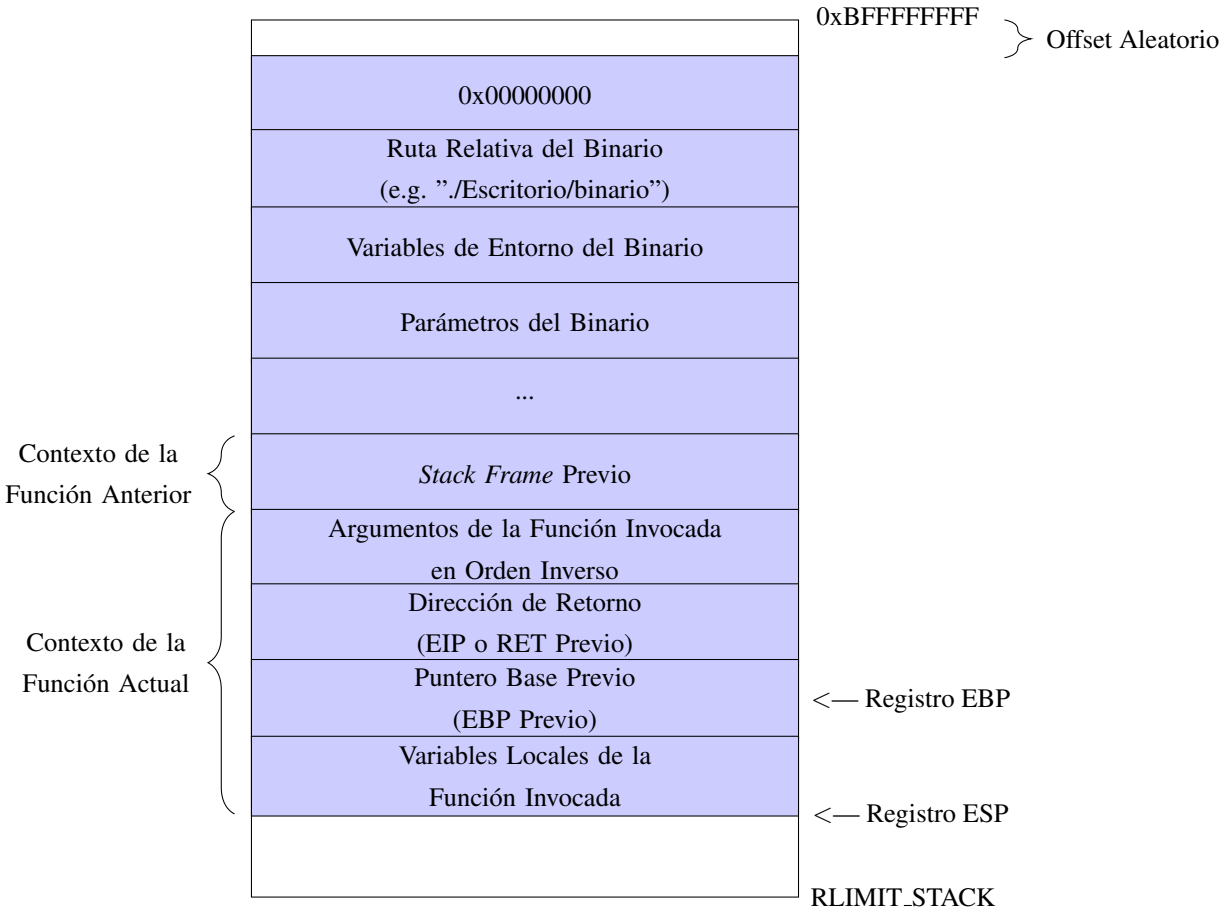


Figura 4.4: Estructura del *stack* en detalle (diseño flexible)

A continuación, detallamos los elementos relevantes de la figura 4.4:

- *Stack Frame*: contexto de una función que tiene almacenado los argumentos y variables locales de manera relativa a su registro EBP.
- Registro EBP: registro de la arquitectura x86 de 32 bits que sirve como referencia para sumar *offset*'s y se puedan alcanzar los argumentos de la función en un *stack frame* concreto o sus variables locales solo conociendo la posición dado el nombre. Por ejemplo, si queremos localizar el primer argumento de una función, debido a que el registro EBP está apuntando al valor del registro EBP anterior y sabiendo que los argumentos se colocan en orden inverso (la razón de ello es porque el *stack* es una estructura LIFO, y debido a que los argumentos están apilados antes que la dirección a la que apunta el registro EBP, esto nos facilita acceder a los argumentos de manera ordenada), si accedemos a $EBP + 8$, llegamos al primer argumento (sumamos bytes, y teniendo en cuenta que el direccionamiento es de 32 bits, tenemos que 4 bytes equivale a 32 bits, por lo que tenemos la longitud de una palabra) porque el primer argumento está justamente encima del valor almacenado EIP del anterior *stack frame*, que está justamente encima del valor almacenado EBP del anterior stack frame, que es donde el registro EBP está apuntando; $EBP + 12$ para el segundo argumento, etc. Habrá que tener en cuenta el tamaño de los argumentos o variables locales para saber cuánto tenemos que añadir al salto para saltar correctamente de una variable a otra. Se almacena el valor actual del registro en el *stack* cuando se va a realizar la llamada a una función para luego poder restaurarlo y poder seguir operando con normalidad.

- Registro EIP: registro de la arquitectura x86 de 32 bits que nos sirve para saber cuál será la siguiente orden a ejecutarse. Cuando se realiza una llamada a una función, se almacena el valor actual más el tamaño de la instrucción actual a la que apunta, la cual será la llamada a la función, y así conseguimos que cuando se restaure el valor una vez se termine la ejecución de la función, continuemos la ejecución de manera normal.
- Registro ESP: registro de la arquitectura x86 de 32 bits que nos sirve para saber dónde está el tope del *stack* en todo momento, para saber en qué dirección almacenar los valores cada vez que se ejecute una instrucción de apilar o desapilar (el valor de este registro se actualiza automáticamente cada vez que se ejecuta una de estas instrucciones).

Una vez entendemos los elementos importantes del *stack*, podemos describir el proceso que se lleva a cabo cuando se llama una función hasta que termina dicha función. Los pasos detallados se describen a continuación:

1. Una función *foo()* llama a una función *bar(1, 2)*.
2. Se apilan en el *stack* los siguientes elementos:
   a) Segundo parámetro: ”2”.
   b) Primer parámetro: ”1”.
   c) Valor actual del registro EIP más el tamaño de la instrucción de llamada a *bar(1, 2)*.
   d) Valor actual del registro EBP que se está utilizando para hacer referencia a elementos de la función *foo()*.
3. Se accede al código de la función *bar(1, 2)*.
4. Se iguala el registro EBP al registro ESP, consiguiendo que EBP apunte al valor del EBP almacenado de *foo()* en el *stack frame* de *bar(1, 2)*.
5. Se reserva el espacio necesario en el *stack* modificando el valor del registro ESP, el cual no debería de modificarse a mano normalmente, pero sí que se realiza para este proceso. El espacio reservado se utilizará para las variables locales y otras tareas (e.g. bytes para alinear el contenido del *stack*, ya que el tamaño de los argumentos varía dependiendo del tipo de datos empleado, pero esto será una decisión del compilador). La razón de reservar el espacio y no dejar que ESP se gestione automáticamente es que, si se hace otra llamada a otra función, sigamos teniendo el espacio necesario para ejecutar la función, ya que si no lo hiciéramos, la llamada a esa función podría machacar los datos de las variables locales que hasta ese momento se habían cargado en *bar(1, 2)*.
6. Ejecutar la función *bar(1, 2)*.
7. Cuando se termine la ejecución de *bar(1, 2)*, se igualará el valor de ESP al valor de EBP, consiguiendo que ahora las acciones de apilar y desapilar se hagan relativas al valor almacenado EBP previo. Con esto, conseguimos que ahora el *stack* haya decrecido.
8. Desapilamos y almacenamos el valor en el registro EBP. Debido a que ESP apuntaba al valor EBP previo, hemos restaurado el registro EBP con el valor previo del registro EBP, es decir, hemos restaurado el valor de EBP que se estaba empleando en la función *foo()*.

9. Desapilamos y almacenamos el valor en el registro EIP. Debido a que ESP apuntaba al valor EIP previo, la cual es la dirección de retorno, ahora EIP apunta a la siguiente instrucción de la llamada a *bar(1, 2)* en la función *foo()*. Con esto, hemos conseguido salir del *stack frame* de la función *bar(1, 2)* por completo y acabamos de restaurar el *stack frame* de la función *foo()*.

Los pasos descritos son los procesos de cargar un contexto de una función y la restauración del contexto anterior (lo que llamamos *stack frame*). Con la información descrita en este apartado, damos por concluida la información necesaria para proceder con los siguientes apartados.

La Vulnerabilidad

Un *buffer overflow* es una vulnerabilidad que puede ocurrir en aquellos lenguajes de programación que no hacen una comprobación de los límites de los *arrays* (e.g. C, C++), cuya situación también es conocida como *bounds checking*. Los *arrays*, o matrices, son una estructura de datos que almacenan múltiples elementos de manera lineal en la memoria. Un ejemplo lo podemos encontrar en la figura 4.5.

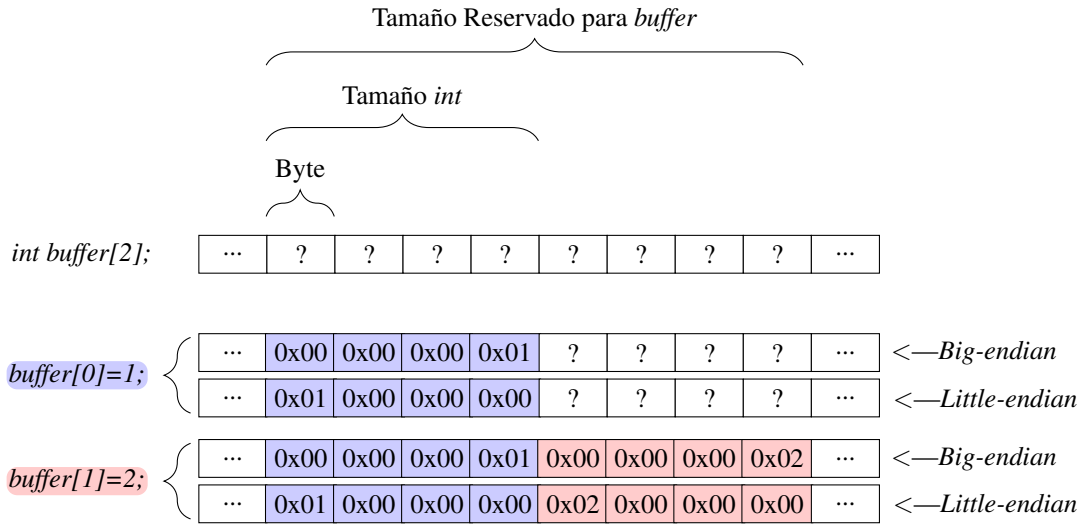


Figura 4.5: Representación *arrays* en memoria (lenguaje C) con *endianness*

Cuando se reserva un tamaño para un *array*, el tamaño reservado se almacenará en el *stack* en caso de poder calcularse en tiempo de compilación, o en el *heap* (podría también almacenarse en el Segmento de Direccionamiento de Memoria) en el caso de calcularse en tiempo de ejecución, lo cual se hace a través de las funciones de la familia de *malloc()* (figura 4.2). Cuando el *array* ya tiene el tamaño asignado, tiene una dirección inicial y una dirección final, y cuando la dirección final, o máxima, se sobrepasa haciendo uso del mismo *array*, decimos que tenemos un *buffer overflow*, que es el caso típico, pero también podemos tener el caso en el que se sobrepase la dirección inicial, lo que nos lleva a un *buffer underflow*, que aunque más atípico, también puede ocurrir y tiene sus riesgos. Centrándonos en el *buffer overflow*, este puede ocasionar los siguientes incidentes:

- Continuar con el comportamiento normal del programa: cuando sobrepasamos el límite de un *array*, no sabemos qué hay a continuación, pero simplemente puede que no haya nada y que, por suerte, no suceda nada. Este caso es muy típico cuando reservamos memoria en el segmento *heap*, ya que el algoritmo que

se emplea para la reserva de almacenamiento dinámico (i.e. *Doug Lea's algorithm* o algoritmo *dlmalloc*) utiliza una cantidad fija de tamaño reservada en *buckets*, y si reservamos una cantidad que no coincide con el tamaño de ningún *bucket*, se nos dará el más cercano y, por lo tanto, si sobrepasamos alguna dirección no ocurrirá nada.

- Comportamiento indefinido: cualquier cosa puede ocurrir cuando se sobrepasa el tamaño asignado a un *array*, y con cualquier cosa nos referimos exactamente a cualquier cosa. Por poner un ejemplo, si sobrepasamos un *array*, podría pasar que se llame a una función que no se llama desde ningún sitio de tu programa pero que está definida, podría ocurrir que se sale la ejecución de un condicional que debería de ejecutarse pero no lo hace, etc. Por estas razones, entre otras, es por lo que a veces es tan difícil de encontrar estas vulnerabilidades.

- Denegación del servicio: se podría cerrar el programa en algún momento, ya sea por un efecto directo (e.g. típico *segmentation fault* por acceder a una dirección a la que no tenemos los permisos necesarios) o colateral (e.g. cambio del valor de una variable, la cual conduce al cierre inesperado del programa).

- Ejecución de código arbitrario: en el peor de los escenarios, un atacante podría ejecutar cualquier orden a través de la vulnerabilidad. Si el programa que sufre de la vulnerabilidad tuviera permisos elevados, el atacante ejecutará dichas órdenes con los mismos permisos.

La correcta explotación de esta vulnerabilidad, normalmente, lleva como mínimo a la denegación de servicio, y en la mayoría de ocasiones a la ejecución de código arbitrario. Centrándonos en la explotación de las vulnerabilidades de tipo *buffer overflow* (aquellas que suceden en el *stack*) y dejando de lado las de tipo *heap overflow* (aquellas que suceden en el *heap*) debido a la complejidad de los mismos, tenemos que tener en cuenta lo explicado anteriormente sobre el diseño de la memoria en el sistema objetivo, siendo en este caso un sistema GNU/Linux con arquitectura x86 de 32 bits. Como se pudo observar en el funcionamiento de las llamadas a funciones, el *stack* es una pieza fundamental para llevarlas a cabo correctamente (figura 4.4). Debido a que el *stack* crece hacia las direcciones bajas, nos podemos aprovechar de esta situación para que, cuando podamos sobrepasar los límites de un *array*, podamos sobreescribir los valores que hay almacenados en el *stack* y así podamos editar a nuestro antojo los valores almacenamos del contexto de la función anterior como puede ser el valor almacenado de los registro EIP y EBP. Modificar estos valores suele ser el objetivo para lograr una ejecución de código arbitrario, pero también podemos simplemente modificar el valor de aquellas variables que estén definidas en la función antes que el *array* que está siendo objetivo de una modificación sobrepasando sus límites. Un ejemplo de *buffer overflow* lo podemos encontrar en la figura 4.6, donde se puede observar como afecta esta vulnerabilidad a los datos almacenados del *stack* y se puede ver un ejemplo típico donde se reescriben los valores almacenados de los registros EIP y EBP, siendo el de mayor interés el del registro EIP debido a que nos permite alterar, de manera arbitraria, el flujo de control del programa.

Hay una gran variedad de técnicas que nos permiten sortear ciertas limitaciones (e.g. poco espacio para almacenar código arbitrario, solo poder reescribir de manera parcial el valor almacenado del registro EBP) que se nos pueden presentar frente a una explotación exitosa. Además, hay que tener en cuenta que, para evitar esta vulnerabilidad tan dañina, se han implementado técnicas para evitar que se lleve a cabo o, al menos, mitigar los efectos o probabilidad de éxito (e.g. ASLR añade aleatoriedad a secciones de memoria que antes estaban definidas de manera estática e intenta evitar que podamos acceder a zonas concretas de la memoria como puede ser una función de la librería estándar; *stack* no ejecutable, lo cual evita que se pueda almacenar en el *stack* código con el fin de ejecutarlo; *stack canaries*, se añade información de control en el *stack* y se comprueba que no se haya modificado para evitar que se reescriban ciertas zonas en el *stack* en las que no está permitido, como

pueden ser los valores almacenados de los registros EIP o EBP). Algunas técnicas se centran en la modificación el registro EIP, otras en el del registro EBP, otras en encadenar llamadas, etc.

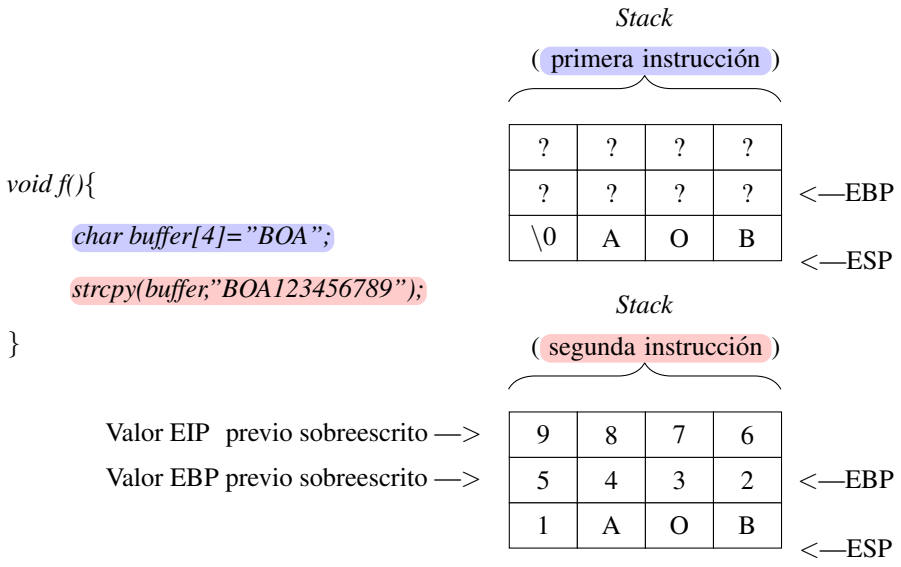


Figura 4.6: *Buffer overflow* en sistema GNU/Linux con arquitectura x86 de 32 bits (*little-endian*) en C

Algunas de las técnicas más empleadas o famosas se describen a continuación:

- Modificación del valor almacenado EIP: el más simple y estudiado de los ataques es la modificación del valor almacenado EIP. Esta explotación suele consistir en modificar el valor almacenado del registro EIP en el contexto de alguna función para que, cuando finalice, inserte dicho valor en el registro EIP y, entonces, se salte la ejecución a algún sitio controlado, como puede ser código que hayamos colocado en algún sitio de manera premeditada o se salte a alguna librería del sistema y, por ejemplo, se nos devuelva una terminal para ejecutar lo que sea necesario.

- Abuso del *Frame Pointer*: hay ocasiones en las que el valor almacenado del registro EIP no es alcanzable porque se hace una correcta gestión o por cualquier otro motivo. Aún así, sigue sucediendo que el valor almacenado del registro EBP sí que es alcanzable, y en estas situaciones sucede que también podemos ejecutar código arbitrario. Cuando tenemos el control del valor almacenado del registro EBP podemos hacernos con el control del registro ESP, el cual controla los valores que se apilan y desapilan en el *stack*. De esta manera, cuando tenemos el control de ESP y hemos hecho que apunta a una dirección donde, por ejemplo, tenemos almacenada la dirección de un sitio que controlamos con código arbitrario, al terminarse la ejecución de la segunda función (cuando la primera función termina, nos hacemos con el control de ESP), hacemos que se desapile el primer valor en el registro EBP de donde apunta ESP pero a continuación, se desapila el siguiente valor en el registro EIP de donde apunta ESP, por lo que conseguimos que esa dirección que almacenamos previamente esté en el registro EIP y, entonces, hemos conseguido que EIP contenga la dirección de una zona que controlamos con código arbitrario. Es una explotación en 2 (o más) pasos, la cual no es tan sencilla de entender como el caso anterior, pero igual de efectiva.

- *Off-by-One*: variante de la técnica de abuso del *frame pointer* que se puede ejecutar en sistemas con *endianness* de tipo *little-endian*. Hay ocasiones en las que solo es alcanzable uno de los bytes del valor almacenado del registro EBP, y en sistemas con *little-endian*, esto puede ayudarnos a hacer saltos muy

pequeños, pero suficientes como para ejecutar código arbitrario. La idea detrás de esta técnicas está en la misma que en el abuso del *frame pointer* pero esta vez almacenando el código resultante o dirección muy cerca del *array*, como por ejemplo en otra variable cercana.

- *Return Oriented Programming* (ROP): esta es una de las técnicas más empleadas en los últimos tiempos debido a la flexibilidad que aporta al atacante. Esta técnica se basa en hacerse con el control del *stack* a partir de buscar instrucciones estáticas en partes del código que se correspondan con una o más instrucciones *pop* (tantas como parámetros necesitemos para ejecutar una función objetivo) seguidas de una instrucción *ret* de ensamblador. Teniendo estas direcciones, se consigue de alguna forma hacerse con el control del registro ESP (aplicando alguna técnica de las anteriores u otras) para que apunte a una de las direcciones que contienen estas instrucciones, y estas instrucciones se van a encargar de ejecutar todas las instrucciones *pop*, y como tenemos el control del registro ESP, esté estará apuntando a los parámetros de la función que se quiera ejecutar, la cual será normalmente alguna función de la librería de C (i.e. libc), y al final, al ejecutarse la instrucción *ret*, en ese momento ESP tendrá que estar apuntando a la dirección de la función que queramos llamar, consiguiendo ejecutar la función con éxito. Cabe destacar que la carga de parámetros para la llamadas a librerías del sistema varía según la convención empleada (*cdecl*: por defecto en compiladores de GNU. Parámetros cargados en el *stack*; *stdcall*: más empleada en Win32. Igual que *cdecl* pero varía en que la función de la librería es la encargada de limpiar el *stack*; *fastcall* usada en algunas partes del núcleo de NT. Los dos primeros parámetros en registros y el resto en el *stack*), por lo que si los parámetros se cargan según otra convención, se tendrá que buscar el conjunto de instrucciones correspondientes, los cuales se les conocen como *gadgets*.

Casos Recientes

En el momento de estar escribiendo estas líneas, hay más de 80 vulnerabilidades relacionadas con un *buffer overflow* con código CVE asignado, lo cual significa que se descubren una media de 5 vulnerabilidades de este tipo por semana. Algunos ejemplos son los siguientes:

- CVE-2020-9760[4]: vulnerabilidad descubierta en la aplicación de mensajería WeeChat, que afecta al rango de versiones con inicio en la 0.3.4 hasta la 2.7, de tipo *buffer overflow*. Esta vulnerabilidad se explotaba cuando se recibía un mensaje IRC 005 con cadenas largas en el prefijo del seudónimo, o *nick*, cuando se indicaba un nuevo modo para un *nick*.
- CVE-2020-9552[5]: una vulnerabilidad de tipo *buffer overflow*, en concreto de tipo *heap buffer overflow*, permitía ejecutar código arbitrario en Adobe Bridge versión 10.0.
- CVE-2020-7450[6]: en el sistema operativo FreeBSD, una mala gestión de las URL's en el protocolo *libfetch* que contenían componentes de usuario y/o contraseña, podía conducir a una vulnerabilidad de tipo *heap buffer overflow* causando comportamientos inesperados o ejecución de código arbitrario.

[4]CVE-2020-9760: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-9760`
[5]CVE-2020-9552: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-9552`
[6]CVE-2020-7450: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-7450`

Fuentes

Para la realización de las secciones anteriores, relacionadas con la vulnerabilidad *Buffer Overflow*, hemos empleado las siguientes fuentes: [10], [11].

Hemos realizado comprobaciones de lo explicado en un sistema GNU/Linux con una arquitectura x86 de 32 bits. En concreto, hemos utilizado una distribución Ubuntu 12.04 con versión de núcleo 3.13.0-32-generic.

4.2.2 Otras Vulnerabilidades

Como ya adelantamos anteriormente, vamos a comentar, de manera superficial, las vulnerabilidades de *Cross-Site Scripting* (XSS) y *SQL Injection*. Estas vulnerabilidades han sido de las más explotadas en el ámbito de la web, y ambas tienen una gravedad muy elevada. Estas tecnologías comparten en común dos características, siendo una la que ya hemos comentado (son del entorno web), y la otra es que ambas se producen por la misma razón: el incorrecto tratamiento de la entrada del usuario. En muchas ocasiones se tiene que trabajar con datos que el usuario nos proporciona directa o indirectamente, y no tenemos que perder de vista que cualquier usuario puede actuar de cierta manera que no hemos previsto, ya sea premeditado o no, y por ello tenemos que realizar un correcto tratamiento de dichas entradas con la prevención como objetivo.

El usuario medio suele actuar con normalidad y según lo esperado, pero no por ello no debemos prevenir las situaciones inesperadas, y es que si a un usuario le preguntas su edad, lo normal será que te responda correctamente, pero es que la verdad absoluta es que no tenemos garantía de ello. Un avión con 300 pasajeros a bordo no puede bloquearse porque un piloto introduzca mal una coordenada, y es que los comportamientos inesperados tienen que ser parte del diseño de cualquier sistema para evitar tragedias. La correcta gestión de los datos introducidos por el usuario es la manera en la que se evita la explotación de estas vulnerabilidades, pero la verdad es que algo que parece tan trivial a la hora de mencionarlo es una de las tareas más complejas y de los mayores retos a los que se suele enfrentar un desarrollador. El problema suele radicar en que somos humanos, y que solemos esperar un comportamiento racional (”si a mi me preguntaran mi nombre, ¡yo diría mi nombre! pero Juanito ha dicho que su nombre era ”1’ or ’1’=’1”, ¿te puedes creer?”), pero hay situaciones que la complejidad técnica también puede ser un obstáculo (e.g. no tener en cuenta los años bisiestos y que la fecha de nacimiento de un usuario se valide como correcta en algunas partes del sistema o en otras no, causando inconsistencias). A continuación, pasamos a explicar estas vulnerabilidades tan vigentes hoy en día en la web.

Cross-Site Scripting (XSS)

La web tal cual la conocemos hoy en día es dinámica. Mucho ha cambiado la web del hoy con la de los años 90, y parte del cambio se debe a una serie de tecnologías, entre la que se encuentra *Javascript* (su nombre oficial es *ECMAScript*), el lenguaje de la parte del cliente (i.e. ejecutado en la máquina del cliente por el navegador, sin necesidad de que un servidor ejecute las órdenes). *Javascript* es el lenguaje de programación que se utiliza junto al lenguaje de marcado HTML, el cual se utiliza para la maquetación de la página web y para indicar la información semántica de la misma página. Podríamos utilizar *Javascript* para realizar cualquier tarea e indicar la salida a través de HTML, pero la verdad es que la mayoría de veces lo que nos interesa es ingresar algún dato en algún formulario HTML y procesarlo en *Javascript* con alguna finalidad (e.g. comprobar que los datos son correctos) y, posteriormente, introducir de nuevo en la página algún dato (e.g. mensaje de

bienvenida al iniciar sesión con tu nombre de usuario), y es aquí justamente donde debemos llevar cuidado.

Una vulnerabilidad XSS puede suceder cuando, de alguna manera, hay un flujo de datos donde la información parte inicialmente del usuario (e.g. a través de un campo en un formulario HTML, a través de un parámetro en una petición GET, a través de un paquete de red, ...), llega parte o totalmente dicha información a *Javascript* y dicha información va a parar, igual o procesada, al código HTML (ejemplo en la figura 4.7) ya sea en el mismo momento o posterior, aunque este último paso es opcional en una vulnerabilidad XSS, ya que existen otras maneras de explotar esta vulnerabilidad sin necesidad de que el código vuelva al código HTML y se explote directamente al llegar a *Javascript* (e.g. función *eval()*). Esta situación es de lo más típico, por lo que, claramente, una vulnerabilidad XSS no sucede simplemente por darse este flujo de información, ya que es una condición necesaria pero no suficiente, y es que también necesitamos que esta información que parte del usuario y acaba en el código HTML, acabe siendo código *Javascript* en el código HTML [12].

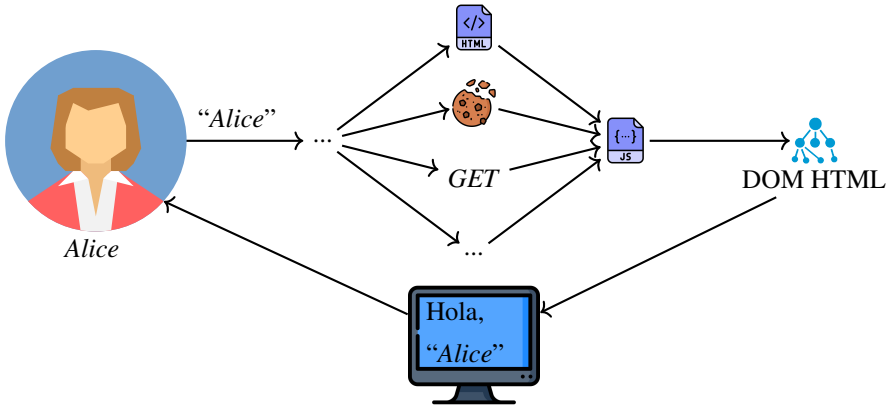


Figura 4.7: Flujo de datos típico en una vulnerabilidad XSS

La forma de evitar estos ataques es escapando aquellos elementos propios del lenguaje *Javascript* (e.g. <, >), ya que en muchas ocasiones es necesario hacer el procesamiento a través de *Javascript*, y es necesario evitar estos ataques. Un ejemplo de ataque siguiendo el mismo esquema que la figura 4.7 lo tenemos en la figura 4.8. El verdadero peligro de esta vulnerabilidad radica en que tenemos control absoluto sobre lo que queremos ejecutar en *Javascript*, a veces con algunas limitaciones (e.g. límite en los caracteres a escribir), por lo que nada nos evita enviar una URL a víctimas y que estas redirijan a algún sitio vulnerable a XSS, donde la URL es el vector de ataque, o que el alcance del XSS sea mayor y sea posible almacenar la vulnerabilidad en el mismo sitio web y todos los usuarios que accedan se vean afectados. El único límite es la imaginación.

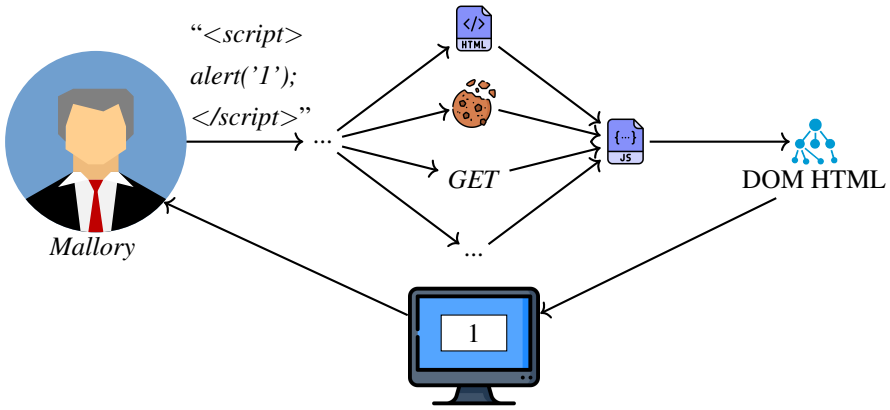


Figura 4.8: Ejemplo de vulnerabilidad XSS (flujo de datos)

Hay diferentes variantes de la vulnerabilidad, de las cuales se van a explicar las más relevantes a continuación:

- XSS no persistente o reflejado: esta variante es la que nos permite explotar una vulnerabilidad XSS de manera que la carga maliciosa (i.e. *payload*, código *Javascript* que va a ser ejecutado por la víctima y escrito por el atacante) esté contenido en algún medio volátil, no persistente, como puede ser una URL.
- XSS persistente o almacenado: esta variante nos permite explotar una vulnerabilidad XSS almacenando la carga maliciosa en el código del sitio web, servidor o algún sitio que al final devolverá un documento HTML con la carga. Esta variante es una de la más peligrosas sino la que más, ya que una vez se inserte el código *Javascript* en la página, todos los usuarios que descarguen esa página o recurso que contenga la vulnerabilidad, se ejecutará la carga maliciosa en el momento de cargarse o cuando las condiciones específicas se den (e.g. hacer *click* en un botón), ya que todo dependerá del alcance de la vulnerabilidad debido a que no siempre es tan sencillo como insertar la vulnerabilidad.
- XSS en el cliente: históricamente, la carga maliciosa se ha colocado en el servidor y ha sido devuelta al cliente en forma de documento web, pero hoy en día se oscilan más los sitios web que siguen un patrón de diseño *single page* o que al menos lo maximizan, y esto quiere decir que evitan cargar un recurso web a través de la carga completa de una página, sino que solo se cargan aquellas partes de la página que lo requieren. Esto ha llevado a ampliar el uso del código que se ejecuta en el lado del cliente, llegando a almacenar los datos necesarios en el mismo, con lo que lleva a evolucionar la vulnerabilidad y a conseguir que la carga maliciosa de la vulnerabilidad XSS se almacene en el cliente y se ejecute en el mismo cuando se den las condiciones necesarias.
- XSS mutado (*mutation* XSS): esta variante se basa en escribir una cadena que, aparentemente, no es peligrosa, pero que al realizarse el proceso de análisis de la misma por el navegador, se transforma en otra cadena que sí que es peligrosa. Debido a lo inesperado de las transformaciones aplicadas por el navegador en el código HTML, resulta muy difícil de detectar este tipo de variantes. Por poner un ejemplo, incluso *Google* ha sufrido de esta vulnerabilidad, y dicha vulnerabilidad sucedía al intentar buscar en el buscador la cadena "*<noscript><p title="</noscript><img src=x onerror=alert(1)>">*".

SQL Injection

Las bases de datos son el activo más importante que tienen las organizaciones, pues en ellas se almacenan los datos de sus clientes como mínimo, y normalmente también todo lo que sus productos utilicen o transmitan. Cada vez que una base de datos, o parte de la misma, es filtrada, la reputación de la organización afectada se ve gravemente perjudicada, y debido a ello han surgido proyectos como *haveibeenpwned*[7], el cual nos permite saber si alguna de nuestras cuentas, e incluso alguna de nuestras contraseñas, se encuentra presente en las grandes brechas de bases de datos conocidas. Pero ¿cómo suceden estas brechas? Pueden haber varías razones, como puede ser que hayan conseguido entrar en el servidor donde se almacena, pero normalmente ocurre debido a una vulnerabilidad *SQL Injection*. Estas vulnerabilidades no son únicamente explotables en un entrono web, ya que cualquier aplicación que no sea web también puede ser víctima de esta vulnerabilidad, pero normalmente tiene más presencia en el ámbito web, y es donde suele ser más prejudicial por el alcance que puede llegar a tener.

[7] Sitio oficial de *haveibeenpwned*: `https://haveibeenpwned.com/`

Una vulnerabilidad *SQL Injection* sucede cuando, de alguna manera, se consigue ejecutar código SQL en la base de datos a través de algún dato introducido por un usuario, lo cual suele llevar a filtraciones de informaciones, aunque no siempre, pero aunque no lleva a una filtración de información, es igualmente peligroso debido a que puede llevar a modificaciones o eliminación de datos. Las filtraciones de información son lo que suele interesarle a los atacantes, y aunque no siempre son sencillas de realizar, existen numerosas técnicas que permiten, como mínimo, realizar filtraciones de informaciones dirigidas a cierto objetivo. El flujo de datos que tiene que producirse es muy parecido al que se produce en la vulnerabilidad XSS, y es que un usuario introduce una cadena en algún elemento, esta pasa a procesarse y, en algún momento, puede que lleguemos al código del servidor, el cual tendrá acceso directo o indirecto a la base de datos, por lo que si conseguimos en ese flujo de información que se ejecute una cadena de lenguaje SQL en el servidor, estaremos en condiciones de explotar una vulnerabilidad *SQL Injection* [13]. Un ejemplo del flujo de datos necesario para una explotación correcta de la vulnerabilidad lo podemos encontrar en la figura 4.9, el cual es la situación típica que nos encontramos cuando estamos frente a esta vulnerabilidad y, como podemos observar, sucede algo similar a lo que sucedía con la vulnerabilidad XSS, ya que la información puede ir a parar a un parámetro, una *cookie* o cualquier elemento que finalmente vaya a parar al servidor. Un error muy típico es solo centrarse en aquellos vectores de entrada que están a la vista, como los parámetros de la URL, pero un atacante no tiene en cuenta solo esos parámetros, sino que también tendrá en cuenta, por ejemplo, los parámetros de las solicitudes *POST*, los cuales no se ven a simple vista pero son modificables fácilmente desde cualquier navegador a través de las herramientas de desarrollador.

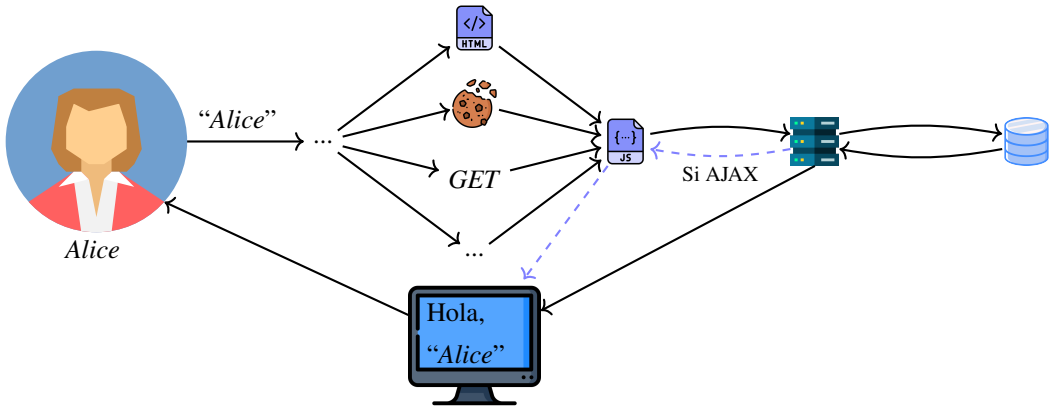


Figura 4.9: Flujo de datos típico en una vulnerabilidad *SQL Injection*

La manera de evitar un ataque por una vulnerabilidad *SQL Injection* suele ser escapar los caracteres propios del lenguaje SQL o que van a modificar de alguna manera la petición SQL que va a realizarse a la base de datos. Las peticiones SQL con datos de usuarios es lo más común y necesario en el día a día de las aplicaciones web, porque ¿cómo vamos a verificar que un usuario puede autenticarse si no es con los datos que él mismo nos ha proporcionado? Las buenas prácticas como emplear diferentes tecnologías que nos protegen de esta vulnerabilidad suele ser la metodología empleada por la mayoría, ya que un *framework* o librería que ya implementa la seguridad contra esta vulnerabilidad seguramente lo haya hecho mucho mejor que nosotros si nos ponemos a hacerlo desde cero. Reinventar la rueda nunca fue garantía de éxito. Un ejemplo de ataque siguiendo el mismo esquema que la figura 4.9 lo tenemos en la figura 4.10. El alcance de esta vulnerabilidad suele ser la fuga de información, pero no es el único de los peligros, porque como acabamos de decir, la base de datos se utiliza también para verificar la autenticación de un usuario, por lo que si podemos alterar esa petición SQL, podremos, seguramente, iniciar sesión con otro usuario que no es el nuestro (situación que podemos observar en la figura 4.10).

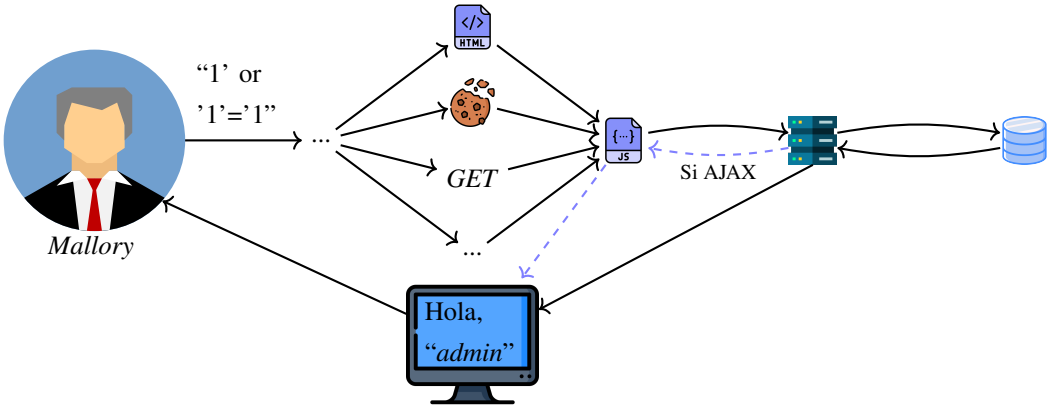


Figura 4.10: Ejemplo de vulnerabilidad *SQL Injection* (flujo de datos)

Hay diferentes variantes de la vulnerabilidad, de los cuales se van a explicar los más relevantes a continuación:

- *SQL Injection* clásico: la manera clásica de explotar *SQL Injection* es realizar una petición a la base de datos y se nos devuelva de alguna manera la respuesta por pantalla. Esta variante ha sido la más ampliamente explotada a lo largo de los años, ya que es la más sencilla. No solo nos permite obtener información, sino que dependiendo de la consulta que realicemos la consulta, también podemos iniciar sesión como otro usuario, actualizar el estado de una suscripción de pago, etc. La manera de explotar esta variante suele ser intentando cerrar la sentencia donde se va a insertar la información por parte del usuario y, a continuación, insertar una nueva sentencia y, finalmente, cerrar el resto de la sentencia original, lo cual se hace normalmente insertando un comentario, que suele ser insertar dos guiones. Un ejemplo podría ser la sentencia SQL ”*SELECT * FROM Users WHERE username=’$1’ AND password=’$2’;*”, donde $1 se reemplazaría por la sentencia ”*admin‘;–*” y $2 por cualquier valor, y así conseguiríamos acceso a la información del usuario ”admin”.

- *SQL Injection* serializado (*Serialized SQL Injection*): diferentes medidas se han implementado a lo largo de los años para evitar el *SQL Injection*, pero no todas las soluciones son perfectas, sino que son prácticas. Una de esas medidas es limitar la cantidad de solicitudes de un cliente a un servidor, ya que el comportamiento de un producto suele estar muy bien definido, y un usuario no suele hacer 500 peticiones en 5 segundos, además de que levanta muchas sospechas. Debido a esto, diversas técnicas se han ido desarrollando con el objetivo de, con muy pocas sentencias SQL, conseguir la cantidad mayor información de datos, que es de lo que se trata esta variante. Hay muchas maneras de hacerlo, muchas de estas maneras son dependientes del sistema gestor de base de datos, como puede ser la concatenación de sentencias SQL, entre otras.

- *SQL Injection* a ciegas (*Blind SQL Injection*): no siempre es posible mostrar la información por pantalla, y la mayor parte de veces es lo que más nos interesa. Aún así, sigue siendo posible obtener información. La manera de hacerlo se realiza a través de esta variante, la cual hace peticiones a través de una vulnerabilidad *SQL Injection*, pero no se muestra por pantalla la información objetivo del atacante, y por ello, para hacer comprobaciones, se realizan breves esperas de pocos segundos que nos permiten saber, por ejemplo, que la primera letra de la contraseña del usuario ”admin” es la ”C”. Esta variante, aunque lenta, es muy útil y, al igual que las anteriores, es automatizable, ya que el hecho de realizar esperas no implica que no podamos realizar un programa que tenga en cuenta esta espera más el sesgo de la conexión al servidor, aunque esto puede llevarnos a falsos positivos dependiendo del *jitter* (i.e. medida que nos indica cómo

de regular es la latencia de la conexión). La parte mala de esta variante es que suele ser dependiente del sistema gestor de base de datos, y que dependiendo de una condiciones u otras podremos explotarla, ya que no siempre se tiene los permisos necesarios en la base de datos para ejecutar ciertas funciones, como puede ser la que se necesite para hacer que el gestor de base de datos tarde unos segundos en devolvernos la respuesta a una de nuestras peticiones SQL.

4.3 Análisis Estático

El análisis estático es el estudio realizado sobre un programa, ya sea sobre el fichero de código fuente o cualquier otra representación (e.g. código objeto, código intermedio), mientras este no se ejecuta. Este tipo de análisis es empleado con diferentes finalidades como puede ser detección de errores comunes (e.g. división por cero, código inalcanzable) o detección de vulnerabilidades. En el caso de los compiladores, la mayoría hacen uso de técnicas de análisis estático con el fin de mejorar la calidad del código, y debido a estas técnicas, los compiladores son capaces de detectar cuando, por ejemplo, declaramos una variable y no se utiliza en un fragmento de código que nunca se ejecuta, entre otras situaciones muy comunes. Aunque no nos demos cuenta, constantemente estamos haciendo uso de este tipo de análisis, y no solo en los compiladores, sino también en los IDE (*Integrated Development System*) como *Visual Studio Code*, en el cual hay una gran variedad de *plugins* que empleamos constantemente y que nos avisan de ciertas características de nuestro código al detectar ciertas características inusuales.

Una primera aproximación de clasificación dentro del análisis estático podría realizarse a partir del análisis intraprocedural e interprocedural:

- Análisis intraprocedural: es el análisis que se realiza sin tener en cuenta las funciones, es decir, se realiza el análisis sobre una única función. Si queremos aplicar este análisis a todo un programa, hay diferentes enfoques, pero lo que normalmente se hace es tratar todo el código como si estuviera encerrado en una única función. El objetivo de este tipo de análisis es simplificar el problema del flujo de control, ya que es un problema a abordar que, en una primera aproximación, puede ser simplemente ignorarlo, y es el enfoque que toma el análisis intraprocedural. Esta es justamente la principal ventaja, ya que el intentar abordar el flujo de control de un programa tiene muchos problemas a resolver, empezando por construir un grafo que represente el flujo del programa. Las técnicas que aplican este análisis suelen ser más sencillas de aplicar, pero los resultados suelen ser más imprecisos o suelen tener más limitaciones. En este análisis se suele emplear alguna representación intermedia del código, no se suele trabajar con el código directamente, y esa representación suele ser el árbol sintáctico abstracto (del inglés *Abstract Syntax Tree* o AST).
- Análisis interprocedural: es el análisis que se realiza teniendo en cuenta la llamada a las funciones a lo largo del programa. Este análisis requiere de un grafo que represente el flujo de control, conocido como grafo de flujo de control (del inglés *Control Flow Graph* o CFG), lo cual no es trivial de construir, pero nos permite tener un nivel de detalle mucho más profundo del comportamiento de nuestro programa debido a que la información de la que disponemos es mayor. El objetivo de este tipo de análisis es aplicar técnicas que nos den unos resultados más acertados, y debido a que estas técnicas tendrán acceso a una cantidad mayor de información contextual (i.e. el CFG, el cual aporta mucha información del contexto del programa), los resultados deberán de ser mejores, pero no necesariamente, ya que las técnicas a aplicar pueden ser más complejas o tener una mayor tasa de falsos positivos y falsos negativos.

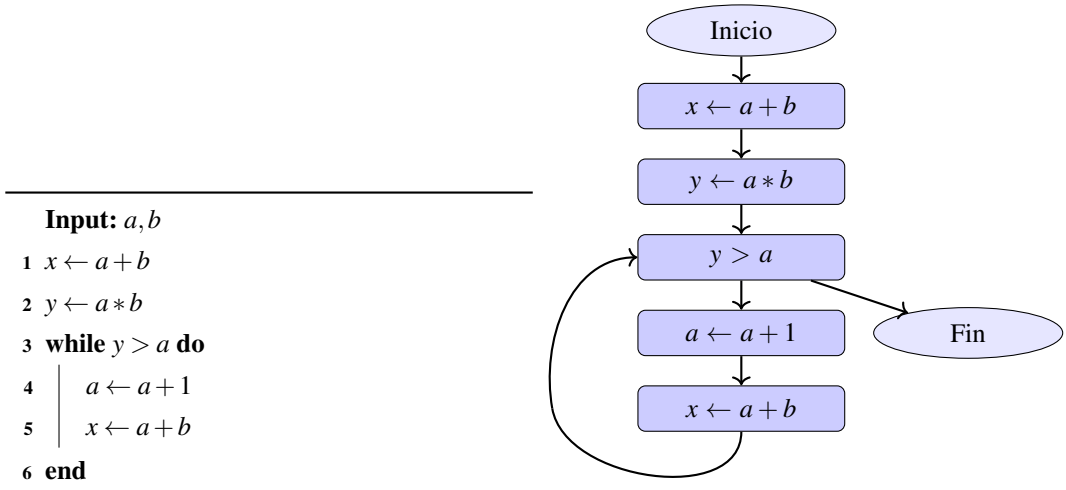
El análisis estático tiene una serie de ventajas, siendo algunas de las más importantes permitir la detección de errores temprana en el desarrollo del *software*, la determinación del lugar exacto donde se encuentra la fuente de los problemas o el análisis automatizado de grandes porciones de código con un coste en tiempo que no dependerá de la complejidad del proyecto, sino de la técnica de análisis estático empleada (e.g. si un proyecto tarda una hora en realizar la inicialización, eso no implica que el análisis estático empleado tarde una hora o más). Por otro lado, tienen una serie de desventajas o limitaciones, como puede ser que las implementaciones son dependientes del lenguaje objetivo, la cantidad de falsos positivos y negativos o la limitación que impone no ejecutar el código, el cual siempre puede llevar a fallos que no son detectados por este tipo de análisis debido a que el código es finito, pero su ejecución puede variar mucho.

4.3.1 Técnicas

Las técnicas que emplean análisis estático pueden ir de lo más formal (e.g. modelo matemático formal) a lo más informal (e.g. búsqueda de cadenas de texto), y centrándonos un poco en aquellas que son formales, vamos a explicar, de manera sencilla e intuitiva, las más relevantes.

Análisis del Flujo de Control (*Control-Flow Analysis*)

En análisis de flujo de control se trata de una formalización donde el objetivo es conseguir una representación del flujo de control del *software* bajo análisis, el cual suele reflejarse en un grafo de flujo de control o CFG (ejemplo en la figura 4.11). Esta técnica está generalizada para que sea posible aplicarse a cualquier lenguaje, ya que el construir un CFG en un lenguaje u otro puede ser más o menos desafiante. Una de las ideas clave de este análisis es el análisis $k-CFA$, donde k significa el nivel de contexto a tener en cuenta (0 significa realizar un análisis de flujo de control sin tener en cuenta el contexto). El utilizar información de contexto le permite al análisis distinguir, por ejemplo, entre los diferentes posibles valores que puede tomar una variable en lugar de quedarse con la primera aproximación que se encuentre (e.g. se define una variable con un valor, pero se modifica dentro de un bucle).



