Estudio e implementación de una biblioteca que permita detectar vulnerabilidades en aplicaciones web escritas en Python mediante el análisis de manchas

Maestría en Ingeniería en Sistemas de Información

Facultad Regional Santa Fe
Universidad Tecnológica Nacional

Julio 2012

por

Ing. Juan José Conti

Santa Fe, Argentina

Director: Dr. Alejandro Russo
Department of Computer Science and Engineering
Chalmers University of Technology
SE - 412 96 Göteborg, Sweden

Co-director: Ms. Susana Romaniz
Facultad Regional Santa Fe
Universidad Tecnológica Nacional
Argentina

E-mail: jjconti@frsf.utn.edu.ar

Índice general

Índice de figuras VI

Resumen VII

Agradecimientos VIII

1. Introducción **1**

1.1. El problema de la seguridad 1

1.2. Análisis de manchas 3

1.3. Presentación de «Sugerencias para el almuerzo» 3

1.4. Objetivo 8

1.5. El lenguaje de programación Python 9

1.6. Organización 9

2. Análisis de manchas **11**

2.1. Conceptos del análisis de manchas 11

2.1.1. Fuentes no confiables 11

2.1.2. Funciones limpiadoras 12

2.1.3. Sumideros sensibles 12

2.1.4. El análisis 13

2.2. Antecedentes 13

3. Una biblioteca para análisis de manchas en Python **18**

3.1. Uso de la biblioteca 18

3.2. Flujos implícitos 27
3.3. Versión segura de «Sugerencias para el almuerzo» 28

4. Implementación **32**
4.1. Mecanismos dinámicos 32
4.2. Generación de clases taint-aware 34
4.3. Decoradores 37
4.4. Funciones taint-aware 42
4.5. Más detalles de implementación 44

5. Limitaciones **45**
5.1. Limitaciones de ámbito 45
5.2. La clase `bool` 46
5.3. Limitaciones en métodos 46
5.4. Problemas específicos 47

6. Conclusiones **49**

Apéndice **51**
A. Código fuente de la biblioteca 51
B. Casos de prueba 57
C. Código fuente de «Sugerencias para el almuerzo» 69

Bibliografía **71**

Índice de figuras

1.1. «Sugerencias para el almuerzo» 4
1.2. Uso normal de «Sugerencias para el almuerzo» 4
1.3. Un problema en la aplicación web: código parte del método `POST` de la clase `add`, ejecutado cuando el usuario hace click en el botón Enviar. 5
1.4. «Sugerencias para el almuerzo» convertido en un navegador de archivos 6
1.5. Uno de los archivos tiene un nombre interesante 6
1.6. «Sugerencias para el almuerzo» usado para ver el contenido de un archivo 7
1.7. «Sugerencias para el almuerzo» usado para ver el contenido de un archivo del sistema 7

2.1. La función `cgi.scape` 12

3.1. Datos manchados con todas las etiquetas 19
3.2. Datos manchados sólo con la etiqueta II (Inyección en el Intérprete) . 20
3.3. ¿Está un valor manchado? 20
3.4. Uso del decorador `untrusted` en una fuente no confiable 21
3.5. Uso del decorador `untrusted` como una llamada a función en una fuente no confiable 21
3.6. Una fuente no confiable usando el decorador `untrusted_args` . . 23
3.7. Una función limpiadora 24
3.8. El validador `is_not_digit` 25
3.9. El validador `is_digit` 26
3.10. Un sumidero sensible 27

3.11. Un flujo implícito 28
3.12. Versión segura de «Sugerencias para el almuerzo» 29
3.13. Ejecución cortada en «Sugerencias para el almuerzo» 30
3.14. Importar una función limpiadora 30
3.15. Uso de la función limpiadora 31

4.1. Propagación de manchas 33
4.2. Definición de etiquetas 33
4.3. Función para generar clases taint-aware 34
4.4. Propagación de la información de las manchas 36
4.5. Clases taint-aware para enteros y strings 37
4.6. Código de `untrusted` 37
4.7. Código de `untrusted_args` 38
4.8. Código de `cleaner` 39
4.9. Código de `validator` 40
4.10. Código de `ssink` 41
4.11. Funciones taint-aware para strings y enteros 43
4.12. Propagación de la información de las manchas a través de diferentes objetos taint-aware 44