(a) Algoritmo del que se obtiene el CFG

(b) Grafo de Flujo de Control (CFG) básico

Figura 4.11: Ejemplo básico de CFG

La manera en la que se realiza este análisis es a través de una serie de pasos, los cuales incluyen realizar una especificación dirigida por la sintaxis (i.e. a través de una sintaxis (cadenas aceptadas por el lenguaje u otras cadenas que puedan asociarse a las del lenguaje), crear una especificación relacionada con la instancia bajo análisis), crear una serie de restricciones (i.e. condiciones necesarias por la forma en la que está definida la instancia bajo análisis, que es el código) y resolver estas restricciones a través de otras técnicas (e.g. interpretación abstracta, resolución de restricciones, sistema de tipos (*type system*)).

Cabe destacar que la creación del CFG es relativamente sencilla en los casos en los que se está realizando un análisis intraprocedural. En los casos de análisis interprocedural es relativamente sencillo si el lenguaje empleado no trata las funciones o procedimientos como instancias de primera clase (i.e. las funciones o procedimientos son tratados como un tipo de datos más y, por lo tanto, pueden asignarse a variables, pasarse como parámetro a funciones, etc.). La razón de ello es que la complejidad disminuye mucho debido a que el conjunto de objetos a examinar disminuye, y, además, no es necesaria la misma cantidad de información de contexto, ya que no es necesario observar el estado de cada una de las variables, ya que en el caso de aquellos lenguajes que tratan a las funciones como instancias de primera clase, habrá que determinar qué variables contienen en su valor una función, y en ese caso, determinar qué función es para poder establecer la conexión en el CFG, por lo que necesitaremos información que al ejecutarse, puede incluso variar. Para concluir, aquellos lenguajes o paradigmas que traten a las funciones como instancias de primera clase (e.g. programación funcional) o aquellos que dependiendo del entorno se ejecutarán unas u otras funciones y/o procedimientos (e.g. programación dirigida por objetos: polimorfismo, métodos virtuales, ...), dificultará la construcción del CFG por las dificultades que presentan ya descritas.

Análisis del Flujo de Datos (*Data-Flow Analysis*)

El análisis de flujo de datos trata de obtener información sobre el posible conjunto de valores en un punto concreto de un programa. La manera en la que se ejecuta esta técnica es teniendo inicialmente un CFG, y una vez lo tengamos, dependiendo de lo que queramos conseguir con el análisis, lo recorreremos de una manera u otra y realizaremos unas u otras acciones. Este análisis es de los más empleados por los compiladores, ya que nos permite llegar a conclusiones como, por ejemplo, si hay variables definidas pero que no se utilizan (*liveness analysis*), o si hay expresiones que ya se han calculado con anterioridad y no es necesario volver a calcularlas (análisis de expresiones disponibles o *available expressions*), etc.

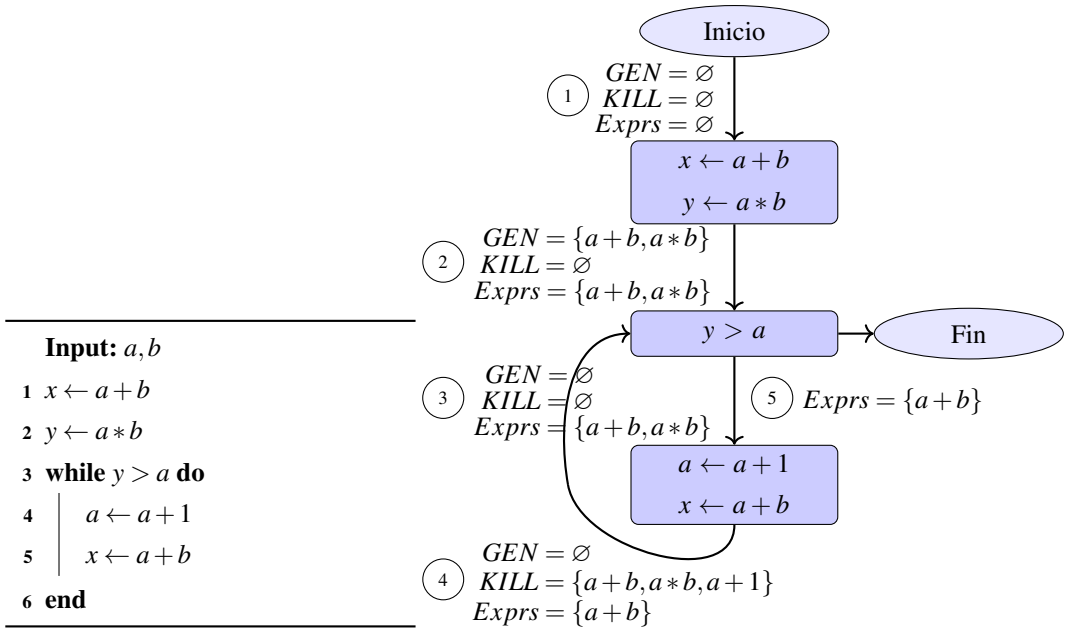
Hay varias formas de realizar este análisis, pero normalmente se parte con el CFG, y a partir del grafo de flujo de control, se pueden formar una serie de ecuaciones donde la solución a dichas ecuaciones es la solución al análisis. Otra posible manera de hacerlo es con un conjunto de restricciones donde se definen, por ejemplo, posibles intervalos de las variables y la solución a estas inecuaciones sería la solución al análisis. Sea cual sea el método aplicado en el análisis, lo que hay que tener claro en este tipo de análisis es la manera en la que se recorre el CFG debido a que es esencial para el tipo concreto de análisis a realizar, ya que hay que tener en cuenta el flujo de control, el camino a recorrer (e.g. tener en cuenta las diferentes ramas en las condiciones) y la información contextual (e.g. mientras recorremos el CFG, si encontramos una función, saltamos hacia la función). Dependiendo de la manera en la que apliquemos estas características, llegaremos a unas conclusiones u otras (i.e. estaremos aplicando un tipo concreto de análisis de flujo de datos). Cuando trabajamos con ecuaciones para el análisis de flujo de datos, la definición de las ecuaciones relevantes son las siguientes:

$$
\begin{aligned}
SUCC(s) &= \{\text{sucesores inmediatos en el CFG de la sentencia s}\} \\
PRED(s) &= \{\text{predecesores inmediatos en el CFG de la sentencia s}\} \\
OUT(s) &= \{\text{sentencias que, inmediatamente después ejecutar la sentencia s, nos llevan a ellas}\} \\
&= TRANS(IN(s)) \\
IN(s) &= \{\text{sentencias que, inmediatamente después de su ejecución, nos llevan a la sentencia s}\} \\
&= JOIN_{s' \in PRED(s)}(OUT(s'))
\end{aligned}
$$

Hay que tener en cuenta que, normalmente, cada tipo de análisis redefine las funciones de transferencia (i.e. *TRANS*) y de unión (i.e. *JOIN*). Un ejemplo sería el análisis de expresiones disponibles, el cual define nuevas ecuaciones y redefine las función de transferencia y de unión de la siguiente manera:

$$
\begin{aligned}
GEN(s) &= \{\text{sentencias disponibles, las cuales no han sido modificadas, tras ejecutar la sentencia s}\} \\
KILL(s) &= \{\text{sentencias no disponibles debido a que en parte o en su totalidad han sido modificadas, tras} \\
&\qquad \text{ejecutar la sentencia s}\} \\
OUT(s) &= GEN(s) \cup (IN(s) \cup KILL(s)) \\
IN(s) &= \bigcap_{s' \in PRED(s)} OUT(s')
\end{aligned}
$$

Como hemos comentado, la manera normal de realizar este análisis es sobre un CFG, pero el CFG no suele estar formado por las sentencias simples, sino que suele estar formado por lo que se conoce como bloques básicos. Estos bloques básicos son un conjunto de sentencias que, mientras no sea necesaria una bifurcación del código o un salto, se seguirán agrupando en el mismo bloque básico, consiguiendo que cada nodo del CFG sea un conjunto de instrucciones que tendrá un único nodo sucesor si se trata de sentencias simples como declaraciones de variables o nodos donde el flujo de control tiene más sucesores o nodos que tienen al nodo como sucesor, como sucede con los bucles o condiciones. A partir de estos bloques se aplican las ecuaciones anteriores en lugar de aplicarse sobre sentencias. El análisis ya queda reducido a recorrer el CFG según se especifique en el tipo de análisis e ir resolviendo las ecuaciones. En el caso del análisis de expresiones disponibles, el CFG se recorre en el orden normal, es decir, se empieza por la entrada y se termina cuando se haya alcanzado un estado de salida en el CFG, y el resultado del análisis serán los valores de las ecuaciones en cada bloque básico. Un ejemplo de este análisis se puede observar en la figura 4.12, donde, paso a paso, calculamos los conjuntos *GEN*, *KILL*, los cuales se calculan teniendo en cuenta el nodo anterior del CFG, y las expresiones disponibles en un momento concreto, las cuales están en el conjunto *Exprs* y se calculan teniendo en cuenta todo el recorrido hasta ese mismo momento. Además, algo que hay que tener en cuenta, como ya se ha comentado, es el modo en el que se recorre el CFG, y para el análisis de expresiones disponibles el flujo es el normal, no inverso, por lo que empezamos por el nodo raíz y recorremos el CFG de manera descendente hasta llegar a la condición de fin, la cual puede ser diferente para cada técnica.



(a) Algoritmo sobre el que se aplica el análisis de expresiones disponibles

(b) Análisis de expresiones disponibles sobre CFG con bloques básicos

Figura 4.12: Ejemplo de análisis de expresiones disponibles (análisis de flujo de datos)

Ejecución Simbólica (*Symbolic Execution*)

La ejecución simbólica está a medio camino entre el análisis estático y el análisis dinámico, por lo que consigue juntar tanto los beneficios de ambos como algunas de sus desventajas, pero no se clasifica como análisis dinámico debido a que no se realiza una ejecución propiamente dicha del programa bajo análisis. La manera en la que este análisis se efectúa es realizando, como su nombre dice, una ejecución simbólica. Inicialmente, las variables definidas por el programa se le asignarán valores simbólicos que, a lo largo de la "ejecución", irán resolviéndose dependiendo del camino que se tome en la ejecución. Al final, lo que tendremos será un árbol donde esté plasmado el recorrido, y mientras vamos recorriendo este árbol, tendremos una serie de condiciones que en un principio no estarán resueltas, pero que su resolución nos guiará por los caminos que podemos y que no podemos recorrer. Los caminos que no se recorrerán serán aquellos que, debido a las condiciones, nunca se cumplan dichas condiciones (e.g. *if (false)*$\{\cdots\}$); otra razón por la que un camino no se recorra puede que sea debido al uso de alguna técnica que evite caer en un bucle infinito, ya que este análisis, al estar realizando una ejecución, aunque no sea real (podríamos decir que está realizando una interpretación), puede caer en un bucle infinito y no salir nunca, y para ello hay diferentes soluciones (e.g. exploración aleatoria, heurísticas).

Los elementos esenciales de este análisis son el dominio simbólico y el árbol de estados. En el dominio simbólico tendremos plasmado el estado simbólico, el cual es el estado en un momento dado de la ejecución, del programa (i.e. el valor simbólico de las variables) y, para cada estado simbólico, tendremos asociado su condición resuelta para el camino que se está tomando (i.e. condiciones booleanas que se cumplen para recorrer ese camino en el programa). Por otro lado, tenemos el árbol de estados, el cual irá ramificándose conforme más caminos vayamos ejecutando, y en los nodos hoja tendremos la solución al análisis, los cuales contendrán los valores de las variables, expresadas simbólicamente o con valores reales en los casos que sea posible inferir los valores (en realidad este árbol de estados será el resultado de ir obteniendo estados

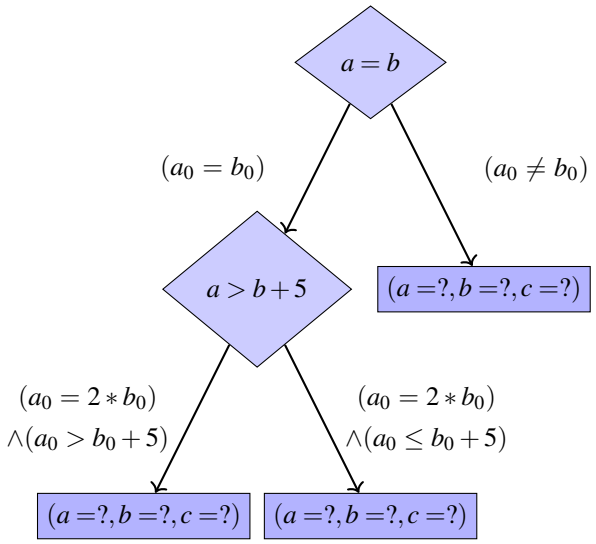
simbólicos, pero sí que es cierto que el resultado se plasmará en el árbol de estados). Un ejemplo de ejecución simbólica lo podemos encontrar en la figura 4.13.

```
   Input: a, b
1  c ← 2 ∗ b
2  if a = c then
3  |  if a > b + 5 then
4  |  |  a ← b
5  |  |  ...
6  |  end
7  end
```

(a) Algoritmo sobre el que se aplica la ejecución simbólica

σ	Condición
$a \mapsto a_0$ $b \mapsto b_0$ $c \mapsto 2 * b_0$	$(a_0 \neq 2 * b_0)$
$a \mapsto a_0$ $b \mapsto b_0$ $c \mapsto 2 * b_0$	$(a_0 = 2 * b_0)$ $\wedge (a_0 \leq b_0 + 5)$
$a \mapsto b_0$ $b \mapsto b_0$ $c \mapsto 2 * b_0$	$(a_0 = 2 * b_0)$ $\wedge (a_0 > b_0 + 5)$

(b) Dominio simbólico



(c) Árbol de estados

Figura 4.13: Ejemplo de ejecución simbólica (sin casos concretos)

La parte que más dificulta este análisis, la cual es la que hasta no hace mucho era inviable debido a que es un problema NP-completo (i.e. subconjunto de problemas que pertenecen al conjunto NP, los cuales son problemas para los que, actualmente, no se sabe si existe algún algoritmo que los resuelva con una complejidad temporal polinómica o menor), es la solución de fórmulas de lógica de primer orden, las cuales son de las que hemos hablado que decidirán los caminos por los que podemos o no podemos ir (i.e. condición de los caminos que decidirán la ramificación del árbol de estados). La manera que tenemos para resolver estas fórmulas es a través de solucionadores SAT o SMT, ambos con complejidades exponenciales. Lo que resuelven estos solucionadores son fórmulas de lógica de primer orden. En el caso de SAT, toda variable se tratará como valor booleano (e.g. un entero los descompondrá en sus bits y tendrá tantas variables booleanas como bits tuviera la representación del entero, lo que sería 32 variables en el caso de un sistema de 32 bits con 4 bytes de tamaño en el entero), y esto hace que el uso de SAT sea inviable en la mayoría de problemas. Lo que ha hecho que recientemente esta técnica sea viable han sido mejoras en los SMT, los cuales son una generalización de los

SAT, donde alguna función y/o símbolo tiene interpretaciones adicionales (i.e. un conjunto de variables binarias se puede sustituir por predicados (i.e. función que devuelve un valor booleano) que trabaje sobre un conjunto que no sea binario), consiguiendo resolver uno de los mayores problemas de los SAT (e.g. ahora, en lugar de tener $8 * 4 = 32$ variables binarias, utilizamos un predicado que trabaje directamente sobre el conjunto de los enteros y trate los números enteros como una única variable y no sobre su representación binaria, pasando de tener que resolver 32 variables a solo tener que resolver 1), ya que un problema muy sencillo en un SAT podía ser inviable de resolver, mientras que en un SMT puede ser totalmente viable.

Para aplicar esta técnica son necesarios casos concretos. En la figura 4.13 no lo hemos hecho por brevedad, pero hay que dar valores a las variables para poder aplicarse.

Fuentes

Para la realización de las secciones anteriores, relacionadas con las diferentes técnicas de análisis estático, hemos empleado las siguientes fuentes: [14], [15], [16], [17], [19].

4.4 Análisis Dinámico

A diferencia del análisis estático, el análisis dinámico se realiza en *software* que realmente se ejecuta. Debido a esto, el análisis estático y dinámico se complementan, siendo dos herramientas diferentes que, como norma general, deberían de emplearse juntas con el fin de obtener los mejores resultados, ya que las limitaciones de la una los suple la otra y viceversa. Al igual que con el análisis estático, estamos acostumbrados a utilizar este tipo de análisis en el día a día, ya que algo tan común como un depurador de código, en realidad es una herramienta que está empleando el análisis dinámico con el fin de darnos información sobre el código que se inspecciona. Los objetivos de este tipo de análisis pueden ser variados, como pueden ser calcular métricas (e.g. cobertura del código) o buscar vulnerabilidades, entre otras. Estas técnicas pueden ser más o menos automatizables, y siguiendo el ejemplo del depurador, esta herramienta es más manual que automática, ya que el fin es ayudar al desarrollador a buscar ciertos errores dándole al desarrollador la libertad de elegir los puntos donde realizar paradas para observar el estado en ese momento entre otras características de los depuradores, pero si un desarrollador no le dice al depurador qué hacer, simplemente ejecutará el programa sin hacer nada especial, por lo que no es una herramienta que podamos clasificar como automática por la naturaleza de la misma. Por otro lado, sí que hay otras herramientas que son más automatizables, pero esto es debido a que su finalidad u objetivos serán otros que sí que permitan la automatización como, por ejemplo, buscar caminos que lleven a un estado de error en el programa, o buscar los caminos del programa que más se ejecutan, etc. Esto no hace que una herramienta sea mejor o peor, sino que son diferentes y buscan realizar tareas distintas.

Como hemos comentado, el análisis dinámico y el estático se suplementan, no es que sean contrarios, por lo que algunas de las ventajas del análisis dinámico son que las implementaciones suelen ser independientes del lenguaje objetivo, nos permite detectar incidentes de naturaleza dinámico que en el análisis estático son falsos negativos o que nos permite tener un conocimiento contextual mayor debido a que se está realizando el análisis sobre el código en ejecución, por lo que sabemos exactamente qué está sucediendo, al contrario que ocurre en el análisis estático. Por otro lado, tiene una serie de inconvenientes que también suplen las ventajas del análisis estático, como son que no suele emplearse en etapas tempranas de los proyectos debido a que el interés de este análisis está en ejecutarlo en el sistema completo, o que el tiempo que tarde en realizarse es el

tiempo que tarde el programa en ejecutarse (e.g. si un proyecto tarda un día en ejecutarse, el análisis tardará un día, o más, en ejecutarse) o que es más complicado determinar la localización exacta de los errores.

4.4.1 Técnicas

Aunque no nos demos cuenta, estamos muy acostumbrados a ver constantemente técnicas de análisis dinámico en el mundo del desarrollo. El gran ejemplo sería el depurador, o más conocido como *debugger*, y es que las técnicas que un depurador emplea son, en su gran mayoría, propias del análisis dinámico. A continuación, vamos a explicar, de manera sencilla e intuitiva, las más relevantes.

Instrumentación

La instrumentación es la técnica a través de la cual se introduce instrucciones, o de manera más general, comportamiento, en un binario. Esta técnica se aplica, normalmente, cuando queremos obtener ciertas métricas de un binario en ejecución, como por ejemplo, los caminos por los que pasa dada una configuración. Las situaciones más comunes en las que se utiliza la instrumentación son: creación de perfiles de eficiencia, optimización, pruebas, detección de errores y virtualización.

En el caso de la detecciones de caminos de un binario, el objetivo puede ser querer saber hasta que punto un conjunto de pruebas pasa por todos los caminos posibles. Para ello, lo que se realiza es, al principio de cada camino (e.g. dentro de cada elemento de control *if*, *while*, etc.), insertar en el binario un contador que vaya aumentando cada vez que se pase por ese camino. Cuando se ejecuten las pruebas, recogemos los valores de los contadores, eliminamos dichos contadores para evitar perder eficiencia, y se obtienen conclusiones de los resultados de los contadores. Las conclusiones que se pueden obtener con una técnica tan sencilla como la descrita pueden ser muy interesantes, como por ejemplo, caminos muy transitados donde habrá que intentar mejorar la eficiencia, caminos poco probados donde habrá que realizar más pruebas o, caminos que tengan un historial de vulnerabilidades detectadas, realizar más pruebas para dichos caminos ya que donde se encuentra una vulnerabilidad, la probabilidad de encontrar otras es elevada.

Hay diferentes maneras de aplicar instrumentación:

- Instrumentación binaria:
  - En ejecución: esta situación es muy típica, por ejemplo, en los depuradores. Cuando insertamos un punto de ruptura o queremos que se vigile el valor de una variable para que se nos notifique cuando cambie su valor, se está realizando este tipo de instrumentación. La razón de utilizar esta instrumentación es que los depuradores utilizan los binarios para poder ejecutarlos, por lo que esto nos dice que estamos ante análisis dinámico, y lo que nos indica que se realiza instrumentación mientras el binario se ejecuta es que el depurador, por defecto, no va a insertar instrucciones en todas las instrucciones para que solo pongamos un punto de ruptura, sino que insertará las instrucciones necesarias en las instrucciones que le digamos en el momento que se lo digamos, ya que de cualquier otro modo, la eficiencia de los depuradores sería mínima.
  - Antes de la ejecución: si lo que se quiere es obtener algún tipo de métrica, como por ejemplo la cantidad de veces que se ejecutan las funciones de un binario, una manera de hacerlo es insertando instrucciones que nos permitan identificar la llamada a una función en el código de la función. Para

realizar esta tarea, lo normal será insertar las instrucciones en las funciones antes de ejecutarlo, y, aunque no es el único modo (podríamos realizar esta tarea en ejecución inicializando los contadores al pasar por primera vez por cada función), tiene sus ventajas y desventajas. Como ventaja tenemos que los resultados serán más completos, ya que tendremos resultados incluso de aquellas funciones que no se ejecuten. Por el contrario, si no lo hacemos de esta manera, los resultados pueden ser más pobres. Esta técnica se conoce como instrumentación estática.

- Instrumentación en el lenguaje intermedio: el lenguaje intermedio es el que está a medio camino entre el lenguaje de alto nivel (e.g. C) y el lenguaje de bajo nivel (e.g. ensamblador). La principal ventaja que obtenemos de realizar la instrumentación con instrucciones de código intermedio es que obtenemos mucho más control sobre lo que realizamos, pero a la vez no bajamos a bajo nivel, donde perdemos mucha flexibilidad.

- Instrumentación en el código fuente: se insertan instrucciones sobre el código fuente, consiguiendo en este caso toda la potencia de un lenguaje de alto nivel. El principal inconveniente que nos encontramos con esta técnica es que si queremos aplicarla sobre un lenguaje compilado, será necesario realizar la instrumentación antes de compilar (hay maneras de conseguirlo, aunque nada sencillas, como podría ser pasar el lenguaje a código máquina e insertarlas en la zona de memoria apropiada). En el caso de los lenguaje dinámicos, será posible aplicar esta técnica directamente mientras se está ejecutando (e.g. función *eval()* en el lenguaje *Javascript* o *PHP*).

Ejemplos de plataformas que nos permiten aplicar esta técnica en entornos Linux son Pin [20], Dyninst [21] y DynamoRIO [22].

Exploración de Caminos (*Fuzzing*)

La técnica de *fuzzing* se basa en la exploración de caminos de un *software*, normalmente se realiza de manera automática. Lo que se suele hacer para aplicar esta técnica es generar cadenas aleatorias o semialeatorias que se le pasarán al sistema con el fin de buscar anomalías. Las anomalías que esta técnica puede encontrar son variadas: vulnerabilidades, fallos que ocasionan un error inesperado, fluctuación en el rendimiento, etc. Algo a destacar de esta técnica es que suele emplearse junto con instrumentación con la finalidad de obtener unos resultados más completos de las pruebas que se quieran realizar en el análisis.

Este análisis ha sido uno de los más empleados en el análisis dinámico y uno de los primeros en emplearse. La primera técnica conocida que utilizaba *fuzzing* es conocida como ”la técnica del mono”, la cual se basa en el teorema del mono infinito, teorema que afirma que si tenemos tiempo infinito y lo empleamos en teclear de manera totalmente aleatoria, la probabilidad de obtener cualquier texto (e.g. *El Quijote*) es muy elevada. Desde que se desarrolló esta técnica, mucho ha evolucionado la técnica del *fuzzing* pasando incluso a utilizar *fuzzers* (i.e. herramientas que aplican *fuzzing*) que comprenden totalmente protocolos y son capaces de realizar una comunicación correcta pero con datos aleatorios o semialeatorios donde es posible. Esta evolución ha permitido mejorar la automatización de la técnica permitiendo aplicarlo sobre sistemas que, si no se realiza una comunicación correcta según un protocolo, no se puede continuar probando el sistema en profundidad por no poder continuar.

A continuación, realizamos una pequeña clasificación de la técnica de *fuzzing* según la manera en la que se aplica:

- *Fuzzer* estático y aleatorio: este tipo de *fuzzer* son los que aplican el teorema del mono infinito. Generan cadenas aleatorias con mayor o menor efectividad dependiendo de la implementación concreta. Una de las mejoras que se suele utilizar es utilizar una librería que dote de cadenas conocidas que suelen activar ciertas vulnerabilidades (e.g. cadenas largas, "1' or '1'='1").

- *Fuzzer* estático y aleatorio basado en plantilla: bajo conocimiento de protocolos, pero basándose en una plantilla, se puede realizar una comunicación simple y que no requiera de intervención dinámica (e.g. cálculo de *checksum*), ya que en este caso es estático. La plantilla lo que nos dice es como realizar la comunicación con el protocolo. Este *fuzzer* es una gran mejora respecto a otros más simples como los que son totalmente aleatorios, ya que nos permite tomar caminos en el sistema que con una aproximación aleatoria, la probabilidad de tomar los mismos caminos es casi nula.

- *Fuzzer* en bloque: tiene el mismo comportamiento que un *fuzzer* estático y aleatorio basado en plantilla con la diferencia de que este sí que es dinámico y puede realizar pequeñas tareas que requieren de un comportamiento dinámico (e.g. cálculo de *checksum*, cálculo de longitud de cadenas).

- *Fuzzer* basado en evolución o de generación dinámica: esta técnica se basa en que, inicialmente, no se tiene conocimiento del protocolo que está siendo utilizado en la comunicación, pero en base a realizar pruebas, la retroalimentación de los errores le ayuda a aprender el protocolo. Es una evolución respecto a otras técnicas que, por ejemplo, se les tiene que especificar mediante una plantilla el protocolo de comunicación. La ventaja frente a otros está clara, y es que es una generalización que nos permite un uso dinámico sin tener que realizar configuraciones concretas para cada protocolo para el que se quiera utilizar. La principal ventaja es que, dependiendo de la complejidad del protocolo, la secuencia de la comunicación podría fallar constantemente.

- *Fuzzer* basado en simulaciones o basados en modelo: esta técnica se basa en el uso de un modelo o una simulación del protocolo. Otra alternativa es tener la implementación completa del protocolo a utilizar. Esta es la técnica más concreta, es decir, menos general, pero al tener un conocimiento total del protocolo, los resultados tendrán una precisión mucho mayor a diferencia de otros.

El uso de una técnica u otra dependerá de las necesidades a la hora de realizar el análisis. Algunos ejemplos de *fuzzers* son Dowsing [23], Intent [24] y SNOOZE [25].

Sustitución o Ampliación (*Hooking*)

Aplicar *hooking* significa sustituir, ampliar o modificar el comportamiento de, normalmente, una función. El ámbito al que se aplique no es relevante, lo relevante es lo que significa, y es que podemos ver esta técnica como una especialización de la instrumentación, pero no se considera lo mismo debido a que no tiene el mismo alcance ni los mismos objetivos. Con esta técnica podemos, si queremos, sustituir totalmente el comportamiento de una librería en nuestro sistema y utilizar nuestra implementación con algún objetivo. Se trata de una técnica ampliamente utilizada y muy útil debido al control que nos proporciona.

La manera de aplicar *hooking* puede variar dependiendo del ámbito sobre el que se aplique. Así pues, hay sistemas que ofrecen un mecanismo estándar que nos permite aplicar *hooking* (e.g. la variable *__malloc_hook* nos permite indicar un *hook* (i.e. dependiendo del alcance puede tratarse de una función, un programa, etc. que se ejecutará en lugar de la función, programa, etc. original) para la función estándar *malloc*[8]).

[8]Documentación de *malloc* (*hooks*): `https://www.gnu.org/software/libc/manual/html_node/Hooks-for-Malloc.html`

Conforme estos *hooks* los podamos definir desde un grado de abstracción mayor (i.e. no es lo mismo asignar un *hook* a nivel de función que aplicarlo a nivel de sistema, ya que a nivel de sistema tendremos control total y a nivel de función, dependemos del orden en el que se ejecuten las instrucciones hasta que se llegue a ejecutar el *hook* que hayamos definido), mayor será el control que tengamos.

Una manera simple que tenemos de realizar esta técnica, en sistemas GNU/Linux, es a través de la variable de entorno *LD_PRELOAD*, la cual es utilizada por el enlazador y, si encuentra definida la variable, utiliza la librería dinámica que esté definida en dicha variable. Hay un orden especificado en los sistemas GNU/Linux para buscar las librerías dinámicas (i.e. ficheros con una extensión "so" o "so.X" (X es la versión de la librería) en /lib, /usr/lib, y contenido del fichero /etc/ld.so.conf en sistemas GNU/Linux que se ejecutan de manera dinámica en los binarios, es decir, que no se encuentran dentro del binario de manera estática, sino que se carga y utiliza cuando se ejecutan), el cual se especifica a continuación:

1. Sección dinámica *DT_RPATH* del binario ELF. Estas secciones pueden contener rutas separadas por punto y coma donde se buscará, en tiempo de enlazado (i.e. cuando el enlazador dinámico, *ld* en sistemas GNU/Linux, esté ejecutándose realizando sus tareas típicas), las librerías, y si se encuentran en dichas rutas las librerías que se utilizan en el binario, se utilizarán. La sección dinámica *DT_RPATH* solo se aplicará si no está definida la sección dinámica *DT_RUNPATH*.

2. Variable de entorno *LD_LIBRARY_PATH*. Esta variable de entorno puede contener rutas separadas por punto y coma donde se buscarán las librerías dinámicas de dichas rutas si se están utilizando en el binario. Esta variable de entorno puede utilizarse junto con la variable de entorno *LD_PRELOAD* si se busca una librería dinámica en concreto, y la librería dinámica que se especifique en esta variable, será la única que se aplique, mientras que el resto, aunque se utilicen en el binario, no se cargarán. Por motivos de seguridad, lo descrito no se aplicará si el binario tiene activos los bits *setuid* o *setgid*, ya que si no fuera así, la sustitución de librerías utilizadas en dichos binarios podrían modificarse y añadir, por ejemplo, un comando que inicie una nueva sesión de terminal, pero al tener alguno de estos bits activos, se abriría con permisos de *root* llevándonos a una rápida y sencilla escalada de privilegios.

3. Sección dinámica *DT_RUNPATH* si está presente. Es la versión actualizada de la sección dinámica *DT_RPATH*, que por motivos de retrocompatibilidad, sigue siendo válida cuando esta nueva sección dinámica no está definida.

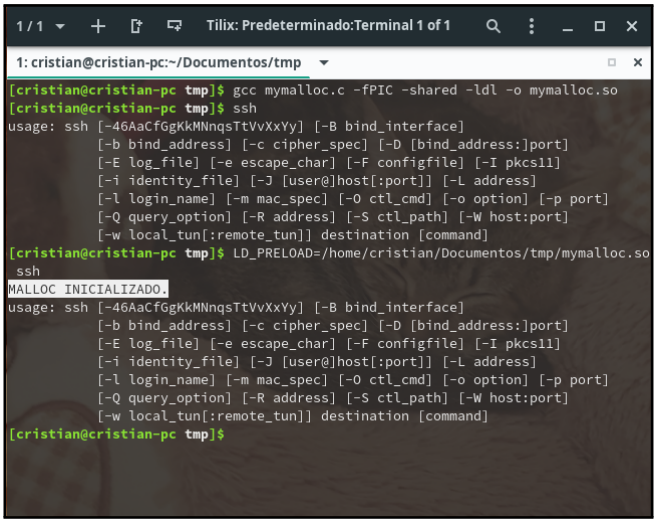
4. Directorios basados en el fichero /etc/ld.so.cache, el cual se basa en el fichero /etc/ld.so.conf que contiene instrucciones para especificar directorios donde buscar librerías dinámicas. Este paso se omite si el binario que se ejecuta se compiló con la opción "*-z nodefaultlib*".

5. Directorios por defecto definidos en el sistema, los cuales son /lib y /usr/lib. Este paso se omite si el binario que se ejecuta se compiló con la opción "*-z nodefaultlib*".

La manera efectiva de comprobar qué librería se está utilizando por el binario es a través del comando *ldd*, el cual muestra la lista de dependencias de librerías dinámicas. Un ejemplo de esta técnica la podemos encontrar en la figura 4.14.

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <dlfcn.h>

static void* (*real_malloc)(size_t
    ) = NULL;

void* malloc(size_t size)
{
    if(real_malloc == NULL)
    {
        real_malloc = dlsym(
            RTLD_NEXT, "malloc");
        fprintf(stderr, "MALLOC
            INICIALIZADO.\n");
    }

    return real_malloc(size);
}
```

(a) Código del fichero *mymalloc.c*



(b) Resultado de aplicar *hooking* sobre la función *malloc* en el binario /usr/bin/ssh con la carga de la variable de entorno *LD_PRELOAD*

Figura 4.14: Ejemplo de *hooking* sustituyendo la función *malloc* en GNU/Linux

Fuentes

Para la realización de las secciones anteriores, relacionadas con las diferentes técnicas de análisis dinámico, hemos empleado las siguientes fuentes: [26], [27], [28], [29].

4.5 Aplicación de la Inteligencia Artificial

En los últimos años, la inteligencia artificial ha sufrido unos avances a gran velocidad en todos, o casi todos, los campos de la informática. No es nada nuevo, ya que desde antes de los años 60 ya se hacían avances en este área, pero la gran revolución vino de la mano del *Deep Learning*, el cual es una ramificación del *Machine Learning* y que se dedica a utilizar redes neuronales con muchas capas ocultas. El avance en potencia computacional nos hizo capaces de poder utilizar estas redes neuronales, las cuales eran inviables hace 40 años, y desde entonces no se ha parado de investigar en este campo. El campo del análisis *software* y la ciberseguridad no son una excepción para la inteligencia artificial y aunque han sido de los campos donde más difícil se ha visto su aplicación, también se ha aplicado, y no sin éxito [30].

La manera en la que suele aplicarse algoritmos de *machine learning* al análisis de *software* con el fin de buscar vulnerabilidades suele basarse en los siguientes pasos:

1. Extracción: se obtiene el código fuente u otra representación del código. Otras posibles representaciones podrían ser el lenguaje intermedio o el lenguaje ensamblador. El nivel de extracción dependerá del objetivo que se quiera alcanzar, pues se puede realizar una extracción a nivel de fichero, de módulo o de sistema.

2. Obtención de métricas: una vez tenemos el código, se extraen ciertas métricas típicas del análisis de *software* como puede ser el número de funciones, el número de lineas, medidas de complejidad del código basado en heurísticas, etc.

3. Aplicar técnica: una vez tenemos las métricas, las utilizamos para aplicar la técnica que se quiera, la cual puede crear un modelo de comportamiento, tomar decisiones en base a las métricas, etc.

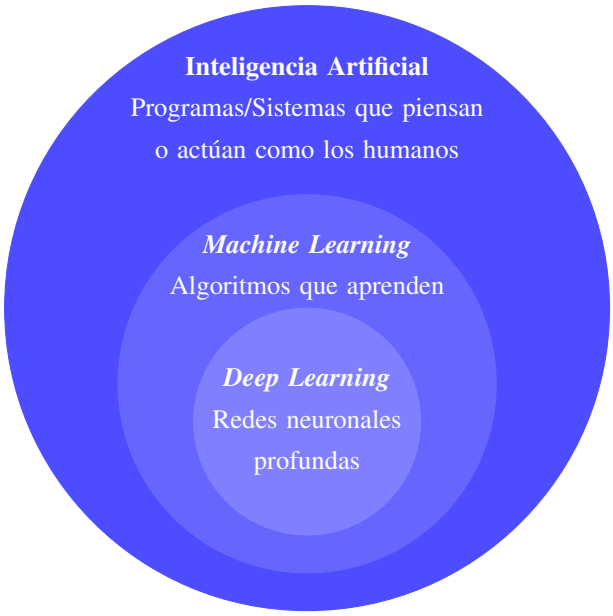


Figura 4.15: Inteligencia artificial, *machine learning* y *deep learning* (clasificación)

La principal ventaja al pasar de utilizar analizadores basados en reglas o patrones, los cuales definen el comportamiento de las vulnerabilidades, es que se reduce en gran medida el número de falsos positivos y falsos negativos. Aún así, en el caso de los algoritmos clásicos de *machine learning*, seguimos necesitando de definir de alguna manera el comportamiento que definen a las vulnerabilidades. El caso del *deep learning* parece ser el más alentador de todos, ya que nos permite no tener que definir de manera manual el comportamiento de las vulnerabilidades, sino que de manera automática lo infiere y es capaz de aprender las características típicas de las vulnerabilidades. [31]

Un ejemplo de analizador que utiliza *deep learning* es VulDeePecker [32]. Este analizador es capaz de detectar múltiples vulnerabilidades, 4 en concreto, de las cuales, 3 de ellas no son detectadas por analizadores clásicos. Al utilizar *deep learning*, no ha sido necesario definir el comportamiento de las amenazas que es capaz de detectar. Además, los resultados obtenidos en la detección de vulnerabilidades mejora la de otros analizadores actuales como VUDDY [33]. El analizador funciona de la siguiente manera:

1. Extracción: utiliza un *dataset* (i.e. conjunto de datos) que contiene programas que son vulnerables y no vulnerables [34], a partir del cual se hace una transformación de sus ficheros, y cada fichero representa un dato. Esta transformación consiste obtener una representación intermedia de los ficheros que contenga las relaciones semánticas originales, a los cuales llama *code gadgets*. Una vez se obtienen estas representaciones intermedias, se obtiene un vector numérico que serán las verdaderas entradas para la red neuronal. Con el fin de ser capaces de obtener la localización de las vulnerabilidades, la granularidad utilizada para obtener los vectores es a nivel de función y a nivel de fichero (una granularidad mayor, como podría ser a nivel de sistema, llevaría a una gran pérdida de información debido a que el

grado de abstracción sería demasiado elevado, conduciendo a perder información más sutil como la línea en la que se encuentra cierta vulnerabilidad).

2. Red neuronal: el problema a abordar es muy complejo, pero algo a tener en cuenta es que se parece mucho al problema del procesamiento del lenguaje natural, ya que igual que el lenguaje natural, los lenguajes de programación son una manera de expresarse. En estos campos es muy importante el contexto, es decir, el orden en el que aparecen las palabras es vital, y recordar lo que ya hemos visto con anterioridad da un significado u otro a lo que se quiere expresar. Basándonos en lo comentado, el analizador se decide por utilizar una red neuronal recurrente (RNN; ejemplo en la figura 4.16), la cual suple de los problemas que hemos comentado al darle importancia al orden y a la memoria. El problema principal a abordar en este tipo de red neuronal es el problema del desvanecimiento del gradiente (*vanishing gradient problem*), lo cual puede llevar a entrenamientos poco eficaces. Para solucionar este problema, de manera parcial, utiliza células de memoria BLSTM en la RNN.

3. Entrenamiento: por último, lo que realiza es el entrenamiento a partir del *dataset*.

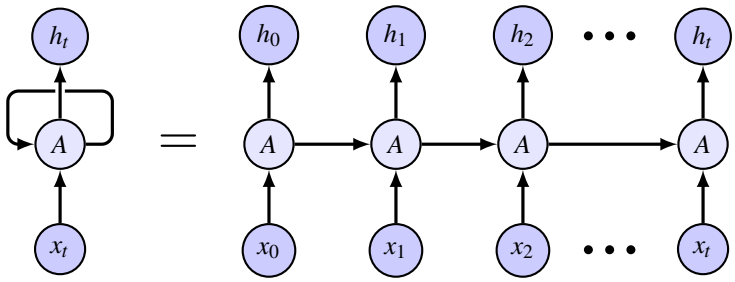


Figura 4.16: Ejemplo de red neuronal recurrente (RNN) plegada y desplegada

A modo de resumen, las fases que VulDeePecker ejecuta son las siguientes:

1. Obtención de *code gadgets* y clasificación en vulnerable (i.e. 1) o no vulnerable (i.e. 0).
2. Obtención de los vectores que representan los *code gadgets*.
3. Entrenamiento de la RNN con células BLSTM.
4. Obtención de los vectores que representan los programas que se quieren analizar.
5. Detección de vulnerabilidades sobre los programas que se quieren analizar.

Al igual que en muchas de las áreas de la informática, lo más seguro es que la inteligencia artificial haya llegado al campo del análisis *software* y de la ciberseguridad para quedarse. Los resultados son bastante prometedores, ya que en muchas ocasiones mejoran los de las herramientas y/o técnicas clásicas.

Capítulo 5

Tecnologías

A lo largo del desarrollo de BOA, el analizador de vulnerabilidades que se discute en el presente documento, se han utilizado una serie de tecnologías que han ayudado a obtener un analizador lo más completo posible. Algunas herramientas han ayudado al desarrollo en sí y otras ha mejorar la calidad del resultado, obtener una documentación del proceso o buscar errores. A lo largo de este capítulo se va a explicar las tecnologías empleadas, la razón de su uso y los problemas que han solucionado.

5.1 Python

Python[1] es un lenguaje de programación que en los últimos años ha ganado mucha popularidad en la comunidad de desarrolladores debido a varios factores, llegando a estar en primera o segunda posición en muchos de los *rankings* que se dedican a medir la popularidad o el uso de los lenguajes de programación. Un ejemplo de *ranking* que mide la popularidad de los lenguajes de programación es TIOBE[2], el cual se basa en la cantidad de búsquedas realizadas sobre los diferentes buscadores web (e.g. Google, Bing) para poder realizar la clasificación de los lenguajes de programación, donde más búsquedas significa mayor popularidad, y así es el caso de Python, el cual, desde el año 2014 hasta la actualidad, no ha dejado de aumentar en popularidad como podemos observar en la figura 5.1.

Python es un lenguaje de programación interpretado, lo que quiere decir que no es compilado, aunque es posible compilar módulos si es necesario (e.g. mayor eficiencia). No tiene un único paradigma, sino que se trata de un lenguaje de programación multiparadigma al cual se le asocian principalmente los paradigmas imperativo (ejecución de instrucciones secuencial, no necesariamente de manera ordenada) y orientado a objetos (creación de instancias a partir de objetos que son descritos por clases), mientras que, de manera más secundaria, el paradigma funcional (funciones de orden superior como *map* o *filter*, funciones *lambda*, funciones como instancias de primera clase, etc.). Otra de las características que diferencia a Python del resto de lenguajes es que separa los bloques de código o ámbitos a partir de espacios o tabulaciones, lo cual no es algo usual en los lenguajes de programación más populares pero gracias a esta manera de funcionar, de una manera visual se puede observar de manera sencilla el flujo de un programa escrito en este lenguaje de programación.

[1] Sitio oficial de Python: `https://www.python.org/`

[2] *Ranking* TIOBE: `https://www.tiobe.com/tiobe-index/`

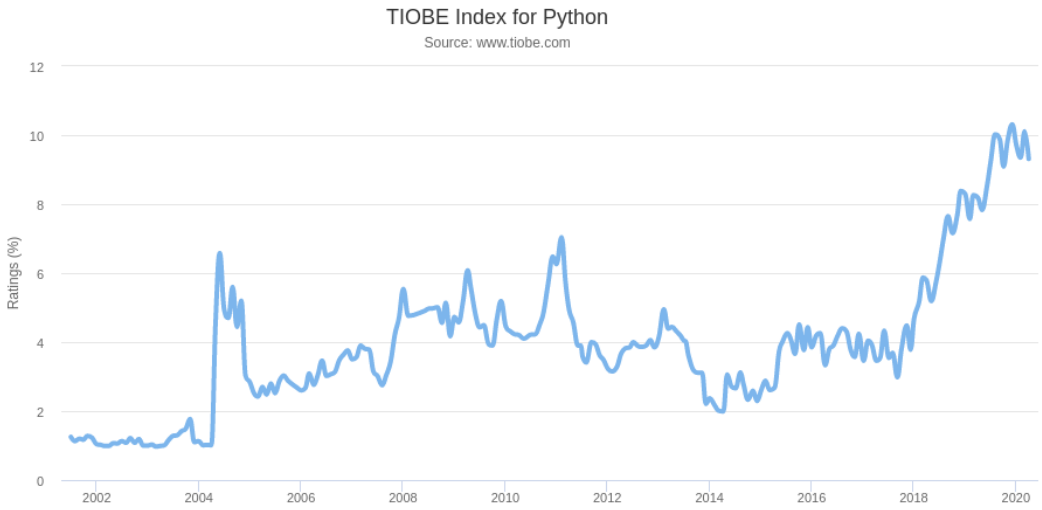


Figura 5.1: Popularidad de Python a lo largo de los años (TIOBE)

Figura 5.2: Logo de Python

Los motivos por los cuales se ha empleado Python para el desarrollo de BOA son los siguientes:

- Curva de aprendizaje poco pronunciada: Python es sencillo y muy flexible. Aprender Python puede llevar muy poco tiempo, ya que no tiene una sintaxis que sea muy distinta del resto de lenguajes y, además, tiene muchas características que son amigables de cara al desarrollador. Por otro lado, hay muchas situaciones en las que nos enfrentamos a problemas sencillos que se resuelven de maneras muy sencillas en la mayoría de lenguajes de programación, pero llevan un trabajo extra que Python simplifica, como puede ser una doble condición de números enteros, y este tipo de características se pueden expresar en Python, normalmente, de una manera sencilla (e.g. $a > 5 \wedge a < 10 \rightarrow 5 < a < 10$) consiguiendo mejorar la legibilidad del código, entre otras características (e.g. recorrido inverso de un *array*: $a[-1]$; obtención de una subcadena donde se elimina el último carácter: $str[0:-1]$).

- Desarrollo ágil: Python ahorra grandes cantidades de tiempo debido a todas las características que tiene implementadas de manera nativa o que tiene en su biblioteca estándar. Lo que en muchas ocasiones se consigue con un lenguaje de programación como C++ en un mes, con Python se puede conseguir en mucho menos tiempo, lo cual no significa que Python sea mejor porque reduce el tiempo de desarrollo frente a otros lenguajes, sino que es diferente y si una necesidad es un desarrollo rápido, Python es una opción a tener en cuenta, como ha sido el caso de BOA donde un requisito era obtener un analizador

completo y con unos objetivos y/o requisitos exigentes en un rango de tiempo acotado de unos pocos meses.

- Módulos de terceros: la comunidad de Python es algo a tener en consideración, ya que la cantidad de módulos disponibles para descargar es inmensa. Todos estos módulos están disponibles en el *Python Package Index*, también conocido como PyPI[3]. En este repositorio podemos encontrar gran cantidad de *software* preparado para ser utilizado al momento a través de utilidades como Pip[4], Conda[5] o Poetry[6]. En el caso de BOA hemos empleado Pip para la descarga de módulos de tercero que han facilitado el desarrollo.

- Importación de módulos: debido a que Python es un lenguaje dinámico, la flexibilidad que nos da es mucho mayor que el de uno estático, y dentro de esta flexibilidad está la de trabajar de manera simple con otros módulos. Además, Python está organizado en módulos, así que facilita la manera en la que se gestionan. Todo esto nos lleva a una gestión flexible y sencilla de los módulos, permitiendo tener un diseño modular de nuestras aplicaciones, lo cual es uno de los objetivos principales que se persiguen con el desarrollo de BOA.

- Lenguaje extendido: con el fin de que cualquiera pueda implementar sus propios módulos para BOA, Python es una buena opción debido a que es un lenguaje ampliamente extendido en la comunidad de desarrolladores. Además, en el área de la ciberseguridad, Python es, junto a otros (e.g. C, Perl), uno de los lenguajes de programación más empleados.

Las ventajas de Python son muchas, y algunas las hemos mencionado en las lineas anteriores, pero claramente también tiene desventajas al igual que todo lenguaje de programación. Las principales desventajas que vamos a mencionar son las siguientes: **poco eficiente** y **pérdida de legibilidad y transparencia** a medida que el proyecto crece. A continuación, discutimos ambas desventajas de cara al lenguaje en sí y de cara a BOA:

- Eficiencia: se sabe que Python no es rápido, de echo es considerado un lenguaje lento a la hora de compararlo con otros, pero es suficiente para la mayoría de proyectos. Para conseguir eficiencia recurre normalmente a lenguajes como C o C++, ya que Python tiene soporte oficial para crea módulos en dichos lenguajes de programación. En el caso de BOA no era un objetivo crear un analizador que fuera eficiente, pero desde luego era algo deseable, y lo mencionamos por ese motivo. Aún así, los módulos implementados en BOA no han sufrido de un impacto crítico en la eficiencia y no es un problema que necesariamente tenga que afrontar BOA

- Pérdida de legibilidad y transparencia: este sí que ha sido un problema importante para el desarrollo de BOA, ya que conforme los ficheros iban creciendo en tamaño y complejidad, la legibilidad se volvía complicada y la transparencia casi nula, ya que no era fácil encontrar los puntos en los que, por ejemplo, los tipos de datos eran correctos o no, y esto es un problema típico de los lenguajes de programación con un tipado dinámico. En las últimas versiones de Python, en concreto en la versión 3.6, se han implementado características para intentar solucionar este problema permitiendo especificar el tipo de las variables en las funciones y el tipo esperado (e.g. $foo(s : str) \rightarrow str$), lo cual aumenta la legibilidad del código, pero no evita que, de manera accidental, se utilicen tipos de datos diferentes debido a que,

[3]Sitio oficial de PyPI: `https://pypi.org/`
[4]Utilidad Pip: `https://pypi.org/project/pip/`
[5]Utilidad Conda: `https://docs.conda.io/projects/conda/en/latest/`
[6]Utilidad Poetry: `https://python-poetry.org/`

por defecto, Python no lanza errores cuando recibe tipos de datos diferentes a los especificados como se especifica en la documentación del módulo *typing*[7]. Este problema ha llevado a refactorizar varios ficheros de BOA en diferentes ocasiones debido a que, para facilitar la ampliación del analizador a través de módulos, lo cual es uno de los objetivos definidos, ha sido necesario mejorar la legibilidad del analizador.

5.1.1 Pylint

Pylint[8] es una herramienta creada para detectar errores, intentar forzar un estándar de estilo de código y buscar *code smells* (i.e. características que indican posibles problemas que, aunque no necesariamente indican un problema ni son un problema en sí, normalmente son indicativos de problemas subyacentes existentes o futuros; ejemplo de ello puede ser la duplicación de código, lo cual es muy probable que en un futuro se modifique parte del código y se olvide modificar el código duplicado, conduciendo a problemas difíciles de depurar). Pylint tiene un comportamiento por defecto, pero este puede ser modificado a través del fichero *pylintrc* (otra opción es el fichero "*.pylintrc*").

Figura 5.3: Logo de Pylint

La detección de errores y *bad smells* en Pylint se realiza basándose en una lista[9] donde define códigos (e.g. E0213: el parámetro *self* no está especificado como primer argumento; E1121: basándose en la configuración concreta, se han sobrepasado el número de parámetros máximos para la definición de una función) para la identificación de los mismos. Respecto al estándar de código que intenta forzar, se basa en los *Python Enhancement Proposal* (i.e. documento destinado a proporcionar información a la comunidad de Python o que describe una nueva característica para Python o para sus subprocesos o entorno [35]), más conocido como PEP o PEPs, y en concreto se basa en el PEP 8[10], el cual define una guía de estilo para programas escritos en Python.

Los códigos de Pylint son la parte más importante, ya que describen los problemas que se encuentran en el código cuando se analiza. Dichos códigos están clasificados y se identifican a partir de una letra mayúscula, la cual es la primera letra que identifica a los códigos seguido de 4 números. Las diferentes categorías de códigos son las siguientes, ordenados por gravedad:

[7] Módulo *typing*: `https://docs.python.org/3/library/typing.html`
[8] Sitio oficial de Pyint: `https://www.pylint.org/`
[9] Descripción de errores y códigos de Pylint: `http://pylint-messages.wikidot.com/`
[10] Python PEP 8: `https://www.python.org/dev/peps/pep-0008/`

1. Fatal (F): algo evita que Pylint pueda seguir procesando.
2. Error (E): posibles errores en el código.
3. Aviso (W): problemas de estilo de código o errores menores de programación.
4. Refactorización (R): métricas de buenas prácticas que no se cumplen (*bad smells*).
5. Convención (C): no se está utilizando los estándares de programación que se debería.
6. Informativos (I): mensajes informativos relativos a Pylint.

A lo largo del desarrollo de BOA hemos empleado Pylint con el objetivo de obtener un código de mayor calidad e intentar evitar errores futuros además de obtener un código con un estilo aceptado por la comunidad. Los resultados han sido favorables, evitando caer en bastantes errores que podrían haber sido bastante difíciles de depurar (e.g. mal uso del operador *"is" vs "=="*). Aún así, gran parte de la comunidad no está de acuerdo con cierta parte de la configuración por defecto (e.g. códigos de errores o advertencias que no siempre funcionan bien), y para ello, hemos hecho uso del fichero "*.pylintrc*" en nuestro proyecto con el fin de realizar algunas modificaciones. Las modificaciones más relevantes son los códigos que hemos deshabilitado en el análisis de Pylint, los cuales son los siguientes:

- F0401: no se ha podido importar un módulo. La manera en la que Pylint intenta detectar si un módulo está disponible o no a la hora de importarlos no es confiable, ya que no hace una correcta detección y en muchas situaciones en las que lanza un error, al ejecutar Python, el módulo es localizado correctamente.
- W0142: uso del parámetro **args* o ***kwargs*. Debido a la naturaleza de BOA, hay funciones que pueden recibir una serie de parámetros indefinidos, ya sea sin nombre (i.e. **args*) o con nombre (i.e. ***kwargs*), y debido a ello esta advertencia no era útil.
- W0212: acceso a miembros protegidos desde fuera de la clase (e.g. *instance.__init__()*). Hay propiedades que son miembros protegidos que son muy útiles para mostrar información relevante, y ejemplos de ello es la propiedad *__class__*, la cual nos da acceso a información de la clase, con lo que conseguimos información relevante y en el caso de BOA, que carga módulos dinámicos, accede a esta información con el fin de dar información relativa a ciertos módulos para, por ejemplo, si sucede algún error, saber qué módulo ha ocasionado el error.
- W0232: clase sin método *__init__()*. Debido a como funciona BOA, hay clases que no tienen necesidad de dicha función.
- W0402: uso de clases obsoletas (*deprecated*). Al igual que ocurría con el código F0401, el sistema de importación de Pylint no se confiable.
- W0403: importación de módulos relativos, no absolutos. BOA espera encontrar ciertos módulos en directorios concretos, por lo que necesitamos importar módulos de manera relativa al directorio donde se encuentre BOA.
- W0614: no se hace uso de un módulo importado con comodín (*wildcard*). Como ya hemos comentado en los códigos F0401 y W0402, el sistema de importación de Pylint no es confiable. Además, ya hay un código definido que indica que importar módulos con comodín es algo que no debería de hacerse (código W0401), por lo que este código es algo redundante debido a que ya hay un código que indica que no debería de estar haciéndose esto, lo que reduce este código a una advertencia de una advertencia.

- W1401: uso de barra invertida en una cadena de texto sin utilizarse para escape y no se ha indicado el carácter ”r” delante de la cadena para indicar que se escapen los caracteres. Un ejemplo claro del mal funcionamiento de este código son los *docstring* (i.e. cadena de texto literal que se sitúa como primera sentencia de un módulo, clase, método o función y se hace accede al mismo a través del atributo *__doc__* [36]), los cuales son detectados con este error.

- R0201: un método de una clase podría ser una función debido a que no hace uso de su parámetro *self*. Debido a como funciona BOA, hay métodos que tienen un comportamiento inicial vacío y que luego pasan a tener su comportamiento real, por lo que Pylint detecta esto como una situación a tener en cuenta, pero en realidad no lo es debido al comportamiento dinámico de BOA.

- R0903: muy pocos métodos públicos definidos. En la mayoría de situaciones, este código tiene sentido, pero si queremos definir una clase para solo almacenar, por ejemplo, códigos de error, este error no tendrá sentido, y por ello lo deshabilitamos.

- C0111: deshabilitación en la comprobación de *docstring* a todos los niveles. Este módulo se deshabilitó con la finalidad de comprobar de una manera rápida la calidad del código sin tener en cuenta que absolutamente todos los elementos (i.e. fichero, clase, funciones, métodos, etc.) tuvieran *docstring*, lo cual era cierto para las etapas tempranas del proyecto, pero actualmente todos los ficheros cuenta con sus *docstring* a todos los niveles.

- C0301: líneas demasiado largas. La razón de deshabilitar este código fue debido a zonas concretas de algunos ficheros donde la complejidad había aumentado demasiado y el código tenía mucho sangrado, por lo que para escribir poco código, era necesario escribir muchas líneas. Una vez se refactorizó el código, este problema se solucionó en gran medida.

- I0011: aviso con fines informativos de que se ha deshabilitado algún código, en concreto aquellos códigos relacionados con el estilo del código (i.e. códigos identificados por la letra C). Este código se deshabilitó con el fin de no mostrar códigos que no fuera necesarios, siendo la causa de ello, la deshabilitación del código C0111.

5.1.2 xmltodict

El módulo xmltodict[11] es una utilidad que permite trabajar con un fichero XML como si fuera un fichero JSON. Se trata de un módulo disponible en PyPI que transforma una cadena de texto que contenga toda la información relativa a un fichero XML en un diccionario de Python, dándonos la flexibilidad que los mismos tienen y facilitando la manipulación de la información del fichero XML para su procesamiento. La razón de utilizar esta utilidad se explica en profundidad en la sección Archivo de Reglas del capítulo Diseño.

Trabajar con un fichero XML directamente no suele ser una tarea simple, lo cual suele conducir a errores difíciles de detectar. Para evitar estos problemas, BOA utiliza este módulo para lidiar con los ficheros XML con los que trabaja, ya que no era objetivo del analizador construir un buen analizador de documentos XML, sino ser un analizador automatizado de vulnerabilidades.

El uso del módulo es muy simple, ya que solo necesitamos cargar el fichero XML en Python, obtener todo el contenido en una cadena de texto, y a partir de un comando, obtenemos un diccionario a través del

[11] Módulo xmltodict: `https://pypi.org/project/xmltodict/`

cual podemos manipular la información de manera sencilla. Los valores que no eran simples en el documento XML original, sino que contenían otros elementos XML se almacenan en listas. En el caso de los atributos, se almacenan como una clave del diccionario donde el nombre de la clave empieza con el carácter '@' y sigue con el nombre del atributo del elemento XML. Hay casos más complejos donde, además de tener atributos, también tenemos directamente texto contenido en el elemento XML, no otros elementos XML, y para identificar el texto, el cual no tiene un nombre debido a que es directamente un valor, xmltodict reserva el nombre "#text" para hacer referencia al texto contenido en una etiqueta XML cuando tiene atributos y texto como valor (en el caso de no haber atributos, el valor se encuentra directamente accediendo al elemento XML). Un ejemplo de xmltodict en acción se puede encontrar en la figura 5.4.

```
<my_xml_document has="an attribute">
  <and>
    <many>elements</many>
    <many>more elements</many>
    <with>
      <inner>elements</inner>
    </with>
  </and>
  <plus a="complex">
    element as well
  </plus>
</my_xml_document>
```

(a) Fichero XML como entrada para xmltodict

```
{
  "my_xml_document": {
    "@has": "an attribute",
    "and": {
      "many": [
        "elements",
        "more elements"
      ],
      "width": {
        "inner": "elements"
      }
    },
    "plus": {
      "@a": "complex",
      "#text": "element as well"
    }
  }
}
```

(b) Resultado de xmltodict

Figura 5.4: Ejemplo de funcionamiento del módulo xmltodict

El principal problema con el que nos encontramos al utilizar este módulo es que utiliza un diccionario de Python, el cual no mantiene el orden de sus elementos. Hay otras estructuras de datos que son diccionarios y sí que mantienen el orden de sus elementos, como es el caso de la clase *Collections.OrderedDict*[12] de Python, pero la naturaleza de los fichero JSON es desordenada, ya que normalmente lo que interesa es el anidamiento de sus datos. Hay situaciones en las que el orden sí que importa, ya que puede ser que en un fichero XML se introduzcan datos que tengan un mismo nivel semántico (i.e. mismo elemento XML padre, lo que significa estar en el mismo nivel de anidamiento) y su procesamiento tenga una dependencia en el orden de aparición. Debido a esto, en BOA hemos intentado suplir este problema, ya que se quería que sí que hubiera un orden, pero conseguir flexibilidad y funcionalidad no siempre es sencillo, y en el caso de BOA se ha apostado por la flexibilidad y se ha conseguido implementar un orden parcial, el cual es optativo.

[12]Documentación de *OrderedDict*: `https://docs.python.org/3/library/collections.html#collections.OrderedDict`

5.2 Sphinx

Sphinx[13] es una herramienta que automatiza la creación de documentación para un proyecto escrito con el lenguaje de programación Python. Se trata de una herramienta ampliamente extendida (inicialmente se creó para generar la documentación oficial de Python) en el mundo Python debido a lo útil y sencilla de utilizar que es. La característica principal y más extendida de Sphinx es la creación de documentación a través de comentarios en el propio código. Esta característica es la que hace que Sphinx sea tan útil, ya que soluciona dos problemas a la vez, que son documentar el código y generar una documentación para el usuario. Es cierto que la documentación generada a partir del código es información técnica y que puede no ser útil para todos los usuarios, pero esta documentación se puede, se debe y se suele ampliar de manera manual, con lo que conseguimos obtener un entorno de documentación unificado a través de Sphinx.

Figura 5.5: Logo de Sphinx

La manera en la que Sphinx funciona es compilando la documentación, es decir, es parecido al procesamiento realizado por LaTeX, ya que coge texto con instrucciones especificas de ficheros de texto (i.e. los ficheros con código Python), los compila y como salida obtiene un documento en otro formato, pero además con las ventajas comentadas debido a la unificación del entorno de documentación. Sphinx hace uso de los *docstrings* de Python, ya que la finalidad de los *docstring* es, justamente, la creación de documentación, y genera la documentación. Internamente, Sphinx hace uso de una estructura jerárquica, en concreto hace uso de una estructura de datos en forma de árbol, donde se definen elementos llamados *toctree* (i.e. *Table of Content Tree*) que son nodos del árbol que hacen referencia a ficheros que se tienen que inspeccionar para obtener la documentación. Los *toctree* son un elemento básico y vital para el correcto uso de Sphinx, y los ficheros ”rst” (i.e. ficheros con código fuente de Sphinx) tienen que hacer un correcto uso de los mismos para poder generar correctamente la documentación, los índices, etc. Otro fichero vital para la configuración de Sphinx es ”config.py”, el cual nos ayuda a especificar o modificar el comportamiento de Sphinx (e.g. carpeta donde se genera la documentación).

Otra de las características a destacar son los múltiples formatos de salida para la generación de Sphinx. De manera nativa, Sphinx puede generar salidas en formato HTML, que es el tipo de salida más común, y se puede personalizar debido a que dispone de diferentes plantillas e incluso se pueden crear nuevas plantillas. Otras de las salidas que Sphinx genera de manera nativa son LaTeX, ePub, Texinfo, XML, texto plano, etc. [37]. Hemos comentado los formatos de salida nativos, pero no son los únicos, ya que Sphinx tiene extensiones, las cuales nos permiten añadirle funcionalidad (e.g. formato de salida PDF con la extensión *rst2pdf.pdfbuilder*).

Sphinx espera estructura en concreto en los *docstring* que encuentre mientras analiza los ficheros que le hemos indicado, y esta estructura que espera se llama formato del *docstring*. Los formatos que Sphinx soporta por defecto son: *NumPy/SciPy docstrings* y *PyDoc*. Dependiendo de las necesidades del proyecto, la documentación tendrá que ser más extensa o menos, más detallada en algunos aspectos (e.g. comportamiento

[13]Sitio oficial de Sphinx: `https://www.sphinx-doc.org/`

general de una función muy detallado) y, quizá, menos en otros (e.g. explicación de los parámetros poco detallado). En el caso de BOA, se ha hecho uso del formato *Google Docstrings*[14], el cual se caracteriza por ser intuitivo y es ideal para documentaciones no muy extensas. Este formato de *docstring* se puede activar incluyendo una extensión en el fichero "config.py", en concreto se trata de la extensión "sphinx.ext.napoleon".

A lo largo del desarrollo de BOA hemos hecho uso de Sphinx para la generación automática de código, la cual ayuda a cumplir con los objetivos definidos para el proyecto. No solo hemos generado la documentación de manera automatizada, sino que la hemos ampliado de manera manual añadiendo secciones que explican los diferentes módulos en los que el proyecto se divide. Además, hemos definido una estructura clara para la generación de la documentación realizando una separación claramente definida de los contenidos. Los detalles se comentan en la sección Documentación del capítulo Implementación.

5.3 Pycparser

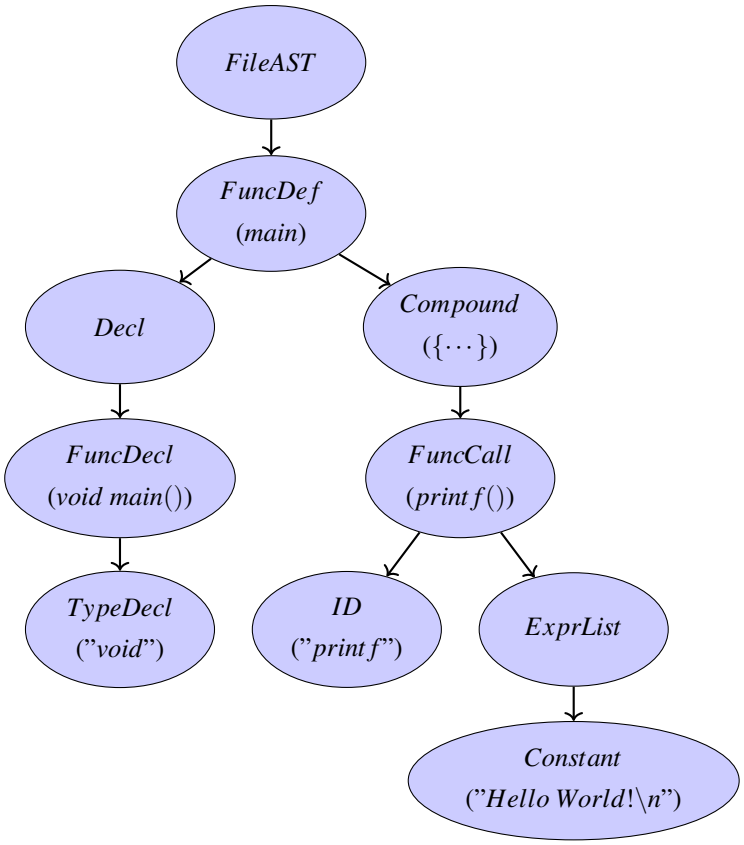
Pycparser[15] es un analizador de código, en concreto para lenguaje C que implementa completamente el estándar C99, conocido formalmente como ISO/IEC 9899:1999. Está escrito totalmente en lenguaje Python, y su principal y única característica es analizar ficheros de lenguaje C y devolver una representación del código de manera que sea más sencilla de procesar que si se procesara el fichero directamente. En concreto, lo que Pycparser nos devuelve es un árbol sintáctico abstracto (del inglés *Abstract Syntax Tree* o AST), el cual representa el fichero de lenguaje de C de una manera mucho más sencilla para procesar debido a que trabajamos con una estructura de datos, que es un árbol, y es mucho más sencillo navegar a través de un árbol que tener que trabajar con una larga cadena de texto que contenga las instrucciones del fichero, ya que habría que hacer mucho trabajo hasta llegar a una representación como la que Pycparser nos ofrece.

Los AST son la representación más común a la hora de analizar ficheros que contienen líneas de lenguajes de programación, y lo que contienen son los elementos sintácticos del lenguaje de programación. Estos elementos sintácticos son los que nos dicen qué tipo de instrucción se está ejecutando, es decir, es como si fuera una clasificación de las sentencias que se pueden ejecutar (e.g. llamada a función, condición, bucle, operación binaria de suma). Esta información está contenida en nodos del árbol, el cual tiene un orden, y la manera de recorrerlo puede variar, pero lo más común es realizar un recorrido en preorden ya que de esta manera estaremos recorriendo el árbol de manera que el orden estará relacionado directamente con el orden en el que se definieron las sentencias. Estos nodos, además de representar ante qué tipo de sentencia estamos, también suelen contener otro tipo de información con el fin de entender completamente lo que las sentencias significan, pero esa información no está completa, pues se tendrá que descender en el árbol para obtener toda la información completa de cada una de las sentencias. Para obtener la información completa de una sentencia habrá que recorrer el árbol de manera recursiva, y no podremos evitar realizar un recorrido recursivo a no ser que sepamos de manera muy concreta qué es lo que estamos buscando en el AST, donde está y que su representación no sea muy variante (e.g. una sentencia donde asignemos un valor a una variable estará formada de nodos que representen operaciones binarias de operaciones binarias ... por lo que su representación es muy variante y será difícil evitar el recorrido recursivo si queremos recuperar toda la sentencia o cualquier información relativa a la misma). Un ejemplo de AST creado con Pycparser se puede observar en la figura 5.6.

[14]Ejemplo de *Google Docstrings*: `https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_google.html`
[15]Repositorio de Pycparser: `https://github.com/eliben/pycparser`