5.1. El problema del operador % 48

Resumen

Las vulnerabilidades en aplicaciones web constituyen amenazas para los sistemas en línea. Los ataques SQL injection (se infiltra código instruso a partir de una vulnerabilidad en la validación de entradas utilizadas para hacer consultas a una base de datos) y Cross Site Scripting (XSS) (se envía al navegador de un usuario código malicioso a partir de una vulnerabilidad en la validación de código HTML incrustado por una página confiable) se encuentran entre las amenazas más comunes hoy en día. A menudo, estos ataques son el resultado de una inapropiada o inexistente validación de las entradas. Para ayudar a descubrir este tipo de vulnerabiliades, los lenguajes de programación más populares utilizados en la web, como Perl, Ruby, PHP y Python, cuentan con análisis de manchas. Este análisis se suele implementar usando técnicas estáticas (por ejemplo, sistemas de tipos) o dinámicas (por ejemplo, monitores de ejecución). Un ejemplo del último caso, son los intérpretes de Perl, Ruby y Python, que han sido modificados para proveer análisis de manchas. Sin embargo, modificar un intérprete puede ser una tarea importante por sí misma. De hecho, es muy probable que las nuevas versiones del intérprete requieran ser adaptadas para proveer este modo de ejecución. Diferenciándome de enfoques previos, muestro cómo proveer análisis de manchas para Python mediante una biblioteca escrita enteramente en este lenguaje y así evitar modificaciones en el intérprete. Los conceptos de clases, decoradores y ejecución dinámica hacen que esta solución sea liviana, fácil de usar y particularmente elegante. Con un esfuerzo mínimo, o incluso inexistente, se puede adaptar la bibiloteca para utilizarse con diferentes intérpretes de Python.

Agradecimientos

Este trabajo no podría haberse completado sin el apoyo de muchas personas.

En primer lugar me gustaría agradecer el apoyo de mi familia y dedicarles este trabajo. A mis padres Raúl y Susana, mi hermana Marisú y mi cuñado Luis. A mi esposa Cecilia con quien me casé durante la confección de este trabajo, pero quien me viene apoyando desde muchísimo antes. A mi familia política Ana María, Raúl y Melisa.

Me gustaría expresar mi agradecimiento a mis directores y guías en la realización de esta maestría. A mi directora de Beca, Marta Castellaro, a mi director de tesis Alejandro Russo y a mi co-directora de tesis Susana Romaniz.

Cuando en el año 2009 obtuve una beca del Programa Bicentenario de Investigación y Posgrado de la UTN, Marta me puso en contacto con Alejandro con quien empezamos a trabajar en enero de 2010, yo en Argentina y él en Suecia. La diferencia horaria era una limitación, por lo que durante varias semanas me levantaba una hora antes de mi horario normal para poder hablar con él antes de ir a mi trabajo.

Afortunadamente nuestro trabajo fue bien recibido y pude viajar Suecia invitado por la Chalmers University of Technology. En Gotemburgo estuve muy a gusto compartiendo con los miembros del grupo de investigación *Computer security*[1] y al escribir este agradecimiento no puedo dejar de recordar a personas como los profesores David Sands y Andrei Sabelfeld y a los entonces estudiantes Phu Phung y Filipo Del Tedesco.

De regreso en Santa Fe, tomé todos los cursos que me faltaban a la vez que participaba del grupo de investigación PID 108 dirigido por Marta y Susana. Vaya también el

[1] `http://www.chalmers.se/cse/EN/research/research-groups/computer-security`

saludo y agradecimiento a los miembros de este grupo. Susana asumió la co-dirección de mi tesis, dándome todo el apoyo local necesario.

Agradezco a Luis Stroppi, Rayentray Tappa, Alejandro Santos, Juan Arce, Ariel Rossanigo y Gustavo Campanelli la lectura desinteresada de mis borradores y sus muchas correcciones.

Gabriel Genellina me ayudó a entender el problema con el operador % del capítulo 5.

El diseño gráfico de la aplicacion «Sugerencias para el almuerzo» es de Manuel Quiñones.

Este trabajo fue financiado por el Programa Bicentenario de Investigación y Posgrado de la UTN. Mi visita a Suecia fue financiada por Chalmersska Forskningsfoden.

Capítulo 1

Introducción

En este capítulo introduzco el problema que afrontaré, comento formas tradicionales de encararlo, planteo el objetivo propuesto, defino el contexto en el que estaré trabajando (el cual está dado por un lenguaje de programación en particular) y explico cómo está organizada la tesis.

1.1. El problema de la seguridad

En los últimos años, ha habido un incremento significativo del número de actividades realizadas en línea. Los usuarios pueden hacer prácticamente todo usando un navegador web (por ejemplo, ver vídeos, escuchar música, realizar transacciones bancarias, reservar vuelos, planear viajes y más). Considerando el tamaño de Internet y su número de usuarios, las aplicaciones web están entre los elementos de software más usados en estos días.

A pesar de su amplio uso, las aplicaciones web sufren de vulnerabilidades que les permiten a los atacantes robar datos confidenciales, romper la integridad de sistemas y afectar la disponibilidad de servicios.