```c
#include <stdio.h>

void main()
{
    printf("Hello World!\n");
}
```

(a) Algoritmo del que obtenemos el AST



(b) Árbol Sintáctico Abstracto (AST)

Figura 5.6: Ejemplo de AST con Pycparser

Internamente, Pycparser representa los diferentes tipos de sentencias de un programa C con diferentes clases en una estructura de datos de tipo diccionario, pero un diccionario que mantiene el orden (utiliza el módulo *Collections.OrderedDict*), y de esta manera se representa el árbol sintáctico abstracto. En concreto, las sentencias que Pycparser utiliza son las siguientes:

- Nodos abstractos de Pycparser:
  - Clase *Node*: clase genérica de la que heredan el resto de nodos. Algo a destacar de todos los nodos es que contienen información relativa a su posición en el fichero de lenguaje C (i.e. atributo *coord*), lo cual es útil para mostrar información relativa al análisis realizado por BOA.
  - Clase *NodeVisitor*: clase que no representa a una sentencia, sino que sirve para poder visitar los nodos. Su implementación permite un recorrido básico del árbol, y en el caso de BOA no hemos hecho uso de esta clase, sino de una propia.
  - Clase *FileAST*: nodo raíz del AST devuelto por Pycparser.
- Nodos comunes:
  - Clase *FuncDef*: nodo raíz de una función. Cuando se define una función, este es el nodo raíz.
  - Clase *ID*: normalmente, contiene el nombre de una variable, pero en casos más complejos (e.g. *struct*) puede contener otras clases anidadas.
  - Clase *Assignment*: representa una asignación (e.g. $int\,foo = "bar"$, $halo = 3$).

- Clase *Constant*: representa un valor constante.
- Clase *Cast*: representa una transformación explícita de un tipo de dato a otro (e.g. *int* $foo =$ (int)"*bar*";).
- Clase *Compound*: representa un ámbito, el cual puede pertenecer a diferentes elementos (e.g. *if*, *for*, definición de una función) e incluso no pertenecer a ninguno (e.g. *int* $foo()\{\cdots\ \{\cdots\}\ \cdots\}$).
- Clase *FuncCall*: representa la llamada a una función.

- Nodos para representar el tipo de datos de una variable:
  - Clase *IdentifierType*: representa el tipo de datos en una declaración simple (e.g. **int** $foo()\{\cdots$ **int** $halo = 3;\ \cdots\}$), pero no de las declaraciones complejas (e.g. $structPoint2Dp$;).
  - Clase *Enum*: representa la palabra clave *enum*.
  - Clase *Enumerator*: representa un elemento de dentro de una lista de una enumeración (e.g. *enum* $Color\{$**Red**, **Green**, **Blue**$\}c =$ *Red*;).
  - Clase *Struct*: representa la palabra clave *struct*.
  - Clase *Typedef*: representa la palabra clave *typedef*.
  - Clase *Union*: representa la palabra clave *union*.

- Declaraciones:
  - Clase *Decl*: representa una declaración, la cual puede ser de una función, una variable, un *array*, etc.
  - Clase *TypeDecl*: representa una declaración de un tipo de datos, pero el valor concreto no estará en este nodo, sino en otro de tipo *IdentifierType*.
  - Clase *FuncDecl*: representa una declaración de una función (e.g. *int* $foo(int\ bar)$;). Es como se indica que un nodo de tipo *Decl* es, en concreto, una función, de manera que se le da un comportamiento especial debido a que las funciones tienen argumentos.
  - Clase *ArrayDecl*: representa una declaración de un *array* (e.g. $char[] =$ "*halo* 3";).
  - Clase *PtrDecl*: representa una declaración de un puntero (e.g. $int * halo$;).

- Referencias a variables o elementos de variables no básicas:
  - Clase *ArrayRef*: representa el acceso a elementos de un *array*. Conforme más profundo sea el *array* (i.e. *array* de *arrays*, lo cual indica la dimensión del *array*), más nodos *ArrayRef* contendrá el AST.
  - Clase *StructRef*: representa el acceso a elementos de una estructura (i.e. *struct*). Conforme más profunda sea la estructura (i.e. estructura que contiene otras estructuras u otros contenedores no básicos), más nodos *StructRef* contendrá el AST.

- Listas:
  - Clase *InitList*: representa la inicialización de una lista (e.g. *int* $a[] = 1,2,3$, $structPoint2Dp = 0,0$), la cual se representa con llaves.
  - Clase *ParamList*: representa una lista de parámetros de una función (e.g. *int* $foo($**int a**, **int b**, **char**$*$ **c**)).
  - Clase *EnumeratorList*: representa una lista de elementos de una enumeración (e.g. *enum Color* $\{$**Red**, **Green**, **Blue**$\}$;).

  - Clase *DeclList*: representa una lista de declaraciones, principalmente utilizado en el bucle *for* (e.g. $for(\textbf{int i} = \textbf{0}, \textbf{j} = \textbf{0}; i < 10 \;\&\&\; j < 100; i{+}{+}, j{+}{=}5)\{\cdots\};$).
  - Clase *ExprList*: representa una lista de expresiones, principalmente utilizado en el bucle *for* (e.g. $for(\textbf{i} = \textbf{0}, \textbf{j} = \textbf{0}; i < 10 \;\&\&\; j < 100; \textbf{i}{+}{+}, \textbf{j}{+}{=}\textbf{5})\{\cdots\};$) y en las llamadas a funciones.

- Operadores:

  - Clase *UnaryOp*: representa un operador que se aplica sobre un único operando (e.g. $i{+}{+}$).
  - Clase *BinaryOp*: representa un operador binario que opera con dos operandos (e.g. $a+b$, $a*c$).
  - Clase *TernaryOp*: representa un operador ternario que toma un valor booleano y, en función de su valor, devuelve su primer o segundo operando (e.g. $is\_weekend()$? "*yeah*!" : "*no* :(").

- Flujo de control:

  - Clase *If*: representa la palabra clave *if*, la cual representa una condición.
  - Clase *For*: representa la palabra clave *for*, la cual representa un bucle *for*.
  - Clase *While*: representa la palabra clave *while*, la cual representa un bucle *while*.
  - Clase *EmptyStatement*: representa de manera explícita la ausencia de un cuerpo, lo cual puede suceder en las estructuras de flujo de control *for* (e.g. $for(int\ i = 0; i < 10; i{+}{+});$) y *while* (e.g. $while(i{+}{+} < 10);$).
  - Clase *DoWhile*: representa la estructura de bucle $do\ \{\cdots\}\ while(\cdots);$.
  - Clase *Continue*: representa la palabra clave *continue*, la cual nos permite pasar a la siguiente iteración de un bucle.
  - Clase *Break*: representa la palabra clave *break*, la cual nos permite detener la ejecución del bucle en ejecución.
  - Clase *Goto*: representa la palabra clave *goto*, la cual sirve para hacer saltos.
  - Clase *Label*: representa una etiqueta, la cual se utiliza para hacer referencia a una parte concreta del código y poder saltar a la misma a través de *goto*.
  - Clase *Switch*: representa la palabra clave *switch*.
  - Clase *Case*: representa la palabra clave *case*.
  - Clase *Default*: representa la palabra clave *default*.
  - Clase *Return*: representa la palabra clave *return*.

- Otros nodos:

  - Clase *CompoundLiteral*: representa un ámbito literal, el cual es una instancia sin nombre con un tipo concreto (e.g. $draw\_line((\textbf{struct Point2D})\{.\textbf{x} = \textbf{0}, .\textbf{y} = \textbf{0}\}, (\textbf{struct Point2D})\{.\textbf{x} = \textbf{5}, .\textbf{y} = \textbf{5}\})$) que se utiliza cuando un *array*, *enum* o unión se necesita únicamente una vez. Se puede ver como el análogo de las funciones *lambda* pero para variables.
  - Clase *EllipsisParam*: representa que una función puede tener una cantidad indefinida de argumentos (i.e. función variádica) y se representa con tres puntos (e.g. $void\ foo(int\ mandatory, ...)\{\cdots\}$).
  - Clase *NamedInitializer*: representa un inicializador designado (e.g. $int\ halo[5] = \{[\textbf{2}] = \textbf{10}\};$), el cual nos permite inicializar partes concretas de un *array*, un *struct* o una unión sin inicializar otras. Es una inicialización parcial en lugar de total en el momento de la declaración.

- Clase *Typename*: representa los tipos de datos que una función acepta cuando se declara y no se indican nombres concretos para las variables (e.g. *int* $foo($**int**$)$;). Se diferencia del nodo *Decl* en que este no tiene un nodo anidado de tipo *ID*, es decir, no tiene un nombre de variable asignado. También podemos verlo en las transformaciones explícitas de tipos de datos (i.e. nodo *Cast*) o funciones como *sizeof()*.
- Clase *Pragma*: representa la directiva de preprocesador *pragma* (e.g. #*pragma once*). Es un caso especial de soporte por parte de Pycparser a una directiva de preprocesador, y la razón de dar soporte a esta directiva es que no es estándar, y para evitar que algún compilador no responda bien ante su preprocesamiento, Pycparser la procesa.

Aunque BOA es un analizador de vulnerabilidades genérico, para la implementación de diferentes módulos se ha utilizado Pycparser con el objetivo de demostrar su funcionamiento, pero BOA no está limitado únicamente a su uso con Pycparser. Para su correcto uso ha sido vital entender los diferentes nodos que se han explicado, ya que su correcta identificación es lo que ha permitido identificar las sentencias que eran necesarias en cada momento.

Pycparser tiene dos problemas principales que hemos tenido que solucionar con BOA. El primero es que no tiene soporte para las directivas de preprocesador de C (i.e. instrucciones que se resuelven en tiempo de compilación; ejemplo de directivas de preprocesador son los siguientes: comentarios, que tienen que eliminarse antes de compilar; instrucciones del tipo $\#define\ CONST\ 5$, que tienen que reemplazar el valor 5 donde se utilice *CONST*; compilación condicional como $\#ifdef\ CONST\ \cdots\ \#endif$, la cual tiene que resolverse en tiempo de compilación), pero esto se soluciona de manera sencilla debido a que, aunque Pycparser no tiene soporte para su procesamiento, sí que tiene soporte para utilizar un compilador como GCC para que le devuelva un fichero donde las directivas de preprocesador ya han sido resueltas (opción "-E" en GCC). El otro gran problema es que no tiene soporte para realizar el análisis sobre aquellas librerías que se importan de la librería estándar en los ficheros C (e.g. $\#include\ <stdio.h>$), pero se nos da una solución, y esa es importar librerías, a través de un compilador como GCC, que son falsas e imitan a las librerías estándar de C (opción "-I /ruta/hacia/cabeceras" en GCC), pero no lo son y no hay ningún problema a la hora de realizar el análisis sobre estas librerías falsas. Con GCC podemos , y la ruta por defecto de estas librerías que imitan a las librerías estándar de C es "utils/fake_libc_include" partiendo de la ruta de la instalación de Pycparser. Para solucionar ambos problemas, Pycparser hace uso de un compilador, el cual puede ser GCC u otro, y esto se puede especificar en las opciones de las directivas de preprocesador, donde el parámetro "cpp_args" nos permite especificar parámetros para el compilador.

5.4 Git

Git[16] es un sistema de control de versiones. La principal finalidad de Git y de otros sistemas de control de versiones (e.g. Subversion) es la de ayudar a gestionar los cambios realizados en los proyectos a partir de versiones. Git está muy extendido en el mundo de *software*, y se trata de una herramienta necesaria para cualquier proyecto, incluso en los más pequeños es útil el uso de un sistema de control de versiones, y Git tiene la ventaja de que es muy sencillo de aprender y utilizar. Una de las grandes ventajas que tenemos al utilizar un sistema de control de versiones es que podemos restaurar diferentes versiones por cualquier motivo

[16]Sitio oficial de Git: `https://git-scm.com/`

(e.g. buscar errores, comprobar una característica que cambia el comportamiento en una versión anterior) y nos permite, a partir de esa versión concreta, crear nuevas ramificaciones para crear proyectos enteros diferentes si así lo deseamos (hay muchos proyectos de código libre que hacen justamente esto partiendo de otro proyecto de código libre).

Figura 5.7: Logo de Git

Internamente, tanto Git como otros sistemas de control de versión utilizan un grafo donde se realiza la gestión de los cambios. Respecto a los cambios/versiones, lo que se almacena son los cambios de una versión a otra, es decir, cuando se crea una nueva versión (i.e. *commit*), los cambios que hay de una versión a otra es lo que se almacena, ahorrando mucho espacio. Una analogía de los cambios almacenados es una copia de seguridad de tipo incremental, la cual se crea de manera parecida al resultado de la orden *diff* de las utilidades de GNU (en concreto, Git tiene diferentes algoritmos para calcular las diferencias, pero el que tiene por defecto es el mismo que se utiliza en la orden *diff*, el cual se llama *Myers* y hace que Git y la orden *diff* sean interoperables [38][39][40]).

La manera en la que hemos hecho uso de Git en BOA es muy sencilla, ya que no hemos hecho uso de ninguna convención donde se hace uso de unas ramas (i.e. subdivisión de versiones) concretas. BOA ha utilizado la rama principal (i.e. rama *master*) en todo momento, y conforme se han ido ampliando las características, se han arreglado los errores o se ha realizado alguna modificación, se han actualizado los cambios. La metodología empleada ha sido realizar actualizaciones periódicas cuando había un cambio pequeño en el proyecto, lo cual ayuda a la búsqueda de errores, y a mantener una lista de versiones/cambios ordenada.

5.4.1 Github

Github[17] es un repositorio remoto que permite subir proyectos creados con el sistema de control de versiones Git. Se trata de uno de los repositorios de proyectos más grandes y más utilizados. Para el desarrollo de BOA, se ha utilizado Github como repositorio para almacenar todas las versiones creadas con Git. En concreto, se ha hecho uso de un repositorio privado.

[17] Sitio oficial de Github: `https://github.com/`

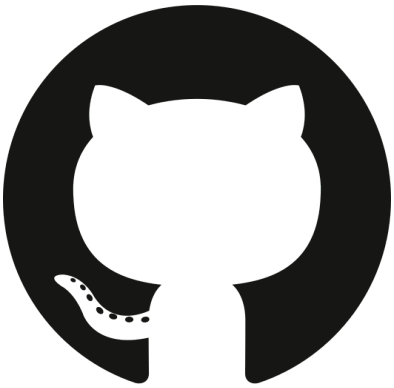
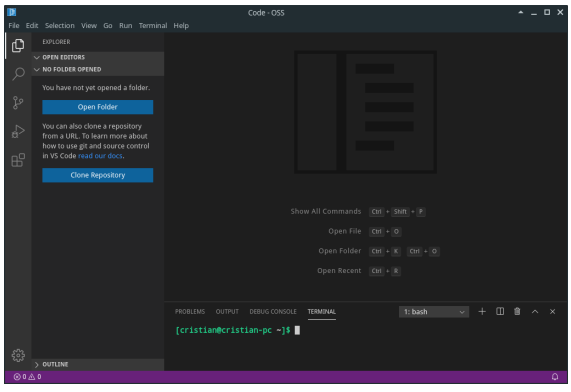


Figura 5.8: Logo de Github

5.5 Visual Studio Code

Visual Studio Code[18] es un editor de código desarrollado por Microsoft[19] que ha ganado mucha reputación a lo largo de los años en la comunidad de desarrolladores como se puede ver en la figura 5.10. Algo que Visual Studio Code hace de manera genial es darle al desarrollador un entorno cómodo, intuitivo y muy potente. No llega a considerarse un IDE (Sistema de Desarrollo Integrado o, del inglés, *Integrated Development System*) debido a que, de manera predeterminada, viene con unas cuantas utilidades, pero no con tantas como otros entornos que sí se consideran IDE.

(a) Logo



(b) Pantalla principal

Figura 5.9: Visual Studio Code

Las características más destacables de Visual Studio Code, o VSCode, son las siguientes:

- Ligero y eficiente: al utilizar VSCode nos damos cuenta en seguida de que es muy ligero. La fluidez con la que nos movemos a través de los diferentes ficheros es algo que no siempre se consigue en otros editores de código o IDE, como es el caso del IDE de Microsoft, Microsoft Visual Studio[20], el cual en muchas ocasiones necesita de una máquina potente para ejecutarse, lo cual no es el caso de VSCode.
- *IntelliSense*: función de autocompletado y colorado del código utilizada tanto en VSCode como en Microsoft Visual Studio. La función de autocompletado no es simplemente una lista de palabras clave de

[18] Sitio oficial de Visual Studio Code: `https://code.visualstudio.com/`
[19] Sitio oficial de Microsoft: `https://www.microsoft.com/`
[20] Sitio oficial de Microsoft Visual Studio: `https://visualstudio.microsoft.com/es/`

un lenguaje de programación, sino que de manera inteligente también autocompleta funciones, módulos y nombres de variables.

- Depurador: incorpora un depurador que cumple con la mayoría de las necesidades de los desarrolladores.
- Soporte nativo para Git: de forma nativa, hay soporte para Git. De forma visual, podemos ver los cambios de una versión a la otra en los mismos ficheros de código. Muy intuitivo y útil.
- Extensible y personalizable: a través de extensiones podemos ampliar el comportamiento de VSCode y personalizarlo. Esta es una de las características más potentes de VSCode, la cual puede hacer de la experiencia con VSCode gratamente satisfactoria, ya que se puede convertir en un editor de código ideal para cualquier tipo de proyecto.
- Terminal nativa incorporada: se pueden añadir terminales directamente a la ventana de VSCode, ya que de manera nativa lo permite. Se pueden añadir tantas terminales como se quiera e incluso se pueden dividir para poder observar diferentes terminales a la vez. Muy útil para tareas que requieran de terminal a la vez que del editor de código (e.g. proyectos que utilicen Node.js y requieran de reiniciar constantemente el servidor).

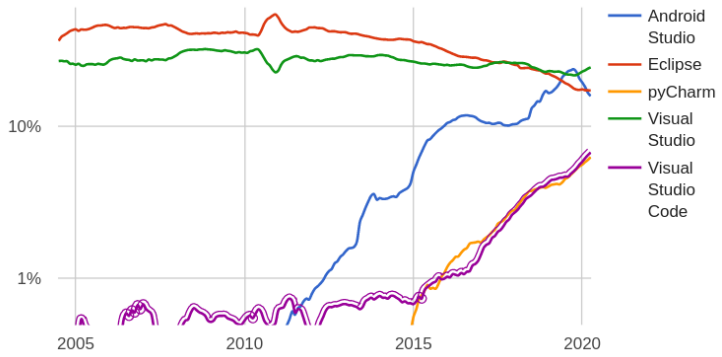


Figura 5.10: Editores de código más populares [41]

VSCode crea un directorio para cada proyecto que utiliza el editor, y este directorio es ".vscode". El fichero más importante que encontramos en este directorio es "launch.json", el cual nos permite configurar ciertas características de nuestro proyecto (e.g. variables de entorno, argumentos).

BOA se ha desarrollado utilizando VSCode. La única extensión que se ha utilizado ha sido *Python*, la cual añade soporte al lenguaje de programación Python y una serie de herramientas propias del lenguaje o del editor (e.g. Linting, depurador, Intellisense).

5.5.1 Depurador

El depurador de VSCode ha sido una de las herramientas que más hemos utilizado a la hora de enfrentarnos a los errores a lo largo del desarrollo de BOA. Una configuración rápida y sencilla en el fichero "launch.json" permite su uso, y debido a su sencillez, familiarizarse con el mismo no es una tarea complicada.

Al igual que la mayoría de depuradores, nos permite añadir puntos de ruptura donde la ejecución se detendrá cuando llegue a dicho punto y, a partir de ahí, se podrá saltar a la siguiente instrucción, al siguiente

punto de ruptura o salir del ámbito de función actual. Otra característica que ha sido realmente útil ha sido la función de vigilar el valor de una variable (i.e. sección *watch*). No se trata de un depurador complejo que permita realizar muchas acciones (e.g. GDB[21] es un depurador con soporte para muchos lenguajes de programación, pero en el caso de la versión para C y C++, permite realizar una gran cantidad de tareas que permiten tener el control total de un binario, con la desventaja de que la curva de aprendizaje es elevada al principio), pero cumple con su cometido, y gracias a su sencillez de uso, permite una inspección rápida y fluida del código, lo cual era lo que necesitábamos para BOA.

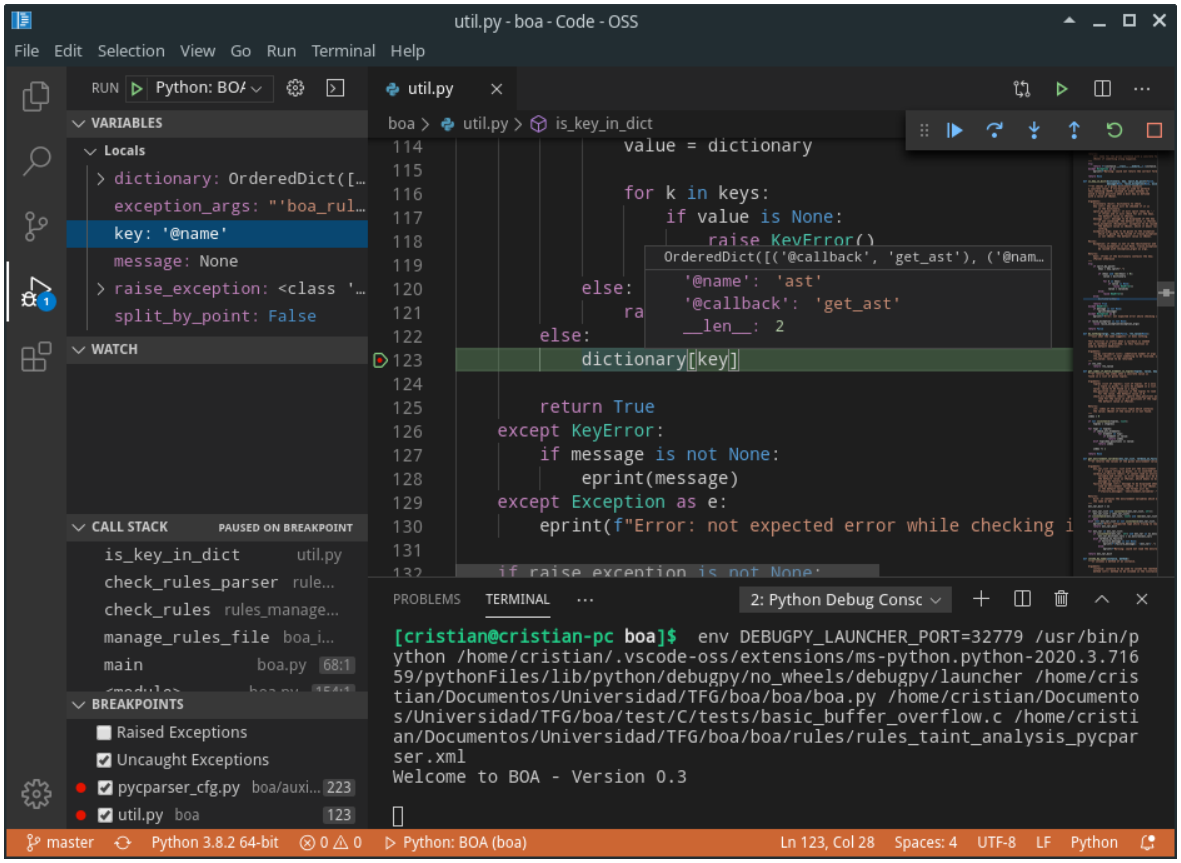


Figura 5.11: Ejemplo del depurador de VSCode con BOA

[21] Sitio oficial de GDB: `https://www.gnu.org/software/gdb/`

Capítulo 6

Diseño

6.1 Introducción

El diseño es uno de los grandes pilares de este trabajo, y es donde vamos a explicar las decisiones tomadas de cara a cumplir con unos objetivos. Las decisiones tomadas durante esta fase condicionan las posteriores, y de ahí surge la necesidad de tomar buenas decisiones para obtener los resultados deseados. Lo explicado en este capítulo se materializará en el capítulo Implementación.

6.2 Objetivos del Diseño

El diseño de BOA se planteó inicialmente con los objetivos en mente mencionados en el capítulo Motivación y Objetivos, y con el fin de poder cumplir dichos objetivos surgen necesidades que tienen que resolverse en la fase de diseño. Además de comentar los diferentes objetivos marcados por esta fase, también se va a explicar, superficialmente, cómo se pretende cumplir dichos objetivos.

Automatización

La automatización es uno de los objetivos primordiales en el desarrollo del presente analizador de vulnerabilidades. Automatizando el análisis de vulnerabilidades se busca facilitar al usuario el análisis de un sistema de manera rápida sin necesidad de realizar un laborioso trabajo manual.

Con el fin de conseguir automatizar el análisis, se pretende definir una serie de etapas que, ejecutadas en un orden concreto, lleve al correcto análisis de un sistema y ofrezca todos los resultados necesarios para la obtención de conclusiones respecto a las vulnerabilidades encontradas o factibles del mismo.

Modularidad

Un diseño modular ofrece muchas ventajas, las cuales van a permitir el correcto desarrollo de otros objetivos de diseño. Con un diseño modular se persigue conseguir la mayor independencia posible entre diferentes módulos, ayudando, por ejemplo, a la definición de fases divididas en diferentes módulos. Trabajando de esta manera conseguimos, entre otra serie de ventajas, detectar de manera concreta errores localizados en su respectivo módulo, lo cual también permite pasar de un módulo a otro si el error no es fatal. Conforme se definan módulos, el nivel de cohesión entre los mismos debe de minizarse en la medida de lo posible, ya que el nivel de cohesión entre los mismos definirá la robustez y flexibilidad de los mismos (un gran nivel de cohesión puede no permitir el correcto funcionamiento de un módulo si falta otro del que depende, lo cual nos lleva a un sistema poco robusto y poco flexible).

La manera en la que este objetivo pretende llevarse a cabo es mediante la definición de interfaces que permitan una comunicación a un alto nivel, pero con un comportamiento concreto en la implementación de dicha interfaz. Se realizará una categorización de las interfaces en función de la finalidad de las mismas, favoreciendo la especialización de los módulos.

Extensibilidad

Permitir ampliar o extender el comportamiento base es algo necesario debido a que uno de los objetivos es conseguir tener un analizador de vulnerabilidades de propósito general. Con permitir que sea de propósito general nos referimos a que sea de utilidad en muchas situaciones, y para ello se pretende permitir la extensión del comportamiento base del analizador. Con la ampliación del analizador conseguimos ampliar su base de conocimiento y su funcionamiento interno si se requiere. La modularidad es un objetivo de diseño que ayudará a la extensión, ya que una manera de realizar la extensión puede ser a través de módulos.

Para permitir que el comportamiento sea extensible se realizará la carga de los nuevos módulos de manera dinámica, y se tendrá que especificar la localización de los mismos a través de algún mecanismo que permita al analizador detectar la presencia de los mismos. A través de las interfaces que ya se han comentado para la modularización, se podrá permitir la ampliación de la misma interfaz con el fin de ampliar el comportamiento.

Comunicación

La comunicación es un mecanismo totalmente necesario para poder especificar configuración y comportamiento al analizador. El usuario, normalmente, querrá especificar cierta configuración antes de realizar el análisis, y para ello se tendrá que comunicar a través de algún mecanismo con el analizador y, en concreto, con algún modo. El mecanismo que se va a utilizar para hacer efectiva la comunicación será un **archivo de reglas**, en el cual se podrá especificar, a través de diferentes directivas claramente definidas, el comportamiento que se desea obtener del análisis a realizar. La comunicación tiene que realizarse entre agentes, y los agentes que se pueden identificar en el ámbito de uso de nuestro analizador son el usuario y los módulos y el analizador en sí.

Con tal de poder realizar la comunicación de manera efectiva y que sirva a todos los propósitos posibles, se podrá realizar de las siguientes maneras atendiendo a los agentes que el analizador puede tener:

- Comunicación Usuario-Analizador: a través del archivo de reglas se podrán especificar valores para directivas claramente definidas que permitan modificar el comportamiento del analizador.
- Comunicación Usuario-Módulo: a través del archivo de reglas se podrán especificar valores que se utilizarán en uno o más módulos con el fin de modificar su comportamiento o, en el caso general y que más se utilizará en este tipo de comunicación, especificar datos de entrada para ampliar el conocimiento de las técnicas utilizadas por los módulos durante el análisis de vulnerabilidades.
- Comunicación Analizador-Módulo: el analizador se comunicará con los módulos con el fin de conducir la ejecución de los mismos teniendo en cuenta las fases que se definan (subsección Automatización).
- Comunicación Módulo-Módulo: se permitirá una comunicación entre módulos con el fin de definir dependencias entre los mismos, lo cual refuerza el objetivo de diseño de flexibilidad pero va en contra del objetivo de modularidad debido a que aumenta la cohesión entre los mismos. La creación de dependencias de módulos será una elección del usuario, por lo que el nivel de cohesión aumentará bajo el criterio del mismo. Este tipo de comunicación permite crear módulos complejos de manera más sencilla, por lo que ayuda a la comprensión de los mismos y esto es debido a que se basa en el principio de divide y vencerás.

Flexibilidad

La flexibilidad es un elemento a tener en cuenta para poder detectar vulnerabilidades, ya que no todas se comportan de igual manera. Permitiendo al usuario una mayor flexibilidad será posible obtener, por ejemplo, configuraciones más concretas y así obtener mejores resultados. Es algo esencial muy relevante en los analizadores, ya que ciertas técnicas tienen parámetros que pueden configurarse y adaptarse mejor a diferentes sistemas (e.g. hay técnicas que permiten realizar un análisis intraprocedural o interprocedural, pero que la aplicación de uno u otro puede variar muchos los resultados de un sistema a otro teniendo en cuenta, por ejemplo, el tamaño del mismo).

A través de la comunicación usuario-módulo, usuario-analizador y módulo-módulo se pretende dar la mayor flexibilidad posible de configuración al usuario. Además, se pretende ofrecer medidas nativas que aumenten la flexibilidad del analizador como, por ejemplo, tener control sobre la ejecución del analizador cuando se detecta una anomalía. Esto se hará en favor de la flexibilidad, ya que en lugar de dejar la decisión al analizador de continuar o detener el análisis, el cual la tomará si no lo hace el usuario, se le dará al usuario la opción de poder tomar la misma decisión si la anomalía es detectada en uno de los módulos del mismo.

Facilidad de Uso

La facilidad o sencillez de uso de cualquier sistema es vital si lo que se persigue es que el usuario quiera utilizar dicho sistema. La manera en la que se intentará obtener dicha sencillez será a través de la intuitividad, y es que muchas veces nombres sencillos de entender en directivas con implicaciones complejas en la configuración es todo lo que necesita un usuario. El problema que en muchas ocasiones aporta el intentar realizar un sistema sencillo de usar es que va en contra de la flexibilidad. La razón es que si el usuario tiene que configurar muchas directivas, lo cual aporta flexibilidad, no será sencillo de utilizar debido a forzará al usuario a comprender todas las directivas, lo cual hace que la curva de aprendizaje empeore, sea más pronunciada, de cara al usuario. Para ello, se intentará, en la medida de lo posible, equilibrar ambos objetivos, y con el fin de no

empeorar la flexibilidad del sistema, se adoptará algún mecanismo como puede ser la definición de valores por defecto.

Además de lo ya comentado, hay muchas medidas que ayudan a mejorar la facilidad de uso de un sistema y a hacer la experiencia de usuario más agradable. Una de estas medidas es la de aportar mensajes claros, concisos y específicos respecto a avisos, mensajes de advertencia, de sugerencia o de error. Esta medida es algo que se intentará aportar en BOA.

Personalización

Hay elementos en los que su personalización puede no ser algo trivial, como es el caso del reporte de resultados. La personalización puede ser un elemento muy importante dependiendo del objetivo concreto de una tarea, y el reporte de resultados es algo que el usuario va a analizar como conclusión del análisis, y en este puede que se quiera más o incluso menos información para evitar redundancia. A través de la extensión se pretende no solo ampliar, sino también sustituir si es necesario. Sustituir puede ser algo que se quiera, y en el caso de la personalización se algo que se puede ayudar. Dependiendo del ámbito, la personalización puede ser más sencilla o más compleja, por lo que hay diferentes elementos que van a favorecer a una correcta personalización. La extensión nos va a ayudar, pero también la comunicación, y de esta manera vamos a lograr personalizar los diferentes ámbitos (e.g. personalización visual de una fase, personalización de mensajes).

Independencia del Lenguaje de Programación

En un principio, el diseño no iba a estar orientado a conseguir una independencia entre el lenguaje de programación y el analizador (de echo, BOA es un acrónimo haciendo referencia a una vulnerabilidad concreta característica del lenguaje de programación C o C++: *Buffer Overflow Annihilator*). La razón de haber reconsiderado esta elección surgió a raíz de diseñar un analizador de propósito general, con lo que ya no era necesario centrarse en un único lenguaje como era C, y, además, conforme los objetivos de diseño que se quería cumplir se iban esclareciendo, la posibilidad de independizar al analizador de un único lenguaje de programación objetivo parecía más factible. El resultado ha sido definir esta idea como un objetivo de diseño propio.

Con la finalidad de independizarse del lenguaje de programación, otros objetivos de diseño son clave para llevarlo a cabo. En concreto, el principio de modularidad y extensibilidad conducen hacia una arquitectura que favorece el diseño de un analizador no sujeto a un lenguaje de programación concreto si se realiza de la manera correcta. Por otro lado, este objetivo de diseño, independizar al analizador del lenguaje de programación, ayuda a la flexibilidad, ya que permite al usuario no estar atado a un lenguaje de programación concreto para poder utilizar BOA.

Algo importante a destacar es que este objetivo de diseño no se llevará a cabo de manera nativa. Con esto se quiere dar a entender que no se va a dar soporte a una gran cantidad de lenguajes de programación, sino que a través de la **interoperabilidad** con otros sistemas/herramientas que permitan el análisis de los diferentes lenguajes de programación, se podrá proceder al análisis de los mismos desde BOA. La interoperabilidad será algo que suceda solo en este caso, y por ello no lo hemos destacado como un objetivo de diseño.

6.3 Enfoque de Analizador

BOA es un analizador de código, pero en concreto es un analizador de código estático. Una de las primeras decisiones, y de las más importantes, a la hora de diseñar un analizador de código es decidir el enfoque que se va a tomar respecto al tipo de analizador a emplear, ya que, como ya se ha discutido con anterioridad, no es lo mismo un analizador de código estático que uno dinámico. Un analizador de código estático aporta unas ventajas y unos inconvenientes, al igual que sucede con un analizador de código dinámico. Las razones por las cuales se decide diseñar e implementar un analizador de código estático se explican a continuación.

Los objetivos generales marcados casan con un analizador de código estático. Al analizar el código de manera estática conseguimos unas ventajas que están mucho más relacionadas con los objetivos, como puede ser la identificación concreta de vulnerabilidades. La razón de ello es que al analizar el código directamente, podemos saber dónde se está identificando una vulnerabilidad y qué vulnerabilidad es (no siempre es posible, ya que dependeremos de la técnica concreta aplicada, pero sí es así para la mayoría de casos), lo cual nos permite indicar al usuario dónde debería de centrar su atención con el fin de poder arreglar la vulnerabilidad. Otra ventaja que favorece a los objetivos es que el análisis estático aporta mayor control al usuario, ya que el usuario puede realizar diferentes configuraciones para, por ejemplo, ajustar el analizador a un sistema propio o ampliar la base de conocimientos de un algoritmo para obtener mejores resultados.

Por otro lado, un enfoque de análisis dinámico es mucho más difícil de relacionar con nuestros objetivos. Una de las principales desventajas es que este tipo de análisis es dependiente del entorno, lo que hace que su automatización sea más difícil de llevar a cabo (un enfoque estático también puede llevar a dependencias con el sistema, pero en mucha menor medida). Además, al igual que con el enfoque estático, también suele ser dependiente del lenguaje de programación, aunque esto no siempre ocurre, y puede que, dependiendo de la técnica empleada, la dependencia sea en su totalidad del sistema. Aún así, la dependencia con el sistema es una barrera muy grande, ya que depender del sistema implica desarrollar la misma técnica múltiples veces si se quiere dar soporte a otros sistemas. Esto no es lo que queremos, debido a que las vulnerabilidades suelen ocurrir debido a problemas no relacionados con el entorno (hay casos en los que esto no es cierto, como las vulnerabilidades relacionadas con la configuración del sistema, pero al centrarnos en el análisis de código, estas vulnerabilidades quedan fuera de nuestro ámbito). Otro gran problema que tienen los analizadores dinámicos es que, sí, son capaces de detectar vulnerabilidades, pero habitualmente no son capaces de decir qué vulnerabilidad es ni dónde se ha detectado (e.g. instrumentar la llamada del sistema *exit()* y realizar *fuzzing* con datos correctos y en los que no se espera una llamada a *exit* puede decir que algo ha sucedido si se realiza la llamada y no termina la ejecución correctamente, pero no nos indica qué ha sucedido ni dónde está el problema). Por supuesto, un analizador estático sufre en mayor o menor medida de algunos de estos problemas, pero en un analizador dinámico es más común estas situaciones.

Como ya se ha comentado, los objetivos definidos en el presente documento están mucho más relacionados con un enfoque estático que con uno dinámico. Debido a ello, el analizador de vulnerabilidades desarrollado, BOA, se centrará en un enfoque de **análisis estático**.

6.4 Ámbito del Analizador

Es importante definir el alcance o ámbito sobre el que el analizador va actuar. En función del ámbito seremos capaces de tomar decisiones tanto de diseño como de implementación debido a que la manera en la que el desarrollo se llevará a cabo será distinta (si definimos un ámbito de actuación de una línea de código, se le pedirá al usuario dicha línea y no será necesario trabajar con el sistema de archivos mientras que si definimos un ámbito de actuación de todo el sistema, será necesario obtener una configuración no solo para el sistema, sino también para el entorno). Esta decisión nos lleva a la construcción de un analizador de vulnerabilidades que será utilizado por usuarios con necesidades concretas, las cuales son cruciales, ya que un usuario que quiere ejecutar un analizador sobre todo un sistema necesitará soporte para dicha tarea, y no le valdrá un analizador que no tenga soporte para un ámbito de sistema.

Teniendo en cuenta que el diseño se va a centrar en un analizador de vulnerabilidades de ámbito general, una decisión con que equilibra la balanza es la de tener un **ámbito de actuación máximo de módulo o fichero**. Esto quiere decir que BOA va a trabajar con ficheros sobre los que se realizarán análisis concretos, y el modo en el que el análisis se efectúe, definirá el ámbito sobre el que trabaje ese módulo concreto. Con ámbito de actuación máximo se quiere hacer referencia a que la limitación será dicho ámbito, pero también se podrá trabajar a unos niveles menores debido a que se tendrá acceso a la información de un fichero, el cual es el ámbito de actuación máximo, por lo que en lugar de utilizarse toda la información, también será posible centrarse de manera más localizada, lo cual significa trabajar en un ámbito menor. Todo ello será una decisión que el usuario deberá de tomar a la hora de incluir un módulo.

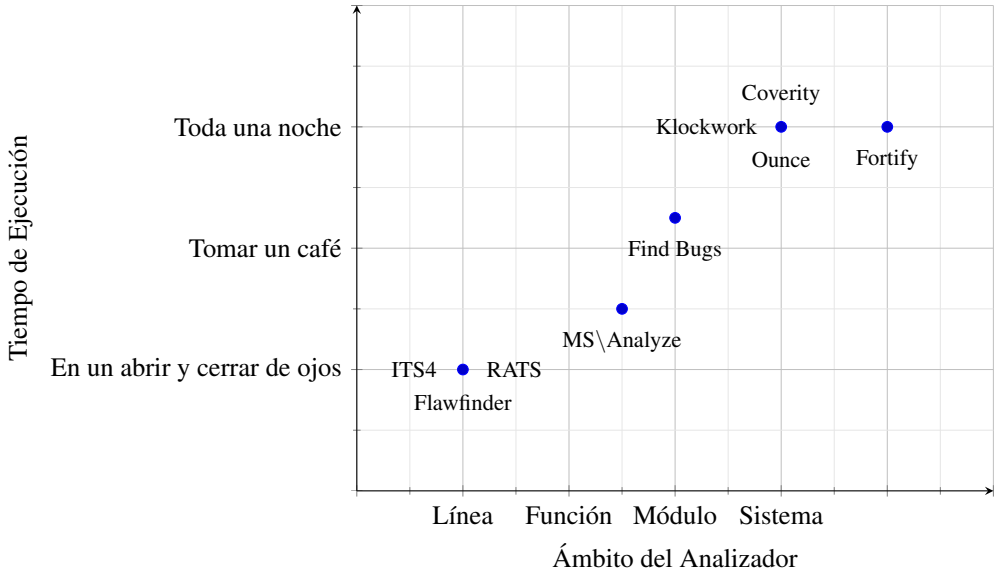


Figura 6.1: Relación entre el ámbito de actuación y eficiencia en diferentes analizadores estáticos [15]

Esta decisión tiene impacto sobre varios aspectos técnicos y sobre algunos de los objetivos de diseño definidos. Por un lado, la eficiencia variará mucho en función del ámbito empleado, como se puede observar en la figura 6.1, donde se ven varios ejemplos de analizadores de código estático según su ámbito de actuación. La eficiencia no es algo que sea un objetivo de diseño, por lo que no es algo preocupante en el caso de BOA, pero es una característica que siempre es deseable, pero no siempre priorizada como es el caso presente. Por otro lado, estamos favoreciendo de manera equilibrada a la flexibilidad y a la facilidad de uso, ya que al utilizar un fichero como ámbito máximo evitamos tener que realizar configuraciones que, aunque mínimas, posiblemente

el usuario debería realizar del entorno si fuera el ámbito de actuación máximo (e.g. variables de entorno), y que sea posible definir módulos que actúen sobre ámbitos menores aporta una mayor flexibilidad al darle al usuario más control (módulos que el usuario, por medio de la extensión, defina).

6.5 Archivo de Reglas

El archivo de reglas es un elemento vital de BOA. Toda la comunicación que proviene del usuario, es decir, el mundo exterior, se realizará a través de este fichero y se podrá realizar con diferentes agentes como se ha explicado en la subsección Comunicación de la sección Objetivos del Diseño. Dicho archivo de reglas podrá modificarse por el usuario para especificar una configuración concreta, por lo que lo normal será tener diferentes archivos de reglas por proyecto o por vulnerabilidades (se pueden detectar varias vulnerabilidades a través de diferentes módulos, pero si tenemos proyectos diferentes en lenguajes de programación distintos, seguramente no se busquen las mismas vulnerabilidades). El flujo de datos con el archivo de reglas, con el usuario como agente principal, se puede observar en la figura 6.2.

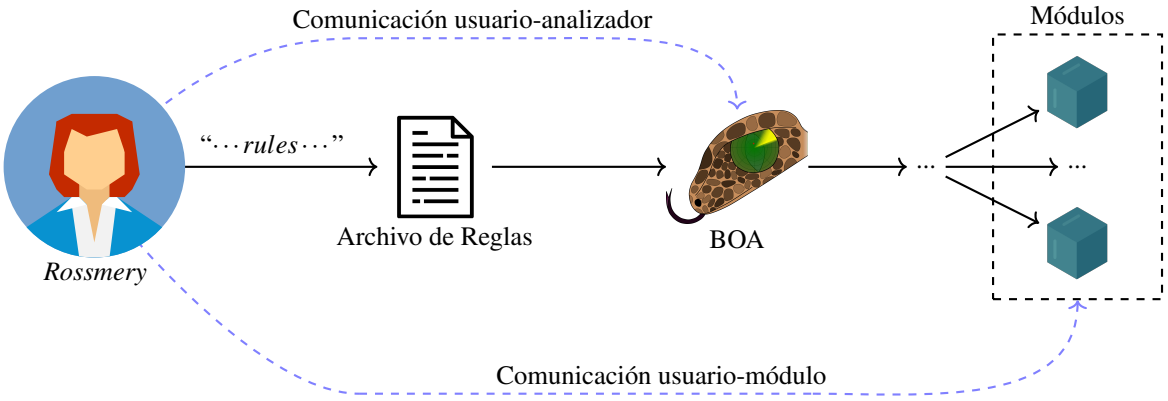


Figura 6.2: Flujo de datos del archivo de reglas en BOA

Hay diferentes formatos que pueden utilizarse para definir el archivo de reglas:

- Formato propio: la definición de un formato propio nos daría la flexibilidad de definir unas directivas propias, con lo que obtendríamos un formato totalmente ajustado a nuestras necesidades. La principal desventaja que surge es que habría que implementar un mecanismo totalmente acoplado a las directivas, y en el momento de que surgiera la necesidad de añadir nuevas directivas, habría que modificar el mecanismo que las procesa, quizá incluso de manera drástica.
- Formato JSON: un fichero con formato JSON tiene la característica de que es muy ligero, por lo que es ideal para entornos web, y que son muy sencillos de trabajar. La principal desventaja que tienen es que, visualmente, son difíciles de seguir, ya que un ámbito o nivel jerárquico se define en un par de llaves, lo cual no indica el significado general de dicho nivel. Además, otra gran desventaja es que no tiene soporte para comentarios, lo cual es algo muy a tener en cuenta si el uso que se va a hacer del fichero es ser leido y modificado a mano por un usuario. En el entorno web esto no suele ser un problema debido a que los ficheros JSON se utilizan para intercambiar información, y para ese propósito es ideal, pero no lo es tanto para un fichero de configuración.

- Formato XML: un fichero con formato XML se parece mucho a un fichero JSON, pero se caracteriza por definir los ámbitos o niveles jerárquicos con etiquetas a las cuales se les da un nombre. Al tener etiquetas que se les puede dar un nombre, ganamos mucho valor semántico, ya que los elementos insertados dentro de una etiqueta estarán relacionados con la etiqueta padre a través de una relación semántica. Además, en cada etiqueta, además de poder asignar un valor, también podemos asignar atributos, con lo que no es necesario crear etiquetas que parezcan totalmente artificiales con el fin de añadir un valor a una etiqueta. La principal desventaja que presentan los ficheros XML es que su análisis es complejo, y obtener los valores de una manera sencilla de trabajar con los mismos no es nada fácil, ya que pueden haber problemas de referencia (e.g. los valores de una etiqueta y sus atributos tienen se les tiene que poder referenciar con un nombre, lo cual para los atributos no es un problema ya que sus nombres tienen que ser distintos entre sí, pero el valor de la etiqueta no tiene ningún nombre y tiene que darse uno si comparte un espacio de nombre común con el de los atributos).

En el caso de BOA, el formato del archivo de reglas será un **fichero XML**. Las ventajas que presenta este formato es lo que necesitamos dadas nuestras necesidades. El formato JSON también podría haber servido, pero el formato XML se acopla mucho mejor a nuestros objetivos, ya que añade una expresividad al archivo de reglas que necesitamos. Además, la limitación de no poder utilizar comentarios en el formato JSON empeoraría la flexibilidad y facilidad de uso de BOA debido a que estamos añadiendo una limitación y perdemos la oportunidad de ofrecerle al usuario un mecanismo a través del cual explicar el porqué o qué hacen ciertas configuraciones de sus ficheros de reglas. La principal razón por la que se va a utilizar un fichero XML es su expresividad de cara al usuario.

Algo importante a destacar es que el archivo de reglas será obligatorio para que BOA funcione, ya que si no se le indica la ruta del mismo, no procederá con el análisis, ya que básicamente no sabrá qué analizar. Aunque sea una configuración mínima, dicha configuración mínima deberá indicarle a BOA de manera explicita dónde se encuentra. Esto no significa que todos los elementos de BOA tengan que tener un valor, ya que algunos, con el fin de mejorar el objetivo de diseño de la facilidad de uso, tendrán un valor por defecto.

6.5.1 Elementos

El archivo de reglas tendrá unos elementos predefinidos que serán válidos, y no todo elemento será válido, solo aquellos predefinidos. El objetivo de ello es poder realizar una comunicación efectiva entre el usuario y el resto del sistema, como si de un protocolo se tratara donde solo hay definidos un conjunto de mensajes válidos. Esto nos aporta la limitación de que por cada elemento que se quiera añadir posteriormente, habrá que modificar el análisis del fichero, pero al tratarse de un fichero XML, la API para acceder a sus elementos será la misma de la que se habrá hecho uso para el análisis de los otros elementos, por lo que los cambios que tendrán que realizarse serán mínimos.

A continuación, explicaremos los elementos que serán válidos en el archivo de reglas. También explicaremos cuál será el cometido de cada elemento, sus detalles. El orden de aparición de los siguientes elementos será de más a menos general, por lo que no coincidirá con el orden de aparición del archivo de reglas.

Elemento *boa_rules*

Se trata del elemento principal. Este elemento será obligatorio y único. Su ámbito será el ámbito global, ya que no tendrá nodos (i.e. etiqueta o elemento) padre. No tendrá ningún atributo.

La principal utilidad de este elemento es recoger toda la configuración del fichero bajo un mismo elemento, otorgando semántica debido al nombre, el cual indica que son las ”reglas para BOA”. Otra utilidad de unificar toda la configuración bajo un elemento común es la de realizar una comprobación básica, de manera que si este nodo no se encuentra en el archivo de reglas, el fichero XML que BOA está analizando no es un archivo de reglas válido.

Elemento *boa_rules.parser*

Se trata de un elemento que engloba una configuración específica para el analizador del lenguaje de programación objetivo, por lo que en este nodo se especificarán datos y metadatos para BOA y para el analizador del lenguaje de programación sobre el lenguaje de programación. Este elemento será obligatorio y único. Su ámbito será *boa_rules*. No tendrá ningún atributo.

Al igual que el elemento *boa_rules*, este elemento tiene la finalidad de agrupar un conjunto de elementos de manera que pertenezcan a una misma categoría semántica, y en este caso dicha categoría semántica tiene relación con el analizador del lenguaje de programación objetivo de un archivo de reglas en concreto. En este elemento se definirán reglas como cuál será el analizador de lenguaje de programación a utilizar, información que se le indicará desde BOA a dicho analizador, metainformación para BOA acerca del analizador para la interoperabilidad, etc. Los elementos internos a este elemento serán los que permitan favorecer la comunicación usuario-módulo y analizador-módulo (el analizador del lenguaje de programación también será un módulo, ya que el diseño será modular), ya que la información especificada en etiquetas internas permitirá que el analizador pueda comunicarse con los módulos relativos al analizador del lenguaje de programación.

Elemento *boa_rules.modules*

Se trata de un elemento que engloba todos los módulos que van a ser cargados en BOA. Este elemento será obligatorio y único. Su ámbito será *boa_rules*. No tendrá ningún atributo.

La categoría semántica para este elemento está relacionada con los módulos que BOA va a cargar. Toda la configuración de los módulos a utilizar por BOA estarán bajo esta etiqueta. La razón de agrupar a no un único módulo bajo una etiqueta diferente sino todos bajo una etiqueta común (i.e. elemento *modules*) es la de poder identificar de manera univoca la localización de los módulos en el archivo de reglas, y una vez localizados, pasar a procesar la configuración de cada módulo de manera individual. Elementos internos a este elemento serán los que permitan favorecer la comunicación usuario-módulo y analizador-módulo, ya que la información especificada en etiquetas internas permitirá que el analizador pueda comunicarse con los módulos.

Elemento *boa_rules.report*

Se trata de un elemento que engloba toda la configuración relativa al informe resultante de utilizar BOA (i.e. los resultados del análisis). Este elemento será obligatorio y único. Su ámbito será *boa_rules*. No tendrá ningún atributo.

Al igual que con otros elementos, englobar la configuración relativa al informe resultante nos ayuda a mejorar la semántica de los elementos bajo este elemento debido a que significa que es una configuración relativa al informe, y esto nos permite, por ejemplo, definir un elemento con un nombre concreto para este elemento y el mismo elemento con el mismo nombre en otro elemento diferente, y aunque ambos tengan el mismo nombre, debido a que el nodo padre será distinto, significarán cosas diferentes. Este elemento se utilizará para aportar comunicación usuario-analizador y analizador-módulo (el informe también será un módulo, ya que el diseño será modular), ya que la información especificada en etiquetas internas permitirá que el analizador pueda comunicarse con los módulos relativos al informe.

Elemento *parser.name*

Se trata de un elemento que contendrá metainformación relativa al analizador empleado que procesará el lenguaje de programación del fichero bajo análisis, la cual será empleada de manera interna por BOA para mensajes de información y no se utilizará para implementar funcionalidad. Este elemento será obligatorio y único. Su ámbito será *boa_rules.parser*. No tendrá ningún atributo.

El contenido de la etiqueta definirá el nombre del analizador del lenguaje de programación empleado (e.g. Pycparser). El mal o buen uso del valor definido en esta etiqueta no implicará ningún comportamiento anormal.

Elemento *parser.lang_objective*

Se trata de un elemento que contendrá metainformación relativa al analizador empleado que procesará el lenguaje de programación del fichero bajo análisis, la cual será empleada de manera interna por BOA para mensajes de información y no se utilizará para implementar funcionalidad. Este elemento será obligatorio y único. Su ámbito será *boa_rules.parser*. No tendrá ningún atributo.

El contenido de la etiqueta definirá el nombre del lenguaje de programación empleado (e.g. C, C++, Java). El mal o buen uso del valor definido en esta etiqueta no implicará ningún comportamiento anormal.

Elemento *parser.module_name*

Se trata de un elemento que contendrá el nombre del módulo que se cargará relativo al analizador del lenguaje de programación empleado. Este elemento será obligatorio y único. Su ámbito será *boa_rules.parser*. No tendrá ningún atributo.

El contenido de la etiqueta definirá el nombre del módulo que contiene la implementación del comportamiento necesario para la interoperabilidad entre BOA y el analizador del lenguaje de programación.

En el caso de no encontrarse el módulo indicado, deberá de avisarse al usuario y detener la ejecución debido a que es una pieza vital para continuar con el análisis.

Elemento *parser.class_name*

Se trata de un elemento que contendrá el nombre de la clase que se cargará relativa al módulo definido en la etiqueta *parser.module_name*. Este elemento será obligatorio y único. Su ámbito será *boa_rules.parser*. No tendrá ningún atributo.

El contenido de la etiqueta definirá el nombre de la clase que contiene la implementación concreta del comportamiento necesario para la interoperabilidad entre BOA y el analizador del lenguaje de programación. En el caso de no encontrarse la clase indicada en el módulo indicado en la etiqueta *parser.module_name*, deberá de avisarse al usuario y detener la ejecución debido a que es una pieza vital para continuar con el análisis. Se utiliza una clase (programación orientada a objetos) con el objetivo de encapsular el comportamiento bajo métodos concretos y conocidos.

Elemento *parser.callback*

Se trata de un elemento que contendrá otros elementos para realizar llamadas a métodos del módulo implementado en BOA que está definido en la etiqueta *parser.module_name*. Este elemento será obligatorio y único. Su ámbito será *boa_rules.parser*. No tendrá ningún atributo.

La razón de indicar otros elementos que contengan los valores definitivos es la de permitir al usuario definir más de una única llamada a un método cuando finalice el procesamiento del módulo (i.e. *callback*) indicado en la etiqueta *parser.module_name*.

Elemento *parser.env_vars*

Se trata de un elemento que contendrá otros elementos para definir nombres de variables de entorno. Este elemento será obligatorio y único. Su ámbito será *boa_rules.parser*. No tendrá ningún atributo.

La razón de indicar otros elementos que contengan los valores definitivos es la de permitir al usuario definir más de un único nombre de una variable de entorno.

Elemento *modules.module*

Se trata de un elemento que contiene toda la configuración relacionada con un módulo concreto. Este elemento será obligatorio (mínimo 1 elemento) y podrá repetirse. Su ámbito será *boa_rules.modules*. No tendrá ningún atributo.

Este elemento contiene toda la configuración relativa a un módulo concreto, y este elemento se tendrá que procesar múltiples veces si se repite. Se indica que tiene que haber 1 elemento mínimo debido a que si no se configuran estos módulos, los cuales hacen referencia a los módulos cuya tarea será detectar vulnerabilidades,

no hay razón de ejecutar BOA. A través de la configuración realizada en este módulo se realiza la comunicación usuario-módulo y analizador-módulo.

Elemento *report.module_name*

Se trata de un elemento que contendrá el nombre del módulo que se cargará relativo al informe generado por BOA. Este elemento será opcional (será obligatorio si se define el elemento *report.class_name*) y único (en el caso de definirse). Su ámbito será *boa_rules.report*. No tendrá ningún atributo.

Con el fin de reforzar el principio de objetivo de extensión y personalización, se permitirá indicar un módulo que defina el comportamiento relativo a la generación del informe de BOA. Para reforzar la facilidad de uso, se asignará un valor por defecto si no se define la etiqueta *report.module_name*. Esta opción le da la flexibilidad al usuario de poder elegir cómo se visualizará (e.g. terminal, HTML, PDF) o qué información se empleará para los informes (e.g. localización, vulnerabilidad, identificadores CVE).

Elemento *report.class_name*

Se trata de un elemento que contendrá el nombre de la clase que se cargará relativa al módulo definido en la etiqueta *report.module_name*. Este elemento será opcional (será obligatorio si se define el elemento *report.module_name*) y único (en el caso de definirse). Su ámbito será *boa_rules.report*. No tendrá ningún atributo.

El contenido de la etiqueta definirá el nombre de la clase que contiene la implementación concreta del comportamiento necesario para mostrar el informe resultante del análisis de BOA. En el caso de no encontrarse la clase indicada, si se define o la indicada en el valor por defecto, en el módulo indicado en la etiqueta *report.module_name*, deberá de avisarse al usuario y detener la ejecución debido a que es una pieza vital para continuar con el análisis. Se utiliza una clase (programación orientada a objetos) con el objetivo de encapsular el comportamiento bajo métodos concretos y conocidos.

Elemento *report.args_sorting*

Se trata de un elemento que contendrá un valor booleano para indicar si los argumentos se ordenan o no. Este elemento será opcional y único (en el caso de definirse). Su ámbito será *boa_rules.report*. No tendrá ningún atributo.

Los elementos del archivo de reglas, debido a limitaciones explicadas en la sección xmltodict del capítulo Tecnologías y detallado en la subsección Orden de los Argumentos del presente capítulo, se cargan de manera desordenada. Con esta etiqueta indicamos si queremos intentar una ordenación a los argumentos definidos en la etiqueta *report.args*. Se trata de un valor opcional que tiene un valor por defecto, y este valor por defecto es el de no realizar ordenación alguna, lo cual significa no hacer nada respecto a la ordenación.

Elemento *report.args*

Se trata de un elemento que contendrá otros elementos que definirán la comunicación usuario-módulo relativo al módulo del informe generado por BOA. Este elemento será obligatorio y único. Su ámbito será *boa_rules.report*. No tendrá ningún atributo.

Los argumentos que el usuario va a asignar al módulo relativo al informe generado por BOA estarán asignados en elementos internos de la presente etiqueta. Los detalles se explican en la subsección Argumentos del presente capítulo.

Elemento *parser.callback.method*

Se trata de un elemento que va a contener un valor relacionado con el *callback* de la clase indicada en la etiqueta *parser.class_name*. Este elemento será obligatorio (mínimo 1 elemento) y podrá repetirse. Su ámbito será *boa_rules.parser.callback*. Tendrá 2 atributos, los cuales serán "*name*" y "*callback*" y ambos serán obligatorios.

El valor de esta etiqueta se expresará por medio de sus atributos. El atributo "*name*" contendrá un nombre identificativo de manera que BOA pueda referenciar al *callback* por medio del nombre asignado. El atributo "*callback*" contendrá el nombre de un método que debería existir en la clase indicada en la etiqueta *parser.class_name*, y en el caso de no existir, se deberá de avisar al usuario y detener la ejecución de BOA debido a que es necesario el valor para proceder. Estos *callback* definidos a través del atributo "*callback*" serán los que BOA utilice para obtener información procesada del analizador del lenguaje de programación, cuya interoperabilidad estará definida en la clase indicada en la etiqueta *parser.class_name* y cuyo *callback* nos devolverá el resultado. Este resultado será visible para todos los módulos cargados relativo a las etiquetas *modules.module* a través del nombre indicado en el atributo "*name*".

Elemento *env_vars.env_var*

Se trata de un elemento que contiene un valor concreto relativo al nombre de una variable de entorno. Este elemento será opcional y podrá repetirse. Su ámbito será *boa_rules.parser.env_vars*. No tendrá ningún atributo.

La razón de definir variables de entorno y no variables directamente es que algunos analizadores de lenguajes de programación utilizan dichas variables de entorno para comunicarse con el usuario en lugar de utilizar directamente variables. Debido a ello, el peor de los casos es utilizar dichas variables de entorno para realizar una interoperabilidad efectiva. El problema es que añade a BOA dependencia con el entorno, y para intentar minimizar el problema, BOA no realiza la tarea de asignar el valor para las variables de entorno, sino que simplemente las carga, minimizando así la dependencia con el sistema y dejando la tarea al usuario de cargar dichas variables de entorno. A través de estas variables de entorno se realizará la comunicación usuario-módulo con los módulos relativos al analizador del lenguaje de programación objetivo.

Elemento *module.module_name*

Se trata de un elemento que contendrá el nombre del módulo que se cargará relativo a la búsqueda de vulnerabilidades. Este elemento será obligatorio y único. Su ámbito será *boa_rules.modules.module*. No tendrá ningún atributo.

Con el fin de reforzar el principio de objetivo de extensión, se permitirá indicar un módulo que defina el comportamiento relativo a la búsqueda de vulnerabilidades. En el caso de no poder cargarse el módulo indicado en la etiqueta, se avisará al usuario y, a través de algún mecanismo, se podrá continuar o detener la ejecución del análisis. La razón de que sea opcional no detener la ejecución es debido a que pueden haber muchos módulos, y si uno falla, mientras no sea una dependencia de otro módulo (incluso si se trata de una dependencia, el usuario podrá elegir continuar sin la dependencia si no es vital para el módulo principal), se podrá continuar sin ningún problema con el análisis pero sin el módulo que no se ha podido cargar.

Elemento *module.class_name*

Se trata de un elemento que contendrá el nombre de la clase que se cargará relativa al módulo especificado en la etiqueta *module.module_name*. Este elemento será obligatorio y único. Su ámbito será *boa_rules.modules.module*. No tendrá ningún atributo.

El contenido de la etiqueta definirá el nombre de la clase que contiene la implementación concreta del comportamiento necesario para la búsqueda de vulnerabilidades. En el caso de no encontrarse la clase indicada, si se define o la indicada en el valor por defecto, en el módulo indicado en la etiqueta *module.module_name*, deberá de avisarse al usuario y, a través de algún mecanismo, se podrá continuar o detener la ejecución del análisis. La razón de que sea opcional no detener la ejecución es debido a que pueden haber muchos módulos con sus respectivas clases, y si una clase falla, mientras no sea una dependencia de otro módulo (incluso si se trata de una dependencia, el usuario podrá elegir continuar sin la dependencia si no es vital para el módulo principal), se podrá continuar sin ningún problema con el análisis pero sin el módulo que no se ha podido cargar que contiene la clase definida en la presente etiqueta. Se utiliza una clase (programación orientada a objetos) con el objetivo de encapsular el comportamiento bajo métodos concretos y conocidos.

Elemento *module.severity_enum*

Se trata de un elemento que contendrá un valor en un formato concreto para cargar un módulo y una clase que definan una interfaz. Este elemento será opcional y único (en el caso de definirse). Su ámbito será *boa_rules.modules.module*. No tendrá ningún atributo.

Nos permite especificar un módulo que defina diferentes niveles de severidad, los cuales serán valores meramente informativos. Habrá un mecanismo a través del cual se definan interfaces con los niveles de severidad y, dicha severidad, sea accesible desde el módulo que está cargando dicha interfaz, presente en la etiqueta *module.module_name*. Debido a que se trata de un elemento opcional, se le proporcionará un valor por defecto que hará referencia a una interfaz genérica. Esta etiqueta añade flexibilidad y personalización al sistema. Se detallará en la sección Módulos Principales del presente capítulo.

El formato que se espera al utilizar esta etiqueta es el siguiente: "*severity_module*"."*severity_class*".

Elemento *module.args_sorting*

Se trata de un elemento que contendrá un valor booleano para indicar si los argumentos se ordenan o no. Este elemento será opcional y único (en el caso de definirse). Su ámbito será *boa_rules.modules.module*. No tendrá ningún atributo.

Los elementos del archivo de reglas, debido a limitaciones explicadas en la sección xmltodict del capítulo Tecnologías y detallado en la subsección Orden de los Argumentos del presente capítulo, se cargan de manera desordenada. Con esta etiqueta indicamos si queremos intentar una ordenación a los argumentos definidos en la etiqueta *module.args*. Se trata de un valor opcional que tiene un valor por defecto, y este valor por defecto es el de no realizar ordenación alguna, lo cual significa no hacer nada respecto a la ordenación.

Elemento *module.lifecycle_handler*

Se trata de un elemento que contendrá un valor en un formato concreto para cargar un módulo y una clase que definan un ciclo de vida para el módulo indicado en la etiqueta *modules.module*. Este elemento será opcional y único (en el caso de definirse). Su ámbito será *boa_rules.modules.module*. No tendrá ningún atributo.

Se permitirá especificar un ciclo de vida de manera que el usuario tenga elección sobre cómo se va a realizar la ejecución del análisis bajo sus propios criterios. Con el fin de aportar flexibilidad, se utilizará un valor por defecto que ejecutará un ciclo de vida por defecto básico. Se detallará en la sección Módulos Principales del presente capítulo.

El formato que se espera al utilizar esta etiqueta es el siguiente: "$lifecycle\_module$"."$lifecycle\_class$".

Elemento *module.dependencies*

Se trata de un elemento que contendrá otros elementos que definirán las dependencias. Este elemento será opcional y único (en el caso de definirse). Su ámbito será *boa_rules.modules.module*. No tendrá ningún atributo.

Los módulos pueden necesitar dependencias con el fin de conseguir comportamientos complejos. Para ello, se pueden definir dependencias en elementos internos de la presente etiqueta.

Elemento *module.args*

Se trata de un elemento que contendrá otros elementos que definirán la comunicación usuario-módulo relativo a los módulos encargados de detectar vulnerabilidades. Este elemento será obligatorio y único. Su ámbito será *boa_rules.modules.module*. No tendrá ningún atributo.

Los argumentos que el usuario va a asignar al módulo relativo a la búsqueda de vulnerabilidades estarán asignados en elementos internos de la presente etiqueta. Los detalles se explican en la subsección Argumentos del presente capítulo.

Elemento *dependencies.dependency*

Se trata de un elemento que define una dependencia concreta para el módulo indicado en la etiqueta *modules.module* que contiene la presente etiqueta. Este elemento será obligatorio (mínimo 1 elemento) y podrá repetirse. Su ámbito será *boa_rules.modules.module.dependencies*. No tendrá ningún atributo.

A través de las dependencias conseguimos hacer efectiva la comunicación módulo-módulo. Estas dependencias son módulos igual que los que tenemos en la etiqueta *modules*, pero cuyo valor va a ser accedido por los módulos que contiene a la presente etiqueta como dependencia. Para que la dependencia sea accedida por el módulo que la define como dependencia, se necesitará una configuración para poder referenciar la dependencia, y dicha configuración estará en elementos interiores de la presente etiqueta.

Elemento *dependency.module_name*

Se trata de un elemento que contendrá el nombre del módulo que se cargará relativo a las dependencias de los módulos relativos a la búsqueda de vulnerabilidades. Este elemento será obligatorio y único. Su ámbito será *boa_rules.modules.module.dependencies.dependency*. No tendrá ningún atributo.

Con el fin de reforzar la flexibilidad y la comunicación, se permite definir módulos cuyos resultados sean accesibles por otros módulos. Estas dependencias solo se podrán definir para los módulos relativos a la búsqueda de vulnerabilidades. En el caso de no poder cargarse el módulo indicado en la etiqueta, se avisará al usuario y, si el módulo que depende de esta dependencia decide continuar con su ejecución, podrá continuar, pero siempre que, a través de algún mecanismo, se haya indicado que el fallo de cargar un módulo no sea un error fatal y se aborte la ejecución del análisis.

Elemento *dependency.class_name*

Se trata de un elemento que contendrá el nombre de la clase que se cargará relativa al módulo especificado en la etiqueta *dependency.module_name*. Este elemento será obligatorio y único. Su ámbito será *boa_rules.modules.module.dependencies.dependency*. No tendrá ningún atributo.

El contenido de la etiqueta definirá el nombre de la clase que contiene la implementación concreta del comportamiento necesario para la búsqueda de vulnerabilidades, pero tratándose de una dependencia de otro módulo. En el caso de no encontrarse la clase indicada, si se define o la indicada en el valor por defecto, en el módulo indicado en la etiqueta *dependency.module_name*, deberá de avisarse al usuario y, a través de algún mecanismo, se podrá continuar o detener la ejecución del análisis. La razón de que sea opcional no detener la ejecución es debido a que pueden haber muchos módulos con sus respectivas clases, y si una clase falla, mientras no sea una dependencia vital para el módulo del que es dependencia, se podrá continuar sin ningún problema con el análisis pero sin el módulo que no se ha podido cargar que contiene la clase definida en la presente etiqueta. Se utiliza una clase (programación orientada a objetos) con el objetivo de encapsular el comportamiento bajo métodos concretos y conocidos.

Elemento *dependency.callback*

Se trata de un elemento que se comporta de igual manera que la etiqueta *parser.callback* a nivel sintáctico, pero no a nivel semántico. Este elemento será obligatorio y único. Su ámbito será *boa_rules.modules.module.dependencies.dependency*. No tendrá ningún atributo.

La razón de indicar otros elementos que contengan los valores definitivos es la de permitir al usuario definir más de una única llamada a un método cuando finalice el procesamiento del módulo (i.e. *callback*) indicado en la etiqueta *dependency.module_name*.

Elemento *dependency.callback.method*

Se trata de un elemento que va a contener un valor relacionado con el *callback* de la clase indicada en la etiqueta *dependency.class_name*. Este elemento será obligatorio (mínimo 1 elemento) y podrá repetirse. Su ámbito será *boa_rules.modules.module.dependencies.dependency.callback*. Tendrá 2 atributos, los cuales serán "*name*" y "*callback*" y ambos serán obligatorios.

El valor de esta etiqueta se expresará por medio de sus atributos. El atributo "*name*" contendrá un nombre identificativo de manera que BOA pueda referenciar al *callback* por medio del nombre asignado. El atributo "*callback*" contendrá el nombre de un método que debería existir en la clase indicada en la etiqueta *dependency.class_name*, y en el caso de no existir, se deberá de avisar al usuario y detener la ejecución de BOA debido a que es necesario el valor para proceder. Estos *callback* definidos a través del atributo "*callback*" serán los que BOA utilice para obtener información de un módulo y que el módulo definido en la etiqueta *modules.module* que contenga las dependencias descritas, tenga acceso a la información del módulo indicado en la etiqueta *dependency.module_name*. Este resultado será visible para el módulo que contenga la dependencia indicado en la etiqueta *module.module_name* a través del nombre indicado en el atributo "*name*".

A través de esta directiva se consigue hacer efectiva la comunicación módulo-módulo, con BOA como intermediario de la información.

6.5.2 Argumentos

Los argumentos que se definen en el archivo de reglas en los elementos *module.args* y *report.args* son una parte vital de BOA. En el caso del elemento *module.args* es la manera principal en la que el usuario será capaz de pasar información a los módulos relacionados con la búsqueda de vulnerabilidades, ya sea para cambiar el comportamiento de los módulos o para ampliar la base de conocimientos con la que el módulo trabaja para la detección de la vulnerabilidad objetivo. En el caso del elemento *report.args* es un mecanismo con el que podemos pasar información concreta para la personalización del informe generado por BOA, pero no tiene mayor relevancia (e.g. formato HTML con tablas de un tamaño concreto). La importancia la encontramos, como ya se ha dicho, en el elemento *module.args*, ya que favorece la comunicación usuario-módulo en gran medida al permitir dicha comunicación.

Con la finalidad de favorecer la flexibilidad, los argumentos podrán definirse de manera recursiva, con lo que conseguiremos tener una cantidad de anidamientos que quedará a elección del usuario. Esta decisión favorece en gran medida la flexibilidad que se le da al usuario, ya que al definir una estructura recursiva dentro

del fichero de reglas, se podrán tener argumentos de argumentos. Para permitir estos anidamientos, definimos 3 tipos de argumentos: elementos *dict*, elemento *list* y elemento *element*.

Elemento *args.dict*

Se trata de un elemento que va a representar un diccionario (i.e. estructura de datos que almacena valores y se accede por un nombre, de la misma manera que funciona una tabla *hash*). Este elemento será opcional y podrá repetirse, pero de manera obligatoria y única tiene que haber un elemento *dict* dentro de la etiqueta *args* si se van a definir argumentos. Su ámbito principal podrá ser *boa_rules.modules.module.args* o *boa_rules.report.args* y, de manera recursiva, también podemos encontrar instancias de esta etiqueta dentro de dichos ámbitos. Los atributos dependerán del elemento padre, por lo que si el elemento padre de una etiqueta *dict* es *args* (solo una etiqueta *dict* puede tener como elemento padre a la etiqueta *args* por lo explicado anteriormente), no tendrá atributos, pero si el padre es otra etiqueta *dict*, habrá un atributo ”name” obligatorio.

Esta etiqueta servirá para poder definir elementos internos dentro de la misma de manera que tengan que identificarse con un atributo ”name”, el cual será obligatorio para todos los hijos directos de la presente etiqueta. La finalidad es poder acceder a elementos a través de un nombre conocido que, gracias a su nombre, le aporte semántica al valor del mismo.

Elemento *args.list*

Se trata de un elemento que va a representar una lista (i.e. estructura de dato que almacena valores de manera ordenada y se puede acceder a los mismos a través de la posición relativa de inserción). Este elemento será opcional y podrá repetirse. Su ámbito será *boa_rules.modules.module.args.dict* o *boa_rules.report.args.dict* y, de manera recursiva, también podemos encontrar instancias de esta etiqueta dentro de dichos ámbitos. Los atributos dependerán del elemento padre, por lo que si el elemento padre de una etiqueta *list* es *dict*, habrá un atributo ”name” obligatorio.

Esta etiqueta servirá para poder definir elementos internos dentro de la misma de manera que no tengan que identificarse con un nombre. La finalidad es poder acceder a elementos a través de la posición, lo cual es muy útil cuando tenemos elementos que están agrupados porque comparten una semántica común, pero que de manera independiente no se diferencian entre sí (e.g. una lista puede contener países, ya que cada nombre de país de manera independiente no tiene semántica propia de manera que se diferencia de la semántica de otro país). Además, en muchas ocasiones, tenemos una serie de datos que necesitamos introducir y no podemos darle un nombre individual a cada uno, como sucede en el diccionario, porque al tener una lista indefinida de elementos, al no ser que el nombre de cada elemento siga un patrón que luego se pueda calcular, no se podrá acceder a nombres que son dependientes del valor concreto (e.g. un país con valor el nombre del país, y nombre a través del cual acceder también el nombre del país).

Elemento *args.element*

Se trata de un elemento que va a representar un elemento básico de información. Este elemento será opcional y podrá repetirse. Su ámbito será *boa_rules.modules.module.args.dict* o *boa_rules.report.args.dict* y, de manera recursiva, también podemos encontrar instancias de esta etiqueta dentro de dichos ámbitos. Los

atributos dependerán del elemento padre, por lo que si el elemento padre de una etiqueta *element* es *dict*, habrá un atributo ”name” obligatorio, y además, habrá un atributo ”value” obligatorio.

Esta etiqueta es la que estará dentro de los contenedores definidos por las etiquetas *dict* y *list* y que contendrá los valores concretos que el usuario quiera especificar en los argumentos. Como mínimo tendrá el atributo ”value”, el cual contendrá el valor que el usuario quiera darle y será accesible desde los módulos de BOA. Si el elemento padre de la presente etiqueta es una etiqueta *dict*, también tendrá un atributo ”name” para identificar el elemento dentro del diccionario.

Orden de los Argumentos

Como ya se comentó en la subsección xmltodict del capítulo Tecnologías, hay muchas ocasiones en las que nos interesa mantener el orden de los elementos respecto a como son definidos. Debido al uso de la utilidad xmltodict, la estructura de datos que utiliza para devolvernos los resultados de un fichero XML es un diccionario nativo de Python, el cual no conserva el orden de los elementos conforme son insertados, por lo que el orden en el que se definen las reglas, y más en concreto los argumentos, que es lo que nos interesa, no estarán ordenados una vez accedamos a los mismos en BOA. Esto puede ser un problema, ya que, por ejemplo, puede que definamos un orden en un elemento *list* donde el orden sea esencial por cualquier razón, y dicho orden se va a perder. Para evitar esta pérdida de información, se ha definido un elemento en el archivo de reglas, el cual es *args_sorting* e indica si se quiere mantener el orden de los argumentos, tanto para los módulos relacionados con la búsqueda de vulnerabilidades como los módulos relacionados con la generación del informe de BOA.

La etiqueta *args_sorting* va a tener un valor por defecto el cual va a indicar que no se realice ninguna operación de ordenación, es decir, que se aplique el comportamiento por defecto de xmltodict. En el caso de indicar un valor que indique que se quiere realizar una ordenación, se intentará ordenar los elementos en la medida de lo posible, pero un orden total no será posible por las limitaciones de los diccionarios nativos de Python, estructura con la que se va a trabajar los argumentos de los módulos, pero se realizará una ordenación parcial.

La manera en la que los diccionarios de Python ordenan los valores es alfabeticamente en función del nombre de los valores almacenados en el mismo. Teniendo esto en cuenta, la única manera de obtener un orden total y no uno parcial será no mezclar elementos *dict*, *list* y *element*, ya que al mezclar dichos elementos es cuando se realiza el ordenamiento de manera alfabética que hemos comentado (los nombres que se darán en el diccionario de Python serán los mismos que las etiquetas, que son *dict*, *list* y *element*, debido a que así es como funciona xmltodict). El orden parcial que se aplicará si se hace uso de la etiqueta *args_sorting* será ordenar todos los elementos de un mismo ámbito para cada ámbito de los argumentos que pertenezcan a un mismo elemento. De esta manera, si el primer elemento encontrado en un ámbito concreto es *element*, todas las etiquetas de tipo *element* aparecerán inicialmente en el ámbito procesado, y si el segundo elemento que se encuentra es *list*, todos los elementos de tipo *list* irán después del último elemento *element* y lo mismo para el elemento *dict*. Como ya hemos dicho, esto es un orden parcial, no total, que lo que hace es agrupar todas aquellas etiquetas en función del primer elemento que se encuentre, y esto se hará así porque esto sí que se puede hacer, pero obtener el orden inicial sin tener que pasar a un procesamiento directo del fichero XML, no.

Un ejemplo del problema lo podemos ver en el siguiente ejemplo:

1. Tenemos los argumentos ordenados, en un ámbito concreto, de la siguiente manera: $element_1$, $list_1$,

$element_2$, $dict_1$, $list_2$.

2. Obtenemos el resultado sin aplicar la etiqueta *args_sorting*: $dict_1$, $list_1$, $list_2$, $element_1$, $element_2$.

3. No nos convence el resultado de xmltodict y aplicamos la etiqueta *args_sorting*: $element_1$, $element_2$, $list_1$, $list_2$, $dict_1$.

La razón de que todo esto ocurra es por una limitación de la utilidad xmltodict, ya que si utilizara una estructura de datos que sí que obtuviera los resultados ordenados, como *Collections.OrderedDict*, o al menos se pudiera elegir utilizar dicha estructura de datos, no tendríamos ningún problema.

6.6 Módulos Principales

Como se comentó en la subsección Modularidad del presente capítulo, la manera en la que se llevará a cabo la modularidad será a través de unos módulos principales. Estos módulos principales tendrán cada uno una finalidad bien definida de modo que cada uno tendrá un ámbito de actuación en BOA. La definición de unos módulos principales ayudará a diferenciar y separar el comportamiento de modo que seamos capaces de entender aquellos módulos que sean interesantes para el usuario, y no que forzosamente haya que implementar un módulo con muchas partes irrelevantes para el usuario debido a que todo estaría muy cohesionado si no se realizara esta división de la que estamos hablando.

Los módulos principales que van a definirse son los siguientes:

- Módulos de búsqueda de vulnerabilidades o de seguridad (BOAM: *BOA Module*): los módulos asociados a la búsqueda de vulnerabilidades son los más importantes, y dichos módulos estarán clasificados según la finalidad de, como se acabo de comentar, la búsqueda de vulnerabilidades. El comportamiento que se definirá en este módulo general estará relacionado con la comunicación a realizar con los módulos concretos o la ejecución de los mismos o ofrecer una interfaz común a todos los módulos relacionados con la búsqueda de vulnerabilidades.

- Módulos de análisis del lenguaje de programación o interoperabilidad (BOAPM: *BOA Parser Module*): la independencia con el lenguaje de programación es uno de los objetivos, el cual lo conseguimos a través de los módulos que se encargan de implementar el comportamiento necesario para la interoperabilidad. Estos módulos estarán clasificados en el módulo general que se clasifica en relación a los analizadores de los lenguajes de programación, y cuyo comportamiento será aportar una interfaz común a todos los módulos cuya tarea será resolver los problemas de interoperabilidad con un analizador concreto para un lenguaje de programación.

- Módulos de ciclo de vida (BOALC: *BOA LifeCycle*): uno de los objetivos de diseño es la automatización, y para ello se comentó en la subsección Automatización que habría que definir una serie de fases. Dichas fases, en su conjunto, vamos a llamarlas un ciclo de vida. Con el fin de reforzar la flexibilidad, la personalización y la extensibilidad, se va a permitir a los usuarios definir sus propios ciclos de vida, los cuales BOA va a ejecutar con el fin de conseguir la automatización. Crear un ciclo de vida puede llevar a conseguir varios objetivos, como puede ser cambiar el nombre de las fases a ejecutar, cambiar el número de veces que cada fase se ejecute o cambiar los datos que cada fase recibe, por lo que crear un nuevo ciclo de vida puede permitir no solo personalizar con el fin de conseguir una mejor semántica en

función del objetivo de las fases, sino también conseguir resultados mucho más elaborados (e.g. definir una fase de procesamiento que se ejecute múltiples veces para refinar el resultado, definir una fase de preprocesado con el objetivo de descartar datos erróneos o de baja calidad).

- Módulos de informe (BOAR: *BOA Report*): el informe generado por BOA es una pieza esencial del análisis, ya que son los resultados. La manera en la que dichos resultados se muestren puede aportar al usuario una gran ayuda o no ayudar para nada, ya sea porque los resultados sean difíciles de leer, no tiene la información que el usuario necesita, etc. A través de la definición de este módulo general, se podrá extender el comportamiento de los informes para añadir más información, obtener otros formatos o realizar la acción que el usuario considere oportuna.
- Módulos de severidad: cualquier vulnerabilidad puede clasificarse en función de la severidad con la que puede afectar a diferentes factores (e.g. sistema, reputación). Debido a ello, los módulos que extiendan el módulo principal de severidad podrán definir sus propios niveles de severidad que podrán emplearse tanto en los módulos BOAM, para asignar un nivel de severidad a las vulnerabilidades localizadas, como en los módulos BOAR, para mostrar los niveles de severidad de las vulnerabilidades reportadas.

6.7 Arquitectura *Software*

La arquitectura *software* empleada por BOA intenta ilustrar todo lo que se ha comentado en el presente capítulo, llegando a un resultado. La manera en la que la arquitectura se ha definido es lo que ha permitido cumplir de manera satisfactoria con los objetivos de diseño establecidos. En la figura 6.3 se puede observar la arquitectura de BOA, la cual está muy influenciada en el *software* Metasploit Framework[1] [42] y la razón de ello es que los objetivos están bastante relacionados y el funcionamiento es muy similar en aspectos generales.

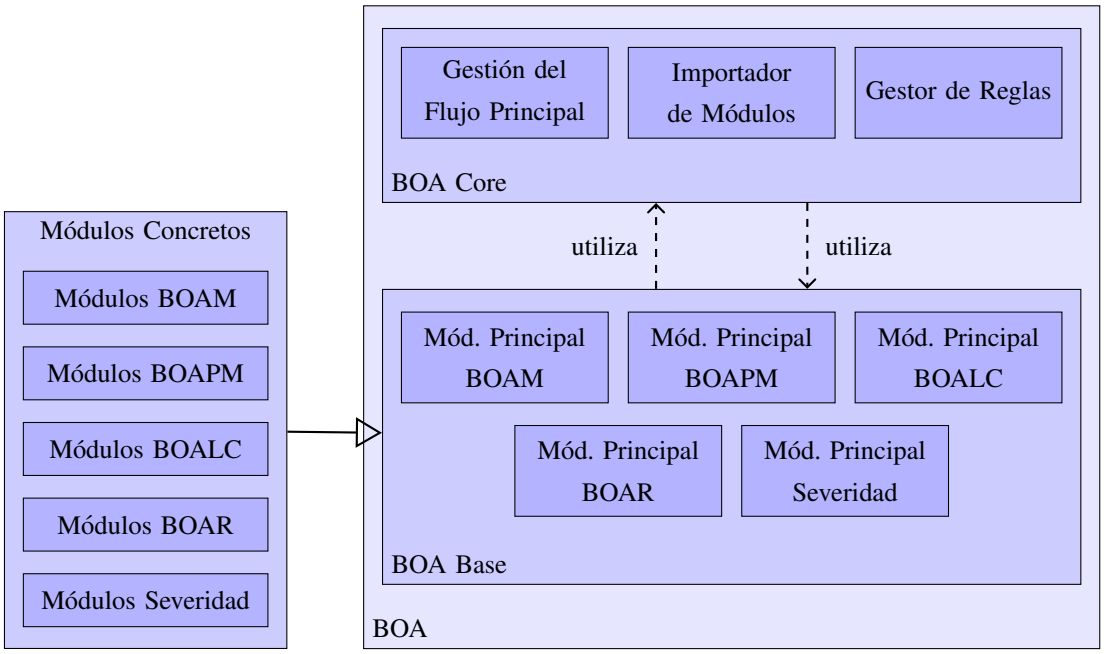


Figura 6.3: Arquitectura *Software* de BOA (notación UML)

[1] Sitio oficial de Metasploit: `https://www.metasploit.com/`

Los elementos de la arquitectura de BOA (figura 6.3) van a ser explicados a continuación:

- BOA *Core*: núcleo de BOA con los elementos más relevantes que hacen que el analizador sea capaz de funcionar. Son los elementos primordiales.
  - Gestión del Flujo Principal: módulo encargado de realizar las tareas básicas de BOA con la finalidad de que todo el flujo principal del analizador sea gestionado de manera correcta.
  - Importador de Módulos: módulo encargado de importar otros módulos. Se encargará de realizar las tareas necesarias para que los diferentes tipos de módulos sean importados correctamente o, si no es posible, reportar los errores.
  - Gestor de Reglas: módulo encargado de analizar el archivo de reglas e interpretarlo en los casos que sea necesario.
- BOA *Base*: elementos que, apoyándose en el núcleo de BOA (i.e. BOA *Core*), dotan de las interfaces necesarias para implementar las diferentes funcionalidades.
  - Módulo Principal BOAM: interfaz o similar que permite definir unas propiedades comunes para todos los módulos BOAM.
  - Módulo Principal BOAPM: interfaz o similar que permite definir unas propiedades comunes para todos los módulos BOAPM.
  - Módulo Principal BOALC: interfaz o similar que permite definir unas propiedades comunes para todos los módulos BOALC.
  - Módulo Principal BOAR: interfaz o similar que permite definir unas propiedades comunes para todos los módulos BOAR.
  - Módulo Principal de Severidad: interfaz o similar que permite definir unas propiedades comunes para todos los módulos de severidad.
- Módulos Concretos: se trata de las implementaciones concretas de elementos de BOA *Base*. Un módulo concreto contendrá el comportamiento de una funcionalidad, dependiendo de la interfaz de la que dependa.
  - Módulos BOAM: módulos que implementan el comportamiento necesario para la búsqueda y/o detección de vulnerabilidades.
  - Módulos BOAPM: módulos que implementan el comportamiento necesario para la interoperabilidad con un analizador de un lenguaje de programación concreto.
  - Módulos BOALC: módulos que implementan el comportamiento necesario para la definición de los ciclos de vida, los cuales definen las diferentes fases, o etapas, que BOA ejecutará a lo largo del análisis de un módulo BOAM.
  - Módulos BOAR: módulos que implementan el comportamiento necesario para obtener un informe del análisis resultante realizado por BOA.
  - Módulos de Severidad: módulos que implementan el comportamiento necesario para la definición de diferentes niveles de severidad que podrán ser empleados en diferentes módulos BOAM.

6.8 Limitaciones

BOA es un analizador de vulnerabilidades de propósito general en el que los usuarios podrán hacer uso de módulos propios. Debido a esto, las limitaciones de BOA estarán marcadas por dichos módulos, ya que se van a proporcionar todos los medios posibles al usuario para poder realizar todas las tareas necesarias. Aún así, sigue habiendo limitaciones, por supuesto, y la principal limitación afecta al análisis de *software* y es la que vamos a discutir a continuación: el **teorema de Rice** [43].

Teorema (Teorema de Rice)**.** *Para cualquier propiedad no trivial de una función parcial, ningún método general ni efectivo puede decidir si un algoritmo puede computar una función parcial con dicha propiedad.*

Teorema (Teorema de Rice, adaptado a la teoría de la computabilidad)**.**
Dado un conjunto S de lenguajes no triviales,

1. *Existe una Máquina de Turing que reconoce un lenguaje en S,*
2. *Existe una Máquina de Turing que reconoce un lenguaje no en S.*

Entonces es indecidible determinar si el lenguaje reconocido por una Máquina de Turing arbitraría pertenece a S.

El teorema de Rice habla de la indecibilidad de las propiedades no triviales (propiedades semánticas) de un lenguaje. Lo que nos quiere decir es que no hay un método general, computable, que nos de un resultado para un predicado (i.e. función que nos devuelve cierto o falso) respecto de una propiedad semántica de un lenguaje (e.g. bucle infinito) en una cantidad de tiempo finito. Se trata de una generalización del clásico problema de la parada en la teoría de la computación.

En el caso de los analizadores de vulnerabilidades, como es el caso de BOA, el predicado que intentan resolver es el siguiente: ¿es vulnerable el elemento (e.g. fichero, sistema, función) bajo análisis? En el mejor de los casos, el mejor de los analizadores, cumpliría con las siguientes propiedades para siempre resolver de manera correcta el predicado citado:

- Robustez (*Soundness*): obtiene todas las vulnerabilidades (i.e. no tiene falsos negativos).
- Completitud (*Completeness*): no obtiene vulnerabilidades que en realidad no lo son (i.e. no tiene falsos positivos).
- Siempre Termina: necesitamos que siempre nos de una respuesta en un tiempo finito, ya que en otro caso, no nos serviría de nada el análisis.

Las características anteriores, en conjunto, no se pueden cumplir por el teorema de Rice, ya que entonces, el analizador perfecto existiría. Aún así, tenemos diferentes problemas para cumplir con cada uno de los puntos anteriores cuando formamos parejas, ya que entonces la propiedad que no se cumpliría sería la tercera que dejamos fuera de la pareja. Por el teorema de Rice, sabemos que no es computable obtener un método general (i.e. respuesta) a las propiedades semánticas no triviales de un lenguaje, por lo que no será posible realizar a la perfección la unión de los puntos anteriores (i.e. no se cumplirán todos los puntos), y por parejas surgen problemas:

- Robustez y Completitud: un analizador que cumpla con estas propiedades (i.e. no tiene falsos positivos ni falsos negativos) sería un analizador casi perfecto, pero el problema surge en que no sabemos si el analizador se detendrá en algún momento. No es que no se vaya a detener, es que no lo sabemos, porque es un problema indecidible.
- Robustez y Siempre Termina: un analizador con estas propiedades (i.e. obtiene todas las vulnerabilidades y termina) sería un analizador casi perfecto, pero el problema surge en que no podremos fiarnos de los resultados, ya que no sabremos si hay resultados que no son vulnerabilidades (i.e. falsos positivos).
- Completitud y Siempre Termina: un analizador con estas propiedades (i.e. no se equivoca en la detección de vulnerabilidades y termina) sería un buen analizador, pero el problema surge en que no podremos fiarnos de los resultados, ya que no sabremos cuantas vulnerabilidades no hemos detectado (i.e. falsos negativos).

Debido a las limitaciones comentadas a raíz del teorema de Rice, lo que se hace son aproximaciones, ya que las aproximaciones sí que son decidibles. Lo normal en un analizador será tener más robustez o más completitud, en equilibrio, pero también hay los que apuestan por uno u otro (e.g. los compiladores suelen aplicar técnicas de análisis de flujo de datos, las cuales cumplen en gran medida la característica de robustez).

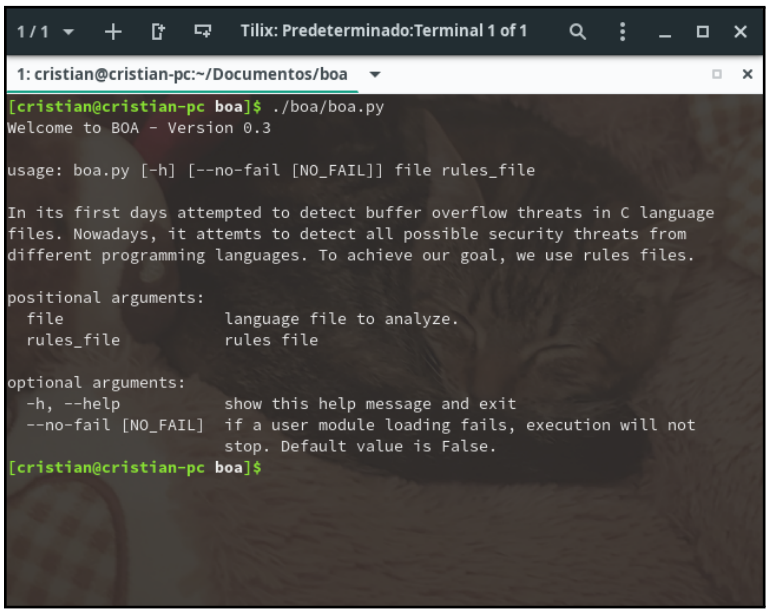
El nivel en el que se cumplan las propiedades discutidas en la presente sección en BOA dependerá de las implementaciones concretas de los módulos de los usuarios.

Capítulo 7

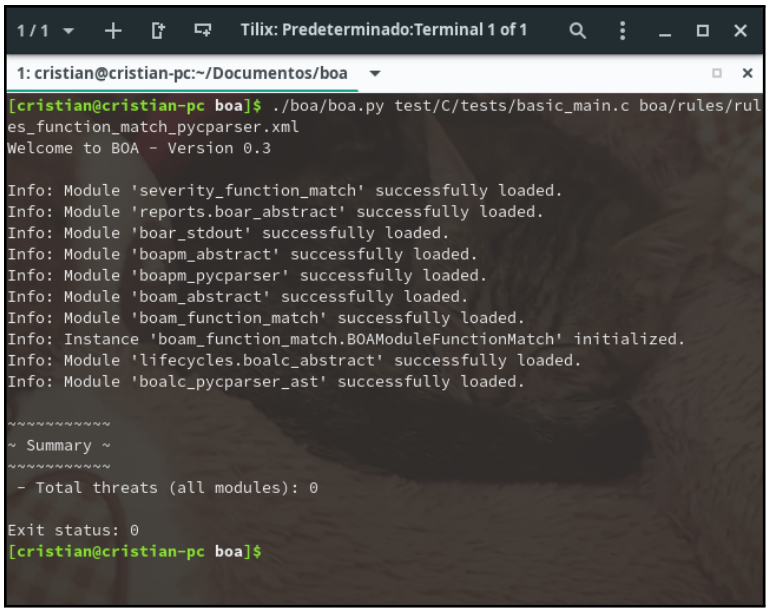
Implementación

7.1 Introducción

Junto al diseño, la implementación es el otro gran pilar discutido en este trabajo. La implementación responde al *¿cómo?* a la hora de desarrollar un proyecto *software*, y eso es lo que vamos a explicar en el presente capítulo desarrollando los procedimientos seguidos para el desarrollo de BOA. Las decisiones de implementación están basadas en los puntos comentados en el capítulo Diseño. Un ejemplo del resultado del presente capítulo se puede observar en la figura 7.1, donde vemos un ejemplo de la ejecución de BOA.



(a) Ejecución de BOA (menú de ayuda)



(b) Ejecución de BOA (ejecución de módulo)

Figura 7.1: Ejemplo de ejecución de BOA

7.1.1 Estructura del Proyecto

La estructura del proyecto consta de una serie de directorios y ficheros principales. Dichos directorios y ficheros principales son los siguientes:

- Directorio *boa/*: directorio que contiene la implementación de BOA y sus respectivos módulos.
- Directorio *boa/docs/*: directorio que contiene la documentación de BOA. Se detalla en la subsección Documentación del presente capítulo.
- Directorio *boa/rules/*: directorio que contiene los diferentes archivos de reglas, aunque no es necesario almacenarlos en este directorio.
- Directorio *boa/modules/*: directorio que contiene la implementación de los módulos relacionados con la búsqueda de vulnerabilidades, además de otros módulos que se encargan de la gestión.
- Directorio *boa/parser_modules/*: directorio que contiene la implementación de los módulos relacionados con la interoperabilidad con analizadores de los lenguajes de programación, además de otros módulos que se encargan de la gestión.
- Directorio *boa/lifecycles/*: directorio que contiene la implementación de los módulos relacionados con los ciclos de vida, además de otros módulos que se encargan de la gestión.
- Directorio *boa/reports/*: directorio que contiene la implementación de los módulos relacionados con la generación de informes, además de otros módulos que se encargan de la gestión.
- Directorio *boa/enumerations/*: directorio que contiene ficheros con enumeraciones que se pueden utilizar por el resto de módulos de BOA.

- Directorio *boa/enumerations/severity/*: directorio que contiene la definición de los niveles de severidad a emplear por los módulos de seguridad. Son enumeraciones, ya que esto nos ofrece la ventaja de poder ordenar los niveles de severidad de más grave a menos e incluso el usuario tiene la opción de realizar el orden que quiera bajo cualquier criterio, no solo el de severidad. El resultado es el orden resultado de las vulnerabilidades reportadas en el informe de BOA.

- Directorio *boa/auxiliary_modules/*: directorio que contiene módulos auxiliares (e.g. utilidades para un analizador de un lenguaje de programación concreto) que pueden emplearse por cualquier módulo de BOA. En este directorio se deberían de almacenar los módulos que un usuario necesite como apoyo.

- Directorio *test/*: directorio que contiene directorios que contienen pruebas. La estructura de directorios que se debería de utilizar dentro de este directorio es un directorio *BOA*, el cual existe y contiene pruebas propias del analizador, y un directorio por lenguaje de programación (e.g. *test/C/*) con la finalidad de poder ser utilizado por BOA para las pruebas. En los directorios de los lenguajes de programación también se debería de hacer la distinción entre pruebas sintéticas y pruebas de sistemas reales (e.g. *test/C/tests/* para pruebas sintéticas y *test/C/real_tests/* para pruebas sobre sistemas reales).

- Fichero *boa/boa.py*: fichero principal que inicia la ejecución de BOA y que realiza las llamadas a todas las subrutinas encargadas de efectuar el análisis de manera que todos los elementos definidos hasta el momento se ejecuten.

- Fichero *boa/boa_internals.py*: fichero que contiene métodos auxiliares utilizados por el fichero *boa.py* para realizar diferentes tareas específicas propias del analizador y que no pueden considerarse métodos generales porque la finalidad es demasiado concreta.

- Fichero *boa/args_manager.py*: fichero que realiza las comprobaciones necesarias y el procesamiento adecuado sobre los argumentos de entrada cuando se ejecuta BOA (i.e. *boa.py* ***args***).

- Fichero *boa/rules_manager.py*: fichero que se encarga de analizar el fichero de reglas. Además del análisis, también detecta una serie de errores semánticos del fichero de reglas (e.g. misma dependencia para un módulo definido múltiples veces) y realiza un procesamiento mínimo necesario para otros módulos (e.g. separación de los argumentos por módulo).

- Fichero *boa/modules_importer.py*: fichero que se encarga de importar los módulos de manera correcta. En el caso de no poder importar algún módulo, la ejecución se interrumpirá o no dependiendo de la elección del usuario.

- Fichero *boa/constants.py*: fichero que contiene constantes utilizadas por el resto de ficheros/módulos de BOA. Contiene información general de BOA (e.g. formato de sus argumentos), códigos de errores, etc.

- Fichero *boa/own_exceptions.py*: fichero que contiene la definición de excepciones propias que se utilizan por el resto de ficheros/módulos de BOA. Dichas excepciones están implementadas con el objetivo de ofrecer una información de errores detalladas y localizable.

- Fichero *boa/util.py*: fichero que contiene utilidades generales utilizadas por el resto de ficheros/módulos de BOA (e.g. la función *do_nothing(*args, rtn_sth=False, rtn_value=None)* es útil cuando necesitamos pasar un argumento que pide un *callback* y lo que queremos cuando dicho *callback* se ejecute es no hacer nada).

7.1.2 Documentación

La documentación de BOA se ha construido a partir de Sphinx. Sphinx utiliza los *docstring* de los ficheros de Python para la generación de la documentación, y BOA ha completado los *docstring* de todos los ficheros en todos los ámbitos que Sphinx admite (módulo, clase y función) siguiendo el format *Google Docstrings*. En el directorio *boa/docs/* tenemos los directorios *source/* y *build/*, los cuales contienen los ficheros con extensión ”rst”, los cuales contienen código *reStructuredText*, y el resultado de la compilación de la documentación, respectivamente.

Cuando se crea un proyecto con Sphinx, este nos proporciona un fichero *Makefile*, el cual nos sirve para utilizar la utilidad ”make”, cuyo resultado es la compilación de la documentación en el formato que se quiera y se almacena, en nuestro caso, en el directorio *build*. El fichero *Makefile* define varias instrucciones cuyo nombre coincide con un nombre descriptivo del formato de salida, por lo que si queremos la documentación en formato HTML, solo tendremos que hacer ”make html” en el directorio *boa/docs/*.

La estructura que BOA tiene en su directorio *boa/docs/source/* es la siguiente:

- Directorio *modules/*: directorio que contiene la documentación de los módulos de BOA.
- Directorio *modules/main_modules/*: contiene los ficheros ”rst” con la documentación de los ficheros de código contenidos en el directorio *boa/*.
- Directorio *modules/main_modules/severity_enums/*: debido a que la documentación de los diferentes niveles de severidad se distancia mucho de los otros módulos principales en relevancia, se han incluido los ficheros de documentación de los mismos dentro del directorio *modules/main_modules/*. Este directorio contiene los ficheros ”rst” con la documentación de los ficheros de código contenidos en el directorio *boa/enumerations/severity/*.
- Directorio *modules/main_modules/auxiliary_modules/*: contiene los ficheros ”rst” con la documentación de los ficheros de código contenidos en el directorio *boa/auxiliary_modules/*.
- Directorio *modules/sec_modules/*: contiene los ficheros ”rst” con la documentación de los ficheros de código contenidos en el directorio *boa/modules/*.
- Directorio *modules/parser_modules/*: contiene los ficheros ”rst” con la documentación de los ficheros de código contenidos en el directorio *boa/parser_modules/*.
- Directorio *modules/lifecycles/*: contiene los ficheros ”rst” con la documentación de los ficheros de código contenidos en el directorio *boa/lifecycles/*.
- Fichero *index.rst*: fichero de documentación inicial donde se muestra una descripción de BOA y ayuda a la navegación a través de la documentación. Un ejemplo del resultado de este fichero de documentación se puede encontrar en la figura 7.2.
- Fichero *footer.rst*: contiene enlaces a los módulos más relevantes de la documentación, o secciones. Este fichero de documentación se incluye en todos los ficheros ”rst” de la documentación de BOA con el fin de que sirva como pie de página y facilite la navegación a través de la documentación.
- Fichero *changelog.rst*: fichero que incluye los cambios a lo largo de las diferentes versiones de BOA. Cada versión incluye una sección de cambios y otra de problemas solucionados, pero para una versión

concreta puede solo estar presente una de las dos secciones. Los cambios son referentes al analizador, no a los módulos concretos de BOA (cambios referentes a módulos pueden ser documentados en su propio fichero ”rst”). Un ejemplo del resultado de este fichero de documentación se puede encontrar en la figura 7.3.

- Fichero *todo.rst*: fichero que incluye posibles características a implementar sobre BOA, las cuales no se garantiza que vayan a ser implementadas, y, en el caso de al final haberse llevado a cabo, se incluye la versión en la que se introdujo la característica.

- Fichero *modules/main_modules.rst*: sirve a modo de índice para los módulos principales y añade una descripción inicial general acerca de los mismos. Cada vez que se añaden nuevos módulos en el directorio de documentación *modules/main_modules/*, hay que actualizar el *(*toctree) (estructura de datos interna utilizada por Sphinx para representar los nodos de la documentación) y los enlaces para que el índice esté actualizado. Este fichero es una ayuda para la navegación en la documentación.

- Fichero *modules/sec_modules.rst*: sirve a modo de índice para los módulos relacionados con la búsqueda de vulnerabilidades (módulos de seguridad) y añade una descripción inicial general acerca de los mismos. Cada vez que se añaden nuevos módulos en el directorio de documentación *modules/sec_modules/*, hay que actualizar el *(*toctree) (estructura de datos interna utilizada por Sphinx para representar los nodos de la documentación) y los enlaces para que el índice esté actualizado. Este fichero es una ayuda para la navegación en la documentación.

- Fichero *modules/parser_modules.rst*: sirve a modo de índice para los módulos relacionados con la gestión de la interoperabilidad entre un analizador de un lenguaje de programación concreto y BOA y añade una descripción inicial general acerca de los mismos. Cada vez que se añaden nuevos módulos en el directorio de documentación *modules/parser_modules/*, hay que actualizar el *(*toctree) (estructura de datos interna utilizada por Sphinx para representar los nodos de la documentación) y los enlaces para que el índice esté actualizado. Este fichero es una ayuda para la navegación en la documentación.

- Fichero *modules/lifecycles.rst*: sirve a modo de índice para los módulos que definen ciclos de vida y añade una descripción inicial general acerca de los mismos. Cada vez que se añaden nuevos módulos en el directorio de documentación *modules/lifecycles/*, hay que actualizar el *(*toctree) (estructura de datos interna utilizada por Sphinx para representar los nodos de la documentación) y los enlaces para que el índice esté actualizado. Este fichero es una ayuda para la navegación en la documentación.

- Fichero *modules/reports.rst*: sirve a modo de índice para los módulos relacionados con el informe que BOA genera y añade una descripción inicial general acerca de los mismos. Cada vez que se añaden nuevos módulos en el directorio de documentación *modules/reports/*, hay que actualizar el *(*toctree) (estructura de datos interna utilizada por Sphinx para representar los nodos de la documentación) y los enlaces para que el índice esté actualizado. Este fichero es una ayuda para la navegación en la documentación.

Se puede observar el resultado de la generación de la documentación en las figuras 7.2 y 7.3 (formato HTML).

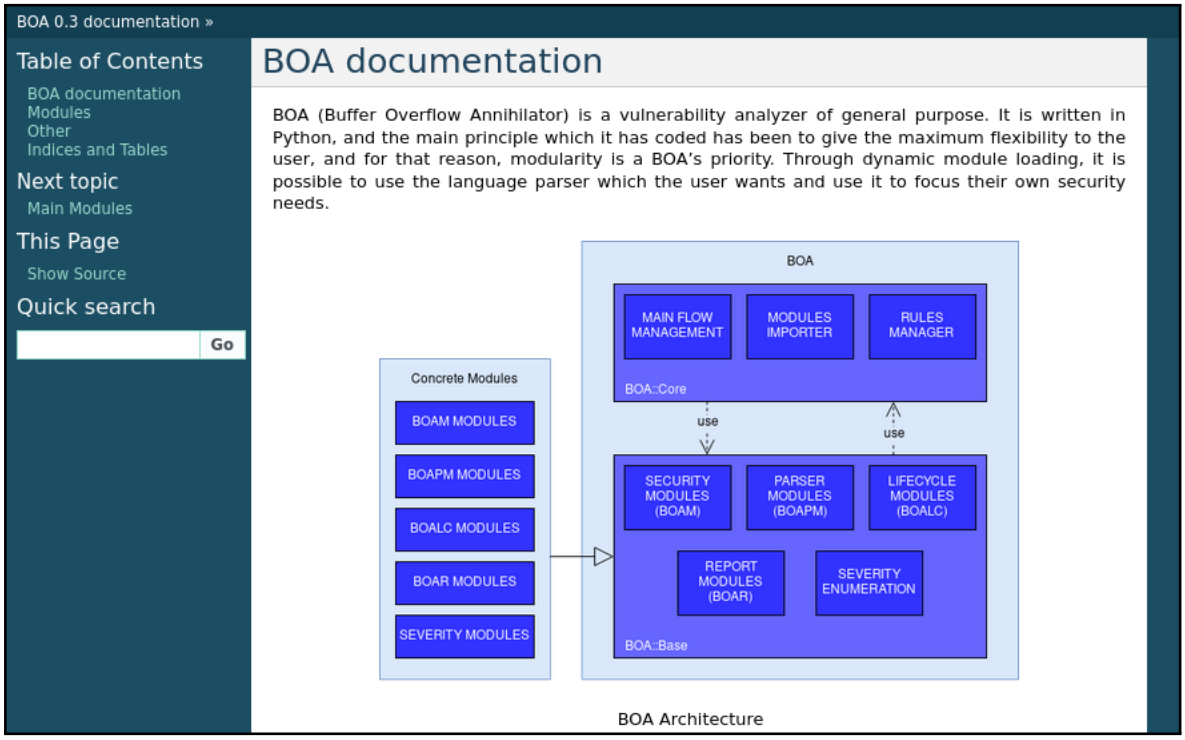


Figura 7.2: Documentación de BOA con Sphinx (pantalla principal)

Changelog

You will find here the main changes from one version to others.

Version 0.3

Dependencies among modules are possible.

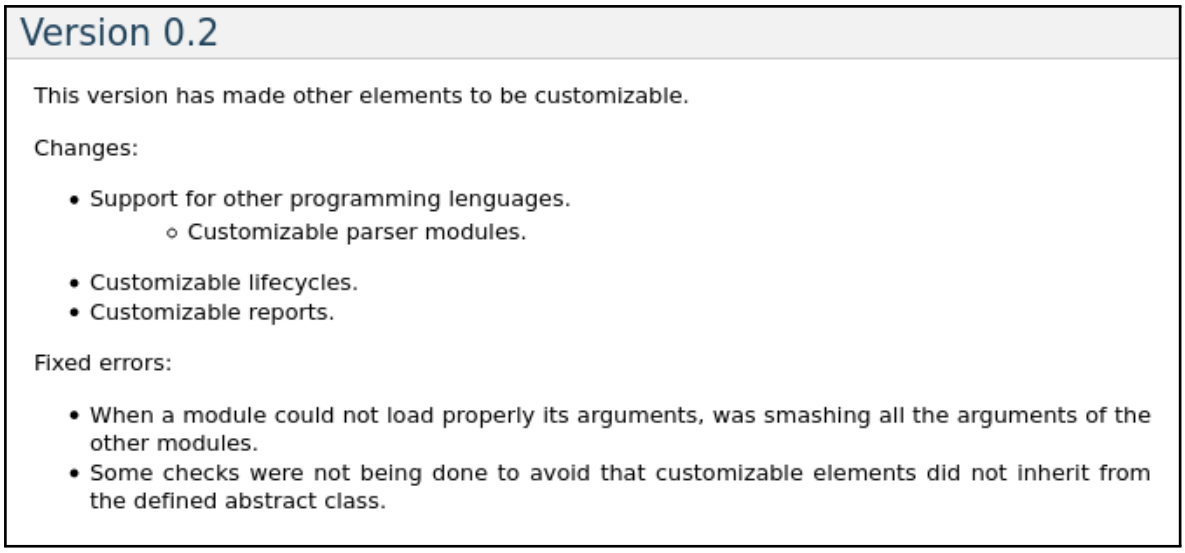
Changes:

- The results of a module can be a dependency for others.

Fixed errors:

- Main argument "--no-fail" was not working as expected.
- When a module loading failed and the execution continued, was not being correctly removed.
- Minnor fixes.

(a) Historial de cambios de BOA (versión 0.3)



Version 0.2

This version has made other elements to be customizable.

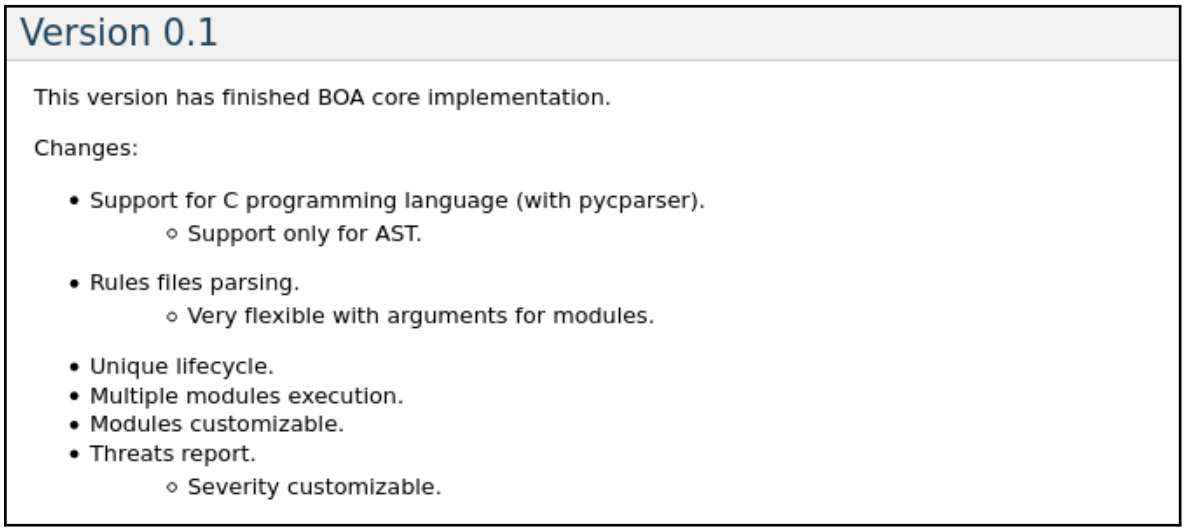
Changes:

- Support for other programming lenguages.
  - Customizable parser modules.
- Customizable lifecycles.
- Customizable reports.

Fixed errors:

- When a module could not load properly its arguments, was smashing all the arguments of the other modules.
- Some checks were not being done to avoid that customizable elements did not inherit from the defined abstract class.

(b) Historial de cambios de BOA (versión 0.2)



Version 0.1

This version has finished BOA core implementation.

Changes:

- Support for C programming language (with pycparser).
  - Support only for AST.
- Rules files parsing.
  - Very flexible with arguments for modules.
- Unique lifecycle.
- Multiple modules execution.
- Modules customizable.
- Threats report.
  - Severity customizable.

(c) Historial de cambios de BOA (versión 0.1)

Figura 7.3: Documentación de BOA con Sphinx (historial de cambios)

7.2 Flujo Principal de Ejecución

El flujo principal de BOA está definido en el fichero *boa/boa.py*, y realiza una serie de tareas que son relevantes para la correcta ejecución del analizador. Esta serie de tareas son la manera en la que todo el análisis realizado por BOA es realizado desde un punto de vista general. A continuación, se explicarán dichas tareas de manera ordenada conforme se ejecutan en BOA, siendo el objetivo comprender de manera general cómo se realiza todo el análisis.

7.2.1 Argumentos de Entrada

Cuando BOA se inicia para realizar un análisis de vulnerabilidades, la primera de las tareas que realiza es analizar los argumentos de entrada obtenidos a través de la llamada al mismo. Este análisis es realizado por la clase *ArgsManager*, la cual está contenida en el fichero *boa/args_manager.py*, y su finalidad es, como ya se ha dicho, analizar los argumentos de entrada. Además, también tendrá contenidos dichos argumentos, por lo que se tendrá que acceder con la instancia de la clase con la que se ha realizado el análisis de los argumentos para poder acceder a los valores concretos.

La manera en la que el análisis de los argumentos de entrada está implementado es a través de la librería nativa de Python argparse, la cual ofrece todo lo necesario para un análisis correcto de los argumentos de entrada. La manera en la que se trabaja con los argumentos es con el fichero que contiene las constantes de BOA (i.e. fichero *boa/constants.py*), en el cual se definen los argumentos de entrada válidos, si son obligatorios u opcionales, si son de tipo booleano y una descripción de los mismos. De esta manera, la ampliación de los argumentos válidos es muy sencilla, y luego acceder al valor de los mismos también lo es siempre que se tenga acceso a la instancia de la clase *ArgsManager* que realizó el análisis. Los parámetros están almacenamos en la variable de instancia llamada *args*.

Algo relevante a comentar es que en esta tarea inicial, que BOA realiza, es donde se implementa el mecanismo que permite al usuario decidir si la ejecución se detiene cuando la carga de un módulo no termina correctamente, entre otros, y en concreto ese mecanismo se activa a través del argumento de entrada "*–no-fail*", el cual acepta los valores "*true*" o "*false*" siendo este último el valor por defecto. Esto ya se comento anteriormente, en el capítulo Diseño, y su finalidad es aportar al usuario una mayor flexibilidad.

7.2.2 Archivo de Reglas

El archivo de reglas es un elemento esencial en BOA. La implementación del módulo que se encarga de su análisis, comprobación y procesamiento está en el fichero *boa/rules_manager.py*, y el procedimiento que se realiza al iniciar un nuevo análisis con BOA es el siguiente:

1. Abrir archivo de reglas: el primer paso a realizar es abrir el propio fichero de reglas, y se realiza gracias a que la ruta del mismo se recibe a través de un parámetro de BOA.
2. Leer archivo de reglas: una vez abierto el archivo de reglas, lo que tenemos que hacer es leer su contenido. El contenido del archivo de reglas se almacena en su totalidad sin procesar y luego se utiliza la utilidad xmltodict para obtener una representación en un diccionario de Python.
3. Cerrar archivo de reglas: una vez leído el contenido solo nos queda cerrar el fichero para liberar el descriptor.

Una vez realizados los pasos iniciales, ya tenemos el contenido del archivo de reglas listo para ser procesado y obtener la información relevante del mismo. Es cierto que hay información que no es necesario procesar, ya que teniendo acceso a la misma, no necesitamos más que el valor contenido. Aún así, en este módulo se hacen todas las comprobaciones necesarias para saber que todos los elementos presentes en el fichero de reglas coinciden con los objetivos de diseño del mismo (e.g. reglas obligatorias definidas en el archivo, reglas no definidas no están permitidas).

Todo el proceso que conlleva el análisis del archivo de reglas se inicia una vez se llama a la función *check_rules()*, la cual realiza las siguientes tareas:

- Comprobaciones iniciales: se comprueba si las etiquetas principales están definidas y si contienen el número de elementos hijos que pueden tener (hay casos en los que se trata de un valor concreto y otros en los que se trata de un rango debido a las etiquetas opcionales).
- Comprobaciones relativas a la etiqueta *boa_rules.parser*: se comprueba si los elementos contenidos en la etiqueta *boa_rules.parser* son los esperados y se comprueba que no haya ningún elemento no permitido.
- Comprobaciones relativas a la etiqueta *boa_rules.modules*: se realizan múltiples comprobaciones relativas a la etiqueta *boa_rules.modules* además de múltiples procesamientos. Entre las tareas que se realizan están las de comprobar que los elementos contenidos son los esperados, se procesan las reglas de los módulos (i.e. etiquetas *modules.module*) y, para cada módulo, sus argumentos (i.e. etiqueta *module.args*), ciclo de vida a ejecutar (se asigna uno por defecto si no se aplica ningún otro), severidad a aplicar (se asigna una por defecto si no se aplica ninguna otra) y dependencias (i.e. etiqueta *module.dependencies*). Se detalla en las posteriores subsecciones del presente capítulo.
- Comprobaciones relativas a la etiqueta *boa_rules.report*: se comprueba si los elementos contenidos en la etiqueta *boa_rules.parser* son los esperados y se comprueba que no haya ningún elemento no permitido. Además, se realiza también un procesamiento de la etiqueta *report.args*, el cual se detalla en la subsección Argumentos del presente capítulo.

Durante el procesamiento del archivo de reglas se hace uso de unas excepciones propias que ayudan a la detección de errores del archivo de reglas. El objetivo de utilizar excepciones propias es utilizar un enfoque de código que no mezcle código con comprobaciones de errores, lo cual es conocido como código espagueti, y definir excepción por cada tipo de error posible en el análisis del archivo de reglas (los comportamientos definidos que lanzan excepciones son no cumplir con el formato especificado para alguna regla, no haber definido alguna regla que debería de estar definida por razones sintácticas o semánticas y otros errores).

Argumentos

El procesamiento de los argumentos es una de las partes más interesantes en el análisis del archivo de reglas, ya que es es el único elemento que se permite definir de manera recursiva (i.e. definir tantos ámbitos como se quiera a través de los elementos *args.dict*, *args.list* y *args.element*). Esto es aplicable a los elementos que tienen argumentos, los cuales son *modules.module* y *boa_rules.report*, procedimiento el cual es idéntico en ambos casos, ya que se ejecuta el mismo procedimiento, el cual se encuentra en la función *check_rules_arg_high_level()*.

Las tareas que se realiza a la hora de analizar, comprobar y procesar los argumentos son los siguientes:

1. Se comprueba si se ha especificado la etiqueta *args_sorting* con el valor ”true”, lo cual significa que se quiere procesar los argumentos de manera ordenada. Se detalla en la subsección Orden Parcial de los Argumentos del presente capítulo.
2. Se recorren los elementos contenidos en los argumentos y se procesan. En este momento, los argumentos que se estarán procesando serán los del ámbito más general, los cuales serán los del elemento *module.args*

o *report.args*, elemento el cual solo debería de contener un elemento hijo de tipo *dict* como se definió en el capítulo Diseño.

a) Se procesa cada argumento de manera individual. Esto se realizará, en concreto, ejecutando la función *check_rules_arg()*. Si algún argumento no es correcto, ya sea sintáctica o semánticamente, se avisará al usuario de que los argumentos son incorrectos a través de un mensaje descriptivo del problema.

3. Se comprueba para el elemento actual si contiene valor. Esto es así ya que, como vamos a ver en pasos posteriores, a partir de este momento se está realizando ya una ejecución recursiva, y esta es nuestra condición de parada.

4. Se procesa cada argumento de manera individual.

a) Se comprueba si estamos ante la primera llamada, la cual se realiza desde el elemento *module.args* o *report.args*, y se realiza un procesamiento no genérico por la restricción asociada que ya se ha comentado.

b) Se comprueba si el argumento es de tipo *dict*, *list* o *element*, y para cualquiera de ellos, se realizan las comprobaciones oportunas a través de la función *check_rules_arg_recursive()*.

c) A través de la función *check_rules_arg_recursive()* se realizan unas comprobaciones básicas de los atributos que deberían de tener los elementos, ya sea *dict*, *list* o *element*. Una vez hecho esto, se realiza la llamada a la función *check_rules_arg()* para el siguiente ámbito, llegando al punto en el que se observa una doble llamada recursiva, y esto es justamente lo que nos permite, con el mismo comportamiento, analizar los argumentos de cada uno de los ámbitos, donde en cada llamada descendemos un nivel, o un ámbito.

d) Por último, para un ámbito concreto se realiza una serie de comprobaciones en base al elemento padre de un argumento. Esto es relevante, ya que nos asegura que la coherencia se mantendrá en base a lo especificado en el diseño del archivo de reglas.

Las tareas anteriores son los puntos relevantes para entender cómo BOA realiza el análisis de los argumentos, ya que es una de las partes más relevantes del mismo. Aún así, no se ha explicado en profundidad, ya que hay detalles innecesarios que son propios de la implementación (e.g. ha sido necesario utilizar copias profundas de instancias, ya que se por defecto Python realiza copias superficiales y, al trabajar sobre una copia superficial, se modificaban todas sus referencias). A lo largo de la ejecución de las tareas también se estaba realizando otra tarea, la cual tiene que ver con el orden parcial de los argumentos que no se ha explicado por cambiar la manera en la que el procesamiento de los argumentos y se explicará en otra subsección del presente capítulo.

Orden Parcial de los Argumentos

Ya se ha comentado en varias ocasiones la limitación que presenta la utilidad xmltodict con respecto al orden de los elementos una vez se analiza un fichero XML. Se ha intentado sortear esta limitación realizando un procesamiento de los argumentos de manera que, conforme se analizan los argumentos, ordenamos los argumentos de manera manual. Aún así, no ha sido posible conseguir una ordenación total de los argumentos debido a las limitaciones que encontramos en la implementación de los diccionarios de Python respecto al orden de los elementos. De todos modos, la solución a la que hemos llegado puede ser útil en más de una ocasión,

la cual consigue un orden total agrupando los diferentes elementos (i.e. *dict*, *list* y *elemento*) por grupos en función del primero encontrado (comprobar subsección Orden de los Argumentos del capítulo Diseño para más detalles de la limitación comentada y ejemplos).

La manera en la que hemos implementado este mecanismo cuyo resultado es un orden parcial consta de los siguientes elementos:

- Cola de llamadas: diccionario de Python que va a contener el orden en el que se tienen que insertar los argumentos. Debido a que se trata de un diccionario, el orden que se aplica en el mismo es el orden alfabético, y conseguimos mantener dicho orden utilizando como clave de acceso al diccionario el indice de la posición del argumento, simulando una cola.
- Lista de tipos de argumentos: lista de Python que va a contener los tipos de argumentos, a partir del diccionario que contiene los argumentos, para un ámbito concreto que se esté ejecutando. La finalidad de esta lista es buscar a través del tipo de elemento su índice en esta lista para obtener su posición.
- Posición del tipo de elemento: contiene la posición del tipo del elemento (*dict*, *list* o *element*). Se utiliza para actualizar la cola de llamadas por cada argumento de un ámbito concreto.

El procedimiento se basa en diferir la inserción de argumentos hasta el final del procesamiento para un ámbito concreto, donde ya tendremos el orden parcial calculado en la cola de llamadas. En ese momento, que es cuando ya se ha terminado el procesamiento de los argumentos para un ámbito concreto pero aún no han sido procesados, es cuando recorremos la cola de llamadas y añadimos los argumentos con su orden parcial.

La razón de realizar este procedimiento de la manera en la que se hace es la de acoplarse al recorrido de los argumentos que se realiza de manera natural. Es decir, a través del recorrido que se hace de los argumentos por defecto, que están desordenados, queremos ordenarlos, y de ahí surge la problemática. La finalidad del procedimiento descrito es que, para un argumento que esté siendo analizado y que, seguramente no esté ordenado (puede que el orden definido y el orden alfabético coincidan, pero no será lo normal si se mezclan diferentes tipos de elementos *dict*, *list* y *element*), conseguir obtener cuál es su posición original, la cual no se puede obtener debido a que, al utilizar la utilidad xmltodict, ya perdimos dicha posición original, por lo que aplicamos un comportamiento voraz ordenando el elemento según el orden en el que aparezca su tipo.

7.2.3 Dependencias

El procesamiento de las dependencias entre módulos es una tarea relevante que permite obtener módulos con un comportamiento más complejo pero mejor organizado. La manera en la que las dependencias son analizadas y procesadas se realiza mientras las mismas reglas están siendo analizadas y procesadas también. En concreto, en el archivo de reglas, mientras los módulos de seguridad están siendo analizados, al llegar al elemento *module.dependencies*, es cuando empieza el análisis y procesamiento de las dependencias. Las tareas realizadas en el archivo de reglas para la correcta gestión de las dependencias son las siguientes:

1. Se comprueba que las sintaxis de los elementos contenidos en la etiqueta *module.dependencies* es correcta. En caso de que no lo sea, se avisa al usuario y se detiene el análisis.
2. Se obtiene un nombre que identifica una dependencia a través del nombre del módulo y su clase ("*module*" ."*class*"). Este formato es el utilizado para referenciar a una dependencia concreta.

3. Se comprueba la dependencia que está siendo analizada no haya sido ya definida anteriormente, lo cual se trata de una comprobación semántica debido a que se basa en una restricción no sintáctica y concreta de las dependencias. En caso de haberse definido con anterioridad, se avisa al usuario y se detiene el análisis.

4. Para los diferentes *callback* definidos en una dependencia concreta, se comprueba si se le ha dado el mismo nombre a más de un *callback*. En caso afirmativo, se le avisa al usuario de dicha condición, lo cual debería de tratarse de un error, pero no se detiene el análisis (el último *callback* con el mismo nombre será el que finalmente se aplique, ya que habrá sobreescrito el valor de los anteriores).

5. Se almacena la dependencia con sus *callback* y los nombres asociados a los mismos. La razón de almacenarse es que, posteriormente, los módulos indicados como dependencias se ejecutarán y los módulos que definan estas dependencias tendrán acceso a los *callbacks* a través de los nombres indicados.

Una vez se han procesado las dependencias en el archivo de reglas, se pasa a un postprocesamiento de las dependencias. Este postprocesamiento se aplica para cada módulo que tenga dependencias. Consiste, inicialmente, en comprobar que las dependencias existan (i.e. las dependencias definidas son módulos que tienen que cargarse en algún momento en el archivo de reglas, sin importar el orden de la definición), que un módulo no se defina a sí mismo como dependencia (i.e. esto causaría una referencia cíclica y la única manera de solucionarlo sería tener que ejecutar el mismo módulo varias veces, lo cual no está permitido para el diseño definido pero que podría ser una característica a implementar en un futuro) y construir un grafo en el que se representen todas las dependencias de todos los módulos. En el caso del grafo no se aplica para cada dependencia, sino que se construye para obtener una representación de todas las dependencias presentes en el análisis.

La parte más interesante de las dependencias es el grafo de dependencias, ya que es el que nos permite evitar problemas a la hora de empezar la ejecución de los módulos (el enfoque más común a utilizar a la hora de trabajar con dependencias es el de utilizar un grafo, el cual es empleado por multitud de herramientas). Una vez tenemos construido el grafo de dependencias, tenemos que realizar unas comprobaciones iniciales. El primer paso es inicializar las dependencias de aquellos módulos que no tienen ninguna dependencia, con la finalidad de obtener un grafo completo (no se hace referencia a grafo completo como concepto de la matemática discreta, el cual es un grafo con cada par de nodos distintos unidos por, al menos, una arista, sino como grafo que contiene todos los datos del ámbito que se está discutiendo) que indique también aquellos módulos que no tengan dependencias y podamos trabajar directamente sobre el grafo sin necesidad de acudir a otras fuentes de información. Una vez tenemos el grafo completo, solo nos queda comprobar si es cíclico (e.g. un módulo A tiene como dependencia a un módulo B, este módulo B tiene como dependencia a un módulo C, y este módulo C tiene como dependencia al módulo A), lo cual no debería de suceder. La manera en la que BOA comprueba si existen ciclos en el grafo de dependencias es utilizando un algoritmo de búsqueda de primera profundidad, lo cual significa recorrer el grafo como si de un árbol de tratara y, si en algún momento volvemos a visitar algún nodo que ya habíamos visitado antes, entonces el grafo contiene, al menos, un ciclo. Hay otros algoritmos para detectar ciclos, como puede ser el algoritmo de numeración, pero el empleado es uno de los más sencillos de implementar de manera recursiva.

Una vez ya tenemos el grafo de dependencias, y estamos seguros de que no contiene ciclos, el siguiente paso es realizar una reordenación de los módulos para que su ejecución se realice en un orden adecuado para que los módulos que contienen dependencias puedan acceder a dichas dependencias, y para ello, las dependencias tendrán que haberse resuelto antes de pasar a la ejecución de dicho módulo. Este paso es

justamente el que hace que no se pueda garantizar un orden ejecución igual al definido en el archivo de reglas. Para obtener el orden de ejecución de los módulos se realiza un recorrido en un bucle el cual solo se detendrá cuando se haya resuelto el orden de ejecución para todos los módulos. En cada iteración busca en el grafo de ejecución un módulo el cual no tenga dependencias o todas sus dependencia ya hayan sido resueltas y, por lo tanto, ya tengan asignado su orden de ejecución. De esta manera, se van añadiendo los módulos a una lista, la cual una vez se termine el bucle, contendrá todos los módulos definidos en el archivo de reglas, pero ordenados de tal manera que las dependencias se cumplen. Es justamente en este bucle donde se ve la problemática de los ciclos, y es que si hubiera un ciclo, acabarían por resolverse el orden de ejecución de todos los módulos menos el de los módulos que están en el ciclo, ya que todos los módulos que estén en el ciclo tendrán dependencias, y no todas sus dependencias estarán resueltas, formando de esta manera un bucle infinito debido a que no se cumplen las condiciones mencionadas para asociar a estos módulos un orden de ejecución. La detección de ciclos es esencial para no caer en este problema.

Una vez ya hemos obtenido el nuevo orden de ejecución, solo queda aplicarlo. Para aplicarlo, hay que cambiar de manera correcta el orden de todos los elementos que afectan a los módulos, y en concreto, por la manera en la que se ha realizado la implementación de los módulos en BOA, esto aplica a los módulos, a la clases de los módulos, a los niveles de severidad definidos para cada módulos y a los ciclos de vida definidos para cada módulo. La solución consiste en aplicar el mismo orden calculado para el grafo de dependencias a todos estos elementos.

7.2.4 Ejecución de Módulos

La ejecución de los módulos es el momento en el que BOA empieza a trabajar en su objetivo de búsqueda de vulnerabilidades. Inicialmente, se realiza la comunicación con el analizador del lenguaje de programación objetivo, con la finalidad de obtener alguna estructura de datos que nos permita trabajar de una manera más cómoda a la hora de buscar las vulnerabilidades (normalmente se trata de un AST). El resultado de ejecutar el módulo relacionado con el analizador del lenguaje de programación objetivo se obtiene a través de realizar las llamadas a los *callback* que están definidos en el archivo de reglas para el elemento *boa_rules.parser*, siendo el resultado final un diccionario donde tenemos los datos necesarios para empezar el análisis real.

Una vez ya tenemos los datos necesarios del analizador del lenguaje de programación objetivo, antes de poder proceder a la ejecución de los módulos de seguridad, tenemos que cargarlos, lo cual se realiza a través de la clase *ModulesImporter*. Esta clase se encarga de cargar y de mantener información relativa a la carga de los módulos. No solo se encarga de cargar los módulos definidos por el usuario, sino también carga una serie de módulos no opcionales que son esenciales para el correcto funcionamiento de BOA y los cuales se discutirán en la sección Clases Abstractas del presente capítulo. Algo importante a destacar es que el resultado obtenido en este momento es instancias de módulos y una instancia de la clase *ModulesImporter* que posteriormente utilizaremos. Hay que tener en cuenta que las instancias de los módulos aún no han sido cargadas.

Cuando se cargan los módulos, alguna operación puede no tener éxito por cualquier razón (e.g. no hay suficiente memoria para cargar el módulo, el nombre del módulo no es correcto), y debido a ello se le da la posibilidad al usuario de escoger si continuar o no con la ejecución (i.e. argumento de entrada "*–no-fail*"). En el caso de continuar con la ejecución cuando la carga de un módulo falla, los módulos que no han sido cargados son eliminados en todos los aspectos posibles del análisis, incluyendo eliminarlos de la instancia *ModulesImporter* para que no hayan problemas a la hora de acceder a un módulo que no ha podido ser cargado

correctamente. Una vez se haya completado, se puede continuar con el análisis sin problema.

El siguiente paso es proceder a inicializar las instancias, las cuales necesitan saber sus argumentos y sus dependencias, por lo que es el momento de utilizar los argumentos que se procesaron en el archivo de reglas y las dependencias, las cuales se comprobó que eran correctas a través del postprocesamiento comentado en este mismo capítulo. Para la inicialización se realizan todas las comprobaciones necesarias (e.g. clase existente, módulo definido a través de la interfaz correcta) y se procede a inicializar con sus argumentos y sus dependencias (por sencillez y limpieza de la interfaz, las dependencias se pasan a través de un argumento con un nombre concreto definido como una constante y se elimina cuando sus dependencias son almacenadas desde la misma instancia; en el caso de existir un argumento con el mismo nombre que el reservado para las dependencias, se le avisa al usuario de que no es posible procesar un argumento con dicho nombre debido a que está reservado para procesamiento interno de BOA). La carga de dependencias se realiza de manera que las instancias de módulos contienen un *callback* que apunta a la instancia de la dependencia, pero en este punto aún no se han ejecutado los módulos, por lo que aunque el *callback* es válido, no se tiene que efectuar la llamada aún ya que, devuelva el valor que devuelva, aún no se ha ejecutado la dependencia y el resultado no será válido, y aquí es justamente donde entra en juego la importancia de tener un orden de ejecución que permita garantizar que un módulo se ejecute después de que lo hagan sus dependencias, y este orden está garantizado por lo explicado en la sección Dependencias del presente capítulo.

Llegado a este paso, las instancias de las clases de los módulos de seguridad ya están listas para ser ejecutadas, por lo que el siguiente paso es definir una serie de fases que automaticen el procesamiento de los módulos de seguridad, y esto se realiza con los ciclos de vida. Cada módulo va a tener definido su propio ciclo de vida o uno por defecto, lo cual va a permitir que los módulos se ejecuten siguiendo un orden concreto, lo que significa que, dependiendo del ciclo de vida definido, cada módulo deberá de tener unos métodos concretos que se acoplen al ciclo de vida que estén utilizando. Esto se garantiza por razones que se explicarán en la sección Clases Abstractas del presente capítulo. Estos ciclos de vida serán ejecutados por un gestor, y el orden aplicado será el ya calculado previamente, y si no ocurre ningún problema en la ejecución de cada instancia, aquí es donde terminará la ejecución de los módulos de seguridad. En el caso de que no termine correctamente algún módulo en alguna fase concreta del ciclo de vida que esté ejecutando, se detendrá la ejecución de la fase concreta y lo que sucederá será que a partir de ese momento no se aplicará el orden normal del ciclo de vida, sino que se ejecutarán solo aquellas fases definidas como esenciales (e.g. fase que se encarga de liberar los recursos).

7.2.5 Ciclos de Vida

Una vez tenemos todas las instancias de los módulos de seguridad preparadas, el siguiente paso es conseguir que cada instancia se ejecute con el ciclo de vida adecuado, el cual será el que se haya definido para cada módulo en su archivo de reglas. Antes de pasar a esa ejecución, hay que cargar las instancias de los mismos ciclos de vida, ya que también son módulos y tienen que pasar por un procedimiento similar al descrito, pero más sencillo. El proceso de cargar las instancias de los ciclos de vida se realiza a través de la clase *ModulesImporter*, pero en lugar de realizarse paso a paso como sucede con los módulos de seguridad, lo cual aporta más control sobre la carga de los mismos al poder hacer diferentes comprobaciones en mitad de los diferentes pasos, para los ciclos de vida se realiza todo en un único procedimiento y se obtienen las instancias de todos los ciclos de vida que se vayan a utilizar.

Una vez tenemos todas las instancias de los ciclos de vida, pasamos a darle el control de la ejecución a la clase *BOALifeCycleManager*, la cual será la encargada de gestionar la ejecución de cada uno de los ciclos de vida para cada módulo de seguridad. Al inicializar la clase *BOALifeCycleManager* se realiza una asociación entre las instancias de las clases de los módulos de seguridad y el ciclo de vida que va a ejecutar cada una, entre otras tareas relacionadas con inicialización de estructuras de datos. Una vez está inicializada, el flujo principal de ejecución se le es otorgado a la instancia de la clase *BOALifeCycleManager* a través del método *handle_lifecycle()*. Este método lo que hace es, para cada asociación calculada en la inicialización entre las instancias de las clases de los módulos de seguridad y sus ciclos de vida, ejecutar dichos ciclos de vida para cada instancia.

La ejecución del ciclo de vida estará totalmente definida por el usuario, ya que ejecutar un ciclo de vida consiste en invocar a una función que estará definida por el usuario, y en esa función se tiene acceso a los métodos de los módulos de seguridad, por lo que este método debería de definir una serie de llamadas a funciones que invoquen métodos de los módulos de seguridad. En concreto, la función que debería de tener estas llamadas a los métodos de las instancias de las clases de los módulos de seguridad de los ciclos de vida es *execute_lifecycle()*. En este método se tendrá acceso a un *callback* que permitirá invocar las funciones de los módulos de seguridad por el nombre de los métodos (e.g. *callback(security_instance, "find_vulnerabilities")*). Los detalles se explican en la sección Clases Abstractas del presente capítulo.

7.2.6 Informe

El último paso consiste en mostrar el informe del análisis. Este paso es el más relevante para el usuario, ya que es donde se muestran los resultados obtenidos debido a la ejecución de los módulos. La única problemática con la que nos encontramos en este procedimiento es la de juntar todos los niveles de severidad que cada módulo tiene, ya que cada módulo tiene asociado un informe propio, que es el que realmente tiene asignado un nivel de severidad sobre el que trabaja cuando se añaden vulnerabilidades a dicho informe.

Cuando se está detectando vulnerabilidades, dichas vulnerabilidades se añaden al informe propio que cada módulo tiene. La razón de que cada módulo tenga un informe propio es la de facilitar la implementación, ya que el informe es dependiente de un nivel de severidad que aplica a la vulnerabilidades que se le van reportando desde cada módulo. Esto hace que el trabajar con los informes desde cada módulo sea muy sencillo y transparente para el usuario, pero tiene el problema de que el resultado final tiene que ser único, es decir, un único informe. Para ello, hay que resolver el problema de que cada informe puede tener, y seguramente tendrá, un nivel de severidad diferente, pero en el informe final se tienen que ver reflejadas todas las vulnerabilidades con sus niveles de severidad apropiados, los cuales son los que los usuarios emplearon a la hora de almacenar las entradas en los informes. La manera en la que BOA resuelve este problema es realizando una relación entre los módulos y la enumeración empleada para los niveles de severidad en el mismo informe que va a ser mostrado al usuario y contiene todas las entradas de todos los informes de los módulos.

Antes de mostrar los resultados del análisis, lo que se realiza es la creación del informe final, el cual tiene que, previamente, pasar por el proceso de realizar las asociaciones entre módulos y sus enumeraciones de severidad. Esta información es empleada posteriormente para almacenar las entradas de todos los informes de todos los módulos, con lo que al final tenemos una relación entre cada entrada del informe final y su enumeración de severidad, y es justamente esta relación la que nos va a permitir obtener la información correcta de cada informe de cada módulo contenida toda en un único informe. Una vez almacenadas todas las entradas

en el informe final, solo queda mostrarlas con el formato configurado, el cual se modifica en el archivo de reglas a través del elemento *boa_rules.report*.

Las entradas de las vulnerabilidades tienen una serie de datos que la describen. Por defecto, los valores que se utilizan para describir a una vulnerabilidad son un nombre característico de la vulnerabilidad, la localización de la vulnerabilidad en el fichero de código a través de su fila y columna, la instancia de la clase del módulo de seguridad que ha detectado la vulnerabilidad, una descripción de la vulnerabilidad, su nivel de severidad o riesgo y un texto descriptivo acerca de cómo solucionar la vulnerabilidad. Algunos de estos valores, los cuales son los utilizados en el comportamiento por defecto, son opcionales.

7.3 Clases Abstractas

El mecanismo a través del cual se ha implementado las interfaces que van a definir el comportamiento por defecto y la manera en la que los módulos concretos van a poder comunicarse son las clases abstractas. Las clases abstractas, propias del paradigma orientado a objeto, son la herramienta utilizada por BOA para implementar los módulos principales, interfaces, que ya se comentaron previamente en la sección Módulos Principales del capítulo Diseño. Al utilizar clases abstractas se puede definir una serie de métodos que, los módulos cuyas clases utilicen una instancia de dicha clase, tendrán que extender en comportamiento, lo cual nos da la ventaja de conocer los métodos a los cuales se puede llamar, e incluso nos da la capacidad de poder definir nuevas interfaces sobre estas para aumentar la extensibilidad del sistema.

La manera en la que las clases abstractas benefician a BOA es el conocimiento de los métodos ya definidos, lo cual permite a una serie de instancias de clases realizar las llamadas oportunas. Las clases abstractas representan los módulos principales en BOA, y estas son las siguientes:

- Clase *BOAModuleAbstract*: clase abstracta utilizada como módulo principal para los módulos dedicados a la búsqueda de vulnerabilidades (i.e. módulos de seguridad). El comportamiento a la hora de inicializar esta clase abstracta es almacenar los argumentos para el módulo de seguridad, los cuales se definieron en su sección concreta del archivo de reglas, almacenar sus dependencias a partir de los argumentos y eliminar su rastro de los argumentos, definir metainformación y definir una variable que permite la detención de la ejecución total del módulo (incluidas las fases del ciclo de vida esenciales) sin necesidad de ocasionar una excepción. Además, se definen una serie de métodos obligatorios a ser implementados por todas las clases que extiendan el comportamiento de la clase abstracta (estos métodos tendrán que ser empleados por los ciclos de vida, pero la implementación puede estar vacía y pasar a emplear los métodos propios del ciclo de vida) y una serie de propiedades.
- Clase *BOAParserModuleAbstract*: clase abstracta utilizada como módulo principal para los módulos de analizadores de lenguajes de programación, dedicados a realizar la interoperabilidad con dichos analizadores. El comportamiento a la hora de inicializar esta clase abstracta es almacenar el fichero del lenguaje de programación objetivo del análisis, la carga de variables de entorno que vayan a ser necesarias y la definición de metainformación. Además, define una serie de métodos obligatorios a ser implementados por todas las clases que extiendan el comportamiento de la clase abstracta, los cuales están relacionados con la inicialización de la interoperabilidad con un analizador de un lenguaje de programación concreto y con el análisis del fichero objetivo del análisis, es decir, proceder a la ejecución de la interoperabilidad.

- Clase *BOALifeCycleAbstract*: clase abstracta utilizada como módulo principal para los módulos que definen los ciclos de vida, dedicados a realizar la ejecución de los módulos de seguridad de manera automática y a través de una serie de métodos previamente definidos. El comportamiento a la hora de inicializar esta clase abstracta es almacenar una instancia del módulo de seguridad que va a ser ejecutado con una instancia de un ciclo de vida, una instancia de un informe ya inicializados, los resultados de ejecutar los módulos asociados con la interoperabilidad entre BOA y un analizador de un lenguaje de programación, un *callback* que será el que se utilizará para ejecutar un módulo de seguridad a partir de una instancia de un ciclo de vida en una fase concreta y definir metainformación. Además, define una serie de métodos obligatorios a ser implementados por todas las clases que extiendan el comportamiento de la clase abstracta, los cuales están relacionados con la ejecución de las fases del ciclo de vida.