Cuando se desarrollan aplicaciones web sin tener en cuenta la seguridad, la presencia de vulnerabilidades aumenta dramáticamente.

Hay varias causas que pueden llevar a que los programadores web no se preocupen por la seguridad. Podemos citar como ejemplos la presión por agregar nuevas características en lugar de arreglar fallas, el desconocimiento o la dificultad de implementar

seguridad, trabajos mal pagos, o simplemente plazos demasiado cortos que obligan a implementar en la forma más básica posible para cumplir el plazo.

Las vulnerabilidades basadas en la web ya han superado a las de otras plataformas [7] y no hay razones para pensar que esta tendencia vaya a cambiar [15]. Atacar una aplicación web es mucho más fácil que atacar aplicaciones de escritorio, en las que el atacante necesita un conocimiento específico, o atacar al sistema operativo, donde el atacante necesita conocimiento de bajo nivel [7].

Según OWASP [37], los ataques de cross-site scripting (XSS) y distintas clases de inyecciones, como SQL injection (SQLI), están entre las vulnerabilidades más comunes en las aplicaciones web.

Aunque estos ataques están clasificados en distintos grupos, son producidos por la misma razón: *los datos provistos por el usuario son enviados a sumideros sensibles sin la sanitización necesaria*. Por ejemplo, cuando se construye una consulta SQL usando cadenas de texto no sanitizadas provistas por un usuario, es posible que ocurran ataques de SQL injection. Posibles consecuencias de este tipo de ataques incluyen:

1. Robo de la identidad de un usuario ante un sistema.
2. Compromiso de datos confidenciales: un usuario no autorizado tiene acceso a datos a los que se suponía no podía acceder.
3. Denegación de servicio: un recurso no está disponible para sus genuinos usuarios.
4. Destrucción de datos.

Por lo tanto, el objetivo de un atacante es alterar los datos de entrada para ganar algún tipo de control sobre ciertas operaciones. Es importante mencionar aquí que el atacante no tiene control sobre el código ejecutado (no puede cambiarlo), sólo sobre los datos de entrada.

Hay distintos escenarios y diferentes sumideros sensibles en los que un atacante puede actuar. El atacante puede manipular datos que serán usados para generar una consulta SQL y obtener alguna información secreta, realizar una inyección en el sistema operativo y ejecutar comandos arbitrarios o explotar una vulnerabilidad XSS y robar las credenciales de un usuario en un sitio web.

1.2. Análisis de manchas

Para asegurar las aplicaciones contra estos ataques, las implementaciones de lenguajes de programación usados en la web, como Perl, Ruby, PHP y Python, cuentan con análisis de manchas. Existen dos formas de análisis de manchas: dinámico y estático. El análisis dinámico suele implementarse con monitores. Éstos no sólo corren el código del intérprete, sino que también realizan chequeos de seguridad [3, 5]. El análisis dinámico tiene la ventaja de producir menos falsas alarmas que el análisis estático. Sus principales desventajas son la sobrecarga producida (ya que el programa y el monitor corren a la vez) y la necesidad de modificar el intérprete para lograr el comportamiento deseado.

El análisis de manchas también puede lograrse mediante análisis estático [18, 19]. Esta técnica no produce sobrecarga ya que se analiza el texto del programa sin necesidad de ejecutarlo y no se necesitan modificaciones en el intérprete. La principal desventaja es que suelen generar más falsas alarmas que los monitores [33].

En otras palabras, los monitores de ejecución tienden a ser más precisos que las técnicas (tradicionales) estáticas. En particular, las técnicas estáticas no pueden lidiar con evaluación dinámica de código (una característica común en los lenguajes de programación utilizados en la web) sin ser demasiado conservadoras. En esta tesis me enfoco en las técnicas dinámicas.

1.3. Presentación de «Sugerencias para el almuerzo»

Presento un ejemplo con el propósito de motivar el uso de análisis de manchas para descubrir y solucionar vulnerabilidades.

«Sugerencias para el almuerzo» (figura 1.1) es una aplicación web usada en una oficina para decidir qué comer en el almuerzo. A la mañana los compañeros de trabajo cargan sus sugerencias y cada día el comprador designado las revisa antes de comprar la comida para todos.