- Clase *BOAReportAbstract*: clase abstracta utilizada como módulo principal para los módulos de informes, dedicados a almacenar entradas relativas a vulnerabilidades o problemas *software* y a mostrarla a través de un formato concreto. El comportamiento a la hora de inicializar esta clase abstracta es inicializar estructuras de datos relacionadas con la información a mostrar en el informe, inicializar estructuras de datos relacionadas con la enumeración de severidad asociada al informe, comprobar que la numeración de severidad extiende a la enumeración base de severidad definida por BOA, almacenar los argumentos definidos para el informe en el archivo de reglas y definir metainformación. Además, define una serie de métodos obligatorios a ser implementados por todas las clases que extiendan el comportamiento de la clase abstracta, los cuales están relacionados con el formato en el que se muestran las vulnerabilidades, o problemas *software*, de manera individual y de manera unida. Otros métodos no obligatorios se definen con la finalidad de poder trabajar sobre cualquier informe de manera general invocando a dichos métodos.

Cabe destacar que, aunque la principal herramienta para la extensibilidad en BOA han sido las clases abstractas, no ha sido la única, ya que también se han empleado las enumeraciones, con la finalidad de poder definir nuevos niveles de severidad. Para las enumeraciones de la severidad se ha empleado una enumeración vacía a partir de la cual se extienden sus valores en otras enumeraciones y se es capaz de poder definir nuevos niveles de severidad en una enumeración totalmente distinta, y la manera de trabajarlas es muy similar a la de las clases abstractas, con la diferencia de que para definir los niveles de severidad, las clases abstractas no nos eran útiles debido a que solo necesitábamos un nombre y un número, lo cual encaja a la perfección con las enumeraciones.

7.4 Módulos de Seguridad

Los módulos de seguridad son aquellos que van a ejecutar el comportamiento que ha definido el usuario para analizar su sistema, y debido a ello, es una de las partes más interesantes de BOA. Algo a destacar es que no se va a comentar el resto de módulos concretos (e.g. ciclos de vida, analizador de lenguajes de programación), no porque no tengan importancia, sino porque el contenido de los mismos ya ha sido descrito extensamente a lo largo del presente capítulo. En el caso de los módulos de seguridad, sí que hay mucha más información a destacar que aún no ha sido explicada debido a que es un contenido independiente del analizador (es dependiente del problema a analizar sobre el sistema *software*). El motivo de implementar algunos módulos de seguridad es aportar una funcionalidad mínima a BOA, para poder probar el sistema en su totalidad, y para que pueda utilizarse nada más ser descargado y no sea necesario para un usuario tener que escribir sus propios módulos para poder empezar a analizar su sistema (esta funcionalidad mínima será dependiente del lenguaje

de programación, por lo que es una limitación de la implementación de dicha funcionalidad mínima). Pero la carencia de módulos de BOA no significaría nada respecto a su funcionamiento, ya que como analizador de propósito general, ya está completo llegado a este punto.

La implementación de estos módulos de seguridad pueden contener el comportamiento que el usuario quiera definir, pero lo normal será que se defina un módulo de seguridad por cada técnica de análisis *software* que se quiera ejecutar. BOA no está limitado a la detección de vulnerabilidades, también puede ser empleado para buscar cualquier tipo de anomalía *software*, pero eso es debido a que podríamos clasificar ambos problemas de una manera genérica que contenga a ambos problemas juntos en una categoría común (e.g. detección de anomalías *software*). Que no sea posible emplearse para otras finalidades no quiere decir que sea lo correcto y que todo vaya a encajar a la perfección, ya que el diseño y la implementación de BOA se ha realizado teniendo en mente el objetivo de ser un analizador de vulnerabilidades *software*, pero nada impide su uso para otras tareas distintas debido a que es un analizador de propósito general, pero no se garantiza que las herramientas necesarias para realizar dichas tareas, diferentes de aquellas en mente con las que diseñó e implementó este analizador, estén disponibles en BOA.

Algo relevante a destacar, que ya se ha comentado anteriormente, es la eficiencia. La eficiencia de BOA dependerá, casi en su totalidad, de los módulos implementados por los usuarios, ya sean los módulos relacionados con los ciclos de vida, o cualquier otro. Los módulos que más afectarán a la eficiencia del análisis de BOA serán, por regla general, los de seguridad, y esto es así debido a que serán los que más carga computacional aporten al análisis debido a que las técnicas empleadas en el análisis de *software*, dependiendo de la técnica, pueden ser muy costosas computacionalmente en términos de tiempo y, seguramente, también de espacio. Algo también a tener en cuenta es el ámbito sobre el que se ejecute el análisis, lo cual se comenta en la sección Ámbito del Analizador del capítulo Diseño. Aún así, las pruebas empíricas realizadas sobre BOA nos arrojan resultados acerca de una eficiencia más que suficiente, pues todos los análisis realizados duran menos de 1 segundo, pero se vuelve a destacar que esto también es debido a las técnicas empleadas. Además, ya se comentó que la eficiencia no era un objetivo de diseño, pero lo comentado se trata más bien de una aclaración.

Los módulos de seguridad que han sido implementados están dirigidos al lenguaje de programación C, y el analizador de lenguaje de programación concreto utilizado ha sido Pycparser. Se han implementado 3 módulos, 2 de ellos dirigidos a encontrar vulnerabilidades genéricas de C pero que, en nuestro caso, han sido dirigidas a buscar vulnerabilidades de tipo *Buffer Overflow* (esto es totalmente configurable a través del archivo de reglas), y el último está pensado para ser empleado como dependencia debido a que no busca vulnerabilidades, sino que construye una estructura de datos con la finalidad de ser analizada por otros módulos.

7.4.1 *Function Match*

Este módulo realiza una tarea muy simple, la cual es buscar llamadas a funciones que, por algún motivo, se consideran vulnerables. Los resultados que se obtienen han sido dirigidos a buscar vulnerabilidades de tipo *Buffer Overflow*, pero esta técnica no está limitada a solo esta vulnerabilidad, sino que es útil para todas las vulnerabilidades que sean detectables a través de llamadas a funciones concretas que, por algún motivo (e.g. función vulnerable, función mal utilizada frecuentemente), su simple invocación puede ser un síntoma de una vulnerabilidad (e.g. condiciones de carrera o *race conditions*).

Se trata de una técnica muy simple pero que puede ayudar en gran medida, ya que puede guiar al usuario a la hora de, por ejemplo, realizar un análisis manual. Un análisis manual de un sistema puede ser

inviable, pero a través de este módulo, el usuario puede ayudarse analizando solo aquellas regiones del sistema que sean sospechosas de albergar alguna vulnerabilidad.

Esta técnica ha sido empleada por una gran multitud de analizadores de vulnerabilidades como pueden ser RATS[1] o Flawfinder[2].

Archivo de Reglas

A la hora de aplicar esta técnica se espera encontrar una serie de argumentos en el archivo de reglas. Estos argumentos especificarán las siguientes características:

1. Nombre de función potencialmente vulnerable. Este elemento estará contenido en una etiqueta de tipo *element* con nombre "*method*".
2. Nivel de severidad en caso de detectarse una llamada a la función potencialmente vulnerable. Este elemento estará contenido en una etiqueta de tipo *element* con nombre "*severity*".
3. Una descripción acerca de la vulnerabilidad detectada. Este elemento estará contenido en una etiqueta de tipo *element* con nombre "*description*".
4. Un texto descriptivo acerca de cómo es posible solucionar la vulnerabilidad detectada. Este elemento estará contenido en una etiqueta de tipo *element* con nombre "*advice*".

Los elementos descritos definirán una función potencialmente vulnerable, y estarán contenidos en un elemento *dict*. Por cada función potencialmente vulnerable tendremos un elemento *dict*, los cuales estarán contenidos en un elemento *list* con nombre "*methods*", y dicha lista estará contenida en el elemento *dict* principal de los argumentos (i.e. elemento *dict* único y obligatorio contenido en el elemento *args* cuando se definen argumentos).

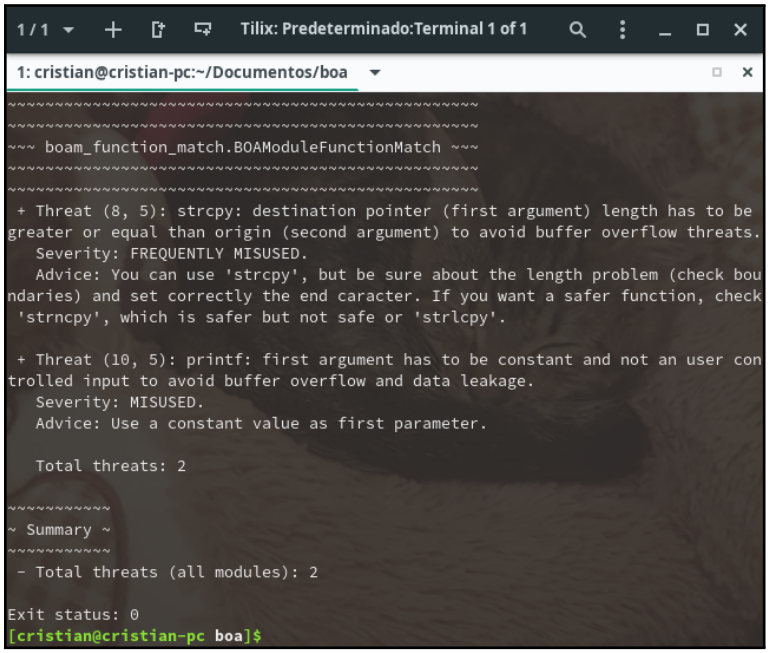
Implementación

La implementación de este módulo está contenida en el fichero *boa/modules/boam_function_match.py*. La inicialización de esta instancia se basa en un pequeño procesamiento de argumentos del archivo de reglas para obtener todas las funciones potencialmente vulnerables. El procesamiento se basa en la obtención de, únicamente, todos los nombres de las funciones potencialmente vulnerables, además de una asociación con el resto de la información relacionada con la función potencialmente vulnerable a la que pertenece dicho nombre.

El procesamiento realizado se basa en realizar un recorrido en preorden en el AST (realizando este recorrido se garantiza realizar el recorrido del AST en el orden en el que los elementos fueron definidos en el fichero de código), recorrido el cual está implementado en el módulo auxiliar contenido en el fichero *boa/auxiliary_modules/pycparser_ast_preorder_visitor.py*, buscando llamadas a funciones. Cuando una llamada a función es detectada, la cual se detecta comprobando si un elemento del AST pertenece a la clase *FuncCall* (comprobar sección Pycparser del capítulo Tecnologías para más información acerca de las clases utilizadas por Pycparser), y si es así, se obtiene el nombre de la llamada a la que se hace referencia y se comprueba si

[1] Repositorio de RATS: `https://github.com/andrew-d/rough-auditing-tool-for-security`
[2] Sitio oficial de Flawfinder: `https://dwheeler.com/flawfinder/`

dicha llamada coincide con alguna de las funciones potencialmente vulnerables. En el caso de detectar una función potencialmente vulnerable en el nodo del AST, se obtiene la información necesaria del mismo para crear una entrada en el informe de la clase y se continua buscando más nodos del AST que contengan llamadas a funciones.



(a) Ejecución del módulo *Function Match*

BOA - Report

boam_function_match.BOAModuleFunctionMatch

Who	Row	Column	Severity	Description	Advice
Threat	8	5	FREQUENTLY_MISUSED	strcpy: destination pointer (first argument) length has to be greater or equal than origin (second argument) to avoid buffer overflow threats	You can use 'strcpy', but be sure about the length problem (check boundaries) and set correctly the end caracter. If you want a safer function, check 'strncpy', which is safer but not safe or 'strlcpy'
Threat	10	5	MISUSED	printf: first argument has to be constant and not an user controlled input to avoid buffer overflow and data leakage	Use a constant value as first parameter
Total threats:					**2**

Summary

Property	Value
Total threats	2

(b) Ejecución del módulo *Function Match* (formato del informe HTML)

Figura 7.4: Ejecución de un módulo de seguridad de BOA (módulo *Function Match*)

En la figura 7.4 se puede observar un ejemplo de la ejecución de este módulo, en formato nativo (i.e. consola, terminal) y en formato HTML (fácilmente configurable a través del archivo de reglas indicando que se utilice el módulo *boar_basic_html* y la clase *BOARBasicHTML* en el elemento *boa_rules.report*).

Limitaciones

Principalmente, encontramos una limitación a la hora de aplicar esta técnica, y tiene que ver con su tasa de falsos positivos. Por lo general, esta técnica tiene un nivel de falsos negativos muy reducida, pero en el caso de los falsos positivos tiene una tasa muy elevada, y es debido a la agresividad de la técnica. La idea principal de esta técnica es que, solo con la llamada a una función, se tienen los indicios necesarios para pensar en una posible vulnerabilidad, lo cual es demasiado agresivo para la mayoría de análisis. Aún así, esta técnica puede ser muy útil cuando se necesita realizar análisis iniciales (en una primera aproximación de un sistema en el cual no se han realizado análisis de vulnerabilidades, es muy probable que contenga vulnerabilidades, lo cual hace que la aplicación de esta técnica sea adecuado debido a que la tasa de falsos positivos será más reducida), análisis sobre un sistema muy grande (la eficiencia de esta técnica puede reducir los tiempos de análisis en gran medida en sistemas complejos) o analizar muchos sistemas (gracias a la eficiencia de esta técnica, se pueden analizar muchos sistemas de manera que se puedan obtener conclusiones en un corto periodo de tiempo).

Otra limitación de esta técnica es que se está aplicando en su versión más simple, ya que las llamadas a funciones se comprueban con el nombre directo que se indicó en el archivo de reglas. Con esto se quiere dar a entender que no se intenta solucionar el problema de los alias a funciones (i.e. este problema es conocido como análisis de punteros), lo cual hace que no funcione para aquellos casos en los que las llamadas a funciones se realicen a través de punteros.

En conclusión, se trata de una técnica simple que es muy robusta, pero con un nivel de completitud muy bajo en el caso general.

7.4.2 *Control Flow Graph* (CFG)

Un grafo de flujo de control (más conocido como *Control Flow Graph* o CFG) es un grafo que contiene información acerca del flujo de control de un programa. Esta información puede ser sumamente útil cuando se realiza un análisis interprocedural, ya que con el CFG somos capaces de saber los posibles caminos que se pueden tomar en tiempo de ejecución e ir de función en función realizando el análisis. No solo es útil para los análisis interprocedurales, también lo es para los intraprocedurales, ya que se pueden aplicar técnicas que necesiten del flujo de control, e incluso se pueden aplicar técnicas propias del análisis interprocedural pero de manera que solo se realice el análisis sobre un procedimiento, simulando así un análisis intraprocedural.

Este módulo está implementado para funcionar con el lenguaje de programación C y, en concreto, con el analizador Pycparser. La implementación se ha realizado de tal modo para que nos sirva tanto para realizar análisis interprocedurales como intraprocedurales. Los nodos empleados por el grafo son simples, es decir, no se realiza ningún procesamiento de manera que se obtenga un conjunto de nodos del AST que formen, por ejemplo, una instrucción completa o un conjunto de instrucciones. Esto tiene sus ventajas e inconvenientes, ya que por una parte definimos el CFG en su aspecto más general, el cual solo soluciona los problemas relacionados con el flujo de control y permite al usuario perfeccionar el módulo si así lo desea, pero por otra parte, el trabajar directamente con esta implementación puede ser costoso y complejo. La razón de que sea difícil trabajar con los nodos básicos del AST en el CFG es debido a que la mayoría de técnicas trabajan a un nivel de ámbito, como mínimo, de línea de programa, y en el caso general se utilizan lo que se conoce como bloques básicos (i.e. nodo del CFG que contiene un conjunto de instrucciones cuyo flujo de control es continuo hasta que la última instrucción del bloque básico cambia de ámbito, o su flujo de control no es directo

a la siguiente instrucción o el flujo de control se bifurca).

Debido a que este módulo se ha implementado con la idea de que se utilice como dependencia para otros módulos de seguridad, su simple ejecución no introduce ninguna entrada en los informes de vulnerabilidad de BOA. Aún así, se ha realizado la implementación de una utilidad gráfica que nos permite observar los resultados del CFG si así se desea. Esta utilidad ha servido para la depuración del propio CFG y poder observar de manera gráfica los resultados, lo cual normalmente es más agradable que leer unos resultados que en modo texto son complejos de entender. Algo a tener en cuenta a la hora de interpretar estos resultados de manera gráfica es que el código crece a lo largo del eje Y (por defecto, 10 unidades por cada instrucción), en concreto hacia los valores positivos, y el ámbito del código crece y decrece a lo largo del eje X (por defecto, 0.5 unidades por ámbito).

Archivo de Reglas

El archivo de reglas para este módulo solo contiene elementos de tipo *element* con directivas de configuración que afectan a cómo el módulo se comporta. Los elementos posibles y su comportamiento se explican a continuación:

- Elemento con nombre "*recursion_limit*": debido al alto uso de la recursividad en el módulo, se permite configurar el nivel de recursividad máximo. Esto puede ser muy útil, pues si se realiza un análisis sobre un sistema que tenga una gran cantidad de ámbitos o el grafo de flujo resultante sea muy complejo, el nivel de recursividad aplicado por defecto puede no ser suficiente para el procesamiento del módulo.
- Elemento con nombre "*display_cfg*": con finalidades de saber cuál es el resultado concreto sobre un sistema, se ofrece un mecanismo a través del cual se puedan observar los resultados en modo texto. Este elemento acepta un valor booleano, siendo el valor por defecto "*false*".
- Elemento con nombre "*plot_cfg*": con la finalidad de obtener un resultado mucho más sencillo y rápido de interpretar que el obtenido en modo texto, se puede activar la obtención del resultado a través de una utilidad gráfica con este elemento. En caso de aplicarse este elemento, es decir, configurar el valor "*true*", se buscará si la librería matplotlib está instalada, la cual es necesaria. Si la librería matplotlib no está instalada, la ejecución el módulo no se detendrá, pero no se mostrará el resultado a través de la utilidad gráfica. Este elemento acepta un valor booleano, siendo el valor por defecto "*false*".
- Elemento con nombre "*lines_clip*": las aristas del CFG que se observan en la utilidad gráfica se extienden en grandes proporciones hasta en los programas más pequeños, lo cual hace que haya una gran cantidad de aristas en la utilidad gráfica. A través de este elemento es posible configurar si solo queremos que se dibujen las aristas que caben en la ventana de la utilidad gráfica, lo cual ayuda a la visualización de los datos pero es más ineficiente y realentiza la utilidad gráfica debido a que tiene que comprobar constantemente si tiene que redibujar las aristas y comprobar cuáles tiene que borrar. Este elemento se aplicará solo si el elemento *plot_cfg* está configurado con el valor "*true*". Este elemento acepta un valor booleano, siendo el valor por defecto "*true*".
- Elemento con nombre "*random_x_offset*": debido a la gran cantidad de ámbitos que se dibujan en la utilidad gráfica, la visualización de las aristas es complicado cuando el dibujado se desplaza de manera fija, pues resulta en que en una misma arista se pueden estar solapando muchas aristas. Activando este elemento, se aplica un pequeño desplazamiento aleatorio en el eje x consiguiendo, en la mayoría de

ocasiones, evitar el solapamiento de aristas. Este elemento se aplicará solo si el elemento *plot_cfg* está configurado con el valor "*true*". Este elemento acepta un valor booleano, siendo el valor por defecto "*false*".

- Elemento con nombre "*random_y_offset*": debido a la gran cantidad de ámbitos que se dibujan en la utilidad gráfica, la visualización de las aristas es complicado cuando el dibujado se desplaza de manera fija, pues resulta en que en una misma arista se pueden estar solapando muchas aristas. Activando este elemento, se aplica un pequeño desplazamiento aleatorio en el eje y consiguiendo, en la mayoría de ocasiones, evitar el solapamiento de aristas. Este elemento se aplicará solo si el elemento *plot_cfg* está configurado con el valor "*true*". Este elemento acepta un valor booleano, siendo el valor por defecto "*false*".

- Elemento con nombre "*propagate_func_call*": este elemento nos permite configurar si queremos que las llamadas a funciones del CFG se propaguen o no. El poder configurar esta característica nos permite elegir cuál va a ser el tipo de análisis a realizar: interprocedural o intraprocedural. Este elemento acepta un valor booleano, siendo el valor por defecto "*true*" (por defecto se aplica un análisis interprocedural).

Implementación

La implementación de este módulo está en el fichero *boa/modules/boam_cfg.py*. La inicialización de esta instancia se basa en configurar los valores de las variables configurables a partir del archivo de reglas, iniciar la estructura de datos del CFG, la cual está en el fichero *boa/auxiliary_modules/pycparser_cfg.py*, y comprobar si la librería matplotlib está disponible para ser utilizada en caso de que se quiera utilizar la utilidad gráfica para observar el resultado del CFG.

La estructura de datos empleada para la construcción del CFG está definida en la clase *CFG*, la cual se ayuda de la clase auxiliar *Instruction* y de unas clases auxiliares que representan nodos del AST de Pycparser. Los nodos auxiliares de Pycparser están representados por las clases *FinalNode* (nodo que indica que la ejecución del programa finaliza, ya sea porque es el final de la función *main* o por llamadas a funciones del sistema como *exit()*), *EndOfFunc* (nodo que indica el final de una función, lo cual ayuda a la construcción del CFG para los análisis interprocedurales), *EndOfLoop* (nodo que indica el final de un bucle, ya sea *for*, *while* o *do while*, con la finalidad de detectar correctamente el ámbito del mismo, ya que puede contener de manera opcional un elemento *Compound*, lo que dificulta el análisis) y *EndOfIfElse* (nodo que indica el final de una declaración *if-else*, y se inserta tanto en la parte del *if* como en la del *else* para que sea más sencillo el análisis, ya que el elemento *Compound* es opcional). La estructura de datos definida en la clase *CFG* contiene todas las funciones definidas en el fichero de código y las instrucciones del AST ordenadas que cada función contiene. Estas instrucciones se representan a partir de la clase *Instruction*, la cual almacena el nodo del AST original al que hace referencia la instrucción y, además, almacena referencias a instrucciones que el flujo de control puede llevar a su inmediata ejecución después de la instrucción a la que se hace referencia, referencias las cuales también las llamaremos indistintamente dependencias de la instrucción principal almacenada (con dependencia nos referimos a una relación de dependencia respecto del flujo de control). Almacenando estas referencias a las siguientes instrucciones que se ejecutarán atendiendo al flujo de control se define una estructura de grafo, y así es como se almacenan los datos del CFG.

Volviendo a la ejecución del módulo, instancia la cual se crea a partir del módulo *boam_cfg.py*, en concreto de la clase *BOAModuleControlFlowGraph*, realiza el proceso atendiendo únicamente a la definición

de las funciones y obviando otros elementos del ámbito global. La razón de solo crear el CFG a partir de las definiciones de las funciones es abstraerse de la definición de estructuras de datos típicas de C en el ámbito global (e.g. estructuras, enumeraciones, uniones), lo cual dificultaría la construcción del CFG, y aunque es posible hacer una definición completa, en este caso se ha centrado la implementación tan solo en el flujo de control de las instrucciones definidas en el fichero de código. Por lo tanto, el proceso se basa en recorrer el AST en un recorrido preorden (realizando este recorrido se garantiza recorrer el AST en el mismo orden en el que los elementos están definidos en el fichero de código), recorrido el cual está implementado en el módulo auxiliar contenido en el fichero *boa/auxiliary_modules/pycparser_ast_preorder_visitor.py*, y cada vez que se encuentre una definición de una función (i.e. clase de Pycparser *FuncDef*), se construye el CFG para esa función.

La manera en la que se construye el flujo de control para una función se basa en una premisa muy sencilla, y es que, como se comentó en la subsección Análisis del Flujo de Control (*Control-Flow Analysis*) del capítulo Estado del Arte, la construcción de un CFG puede ser muy sencilla si se dan las condiciones necesarias. Tal como hemos definido que se va a realizar el análisis para construir el CFG es como si de un análisis intraprocedural se tratara, lo cual hace que, unido al lenguaje de programación para el que se está construyendo el CFG, el cual es el lenguaje de programación C, sea muy sencillo, ya que no presenta ninguno de los inconvenientes descritos para el análisis de flujo de control. El paradigma al que pertenece el lenguaje de programación C es el imperativo o procedimental, lo cual significa que las instrucciones se ejecutan en un flujo de ejecución secuencial no necesariamente ordenado,lo cual unido a la metodología intraprocedural que se va a seguir para la construcción del CFG, hace sencilla la construcción del CFG. Una vez dicho esto, por cada función que va a ser procesada para construirse el CFG, se crea una instrucción creando una instancia de la clase *Instruction* y se añade a la estructura de datos *CFG*, cuya instancia está almacenada en la instancia de la clase del módulo. De esta manera, ya tenemos un procesamiento básico para el CFG para todas las funciones del fichero de código.

Aún siendo sencilla la construcción de un CFG para un lenguaje de programación como C, nos hemos encontrado con una gran cantidad de problemas para llevar a cabo de la construcción de un CFG de calidad que reflejara realmente cómo es el flujo de control de un programa. Teniendo en mente la manera sencilla de añadir instrucciones y sus instrucciones sucesivas como dependencias del flujo de control, se define el comportamiento general para casi todos los tipos de nodos del AST, pero hay ciertos nodos, los cuales justamente pertenecen a la categoría de modificadores del flujo de control, que han necesitado de un tratamiento especial en todas sus formas. Estos elementos, los cuales han necesitado un tratamiento especial para que el CFG reflejara realmente el flujo de control, han sido los siguientes (comprobar sección Pycparser del capítulo Tecnologías para más información acerca de las clases de Pycparser):

- Clase *If*: los bloques *if-else* representan un problema difícil de resolver a la hora de construir el CFG, y es que se pueden definir de varias maneras y tiene mucha flexibilidad, lo cual complica el análisis del flujo de control. La principal problemática encontrada con este bloque es que hay varias formas de definirse (e.g. solo el bloque *if*, bloque *if-else*, el bloque *if* solo contiene una instrucción y no se utilizan llaves y debido a ello no se utiliza un elemento *Compound*, el bloque *if* solo contiene una instrucción y se utilizan llaves y debido a ello se utiliza un elemento *Compound*, ...), siendo todas ellas correctas. Lo que se ha hecho para solucionar el problema de la manera en la que se identifican los bloques *if-else* ha sido realizar una normalización del mismo, lo cual consiste en utiliza siempre bloque *else* (si no está presente, se inserta de manera que sea identificable) y en utilizar siempre llaves para los bloques *if-else* (si no existe elemento *Compound*, se inserta en ambos bloques, con la excepción del bloque *else*, el cual si no existe, no se añade, pero si solo tiene una instrucción y no tiene elemento *Compound*, entonces

sí que se añade). Además, para identificar el final de los bloques *if-else* se añade al final del bloque *if* y del bloque *else* un elemento definido en específico para la tarea de indicar que se terminan ambos bloques, el cual es *EndOfIfElse*. Una vez terminamos esta normalización, es sencillo realizar el análisis de flujo de control. En la condición del bloque *if* se añaden dependencias a la primera instrucción del bloque *if* y a la primera instrucción del bloque *else* (en el caso de que el bloque *else* se haya construido artificialmente, solo contendrá un nodo de tipo *EndOfIfElse*, y aquí es también donde vemos la ventaja de este nodo creado artificialmente). Por último, a las últimas instrucciones de los bloques *if* y *else* se les añaden dependencias hacia la primera instrucción que hay al terminarse el bloque *if-else*.

- Clase *For*: los bloques *for* representan un problema difícil de resolver a la hora de construir el CFG, y es que se pueden definir de varias maneras y tiene mucha flexibilidad, lo cual complica el análisis del flujo de control. La principal problemática con la que nos encontramos es la misma que con la clase *If*, por lo que la solución adoptada es muy similar y se trata de normalizar. La normalización del elemento *for* consiste en añadir, inicialmente, un elemento *Compound*, lo cual será necesario si no se han insertado las instrucciones en un bloque. Posteriormente, se añade como última instrucción del bloque *for* un elemento *EndOfLoop*, elemento el cual nos permitirá comprobar cuándo se termina el bloque de instrucciones. Una vez hecha esta normalización, ha sido necesario implementar todas las posibles formas de un bucle *for*, el cual tiene múltiples formas de definirse (e.g. *for(;;);*, *for;;{}*, *for(;true;);*, *for(i=0;i<10;i++){⋯}*, *for(i=0;;i++){⋯}*) y cada una de ellas tiene un control del flujo diferente. Esta clase ha sido una de las más complejas de implementar por los problemas comentados. La manera en la que el flujo de control ha sido controlado se explica a continuación:
  - *for(;;);*: dependencia del nodo *For* al nodo *EndOfIfElse* y dependencia del nodo *EndOfIfElse* al nodo *For*, creando un bucle infinito en el CFG.
  - *for(int i=0;i<10;i++){⋯}*: dependencia del nodo *For* al nodo de la inicialización, dependencia del nodo de la inicialización al nodo de la comprobación, dependencia del nodo de la comprobación a la primera instrucción del cuerpo del bucle *for* y hacia la primera instrucción de fuera del bucle *for*, dependencia de la última instrucción del bucle *for* hacia la primera instrucción de fuera del bucle *for* y hacia el incremento y dependencia desde el incremento hacia la comprobación.
  - *for(;i<100;);*: dependencia del nodo *For* al nodo de la comprobación, dependencia del nodo de la comprobación hacia el nodo *EndOfIfElse* y dependencia del nodo *EndOfIfElse* hacia el nodo de la comprobación.
  - ...
- Clase *While*: los bloques *while* siguen representando la misma problemática que la encontrada en el bloque *for*, con la diferencia de que esta vez, los bloques *while*, no tienen un grado de libertad tan amplio como sucede con los bloques *for*. Al igual que antes, la manera en la que hemos resuelto estos problemas es con la normalización. Lo primero que se hace es comprobar si es necesario añadir un elemento *Compound* y, una vez se ha añadido si es necesario, se añade un elemento *EndOfLoop* como última instrucción. La manera en que se realiza el análisis de flujo de control es añadiendo una dependencia del nodo *While* a la condición, dependencia de la condición a la primera instrucción del cuerpo del bloque *while* y hacia la primera instrucción de fuera del bucle *while* que lo continúa, y dependencia de la última instrucción del bloque *while*, la cual será un nodo *EndOfLoop*, a la condición del bloque *while*.
- Clase *DoWhile*: igual que los nodos *While*. La única diferencia es que el flujo de control varía en que la primera dependencia del nodo *DoWhile* es con la primera instrucción del cuerpo de instrucciones en lugar de con la condición del bucle.

- Clase *Switch*: los bloques *switch* han presentado la misma problemática que la encontrada en el resto de bloques, pero esta vez se han resuelto de manera diferentes debido a la peculiar forma de definir este elemento, pero ha sido junto al elemento *For* uno de los más complicados de resolver. Los bloques *switch* se definen de manera que contienen elementos *case* y *default*, por lo que la manera en la que se ha resuelto el análisis de flujo de control del elemento *switch* ha sido buscando estos elementos. La búsqueda de estos elementos, *case* y *default*, ha tenido un gran problema, el cual ha sido la flexibilidad del lenguaje C con este bloque, y es que, aunque es muy inusual, un bloque *switch* puede contener otros elementos *switch*, lo cual ha complicado en gran medida la búsqueda de elementos *case* y *default*. La manera en la que se ha resuelto la búsqueda de estos elementos ha sido obteniendo todos los elementos de tipo *case* y *default*, se ha calculado los elementos padre de cada uno de ellos, y se ha comprobado si el elemento padre coincidía con el bloque *switch* bajo análisis (la comparación se ha tenido que realizar a través de referencias, no de valor, ya que una comparación por valor puede arrojar un falso positivo). Una vez resuelto el problema de los elementos *case* y *default*, se ha procedido a realizar el análisis del flujo de control. La manera en que se realiza el análisis de flujo de control es añadiendo una dependencia del nodo *Switch* a la condición y dependencia de la condición a todos los elementos *case* y *default*. Es cierto que realizando esta serie de dependencias no queda completo el análisis de flujo de control del elemento *Switch*, ya que quedará un elemento *Switch* donde no se tengan en cuenta los elemento *Break* de los que tanto depende. Esto se ha realizado así porque el elemento *Break* tiene su propio procesamiento para realizar el análisis de flujo de control y, cuando se realice el análisis de flujo de control de dicho elemento, será cuando, de manera indirecta, el análisis de flujo de control del elemento *Switch* se completará.

- Clase *Break*: el modo en el que se ha resuelto el análisis de flujo de control para el elemento *Break* se ha basado en la suposición de que dicho elemento tiene que estar contenido en un elemento *For*, *While*, *DoWhile* o *Switch*, cuya suposición será correcta para un fichero de código compilable. La manera en la que se realiza el análisis de este elemento es buscando los elementos padre y, cuando se encuentre un padre que sea un elemento *For*, *While*, *DoWhile* o *Switch*, entonces, la siguiente instrucción de estos elementos será la dependencia del elemento *Break*. Se trata de una manera sencilla de solucionar el comportamiento de este elemento, el cual puede ser difícil de manejar en ciertas situaciones (e.g. *for(;;){{{break;}}}*), y una comprobación directa del padre del elemento *Break* no funcionaría en la mayoría de ocasiones (en el caso de que estuviera contenido en un elemento *Switch*, no funcionaría nunca, ya que el elemento *Break* debería de estar contenido en un elemento *case* o *default*).

- Clase *Continue*: mismo mecanismo de funcionamiento que con el elemento *Break*, pero no tiene en cuenta si el padre es un elemento *Switch*. La dependencia del elemento *Continue* es un poco más difícil de obtener, ya que dependerá de si el elemento padre *For*, *While* o *DoWhile* tienen o no una condición definida (en el caso de los elementos *While* y *DoWhile* es obligatorio que tengan una compilación para que se pueda compilar el código). En realidad, el único caso a tener en cuenta es el elemento *For*, ya que los otros deberían de tener una condición definida. Para el caso del elemento *For* se sigue la misma política que la aplicada para el mismo elemento *For*, y es que se intenta buscar la condición, si no está disponible se intenta buscar el incremento y si no está disponible, se obtiene la primera instrucción del bucle *For* como dependencia.

- Clase *Goto*: el tratamiento especial es esta clase ha sido uno de los más sencillos, ya que lo que se ha hecho ha sido buscar la etiqueta empleada en la declaración *goto* y, una vez se ha encontrado, se ha añadido a la instrucción de la clase *Goto* la dependencia de la instrucción de la etiqueta.

- Clase *FuncCall*: la manera en la se realiza el procesamiento de las llamadas a funciones, junto con el

procesamiento de las declaraciones *return*, definen si se trata de un procesamiento intraprocedural o interprocedural. Antes hemos comentado que, en principio, se iba a realizar un análisis intraprocedural con el fin de facilitar el mismo, pero esta es una excepción con la finalidad de obtener, también, un análisis interprocedural. Cuando se encuentre una llamada a función, se comprobará si la llamada se trata de una llamada simple (no se va a realizar un análisis de punteros con el fin de resolver el problema de los alias de funciones), y en tal caso, se comprobará si la llamada a función se trata de una llamada a una función definida en el fichero de código. En caso de que la función exista, se añadirá una dependencia de la llamada a función a la declaración de la función. Esto nos aportará mucha más información en global del fichero de código, consiguiendo así un análisis intraprocedural. Este análisis intraprocedural está activado por defecto, pero puede desactivarse a través del argumento del archivo de reglas llamado "*propagate_func_call*". No siempre interesa este análisis interprocedural, y debido a ello se da la opción de desactivarse. Para el caso de aquellas funciones que no se reconozcan, lo cual será muy habitual (e.g. funciones del sistema, alias a funciones), se aplicará el comportamiento general de añadir la siguiente instrucción como dependencia.

- Clase *Return*: sucede algo similar a lo del elemento *FuncCall*. Este elemento nos permite definir un análisis interprocedural, y si este está activado, el flujo de control de los elementos *Return* tendrán dependencias a todas las siguientes instrucciones de los elementos *FuncCall* del programa cuya función invocada sea la que contiene el elemento *Return*. En el caso de no estar activado el análisis interprocedural, la dependencia del elemento *Return* sera la que se añada por el comportamiento general, es decir, la siguiente instrucción, la cual puede ser cualquier tipo de instrucción o, en el caso general (elemento *Return* como última instrucción de la función), se tratará de un elemento *EndOfFunc*.

Algo a destacar de la normalización aplicada a los diferentes elementos es que ha habido una doble problemática para llevarla a cabo. La razón de ello es que no solo hemos tenido que insertar elementos en la estructura de datos propia *CFG*, sino que también ha sido necesario añadir estos elementos en el AST de Pycparser. Llevar a cabo estas tareas de manera que el orden de las instrucciones y de los nodos del AST se siguiera manteniendo ha sido un reto en numerosas ocasiones, ya que no solo era una tarea completa al tener que modificar diferentes estructuras de datos, sino que también se ha tenido que aplicar un tratamiento especial para cada normalización de cada elemento distinto. El resultado final nos ha permitido obtener un CFG de calidad, el cual, a través de numerosas pruebas empíricas, nos garantiza robustez. Es cierto que para la implementación de muchas secciones del CFG se ha supuesto que el código del fichero de lenguaje C era correcto, lo cual no tiene porque ser así y un analizador de flujo de control tendría que ser resistente a ciertos fallos, pero aún así el resultado es satisfactorio.

En la figura 7.5 se puede observar un ejemplo de la ejecución de este módulo. En concreto, se puede observar la utilidad gráfica donde se muestran todos los nodos, los cuales representan los elementos del AST, y, mediante el recorrido de las flechas, el flujo de control definido para un simple programa que comprueba si la cantidad de argumentos es mayor que 1, en cuyo caso devuelve el valor 0 en la función *main()*, y en caso contrario devuelve el valor 1. Este ejemplo, aunque simple, muestra algunos de los problemas que se han comentado con las estructuras de control. En este caso solo hay una función definida, pero si hubieran más funciones, dependiendo de la configuración, se verían más nodos enlazados en función de las llamadas a funciones (habrían dependencias hacia las definiciones de las funciones desde los nodos de tipo *FuncCall* y dependencias salientes de los elementos de tipo *Return* hacia los siguientes nodos que llamaban a la función concreta).

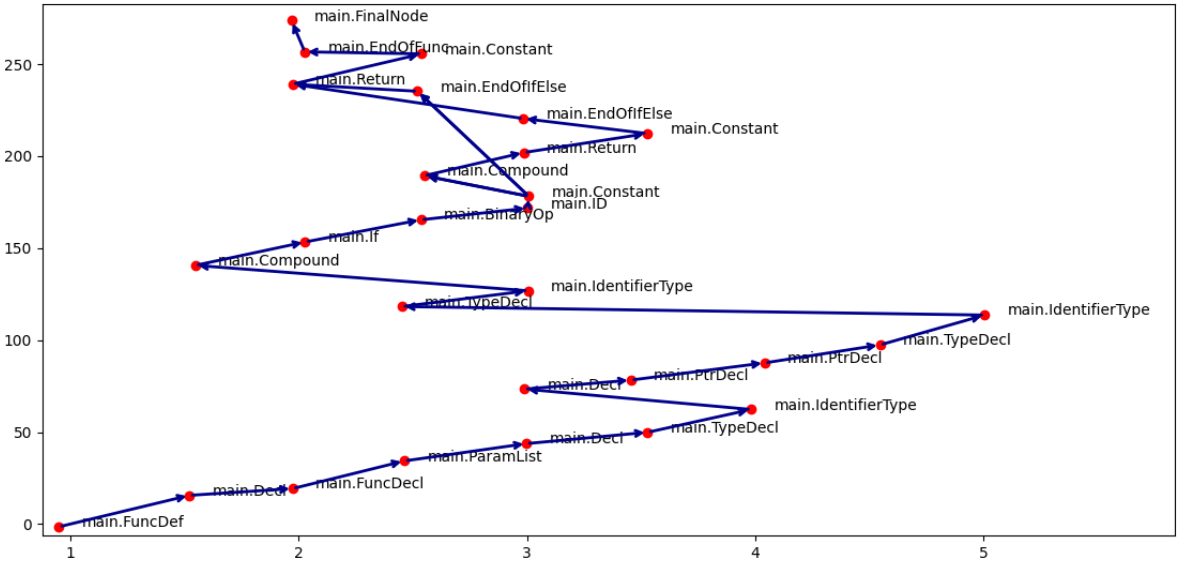


Figura 7.5: Ejecución de un módulo de seguridad de BOA (módulo *CFG*)

Limitaciones

La principal limitación que encontramos en esta técnica es debido a la manera en la que se ha implementado. La manera formal de implementar un CFG es utilizando bloques básicos, los cuales contienen un conjunto de instrucciones hasta que se encuentra una que modifica el flujo de control, pero en nuestro caso hemos implementado nodos simples que contienen únicamente un nodo del AST, lo cual puede hacer que el CFG sea bastante difícil de procesar. Aún así, la implementación del CFG es lo más genérica posible, permitiendo construir otro módulo que utilice este como dependencia y construir a partir del CFG base los bloques básicos o, si se quiere, nodos a partir de instrucciones completas, no a partir de solo nodos del AST.

7.4.3 *Taint Analysis*

El análisis de manchas, o más conocido como *taint analysis*, es una técnica donde lo que se intenta es hacer una traza de la información que, directa o indirectamente, ha sido o podido ser introducida o manipulada por el usuario. La traza se realiza teniendo en cuenta ciertos aspectos, como saber si la información llega a algún punto donde puede ser peligroso, ya que si no se tuviera en cuenta esta consideración, la cantidad de falsos positivos sería muy elevada debido a que todo programa que interactúe con un usuario tendrá, en mayor o menor medida, información introducida por el mismo. Otro aspecto que tienen en cuenta son los puntos que introducen información por parte del usuario y puntos que hacen que esa información ya no se considere manipulada (e.g. comprobaciones con condiciones, métodos ”desinfectantes” o ”depuradores”) o puntos que hacen que la información manipulada pase de un contenedor a otro (e.g. función *strcpy* en el lenguaje de programación C), como si de una ”infección” se tratara. Esta técnica es muy empleada debido a su potencial para encontrar vulnerabilidades, aunque no es su única aplicación, pero sí la más empleada. Aunque la teoría de la técnica es muy sencilla de entender, su implementación tiene varias complicaciones que se explicarán conforme sea necesario.

Algo a tener en cuenta de esta técnica es que se realiza con la ayuda de un CFG, por lo que será necesario tener un módulo que nos permita obtener dicho CFG. En nuestro caso, tenemos un módulo que

nos calcula el CFG de un programa, por lo que podemos aplicar esta técnica utilizando dicho módulo como dependencia. El análisis que se va a realizar puede ser interprocedural, el cual es el que emplea *taint analysis*, o intraprocedural, el cual facilita la aplicación de la técnica y es el que se aplica en la presente implementación.

La terminología empleada en el *taint analysis* es la siguiente:

- Fuentes (*sources*): puntos del programa (e.g. funciones) que introducen información al sistema que, de manera directa o indirecta, puede estar manipulada por el usuario o, directamente, es información que proviene del usuario (e.g. función *gets()*).
- Sumidero (*sink*): puntos del programa (e.g. funciones) que son vulnerables y ofrecen un riesgo potencial si información manipulada por el usuario los alcanza (e.g. función *system()*).
- Mancha (*taint*): contenedor que contiene información manipulada por el usuario, directa o indirectamente. Con contenedor nos referimos a variables, pero se generaliza a contenedor debido a que otras estructuras más complejas también pueden tener esta característica de estar ”manchadas”. Al final, esta característica se generaliza a cualquier tipo de variables (e.g. estructuras, uniones, variables simples, enumeraciones).

El objetivo final de esta técnica es detectar si existe un camino entre una fuente y un sumidero de modo que una mancha alcance dicho sumidero. Conforme estos caminos entre fuentes y sumideros se recorren, habrá que detectar los cambios de estado de las manchas, las cuales pueden estar manchadas (i.e. estado ”T”), no manchadas (i.e. estado ”NT”), manchadas o no manchadas (i.e. estado ”MT”, el cual indica que debido a la mezcla de estados, no sabemos el estado concreto en un punto del programa) o desconocido (i.e. estado ”UNK”). Los cambios de estado de las manchas se pueden realizar de muchas maneras, pero principalmente se realizan a través de comprobaciones con condiciones y a través de llamadas a funciones que sabemos que, de alguna manera, ”limpian” la variable (estas funciones que ”limpian” las variables suelen añadirse a través de configuración).

Archivo de Reglas

A la hora de aplicar esta técnica se espera encontrar una serie de argumentos en el archivo de reglas, los cuales se explican a continuación:

- Elemento *list* con nombre ”*sources*”: esta lista contiene la representación de todas las fuentes para el análisis. Como se ha dicho, las fuentes son aquella manera que el usuario tiene de insertar información en el sistema bajo análisis. En concreto, esta lista contendrá una serie de elementos de tipo *element* cuyo valor podrá contener hasta un máximo de 6 valores diferentes separados por el carácter ”@”, y si algún valor no se quiere indicar (hay valores opcionales), simplemente se deja una cadena vacía (e.g. ”@@”, ”@...”). Un ejemplo de fuente sería ”*argc@variable@main*” o ”*strcpy@function@@targ@0@2*”. Los posibles valores para configurar una fuente se explican a continuación:
  1. Nombre: indica el nombre de la fuente concreta a buscar en el sistema bajo análisis, el cual puede pertenecer a una función concreta o un nombre de variable concreto. Se trata de un valor obligatorio.
  2. Tipo: indica el tipo de la fuente, el cual admite los valores ”*function*” o ”*variable*”, y hacen referencia al tipo del contenedor que se va a buscar y cuyo nombre se ha indicado. Se trata de un valor obligatorio.

3. Nombre de función: indica el nombre de la función que contiene la fuente. Este valor puede permitir al usuario personalizar el módulo a su sistema, ya que, por norma general, será poco útil debido a que limita la fuente a solo ser encontrada en la función que se indique (una excepción son los argumentos de entrada del programa en la función *main*). Se trata de un valor opcional.
4. Cómo afecta: indica el modo en el que la fuente afecta a otros contenedores, y es que puede que, por ejemplo, una función que sea una fuente no sea siempre peligrosa, sino bajo ciertas circunstancias, y esto es justamente a lo que ayuda este parámetro. Los valores aceptados son "*argument*" y "*targ*", los cuales indican que la fuente se aplica a un argumento en concreto (comportamiento por defecto de una fuente) y que la fuente se aplica a un argumento en concreto si otro argumento tiene un estado de mancha "T" o "MT" (el ejemplo claro se ve con la función *strcpy()*, la cual se tratará de una fuente si su segundo parámetro tiene un estado "T" o "MT", el cual "contagiará" dicho estado al primer parámetro además de convertirse en fuente), respectivamente. Se trata de un valor opcional.
5. Afectado: argumento sobre el que se aplica la condición de fuente cuando se indica la manera en cómo afecta una fuente concreta. Se trata de un valor opcional, pero si se indica un valor en el elemento que indica cómo afecta la fuente, se tratará de un valor obligatorio. Los valores aceptados son números mayores o iguales que 0, significando 0 la referencia al contenedor de una asignación (e.g. *int foo = source();*), 1 el primer argumento, 2 el segundo argumento, etc.
6. Contenedor manchado: característica específica del valor "*targ*". Este valor indica la posición del parámetro el cual se comprobará si está manchado para aplicar la condición de fuente al parámetro afectado, además de "contagiar" el estado "T" o "MT". Se trata de un valor opcional, pero si se indica un valor en el elemento que indica cómo afecta la fuente, se tratará de un valor obligatorio. Los valores aceptados son números mayores o iguales que 1, significando 1 el primer argumento, 2 el segundo argumento, etc.

- Elemento *list* con nombre "*sinks*": esta lista contiene la representación de todos los sumideros para el análisis. Como se ha dicho, los sumideros son puntos del sistema que si una mancha alcanza, existe un riesgo de tener un punto vulnerable en el sistema. En concreto, esta lista contendrá una serie de elementos de tipo *element* cuyo valor contendrá 2 valores diferentes separados por el carácter "@". Un ejemplo de sumidero sería "*system@0*" o "*getenv@1*". Los posibles valores para configurar un sumidero se explican a continuación:

  1. Nombre: nombre del sumidero, el cual puede referirse a diferentes tipos de contenedores, pero en el caso de la presente implementación se trata solo de nombres de funciones. Se trata de un valor obligatorio.
  2. Posición del argumento: posición del argumento vulnerable, el cual hay que detectar si recibe un parámetro cuyo estado sea "manchado" (i.e. estado "T" o "MT"). Se trata de un valor obligatorio. Los valores aceptados son números mayores o iguales que 0, significando 0 cualquier posición del sumidero (i.e. parámetros) es vulnerable, 1 el primer parámetro es vulnerable, 2 el segundo parámetro es vulnerable, etc.

- Elemento *element* con nombre "*append_tainted_variables_to_report*": se trata de un valor que añade información adicional al informe de resultados. Si esta variable contiene el valor "*true*", todos los contenedores cuyo estado sea distinto de "NT" y "UNK" será reportado en el informe como información, ya que su nivel de severidad será el de depuración. El valor por defecto de este elemento es "*false*".

Implementación

Como ya se ha comentado, la teoría detrás de esta técnica es muy sencilla de entender, pero llevarla a la práctica ha sido complicado como mínimo. La implementación de la técnica se ha basado en utilizar un algoritmo de lista, en concreto se trata del algoritmo de lista de *Kildall* [44]. Para poder realizar una implementación que fuera efectiva, hemos tenido que resolver el problema de conseguir representar los nodos del módulo *CFG* como conjuntos, siendo cada conjunto el contenido de todos los nodos del AST que forman una instrucción completa en el fichero de código. Una vez resuelto el problema del CFG, se ha aplicado el algoritmo de la lista de *Kildall* para cada función, el cual resuelve el *taint analysis*.

Algoritmo: lista de *Kildall*

Input: $program, initial\_dataflow\_information$

1 **foreach** *Instruction i in program* **do**
2 $\quad input[i] \leftarrow \bot$
3 **end**
4 $input[first\_instruction] \leftarrow initial\_dataflow\_information$
5 $worklist \leftarrow \{first\_instruction\}$
6 **while** *not* $worklist.empty()$ **do**
7 $\quad i \leftarrow worklist.remove()$
8 $\quad output \leftarrow flow(i, input[i])$
9 $\quad$ **foreach** *Instruction* j *in* $succs(i)$ **do**
10 $\quad\quad$ **if** $output \not\sqsubseteq input[j]$ **then**
11 $\quad\quad\quad input[j] \leftarrow input[j] \sqcup output$
12 $\quad\quad\quad worklist.append(j)$
13 $\quad\quad$ **end**
14 $\quad$ **end**
15 **end**

La implementación que hemos realizado del *taint analysis* es independiente de función a función, pero se aplica a todas las funciones. Por cada función definida en el fichero de código, se le aplica el algoritmo de la lista de *Kildall* para obtener los resultados, pero el análisis realizado es totalmente intraprocedural, lo cual facilita sumamente la realización del *taint analysis*. Los pasos realizados para aplicar la técnica *taint analysis*, los cuales se han aplicado a cada función, han sido los siguientes:

1. Obtener la información de las fuentes iniciales, lo cual se realiza a través de la información obtenida en los argumentos del archivo de reglas. También se obtiene el CFG, el cual es necesario para continuar con el análisis.

2. Inicializar el algoritmo de la lista de *Kildall* obteniendo la información inicial del flujo de datos. Esto consiste en formar una tabla con todas las variables de la función e inicializar su estado a desconocido (i.e. estado ”UNK”), o estado concreto si es conocido debido a las fuentes (e.g. argumento *argc* de la función *main()*). En esta tabla se realiza ya el análisis inicial de los argumentos de la función bajo análisis para comprobar si algún parámetro es una fuente. La información inicial obtenida en este paso es la información inicial del flujo de datos. Este paso hace referencia a las líneas 1, 2, 3 y 4 del algoritmo lista de *Kildall*.

3. Crear una lista de trabajo que nos sirva para añadir las instrucciones a inspeccionar a lo largo del *taint analysis*. Se inicializa con la primera instrucción de la función bajo análisis, la cual ya se analiza para comprobar si se trata de una fuente. Este paso hace referencia a la línea 5 del algoritmo lista de *Kildall*.

4. Mientras la lista de trabajo no esté vacía (este paso hace referencia a la línea 6 del algoritmo lista de *Kildall*), aplicamos los siguientes pasos:

   a) Obtenemos la primera instrucción de la lista de trabajo y la eliminamos. Los pasos posteriores a este trabajarán sobre esta instrucción que se obtiene o sobre información relacionada con esta instrucción, lo cual hay que tener en cuenta cada vez que se hable de comprobaciones o de cualquier información, ya que se hará referencia a esta instrucción a no ser que se diga lo contrario. Este paso hace referencia a la línea 7 del algoritmo lista de *Kildall*.

   b) Controlar las estructuras de control de flujo (e.g. *if*, *for*, *while*, ...) debido a que los estados de ”mancha” de las variables se heredan para las instrucciones contenidas en dichas estructuras. Esta comprobación se realiza en los casos especiales en los que los bloques de control de flujo adquieren un estado como ”manchado” (i.e. estado ”T” o ”MT”), estado el cual se hace referencia a los bloques que definen si se ejecuta o no un bloque (e.g. condición del bloque *if*, condición del bloque *while*, elementos del bloque *for*). En los casos en los que un bloque de control de flujo se considere ”manchado”, se aplicará un comportamiento especial a todas las instrucciones de dentro del bloque, ya que, indirectamente, todas las variables de dentro del bloque estarán ”manchadas” debido a que el bloque de flujo de control que las contiene, está ”manchado”. Esto es así porque si algún elemento de un bloque de control de flujo tiene una variable ”manchada”, la ejecución del cuerpo de instrucciones de dicho bloque, puede estar condicionada a ejecutarse o no ejecutarse en función de los datos de la variable ”manchada” que está siendo utilizada para cierta condición del bloque (si la inicialización de una variable en un bucle *for* está ”manchada”, ello puede condicionar el número de iteraciones, consiguiendo así un comportamiento posiblemente manipulado y, por lo tanto, haciendo que el comportamiento del bucle *for* esté ”manchado”). A nivel de implementación todo esto significa que si un bloque de flujo de control está ”manchado”, todas las instrucciones contenidas heredarán el mismo estado de ”manchado”. Para aplicar este comportamiento se tiene muy en cuenta que diferentes estructuras de control se pueden anidar (e.g. *while*$(\cdots)\{$*for*$(\cdots;\cdots;\cdots)\{$*if*$(\cdots)$ $\{\cdots\}\}\}$), y cuando ello suceda, lo que va a ocurrir es que el estado de ”manchado” se va a heredar también a las diferentes estructuras de flujo de control, manteniéndose aquellos estados más restrictivos. Todos estos estados heredados se aplican solo mientras estemos en cada estructura de flujo de control, y dichos estados de ”mancha” se almacenan para la posterior toma de decisiones del algoritmo y para comprobar si las variables ”manchadas” llegan a algún sumidero. Este paso, junto a otros, hace referencia a la línea 8 del algoritmo lista de *Kildall*.

   c) Controlar las llamadas a funciones. Las llamadas a funciones tienen un procesamiento diferente al resto debido a que hay que comprobar varias cosas que solo es necesario en dichas llamadas. Por un lado, hay que comprobar si la llamada a función se trata de una fuente, en cuyo caso se comprobará qué variables están ”manchadas” y se actualizará y almacenará el estado de las mismas para la posterior toma de decisiones del algoritmo y para comprobar si las variables ”manchadas” llegan a algún sumidero. Por otro lado, habrá que comprobar si la llamada a función se trata de una función que es un sumidero, en cuyo caso habrá que comprobar si alguna de las variables está ”manchada” y, si alguna lo está, habrá que comprobar si esas variables ”manchadas” están en algún parámetro considerado vulnerable para el sumidero, lo cual se indicaba en el archivo de reglas, y puede que todos los argumentos sean vulnerables, pero habrá que comprobarlo. El orden aplicado entre una

tarea y otra es importante, ya que es necesario primero buscar las fuentes y luego los sumideros, ya que si se hiciera al revés, podría ser que se detectara que la llamada a función es un sumidero, pero que no hay peligro porque no hay ninguna variable ”manchada”, para posteriormente, en la búsqueda de fuentes, detectar que alguna variable sí que está ”manchada”. Esta situación nos llevaría a una tasa mayor de falsos negativos, por lo que el orden a aplicar tiene que ser primero buscar fuentes y luego sumideros. Este paso, junto a otros, hace referencia a la línea 8 del algoritmo lista de *Kildall*.

d) Controlar las declaraciones y asignaciones de variables. Al igual que sucede con las funciones, las asignaciones a variables necesitan un tratamiento diferenciado. Algo a tener en cuenta es que una variable se puede expresar de una manera simple (e.g. *int i; i = 0;*) o compleja (e.g. *struct Point2D p; p.x=1;*), y en la presente implementación se diferencia el procesamiento que se hace de una del que se hace de la otra. La principal diferencia se aplica sobre las variables complejas, y es que no se va a hacer una distinción de entre todas las posibles variables que puede contener, por ejemplo, una estructura, sino que cuando se detecte alguna de estas variables con un estado ”manchado”, se aplicará dicho estado de ”manchado” a toda la estructura, no solo a parte de ella (se trata de una generalización que simplifica el procesamiento de las asignaciones). El procesamiento que se hace sobre las declaraciones y asignaciones de variables es asignar un valor de ”mancha”, si es que no tiene uno (esto sucederá siempre con las declaraciones, tenga o no tenga una asignación), el cual puede ser positivo (i.e. ”T” o ”MT”) o negativo (”NT” o ”UNK”), o actualizar el estado de ”mancha” en función del valor asignado. Una vez inicializado o actualizado el valor de ”mancha” de la variable, se comprueba si la variable es o ha sido modificada a través de una fuente, lo cual también puede actualizar su estado de ”mancha”. Este paso, junto a otros, hace referencia a la línea 8 del algoritmo lista de *Kildall*.

e) Una vez llegado a este punto, ya tenemos los valores que representan un estado completo dentro del análisis que estamos realizando. La razón por la que tenemos un estado completo es porque ya hemos obtenido los estados de ”mancha” de las variables. A partir de la instrucción que estamos analizando, la información de ”mancha” de todas las variables y del resultado actual del análisis, creamos una representación del estado actual del análisis y lo almacenamos. El objetivo de ello es saber cuándo estamos en una iteración que no nos está aportando nada de información nueva, lo cual significaría no añadir las instrucciones que son dependencia de la actual instrucción bajo análisis a la lista de trabajo. Con esto, conseguimos no caer en bucles infinitos, ya que si no se realizara un seguimiento del estado, no podríamos comprobar cuándo no debemos insertar una dependencia de una instrucción a la lista de trabajo. Este paso hace referencia a la línea 10 del algoritmo lista de *Kildall*.

f) Obtenemos las dependencias de la actual instrucción bajo análisis, lo cual se hace a través del CFG. Algo a mencionar es que el algoritmo lista de *Kildall* está pensando para ser ejecutado sobre un CFG que trabaje, como mínimo, a un nivel en el que se representen instrucciones completa, y como ya se ha comentado anteriormente, el módulo *CFG* funciona a nivel de nodo de AST. Esto ha sido una de las mayores dificultades a resolver, ya que ha sido necesario hacer una construcción de nodos más complejos que representen una instrucción completa. Una vez se consigue una representación de la función a partir de nodos que representan instrucciones completas, la búsqueda de nodos que sean dependencia de una instrucción completa se ha realizado buscando entre las dependencias de todos los nodos de la instrucción completa y buscando entre todas las instrucciones completas para encontrar cuáles contenían dichos nodos que son dependientes. Por cada instrucción completa que es dependencia de la que está bajo análisis (este paso en concreto hace referencia a la línea 9 del

algoritmo lista de *Kildall*), se aplican los siguientes pasos:

1) Se comprueba si se tiene información de las instrucciones que son dependencias de la principal bajo análisis. En caso de no tener información, se realiza una inicialización de los valores de las instrucciones. Este paso no hace referencia a ninguna línea en concreto del algoritmo lista de *Kildall*, pero es necesario para poder proceder con los datos correctamente inicializados.
2) Por cada estado de ”mancha” que haya sido calculado en la presente iteración, se comprueba si dichos cambios resultan en nueva información acerca del estado anterior de ”mancha” de las variable que es dependencia. En el caso de que la información calculada sea nueva (e.g. el estado de ”mancha” almacenado de la dependencia es ”UNK” y en la presente iteración, con los datos procesados, se ha detectado que ha cambiado su estado de mancha a ”T”), se actualizan los nuevos datos de la dependencia. Además, si se actualizan los datos, eso quiere decir que la dependencia aún puede volver a cambiar de estado, ya que no ha alcanzado un estado estable, por lo que se añadirá a la lista de trabajo. En el caso de que no se detecte ningún cambio en la dependencia, aún es posible que dicha dependencia se añada a la lista de trabajo, y esto será así comprobando si el estado completo en el que nos encontramos ya ha sido visitado antes, es decir, si hemos entrado en un bucle de estados, en cuyo caso, no se añadirá la dependencia a la lista de trabajo. Si por el contrario estamos en un estado completo no alcanzado previamente, significa que puede que el estado de ”mancha” de la dependencia cambie si volvemos a llegar a la dependencia a través de otro camino, o incluso repitiendo el mismo, por lo que sí que se añadirá a la lista de trabajo. La razón de no añadirse si estamos en un estado completo visitado previamente es que se ha detectado un bucle de estados con la dependencia, lo que quiere decir que, si se vuelve a añadir a la lista de trabajo, volveremos a repetir el mismo bucle de estados indefinidamente cada vez que lleguemos a procesar la misma dependencia. Esta paso hace referencia a las líneas 10, 11, 12 y 13 del algoritmo lista de *Kildall*.
3) En los casos en los que las dependencias se añadan a la lista de trabajo, se realizará un procesamiento especial de los bloques *if-else* a la hora de añadirse a la lista de trabajo. El resto de instrucciones no necesitan ningún procesamiento especial a la hora de añadirse a la lista de trabajo, simplemente se añadirán como si de una lista normal se tratara, pero con los bloques *if-else* tenemos una limitación. Resulta que cuando se está procesando un bloque *if-else* tenemos dos bloques, los cuales son la parte del bloque *if* y la parte del bloque *else*. Este simple hecho complica en gran cantidad el procesamiento del bloque *if-else*, ya que cuando el *taint analysis* está recorriendo el CFG en un bloque *if-else*, cuando termina alguno de los dos bloques, la dependencia de esta última instrucción será la primera instrucción del final del bloque *if-else*, lo que quiere decir que se habrá analizado por completo el bloque *if* o el bloque *else* de manera exclusiva, pero no el otro, posponiendo el procesamiento del análisis para el otro bloque hasta que todo el *taint analysis* haya finalizado. Esto es un problema, pues modifica por completo los resultados obtenidos del análisis. La solución a ello es detectar en las dependencias cuándo tenemos un elemento *EnfOfIfElse*, el cual indica que se va a terminar un bloque *if* o un bloque *else*, y en lugar de insertar dicha dependencia a la lista de la manera habitual, realizamos un procesamiento concreto. Este procesamiento concreto consiste en buscar el elemento *If* cuyo bloque *if* o bloque *else* contenga en su cuerpo el nodo *EndOfIfElse* que estamos analizando como dependencia. En el caso de no encontrarlo, el cual es el comportamiento por defecto pero que no debería de ocurrir, será el de añadir el nodo al final de la lista en lugar de añadirlo al principio. Si se encuentra el elemento *If*, cuyo cuerpo

contiene la dependencia de tipo *EndOfIfElse*, entonces se busca en la lista de trabajo el primer elemento que no pertenezca al cuerpo del elemento *If*, con el objetivo de añadir la dependencia justo antes de esa instrucción que no pertenece al cuerpo del elemento *If*. La razón de aplicar este comportamiento es conseguir que, en lugar de que la siguiente instrucción a ejecutar de la lista de trabajo sean las dependencias de la instrucción de tipo *EndOfIfElse*, lo cual nos llevaría a no ejecutar el otro bloque del bloque *if-else* como ya hemos comentado. En su lugar, se ejecutarán el resto de instrucciones de la lista de trabajo, las cuales serán las del otro bloque del bloque *if-else* u otras instrucciones, pero el orden será el adecuado y no se realizará un orden que dejará instrucciones por analizar pasando a instrucciones posteriores que no están reflejadas en el flujo de control de CFG. El único caso especial con el que sucede esto son con los bloques *if-else*, y son los únicos que no se insertan tal cual se indica en el algoritmo lista de *Kildall*. Este paso no hace referencia a ninguna línea del algoritmo lista de *Kildall*, más bien es un ajuste necesario debido a la semántica del lenguaje de programación que estamos tratando.

Toda la explicación esencial ha quedado resuelta en los pasos anteriores, los cuales indican la manera en la que el algoritmo lista de *Kildall* se aplica con la finalidad de resolver la técnica del análisis de manchas o *taint analysis*. Además, todos los problemas principales los cuales se han encontrado conforme se realizaba la implementación, han sido desarrollados junto a la solución de los mismos. La manera en la que esta técnica ha sido implementada ha sido siguiendo una teoría formal, la cual ha resultado en unos resultados satisfactorios.

En la figura 7.6 se puede observar un ejemplo de la ejecución de este módulo en formato HTML, junto al fichero de código utilizado. Los resultados son perfectos para el caso que se muestra, el cual es un caso sencillo, pero demuestra el potencial de la técnica. Es interesante como la variable *z* obtiene el estado de ”manchada” en la línea 10 debido al bloque *if-else*, el cual tiene el estado de ”manchado” debido a que se hace uso de la variable *y*, la cual se ”manchó” en la línea 7 debido a inicializar la variable asignado el valor de la variable *x*, la cual se ”manchó” en la línea 5 al utilizarse en la función *gets()*, la cual está definida como una fuente en el archivo de reglas. A pesar de tratarse de un ejemplo sencillo, es más complejo que los vistos hasta el momento, y la detección de la línea donde se manchan las variables se realiza perfectamente.

Limitaciones

La técnica del análisis de manchas o *taint analysis* ha sido muy empleada a lo largo de los años. A diferencia de otras técnicas más agresivas, esta técnica intenta mantener un equilibrio entre la robustez y la completitud (i.e. nivel de falsos positivos y falsos negativos equilibrado). La principal limitación de esta técnica es el problema conocido como ”*overtainting*”. Este problema pone al descubierto el inconveniente de una herencia del estado de ”mancha” demasiado agresivo, lo cual nos lleva a situaciones donde todos los contenedores quedan ”manchados”. Esta situación no es la ideal, ya que la cantidad de falsos positivos aumenta sobremanera. La manera de solucionarlo consiste en aplicar técnicas contra el ”*overtainting*”, las cuales consisten en realizar un análisis menos agresivo y más selectivo a la hora de asignar el estado de ”mancha”. También hay que tener en cuenta que abusar de estas técnicas conduciría a unos resultados donde los falsos negativos serían el principal problema, ya que si los criterios para que una variable esté ”manchada” son demasiado exigentes, hasta las situaciones más sencillas de detectar un estado de ”mancha” se dificultarían debido al mal uso de estas técnicas. Algunas de estas técnicas se basan en no simplificar las estructuras de datos complejas (solo se ”mancha” la parte concreta de la estructura de datos que se ha detectado y no toda la estructura). En la presente implementación, no se ha aplicado ninguna técnica contra el ”*overtainting*”.

```c
void main()
{
    char x[20];

    gets(x);

    char* y = x;
    int z;

    if (y == 0)
    {
        z = 0;
    }
    else
    {
        z = 1;
    }

    system(z);
}
```

(a) Código del fichero analizado para aplicar la técnica *taint analysis*

BOA - Report

boam_taint_analysis.BOAModuleTaintAnalysis

Who	Row	Column	Severity	Description	Advice
Threat	19	5	ALERT	function 'main': a sink (function 'system') with a tainted value has been found, in the parameter with position '1' (the first parameter starts with 1)	try to avoid that the user has access to information which could reach dangerous functions unless that is your goal
Threat	3	10	INFORMATIONAL	function 'main': variable 'x' is tainted with status 'T' which means that the variable is, if is not a false positive, tainted	None
Threat	7	9	INFORMATIONAL	function 'main': variable 'y' is tainted with status 'T' which means that the variable is, if is not a false positive, tainted	None
Threat	10	5	INFORMATIONAL	function 'main': variable 'z' is tainted with status 'T' which means that the variable is, if is not a false positive, tainted	None
Total threats:					**4**

Summary

Property	Value
Total threats	4

(b) Ejecución del módulo *Taint Analysis* (formato del informe HTML)

Figura 7.6: Ejecución de un módulo de seguridad de BOA (módulo *Taint Analysis*)

Capítulo 8

Resultados

Las pruebas a realizar sobre un sistema son importantes, pues nos dicen cómo de efectivo es el producto. Con la finalidad de evaluar los módulos de seguridad implementados para BOA, los cuales se comentó que tenían la finalidad de aportar una funcionalidad mínima, se va a realizar una serie de pruebas. Las pruebas a realizar se van a basar en unas pocas pruebas sobre ficheros de código C sintéticos (creados con el propósito específico de probar BOA) y sobre un gran conjunto de datos (*dataset*) de casos reales.

Las pruebas se van a realizar sobre los módulos de seguridad *Function Match* y *Taint Analysis* (este último hace uso del módulo *CFG*). La vulnerabilidad a detectar por ambos módulos es *Buffer Overflow*. Para este fin, ambos módulos tienen una configuración concreta en sus archivos de reglas. En el caso del módulo *Function Match* está configurado para detectar las siguientes funciones: *gets*, *printf*, *sprintf*, *vsprintf*, *strcpy*, *strncpy*, *strcat*, *strncat* y *strcmp*. Todas las funciones citadas tienen en mayor (e.g. *gets*) o menor (e.g. strncpy) medida cierto riesgo de contener un *Buffer Overflow*. En el caso del módulo *Taint Analysis*, tiene una serie de fuentes y sumideros configurados, los cuales son los siguientes:

- Fuentes: variable *argc* en la función *main()*, variable *argv* en la función *main()*, función *gets()* que afecta a la asignación y parámetros, función *strcpy()* que afecta a la asignación y primer argumento si el segundo argumento está ”manchado”, función *strncpy()* que afecta a la asignación y primer argumento si el segundo argumento está ”manchado”, función *sprintf()* que afecta a la asignación y primer argumento si el segundo argumento está ”manchado”, función *vsprintf()* que afecta a la asignación y primer argumento si el segundo argumento está ”manchado”, función *strcat()* que afecta a la asignación y primer argumento si el segundo argumento está ”manchado”, función *strncat()* que afecta a la asignación y primer argumento si el segundo argumento está ”manchado”, función *strcmp()* que afecta a la asignación si el primer o segundo argumento está ”manchado”, función *getenv()* que afecta a la asignación, función *fopen()* que afecta a la asignación, función *scanf()* que afecta a los argumentos a partir del segundo (en concreto, se comprueba hasta el sexto argumento), función *fscanf()* que afecta a los argumentos a partir del tercero (en concreto, se comprueba hasta el séptimo argumento), función *fgets()* que afecta a la asignación y al primer argumento, función *fgetc()* que afecta a la asignación, función *fread()* que afecta al primer parámetro, función *getline()* que afecta al primer parámetro.
- Sumideros: función *strcpy()* que afecta al segundo parámetro, función *strncpy()* que afecta al tercer parámetro, función *system()* que afecta a todos sus parámetros, función *printf()* que afecta al primer

parámetro, función *fprintf()* que afecta al segundo parámetro, función *sprintf()* que afecta al segundo parámetro, función *vsprintf()* que afecta al segundo parámetro, función *getenv()* que afecta al primer parámetro, función *getline()* que afecta al segundo parámetro, función *fopen()* que afecta al primer parámetro, función *fgets()* que afecta al segundo parámetro, función *fread()* que afecta al segundo y tercer parámetro, función *strcat()* que afecta al segundo parámetro y función *strncat()* que afecta al tercer parámetro.

La configuración descrita para los módulo *Function Match* y *Taint Analysis* está dirigida a detectar *Buffer Overflow*, pero es muy sencillo realizar otras configuraciones dirigidas a otras vulnerabilidades, pues las técnicas implementadas en estos módulos no son específicas de esta vulnerabilidad, sino que es posible detectar otras muchas vulnerabilidades, como puede ser las condiciones de carrera (*race condition*), ya que hay un serie bien definidas que se sabe que sufren de este problema. Otra posibilidad para el módulo *Taint Analysis* es aplicarlo directamente sobre un ámbito de actuación más amplio, lo cual requeriría ampliar las fuentes y sumideros con el objetivo de que el análisis efectuado fuera más general y no tan concreto hacia la vulnerabilidad *Buffer Overflow*.

8.1 Pruebas

El objetivo de realizar pruebas sobre un pequeño conjunto de pruebas sintéticas es la de poder evaluar las fortalezas y debilidades de los módulos que están siendo evaluados. El diseño de las pruebas se ha realizado teniendo esto en mente para que se pueda observar con facilidad en los resultados.

Las pruebas se han realizado sobre 4 ficheros de código distintos, código de los cuales se puede observar en la figura 8.1. Los resultados de los mismos se pueden observar en la figura 8.2, 8.3, 8.4 y 8.5. A continuación, detallamos la información relevante de las pruebas (figura 8.1):

- Prueba #1: prueba básica de una vulnerabilidad *Buffer Overflow* a través de la función *strcpy()*.
- Prueba #2: prueba más elaborada que la prueba #1. Contiene una vulnerabilidad *Buffer Overflow* a través de una función *strcpy()*, pero hay 2 llamadas a dicha función, siendo una vulnerable y la otra no. A través de esta prueba podremos comprobar los falsos positivos.
- Prueba #3: esta prueba contiene una ejecución a la función *system()* en función de un parámetro manejado por el usuario, cuestión ya preocupante de por sí, pero puede ser el objetivo del programa. Obviando la llamada a *system()*, la cual se realiza a través de 2 funciones, también tenemos un *Buffer Overflow* debido a un mal uso de la función *strncpy()*. En esta prueba hay un *Buffer Overflow* basado en el *Heap*, no en el *Stack*, pero la metodología, en este caso, es como la aplicada a los que están basados en el *Stack*, por lo que es detectable de igual manera.
- Prueba #4: esta prueba contiene el mismo comportamiento que el de la prueba #3, con la diferencia de que ahora todo el comportamiento está contenido en la función *main()*. La principal diferencia entre la prueba #3 y la #4 va a estar en los resultados, ya que a diferencia de la prueba #3, esta prueba sí que va a arrojar resultados positivos debido a que no hay información interproceso. Este echo lleva a que un análisis intraprocedural, el cual es el que efectuamos en todo momento, sea más efectivo.

```c
int main(int argc, char** argv)
{
    char name[20];

    if (argc > 1)
    {
        strcpy(name, argv[1]);
    }
    else
    {
        strcpy(name, "Master Chief");
    }

    printf("Hello, %s.\n", name);

    return 0;
}
```

(a) Prueba #1: *test_basic_buffer_overflow.c*

```c
int main(int argc, char** argv)
{
    int a = argc;    // T
    int b = 2;       // NT

    for (int i = b; i < 10; i++) // T
    {
        if (a)  // T
            a = b;  // still T
        if (b)  // NT
            a = b;  // from T to NT!
    }

    char* d = argv[0];
    char* e = d;
    char f[5] = "Halo 3\0";
    char copy[20];
    strcpy(copy, e);
    strcpy(copy, f);

    return 0;
}
```

(b) Prueba #2: *test_taint_control_flow_structures.c*

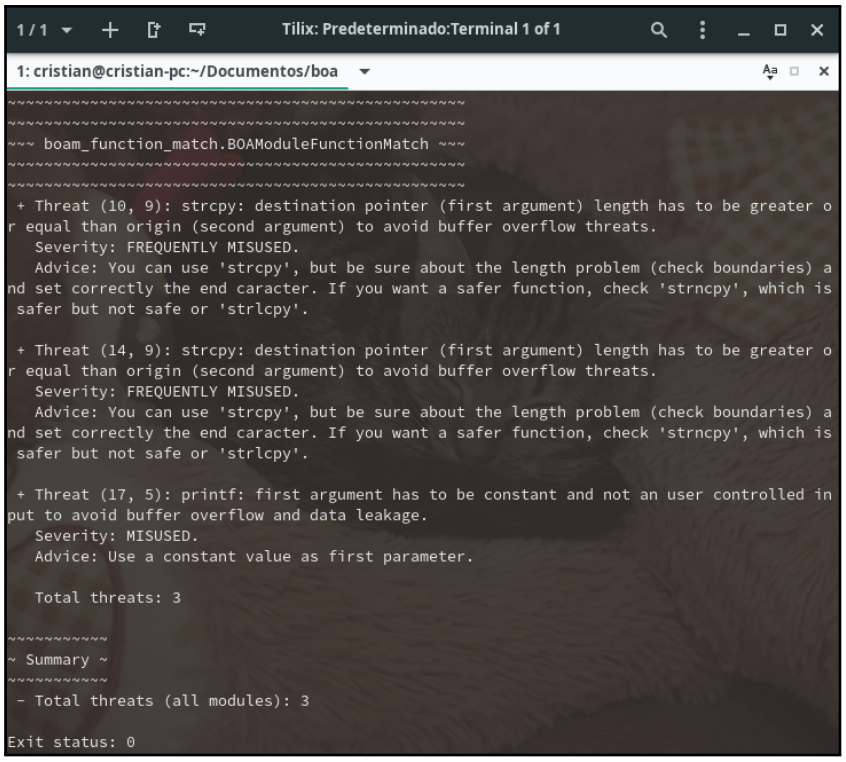
```c
int execute_command(char* command, int
     size)
{
    char* buffer = (char*)malloc(size
        * sizeof(char));
    strncpy(buffer, command, strlen(
        command));

    system(buffer);
}

int main(int argc, char** argv)
{
    if (argc != 3)
        return 1;

    char* size = argv[1];
    int command_size = atoi(size);

    execute_command(argv[2],
        command_size);

    return 0;
}
```

(c) Prueba #3: *test_buffer_overflow_dyn_mult_funcs.c*

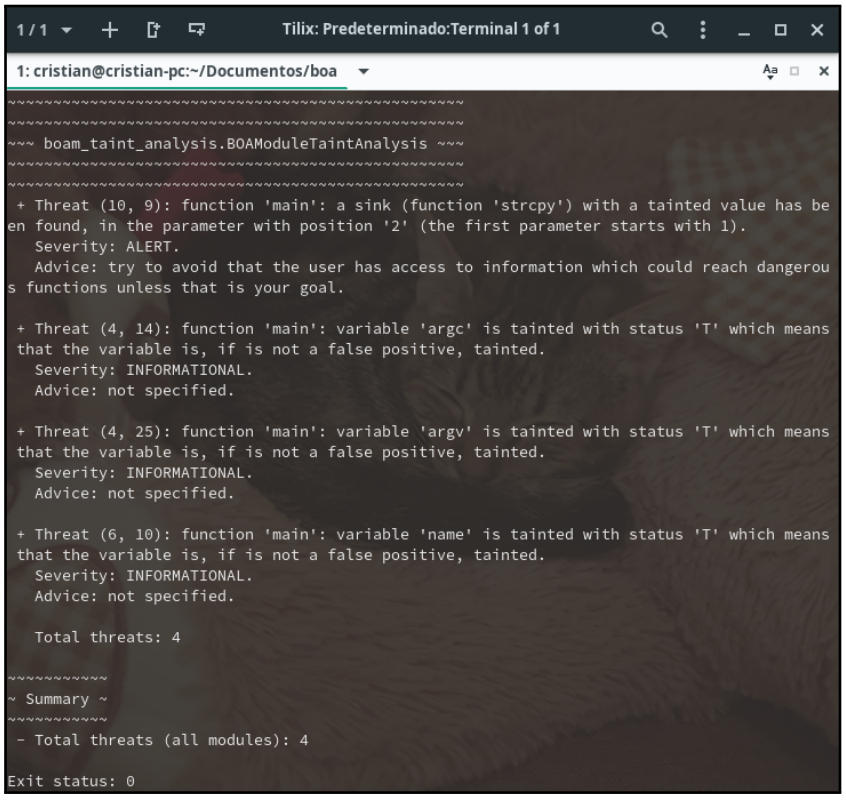
```c
int main(int argc, char** argv)
{
    if (argc != 3)
        return 1;

    char* size = argv[1];
    int command_size = atoi(size);
    char* command = argv[2];
    char* buffer = (char*)malloc(
        command_size * sizeof(char));
    strncpy(buffer, command, strlen(
        command));

    system(buffer);

    return 0;
}
```

(d) Prueba #4: *test_buffer_overflow_dyn_single_func.c*

Figura 8.1: Pruebas sintéticas (código)

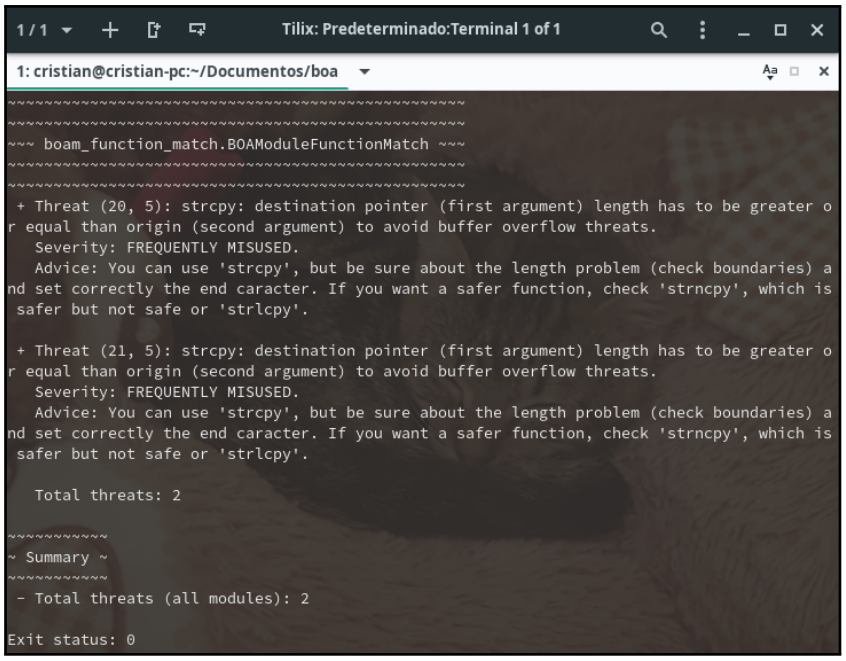


(a) Módulo *Function Match*

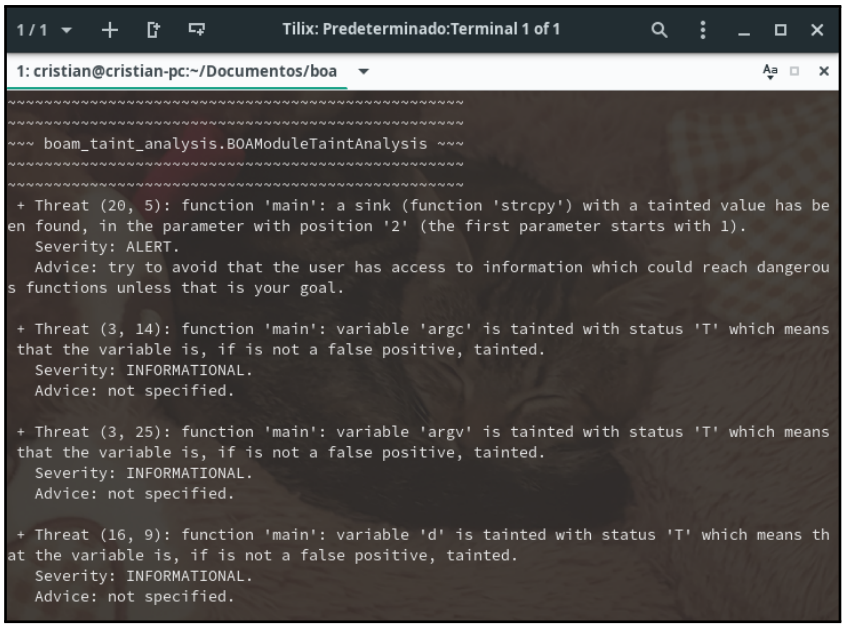


(b) Módulo *Taint Analysis*

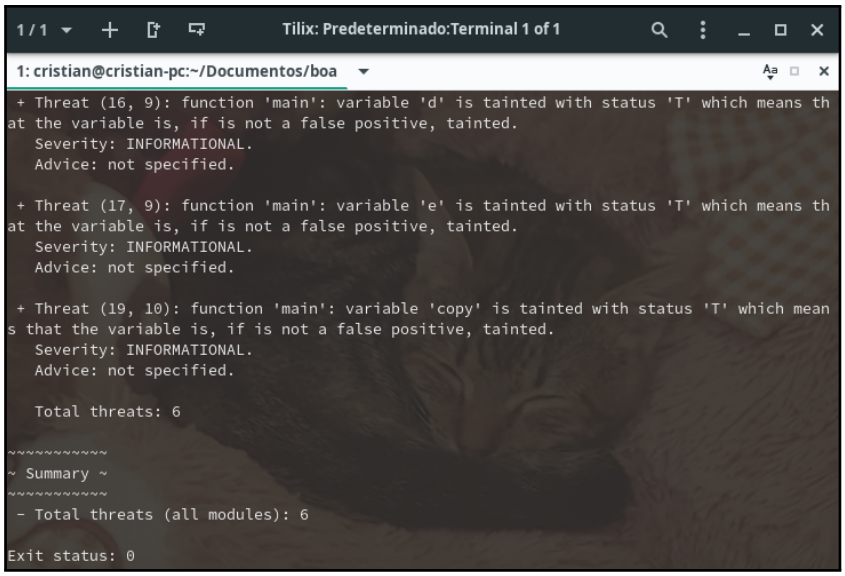
Figura 8.2: Resultados prueba sintética #1



(a) Módulo *Function Match*

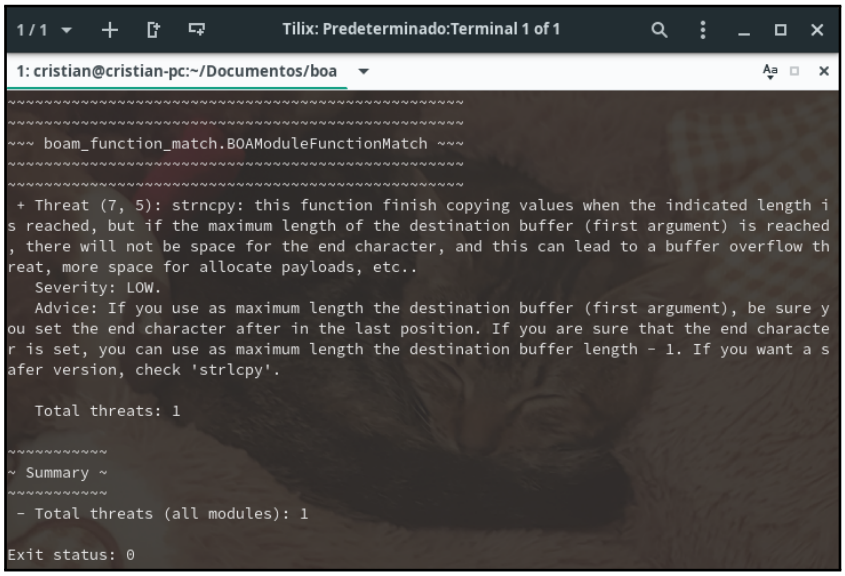


(b) Módulo *Taint Analysis* (1)

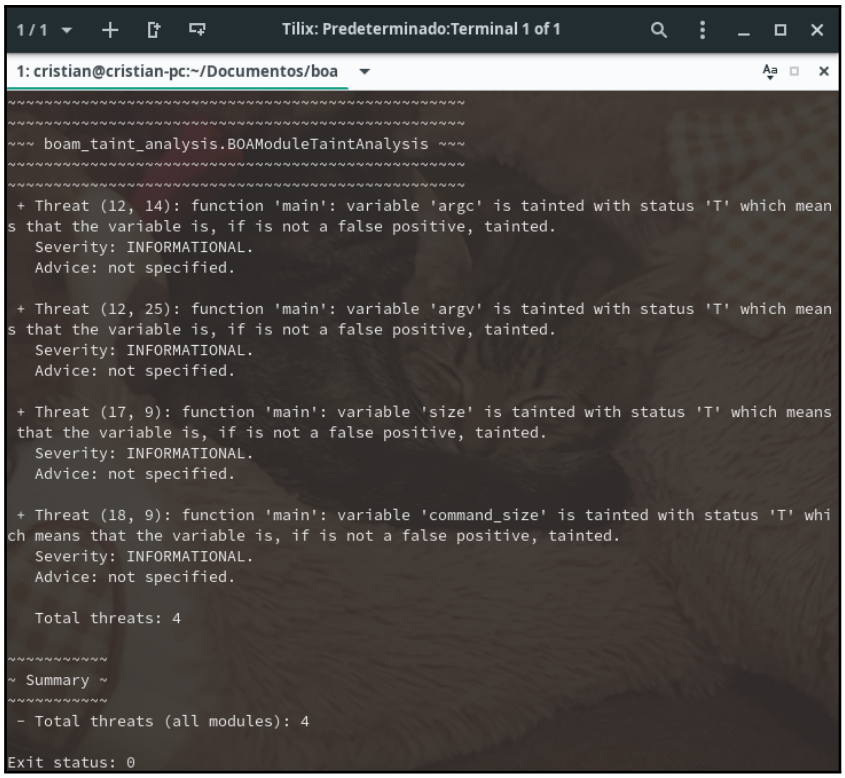


(c) Módulo *Taint Analysis* (2)

Figura 8.3: Resultados prueba sintética #2

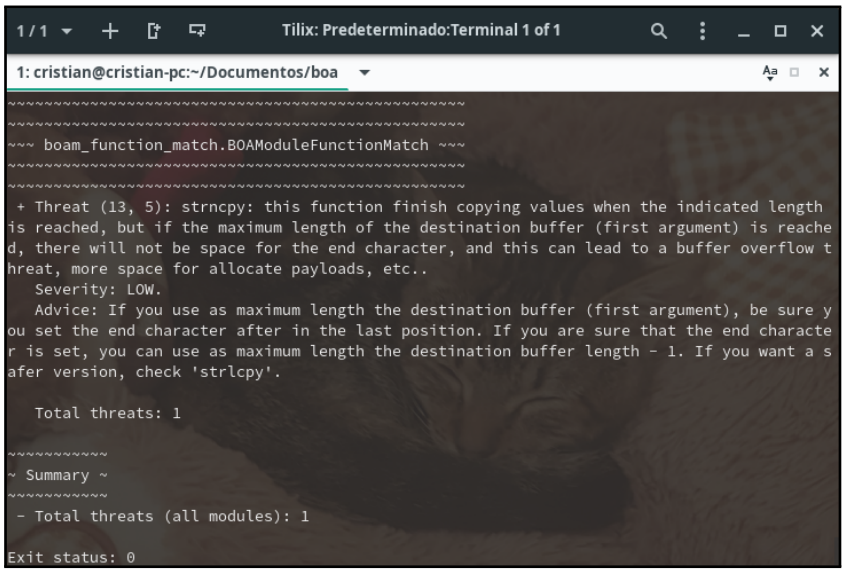


(a) Módulo *Function Match*

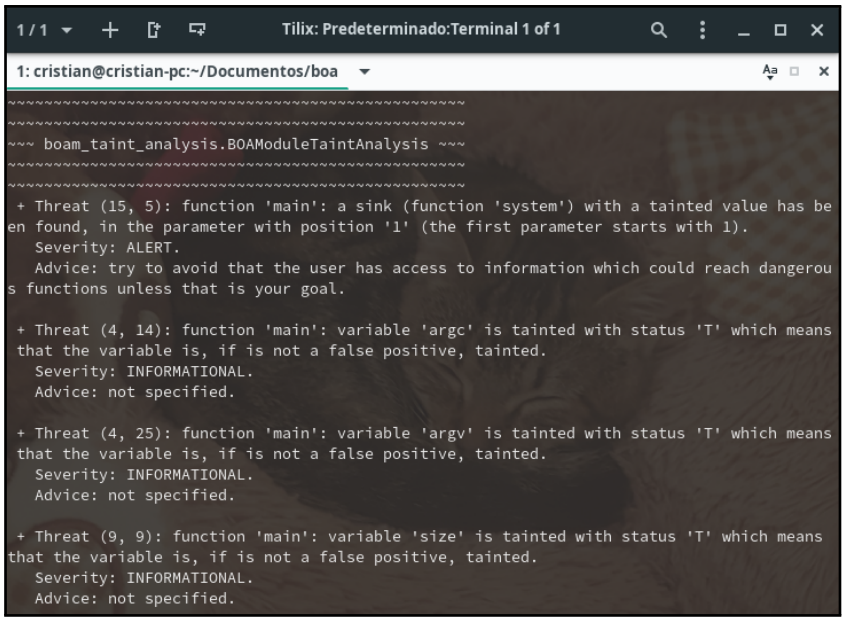


(b) Módulo *Taint Analysis*

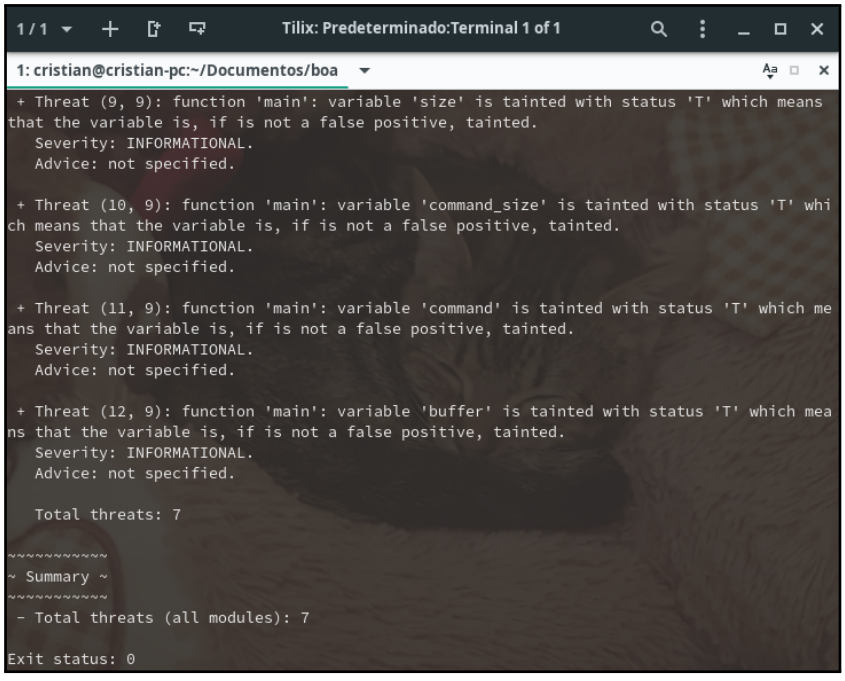
Figura 8.4: Resultados prueba sintética #3



(a) Módulo *Function Match*



(b) Módulo *Taint Analysis* (1)



(c) Módulo *Taint Analysis* (2)

Figura 8.5: Resultados prueba sintética #4

Una vez vistos los resultados, las conclusiones y hechos relevantes son los siguientes:

- Resultados prueba #1 (figura 8.2): se detecta la vulnerabilidad en ambos módulos.
  - Módulo *Function Match*: al tratarse de una técnica agresiva, se detecta la vulnerabilidad (línea 10) y se detectan varios falsos positivos (líneas 14 y 17).
  - Módulo *Taint Analysis*: se detecta solo la vulnerabilidad (línea 10) y no hay falsos positivos (línea 14). Además, se detectan correctamente los estados de ”mancha” de las variables.
- Resultados prueba #2 (figura 8.3): se detecta la vulnerabilidad en ambos módulos.
  - Módulo *Function Match*: al tratarse de una técnica agresiva, se detecta la vulnerabilidad (línea 20) y se detecta un falso positivo (línea 21).
  - Módulo *Taint Analysis*: se detecta solo la vulnerabilidad (línea 20) y no hay falsos positivos (línea 21). Además, se detectan correctamente los estados de ”mancha” de las variables. Se destaca el estado de ”mancha” de la variable ”a”, la cual se ”mancha” y luego pasa a un estado ”limpio” debido al recorrido del bucle *for*.
- Resultados prueba #3 (figura 8.4): solo el módulo *Function Match* detecta la vulnerabilidad.
  - Módulo *Function Match*: detecta la vulnerabilidad y no detecta falsos positivos.
  - Módulo *Taint Analysis*: no se detecta la vulnerabilidad debido a que el estado de mancha no se hereda entre funciones (i.e. no se realiza un análisis interprocedural). Si se tiene en cuenta que el análisis realizado es intraprocedural, la detección de los estados de ”mancha” son correctos.
- Resultados prueba #4 (figura 8.5): se detecta la vulnerabilidad en ambos módulos.
  - Módulo *Function Match*: detecta la vulnerabilidad y no detecta falsos positivos.
  - Módulo *Taint Analysis*: a diferencia de los resultados de la prueba #3, ahora está toda la información en una misma función, con lo que al realizar el análisis intraprocedural, se detecta la vulnerabilidad correctamente. Además, se detectan correctamente los estados de ”mancha” de las variables.

8.2 Casos Reales

Para la realización de las pruebas sobre casos reales se ha hecho uso del *dataset* SARD[1]. Este *dataset* consta de un gran conjunto de elementos con ficheros de código que contienen vulnerabilidades *software*. El *dataset* contiene multitud de lenguajes de programación y multitud de vulnerabilidades. A través de su buscador[2] se pueden especificar diferentes atributos para evitar tener que descargar todo el *dataset*, pues en su totalidad tiene un tamaño del orden de los *gigabytes*.

El subconjunto que se ha utilizado para realizar las pruebas con BOA han sido ficheros escritos en lenguaje de programación C, los cuales contenían resultados con vulnerabilidades y sin ellas (la cantidad de elementos que, o no contenían vulnerabilidades, o solo contenían vulnerabilidades, es despreciable, ya que en su mayoría contienen la vulnerabilidad y la solución a la misma en el mismo fichero), y cuya vulnerabilidad

[1] Sitio oficial de SARD: `https://samate.nist.gov/SARD/`
[2] Buscador de SARD: `https://samate.nist.gov/SARD/search.php`

está clasificada según el código CWE-120[3] (en el buscador, este código también hacia referencia a los códigos CWE-121[4] y CWE-122[5], por lo que estos códigos también pertenecen al subconjunto de resultados). Los códigos CWE (*Common Weakness Enumeration*) son una clasificación de las vulnerabilidades más comunes, y las que están presentes en el subconjunto de elementos que han sido utilizados en las pruebas con casos reales significan lo siguiente:

1. CWE-120: vulnerabilidad *Buffer Overflow* clásica.
2. CWE-121: vulnerabilidad *Buffer Overflow* basada en el *Stack*.
3. CWE-122: vulnerabilidad *Buffer Overflow* basada en el *Heap*.

La cantidad total de casos concretos de las vulnerabilidades utilizados para las pruebas ha sido de 7182, de los cuales 4342 contienen una vulnerabilidad CWE-121 (solo 27 de ellos no tienen vulnerabilidades) y 2840 contienen una vulnerabilidad CWE-122 (solo 15 de ellos no tienen vulnerabilidades). Un 99.4 % de los ficheros contienen vulnerabilidades, por lo que el otro 0.6 % es despreciable. De los 7182 casos, solo hubo problemas con aquellos que hacían uso de ciertas librerías para las cuales no se tenían las cabeceras (”*apr.h*”, ”*glib.h*”, ”*apr_pools.h*”, ”*config.h*”, ”*epan/timestamp.h*”, ”*postgres.h*”, ”*gtk/gtk.h*”, ”*tree.h*”, ”*column-utils.h*”, ”*svn_private_config.h*”, ”*cryptlib.h*”, ”*evp_locl.h*”, ”*avcodec.h*”, ”*apr_general.h*”, ”*apr_strings.h*”, ”*avformat.h*”, ”*timestamp.h*”, ”*avdevice.h*”, ”*apr_uri.h*”, ”*audio.h*”, ”*apr_want.h*” y ”*eng_int.h*”), por lo que se descartaron (se podría haber creado las cabeceras de las librerías, pero el número de errores era despreciable). Debido a que hay casos en los que para una misma vulnerabilidad concreta se aportan varios ficheros vulnerables, la cantidad total real de ficheros utilizados para las pruebas ha sido de 9660.

Los resultados al analizar los ficheros se pueden observar en las figuras 8.6 (resultados generales) y 8.7 (cantidad de vulnerabilidades detectadas por fichero). Suponiendo que aquellos resultados que detectan alguna vulnerabilidad, detectan la vulnerabilidad del fichero, la cantidad de vulnerabilidades detectadas por los módulo *Function Match* y *Taint Analysis* es bastante pobre. En el primer módulo se detectan vulnerabilidades en 1260 ficheros, lo que equivale a una detección del 13 %, lo que deja un 87 % de ficheros en los que no se ha detectado ninguna vulnerabilidad. En el segundo módulo se detectan vulnerabilidades en 12 ficheros, lo que equivale a una detección del 0.1 %, lo que deja un 99.9 % de ficheros en los que no se ha detectado ninguna vulnerabilidad.

La razón detrás del porqué de los resultados tiene que ver con la manera en la que se está intentando detectar la vulnerabilidad *Buffer Overflow*. Esta vulnerabilidad ha sido muy estudiada, y existen taxonomías que intentan hacer una clasificación de dicha vulnerabilidad en base a una serie de atributos [45][46]. Estas taxonomías están formadas por muchos atributos que intentan clasificar esta vulnerabilidad, y en el caso de los módulos implementados, la mayoría de atributos no se tienen en cuenta. Debido a lo explicado, los resultados obtenidos no son lo esperado, y, además, esta vulnerabilidad se puede presentar de muchas maneras (e.g. llamada a funciones no seguras, cálculo de índices incorrectos superiores o inferiores, *casting* incorrecto de enteros, ...), y los módulos implementados solo buscan algunas de las maneras más comunes en las que se puede presentar, pero no es efectivo en el caso general.

[3]CWE-120: `https://cwe.mitre.org/data/definitions/120.html`
[4]CWE-121: `https://cwe.mitre.org/data/definitions/121.html`
[5]CWE-122: `https://cwe.mitre.org/data/definitions/122.html`

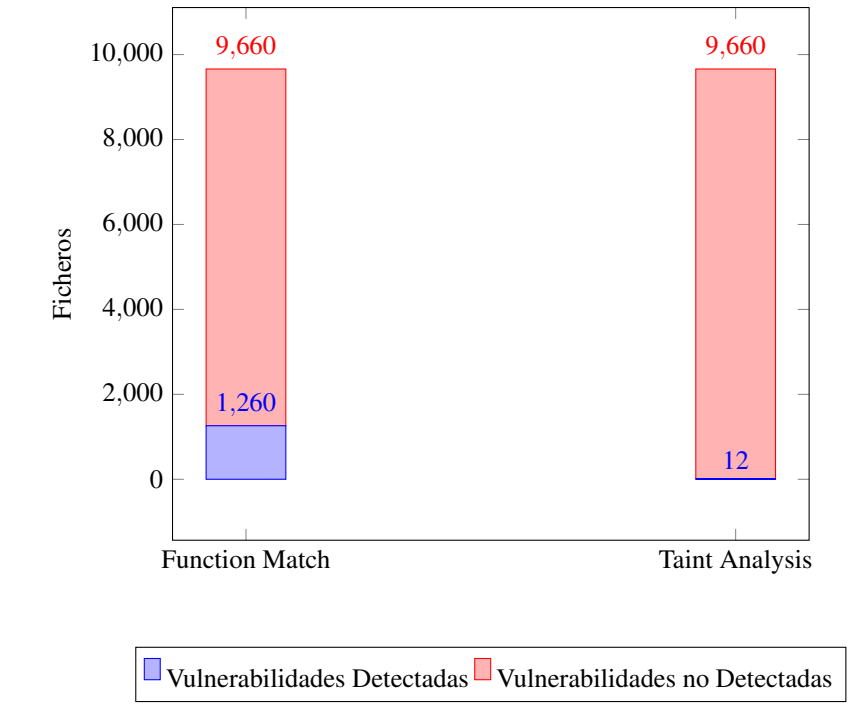


Figura 8.6: Resultados generales de los casos reales

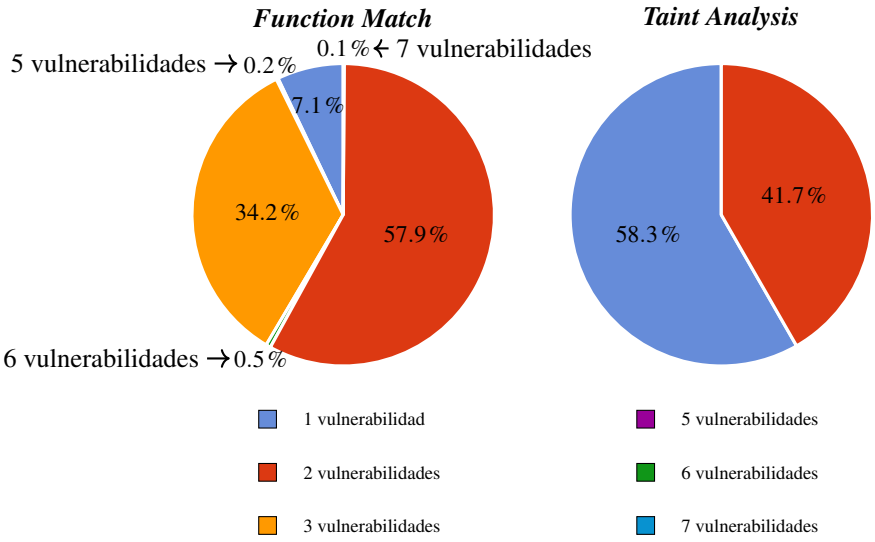


Figura 8.7: Vulnerabilidades por fichero

En el caso del módulo *Function Match*, se intenta, de una manera muy agresiva y basándose en experiencias previas empíricas, detectar la vulnerabilidad. Se hace un gran resumen de todos los atributos que puede tener en cuenta una taxonomía y, debido a ello, puede detectar muchos más resultados, pero mucho más inexactos. Esta manera tan agresiva de intentar detectar un *Buffer Overflow* puede ser muy efectiva a la hora de realizar un análisis de un gran sistema en el cual hayan muchos usuarios involucrados, pues el mal uso o no comprensión total de algunas funciones usuales como *strncpy()*, la cual muchos usuarios consideras segura, lo cual no siempre cierto (no se considera segura debido a que no introduce el carácter de fin de cadena, y tampoco es que tenga que ser utilizada siempre en sustitución de *strcpy()*, ya que hay casos en los que se preferirá *strcpy()* por necesidades de eficiencia), y puede llevar a usuarios a la inserción de vulnerabilidades. También puede ser muy efectiva al analizar un gran conjunto de ficheros, ya que muchas vulnerabilidades pueden salir a la luz con

esta simple técnica. En conclusión, es una técnica rápida y sencilla que puede ser útil en ciertas ocasiones, pero no es una solución ni mucho menos a la detección de cualquier vulnerabilidad *Buffer Overflow*.

Por otro lado, el módulo *Taint Analysis* intenta equilibrar los resultados obtenidos en cuestión de robustez y completitud. Debido a ello, la cantidad de vulnerabilidades detectadas será menor en el caso general, pero será capaz de obtener unos resultados de mayor calidad y fiabilidad. En el casos de una vulnerabilidad *Buffer Overflow* puede ser muy útil, pero a diferencia del módulo *Function Match*, esta técnica no hace un resumen de todos aquellos atributos que pueden identificar a una vulnerabilidad *Buffer Overflow*, sino que hace referencia a solo un subconjunto o incluso solo un atributo que identifique a dicha vulnerabilidad (hay taxonomías que identifican el estado de las variables respecto a si están ”manchadas” por el usuario como un atributo para identificar un *Buffer Overflow* [46]). Los resultados obtenidos por este módulo son mucho menores que en el módulo *Function Match*, pero eso es debido a que el otro módulo actúa de una manera diferente a este, y las pocas vulnerabilidades detectadas por este módulo deberían de tenerse muy en cuenta, pero no son concluyentes.

Teniendo en cuenta que los ficheros analizados por BOA, menos una cantidad despreciable, todos contenían vulnerabilidades, los resultados no son positivos. Las razones han sido explicadas, y otras técnicas en unión a las aplicadas aquí deberían de aplicarse con la finalidad de obtener unos mejores resultados. Hay que tener en cuenta que, a diferencia de las pruebas sintéticas, en estos casos reales se han utilizado ficheros que contienen vulnerabilidades difíciles de detectar, ya que no suelen seguir el esquema más simple y general que se ha explicado a lo largo del presente trabajo. En su lugar, las formas en las que estas vulnerabilidades se presentan en estos casos reales suelen ser, en su mayoría, complejas (e.g. errores de *casting*, errores *off-by-one*, doble *free()*, valores no inicializados). Aún así, a través de la perfección de las técnicas empleadas y con la implementación de módulo alternativos, los resultados podrían mejorar.

Capítulo 9

Conclusión

En el presente trabajo se detalla el desarrollo realizado para el diseño e implementación de un analizador de vulnerabilidades. A partir de una serie de objetivos, se ha realizado el diseño de un sistema de propósito general que permite, a cualquier usuario, hacer uso de herramientas preventivas en materia de ciberseguridad. El resultado obtenido ha sido un analizador simple de utilizar, configurar y potente, pues se puede adaptar fácilmente a las necesidades de cualquiera que lo utilice.

Debido al diseño realizado, se ha obtenido un analizador de propósito general que permite analizar diferentes sistemas. Además, las tecnologías empleadas permiten al usuario interesado comprender el trabajo realizado en el desarrollo y cualquiera interesado en extender el comportamiento base no debería de tener grandes dificultades.

Por último, comentar que, personalmente, el resultado ha sido totalmente satisfactorio. En un corto periodo de tiempo ha sido posible realizar el diseño e implementación de una herramienta que puede dar solución a problemas reales en el campo de la ciberseguridad. Han habido momentos en estos últimos meses en los que se han presentado retos, los cuales han acabado siendo solucionados. El verdadero reto no ha sido la implementación del sistema, sino su diseño, pues es el que verdaderamente ha conducido hacia la solución final.

9.1 Relación con Estudios Cursados

En las actuales competencias específicas de la carrera de Ingeniería Informática no hay ninguna especialización que trate la ciberseguridad en sí, al menos no implementada en la mayoría de universidades, sino que hay asignaturas que dan pinceladas. En lo personal, creo que el campo de la ciberseguridad requiere de una alta comprensión teórica de la computación y práctica en las tecnologías de la información. Debido a ello, y con la finalidad de obtener una formación que cumpla con dichos criterios, he cursado las especialidades de Computación y Tecnologías de la Información en la Universidad de Alicante. En ambas especialidades hay asignaturas que ofrecen una gran formación que creo necesaria para especializarse en ciberseguridad, e incluso necesaria como formación básica para un ingeniero informático. Por un lado, la teoría es importante a pesar de que pueda parecer lo contrario, y debido a ello, asignaturas como Teoría de la Computación, la cual ha sido la

que más me ha gustado sin lugar a dudas a lo largo de la carrera, puede ser vital para comprender de verdad qué es lo que hace que un problema sea computable y saber formalizar un problema como los vistos en el presente trabajo. Por otro lado, la práctica también es esencial, y asignaturas como Gestión e Implantación de Redes de Computadores ayuda a desarrollar los conocimientos técnicos necesarios que un experto en ciberseguridad, personalmente, creo que debería de adquirir a lo largo de su formación.

Tampoco hay que dejar de lado la formación básica, pues multitud de asignaturas cursadas en los primeros años del grado pueden parecer no tener utilidad directa e inmediata, pero la tienen. Ejemplos de ello son asignaturas como Lenguajes y Paradigmas de Programación y Programación de Estructuras de Datos, en las cuales se ven conceptos teóricos y prácticos útiles que han sido utilizados en el presente trabajo. Muchas de estas asignaturas básicas nos ayudan a desarrollar nuestro pensamiento crítico y técnico como ingenieros en informática, y debido a ello, son necesarias. La cantidad de conceptos utilizados para el desarrollo de BOA han sido tanto teóricos como prácticos, y la cantidad de asignaturas que han sido útiles para ello son multitud por no decir que todas ellas, en mayor o menor medida, han aportado algún conocimiento necesario.

9.2 Trabajo Futuro

Como trabajo futuro, se considera realizar una serie de mejoras sobre el analizador. Estas mejoras consideran ampliar el ámbito de actuación sobre los sistemas a analizar, permitiendo analizar no solo ficheros, sino también sistemas complejos. Otras de las posibles mejoras consideradas es ampliar el ámbito del analizador a realizar análisis dinámico, ya que actualmente está muy ligado al análisis estático, pero una herramienta que permita ambos tipos de análisis es posible. Por último, se considera realizar una integración con otras herramientas similares, pues hay una gran cantidad de analizadores que son específicos de vulnerabilidades concretas en un conjunto reducido de lenguajes de programación que podrían ser utilizados directamente desde BOA, obteniendo como resultado un analizador desde el cual poder controlar los resultados obtenidos por otros analizadores junto a módulos propios de BOA.

Bibliografía y Referencias

[1] Larry Boettger. “The Morris worm: How it affected computer security and lessons learned by it”. En: *URL http://www.giac.org/paper/gsec/405/morris-worm-affected-computer-security-lessons-learned/100954* (2000).

[2] Ivan Victor Krsul. *Software vulnerability analysis*. Purdue University West Lafayette, IN, 1998.

[3] Allen Householder, Kevin Houle y Chad Dougherty. “Computer attack trends challenge Internet security”. En: *Computer* 35.4 (2002), sulp5-sulp7.

[4] D Richard Kuhn, Mohammad S Raunak y Raghu Kacker. “An Analysis of Vulnerability Trends, 2008-2016”. En: *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE. 2017, págs. 587-588.

[5] Igor Kononenko. “Machine learning for medical diagnosis: history, state of the art and perspective”. En: *Artificial Intelligence in medicine* 23.1 (2001), págs. 89-109.

[6] Peter Szolovits, Ramesh S Patil y William B Schwartz. “Artificial intelligence in medical diagnosis”. En: *Annals of internal medicine* 108.1 (1988), págs. 80-87.

[7] David Silver y col. “Mastering the game of go without human knowledge”. En: *Nature* 550.7676 (2017), págs. 354-359.

[8] Samuel Melamed y col. “Objective and subjective work monotony: effects on job satisfaction, psychological distress, and absenteeism in blue-collar workers.” En: *Journal of Applied Psychology* 80.1 (1995), pág. 29.

[9] George Ellwood Dieter, Linda C Schmidt y col. *Engineering design*. McGraw-Hill Higher Education Boston, 2009.

[10] Daniel Plerre Bovet y Marco Cesati. *Understanding the Linux Kernel*. O’Reilly Media, 2005.

[11] David Puente Castro. *Linux Exploiting*. 0xWORD, 2013.

[12] Enrique Rando González. *Hacking Web Applications: Client-Side Attacks*. 0xWORD, 2017.

[13] Enrique Rando González, José María Alonso Cebrián y Pablo González Pérez. *Hacking de Aplicaciones Web: SQL Injection*. 0xWORD, 2016.

[14] Flemming Nielson, Hanne Riis Nielson y Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[15] Brian Chess y Jacob West. *Secure Programming with Static Analysis*. Addison-Wesley Educational Publishers Inc, 2007.

[16] Steven Stanley Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[17] Edward Ashford Lee y Sanjit A. Seshia. *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. MIT Press Ltd, 2016.

[18] Armin Biere y col., eds. *Handbook of Satisfiability*. IOS Press, 2009.

[19] Clark Barrett y col. En: *Handbook of Satisfiability*. Ed. por Armin Biere y col. IOS Press, 2009. Cap. Satisfiability Modulo Theories.

[20] Chi-Keung Luk y col. "Pin: building customized program analysis tools with dynamic instrumentation". En: *Acm sigplan notices* 40.6 (2005), págs. 190-200.

[21] Andrew R Bernat y Barton P Miller. "Anywhere, any-time binary instrumentation". En: *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*. 2011, págs. 9-16.

[22] Derek Bruening, Qin Zhao y Saman Amarasinghe. "Transparent dynamic instrumentation". En: *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*. 2012, págs. 133-144.

[23] Istvan Haller y col. "Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations". En: *Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13)*. 2013, págs. 49-64.

[24] Raimondas Sasnauskas y John Regehr. "Intent fuzzer: crafting intents of death". En: *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*. 2014, págs. 1-5.

[25] Greg Banks y col. "SNOOZE: toward a Stateful NetwOrk prOtocol fuzZEr". En: *International Conference on Information Security*. Springer. 2006, págs. 343-358.

[26] DA Andriesse. "Analyzing and securing binaries through static disassembly". En: (2017).

[27] Ari Takanen, Jared D. Demott y Charles Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House Publishers, 2008.

[28] *The Linux Documentation Project: Shared Libraries*. `https://tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html`. Accessed: 2020-04-20.

[29] Michael Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press, Inc, 2010.

[30] R Ravi y col. "A performance analysis of Software Defined Network based prevention on phishing attack in cyberspace using a deep machine learning with CANTINA approach (DMLCA)". En: *Computer Communications* 153 (2020), págs. 375-381.

[31] Zhen Li y col. "A Comparative Study of Deep Learning-Based Vulnerability Detection System". En: *IEEE Access* 7 (2019), págs. 103184-103197.

[32] Zhen Li y col. "Vuldeepecker: A deep learning-based system for vulnerability detection". En: *arXiv preprint arXiv:1801.01681* (2018).

[33] Seulbae Kim y col. "Vuddy: A scalable approach for vulnerable code clone discovery". En: *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2017, págs. 595-614.

[34] *VulDeePeck dataset*. `https://github.com/CGCL-codes/VulDeePecker`. Accessed: 2020-04-20.

[35] *Python PEP: PEP 1*. `https://www.python.org/dev/peps/pep-0001/`. Accessed: 2020-04-24.

[36] *Python PEP: PEP 257*. `https://www.python.org/dev/peps/pep-0257/`. Accessed: 2020-04-25.

[37] *Sphinx: build manual*. `http://www.sphinx-doc.org/es/stable/man/sphinx-build.html`. Accessed: 2020-04-25.

[38] Yusuf Sulistyo Nugroho, Hideaki Hata y Kenichi Matsumoto. "How different are different diff algorithms in Git?" En: *Empirical Software Engineering* 25.1 (2020), págs. 790-823.

[39] Eugene W Myers. "AnO (ND) difference algorithm and its variations". En: *Algorithmica* 1.1-4 (1986), págs. 251-266.

[40] *GNU: Diffutils*. `https://www.gnu.org/software/diffutils/manual/diffutils.html`. Accessed: 2020-04-26.

[41] *Top IDE by Popularity*. `https://pypl.github.io/IDE.html`. Accessed: 2020-04-27.

[42] *Metasploit Framework: Architecture*. `https://www.offensive-security.com/metasploit-unleashed/metasploit-architecture/`. Accessed: 2020-04-30.

[43] Henry Gordon Rice. "Classes of recursively enumerable sets and their decision problems". En: *Transactions of the American Mathematical Society* 74.2 (1953), págs. 358-366.

[44] Gary A Kildall. "A unified approach to global program optimization". En: *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1973, págs. 194-206.

[45] Matt Bishop y col. "A taxonomy of buffer overflow characteristics". En: *IEEE Transactions on dependable and secure computing* 9.3 (2012), págs. 305-317.

[46] Kendra Kratkiewicz y Richard Lippmann. "A taxonomy of buffer overflows for evaluating static and dynamic software testing tools". En: *Proceedings of Workshop on Software Security Assurance Tools, Techniques, and Metrics*. Vol. 500. 2006, págs. 44-51.

Anexos

Anexo A

Códigos de Error de BOA

BOA define una serie de códigos de error con la finalidad de poder guiar al usuario en caso de que el análisis de BOA no tenga éxito por cualquier motivo. Estos códigos de error se utilizan con la llamada de sistema *exit()*, y dicho código de error se muestra antes de finalizar la ejecución del análisis. En la actual versión de BOA, versión 0.3, los códigos de error definidos son los siguientes:

- Códigos de error generales:
  - Código 0: no es un código de error, pero lo indicamos aquí porque es el código que inicia la numeración. Se trata del código que indica que no hay ningún error, el cual es el valor 0 por convención.
  - Código 1: error desconocido.
  - Código 2: fichero no encontrado (error genérico de fichero).
- Códigos de error de los argumentos de entrada:
  - Código 10: error genérico de los argumentos de entrada (e.g. no se introducen todos los argumentos obligatorios).
  - Código 11: formato de los argumentos incorrecto.
- Códigos de error en los módulos (general):
  - Código 200: no se han podido cargar algunos módulos obligatorios y algunos módulos de usuario (al menos uno de ambos).
  - Código 201: no se ha podido cargar algún módulo obligatorio (al menos uno).
  - Código 202: no se ha podido cargar algún módulo de usuario (al menos uno).
  - Código 203: no se ha podido cargar alguna instancia (al menos una).
  - Código 204: error al intentar eliminar un módulo que no ha podido ser cargado.
  - Código 205: no se ha podido cargar una instancia de alguna de las clases abstractas.
  - Código 206: una instancia no está utilizando la clase abstracta que se supone que debería de estar utilizando.

    - Código 207: no ha sido posible inicializar el importador de módulos.

- Códigos de error en los módulos (dependencias):
    - Código 210: se intenta utilizar un módulo que no existe como dependencia.
    - Código 211: se intenta utilizar un módulo como dependencia de sí mismo.
    - Código 212: se detecta una dependencia cíclica.
    - Código 213: no se encuentra algún *callback* de los indicados en las dependencias en el archivo de reglas.
- Códigos de error del archivo de reglas:
    - Código 30: el número de módulos y de clases no coincide.
    - Código 31: el formato empleado para hacer referencia a un módulo y una clase a través de una única cadena de texto no está bien formado y no permite encontrar la referencia.
    - Código 32: no se ha podido abrir el archivo de reglas (e.g. no se tienen los permisos adecuados).
    - Código 33: no se ha podido leer el archivo de reglas.
    - Código 34: no se ha podido cerrar el archivo de reglas.
    - Código 35: el archivo de reglas no cumple con los requisitos sintácticos y/o semánticos especificados por BOA en el análisis del mismo.
    - Código 36: no se encuentran los argumentos para los módulos.
- Códigos de error de los ciclos de vida:
    - Código 40: algún módulo de seguridad ha lanzado una excepción durante la ejecución de un ciclo de vida.
    - Código 41: un ciclo de vida no ha podido finalizar correctamente.
    - Código 42: un ciclo de vida no ha podido encontrar algún módulo cuando se iba a ejecutar.
    - Código 43: no se ha podido cargar la instancia de clase abstracta de los ciclos de vida.
    - Código 44: no se ha podido cargar alguna instancia de ciclo de vida.
    - Código 45: alguna instancia de un ciclo de vida no estaba extendiendo la clase abstracta de los ciclos de vida.
- Códigos de error de los informes (general):
    - Código 500: argumentos obligatorios (módulo que ha localizado una vulnerabilidad, descripción de la vulnerabilidad y nivel de severidad de la vulnerabilidad) sin valor.
    - Código 501: tipo de datos de los argumentos no esperado (e.g. si se especifica la fila donde se ha localizado una vulnerabilidad, el tipo de datos tiene que ser un entero).
    - Código 502: argumento que indica el módulo que ha localizado una vulnerabilidad no cumple con el formato esperado.
    - Código 503: no se ha podido añadir correctamente una entrada al informe.
    - Código 504: error genérico.
    - Código 505: no se ha podido cargar la instancia de la clase abstracta del informe.

    - Código 506: no se ha podido encontrar un módulo de informe.
    - Código 507: alguna instancia de un informe no estaba extendiendo la clase abstracta de los informes.

- Códigos de error de los informes (niveles de severidad):
    - Código 510: no se ha podido establecer una asociación entre un nivel de severidad asignado por el usuario y entre los niveles de severidad disponibles.
    - Código 511: alguna instancia de los niveles de severidad no está extendiendo la instancia base u otra instancia de los niveles de severidad.
    - Código 512: no se ha podido encontrar un módulo de nivel de severidad.
    - Código 513: no se ha podido cargar alguna instancia de nivel de severidad.
- Códigos de error de los módulos de analizadores de lenguajes de programación:
    - Código 60: no se ha podido encontrar un módulo de analizador.
    - Código 61: no se ha podido cargar un módulo de analizador.
    - Código 62: no se ha podido cargar la instancia de la clase abstracta del analizador.
    - Código 63: alguna instancia del analizador del lenguaje de programación no está extendiendo la instancia de la clase abstracta de los analizadores de lenguajes de programación.
    - Código 64: alguno de los *callbacks* definidos en el archivo de reglas no se han podido ejecutar correctamente.
    - Código 65: ninguno de los *callbacks* definidos en el archivo de reglas se han podido ejecutar correctamente.
    - Código 66: el analizador de lenguaje de programación no ha finalizado con éxito el análisis del fichero de código.
    - Código 67: la ejecución del analizador no ha finalizado correctamente.
- Otros códigos de error:
    - Código 1001: se hace uso de alguna palabra clave reservada en algún elemento en el que no está permitido.

Los códigos de error que se han definido para BOA se han definido con la intención de que haya un prefijo común en la numeración que indique una categoría general del error (e.g 32x significa códigos de error relacionados con las dependencias de los módulos, 4x significa códigos de error relacionados con los archivos de reglas). Este prefijo se incrementa en potencias de 10 conforme es necesario para seguir añadiendo más códigos de error. Algo importante a destacar es que algunos de los códigos de error se utilizan en solo un sitio concreto del código de BOA, mientras que otros pueden ser utilizados libremente en cualquier parte del código, lo cual es algo a tener en cuenta a la hora de detectar un error concreto. Algunos de los código de error son muy generales, y habrá que hacer uso del mensaje de error, ya que muchos están personalizados con la finalidad de poder detectar el error fácil y rápidamente.

Debido a que BOA es extensible y fácilmente modificable (e.g. modificación de valores del fichero *constants.py*), esto puede llevar a ciertos errores que se detectan a través de algunos de los códigos de error que se han definido. En el comportamiento por defecto de BOA, alguno de estos errores nunca se verán, y solo se verán si se realiza alguna modificación que conduce a un comportamiento inesperado.

Anexo B

Utilizar Otros Analizadores en BOA

En varias ocasiones se ha comentado que BOA es independiente del lenguaje de programación. En su lugar, hace uso de los módulos relacionados con los analizadores de lenguajes de programación, cuya tarea es encargarse de la interoperabilidad con dichos analizadores de lenguajes de programación. En el presente apéndice se va a mostrar un ejemplo donde se utiliza otro analizador que no es Pycparser.

El analizador de lenguaje de programación que se va a utilizar es Javalang[1], el cual analiza el lenguaje de programación Java. Al igual que con Pycparser, Javalang procesa código Java y nos devuelve estructuras de datos más sencillas de manejar en un análisis *software*, y de nuevo, al igual que Pycparser, la estructura de datos que Javalang nos devuelve es un AST.

Con la finalidad de no tener que insertar una dependencia a nuestro sistema, esta vez no se instalará directamente Javalang sobre el mismo, sino que se utilizará un entorno virtual de Python[2]. Un entorno virtual de Python es una utilidad de Python que nos permite crear un entorno aislado del sistema, y la manera en la que lo hace es copiando los ficheros necesarios para obtener una funcionalidad mínima (un ejemplo donde se aplica la misma filosofía es en aquellos servidores que se enjaula a los usuarios a través de *chroot* y se realiza una copia de las librerías y binarios mínima necesarios para poder ejecutar un sistema totalmente aislado del resto de entornos del servidor). En la figura B.1 se puede observar como se crea el entorno virtual "*boa-javalang*" e instala las dependencias necesarias (Javalang y xmltodict), las cuales solo estarán disponibles en este entorno virtual y fuera del mismo no se podrá ejecutar correctamente cualquier módulo que depende de las dependencias instaladas. Una vez está creado el entorno virtual de Python, ya podemos pasar a implementar los módulos necesarios para utilizar Javalang.

Para este ejemplo se va a volver a implementar el módulo *Function Match*, pero esta vez para que funcione con el analizador Javalang. La razón de que haya que volver a implementar un módulo de seguridad es que, en mayor o menor medida, los módulos de seguridad dependen del analizador del lenguaje de programación utilizado, pues incluso la más mínima comprobación de tipos de nodos ya hace que dicho módulo sea dependiente del analizador. Esto se puede evitar realizando asociaciones que permitan relacionar tipos de datos o lo que sea necesario entre diferentes analizadores de lenguaje de programación (e.g. el tipo de dato *FuncCall* de Pycparser se relacionaría con el tipo de dato *MethodInvocation* de Javalang). Esto es un

[1]Repositorio de Javalang: `https://github.com/c2nes/javalang`

[2]Documentación de los entornos virtuales de Python: `https://docs.python.org/3/tutorial/venv.html`

problema de portabilidad, y diferentes técnicas pueden ser aplicadas como la mencionada (crear asociaciones que permitan definir relaciones) u otras (e.g. hacer comparaciones con cadenas de texto, hacer separación de código en función del analizador, volver a implementar el módulo).

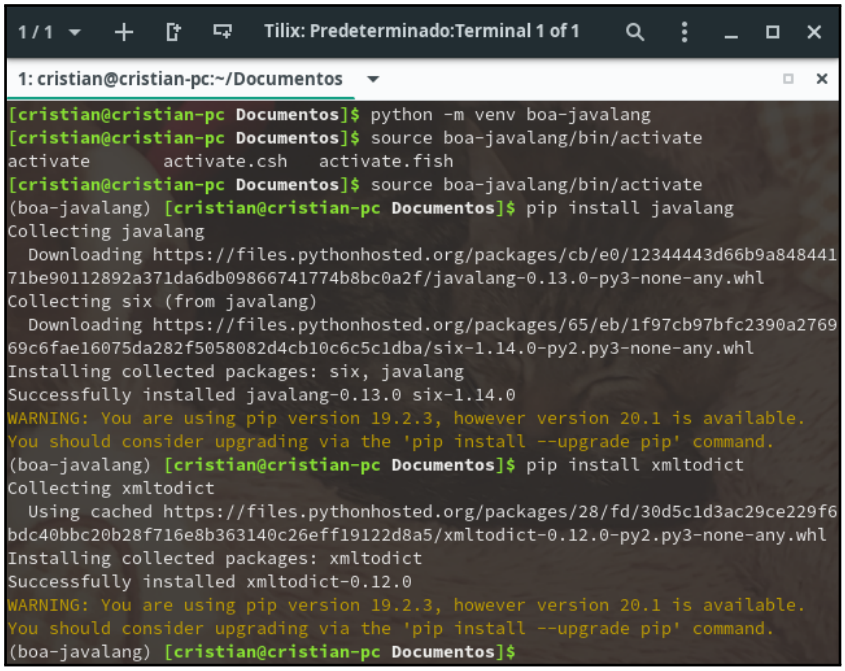


Figura B.1: Entorno virtual de Python *boa-javalang*

Lo primero a realizar es crear un módulo de analizador, el cual se va a llamar *boamp_javalang* y va a contener la clase *BOAPMJavalang*. Este módulo va a implementar la interoperabilidad con Javalang y nos va a permitir obtener el AST del fichero de código Java. Lo siguiente es crear una versión alternativa al módulo *FunctionMatch*, el cual funciona con Pycparser. En este nuevo módulo que va a funcionar con Javalang es casi idéntico al que utiliza Pycparser, pues las pocas diferencias que hay que hacer para obtener el mismo comportamiento es utilizar el tipo de dato que se corresponde con las llamadas a métodos de Java en lugar de las llamadas a funciones de C. Además de esa diferencia, también está la diferencia de cómo se obtienen las filas y columnas de los elementos del AST de Javalang, las cuales se obtienen accediendo a los elementos *position.line* y *position.column* en lugar de *coord*. Como se puede observar, la equivalencia es fácil de ver e implementar, pero para este ejemplo sencillo se ha optado por volver a implementar todo el módulo, lo cual no debería de hacerse para módulos más complejos si se quiere portar su comportamiento a otro analizador de lenguaje de programación. Por último, lo que queda es crear un nuevo fichero de reglas parecido al utilizado por Pycparser, pero en el nuevo se indicará que el analizador a utilizar es Javalang indicando que el módulo de analizador de lenguaje de programación a emplear será *BOAPMJavalang*. Otro cambio en este archivo de reglas es que las funciones consideradas como peligrosas serán las que se consideren peligrosas por Java. Para este ejemplo solo se ha indicado la función ”*exec*” como peligrosa.

Una vez realizado todo lo anterior, ya tenemos todo lo necesario para poder ejecutar el módulo *FunctionMatch* sobre ficheros de lenguaje Java. Un ejemplo de ejecución de este módulo se puede observar en la figura B.2, el cual encuentra una llamada a la función ”*exec*” y lo indica de la misma manera que hemos visto hasta el momento. El fichero de código empleado contiene una función *main()* y una función *execute()*, donde *main()* le pasa el primer argumento de la entrada a *execute()* y este ejecuta el método ”*exec*” a través del entorno de ejecución (i.e. método *Runtime.getRuntime()*).

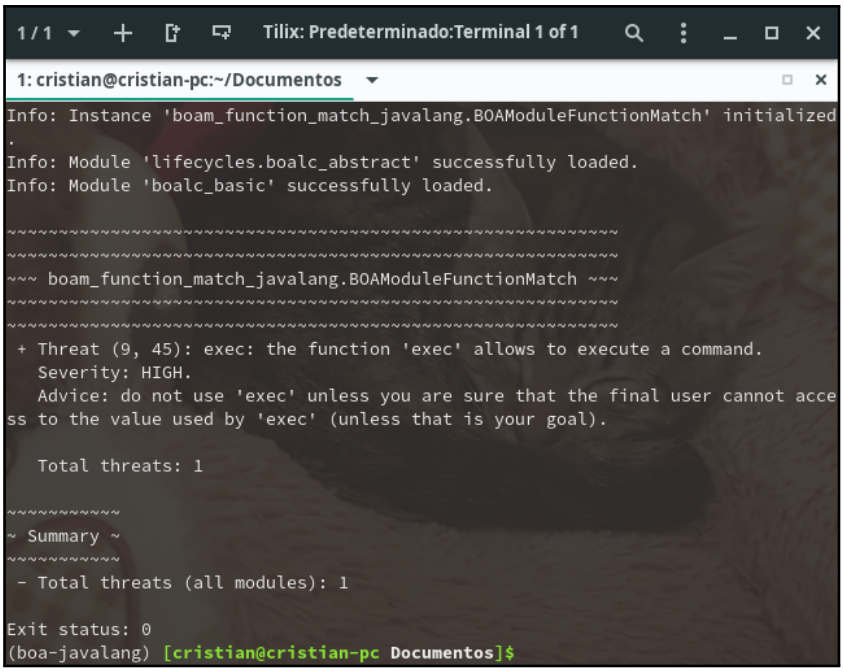


Figura B.2: Ejecución de un módulo de seguridad (*Function Match*) con Javalang

Los pasos realizados para el ejemplo que se ha visto no deberían de variar mucho a la hora de utilizar otros analizadores de lenguajes de programación. También comentar que el entorno virtual es un paso totalmente opcional, pero que nos aporta la ventaja de no tener que instalar módulos de Python en el sistema e incluso nos permite mantener las versiones de los módulos si estos se actualizan en el sistema. El mantener las versiones de los módulos es algo muy interesante, pues cuando se actualiza algún componente *software*, siempre existe el riesgo de que algún cambio no tenga en cuenta la retrocompatibilidad, y los elementos que hagan uso de dicho componente, pueden dejar de funcionar correctamente.