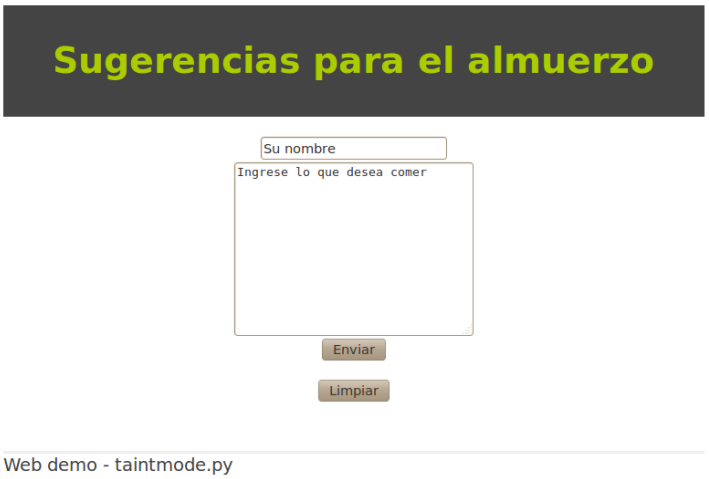

Figura 1.1: «Sugerencias para el almuerzo»

Parece funcionar bastante bien (figura 1.2), pero la aplicación tiene un problema.

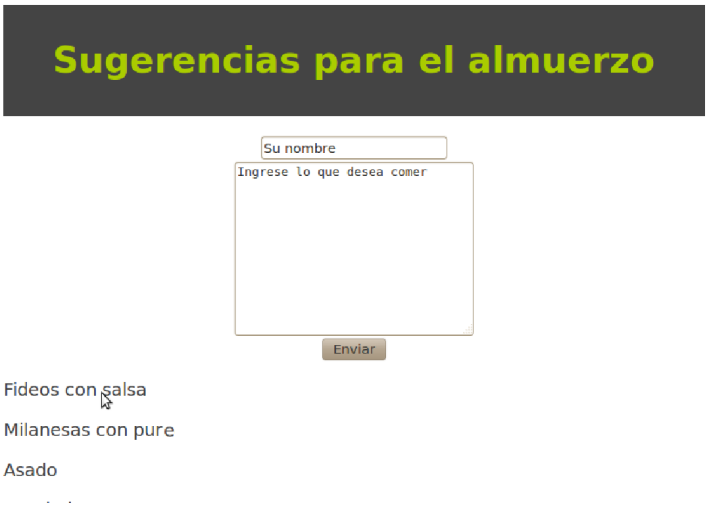

Figura 1.2: Uso normal de «Sugerencias para el almuerzo»

Como se ve en la figura 1.3, utiliza un archivo de texto para almacenar los datos y usa comandos del sistema operativo para almacenar la información en el archivo.

```
1        user = web.input().user
2        meal = web.input().meal
3        # save it to the file of the day
4        dayfile = datetime.today().strftime('%Y%m%d') +
             '.txt'
5        os.system("echo " + meal + " >> " + dayfile)
```

Figura 1.3: Un problema en la aplicación web: código parte del método `POST` de la clase `add`, ejecutado cuando el usuario hace click en el botón Enviar.

Podemos ver que se reciben datos desde una fuente no confiable (`web.input`) y son usados sin ninguna validación para construir un comando que más tarde se usará en el sumidero sensible `os.system`. Este es un objeto Python que permite al programador ejecutar comandos crudos contra el sistema operativo y así ser capaz de reutilizar servicios y herramientas que este provee.

Ataques

Un atacante puede abusar de esta aplicación para realizar distintos ataques. Por ejemplo, en lugar de introducir el almuerzo que desea, puede introducir `; ls` para hacer que el comando ejecutado liste los archivos en el directorio raíz de la aplicación y guarde ese listado en el archivo de sugerencias (figura 1.4).

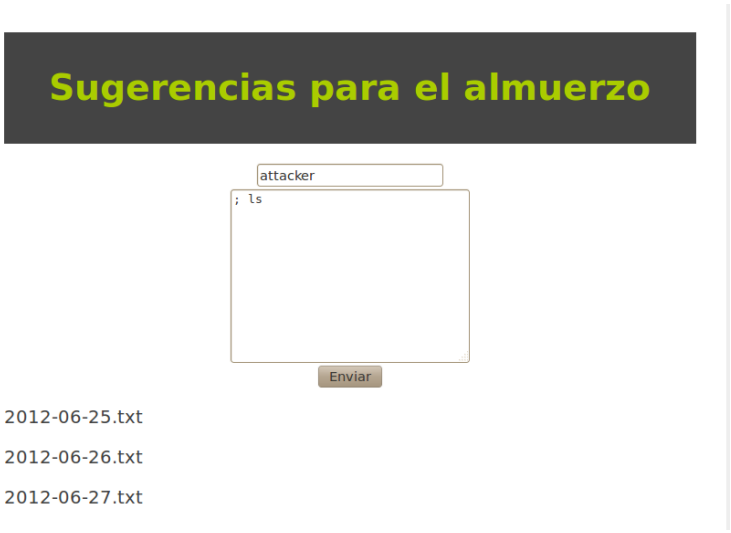

Figura 1.4: «Sugerencias para el almuerzo» convertido en un navegador de archivos

Si encuentra algo interesante (figura 1.5), por ejemplo un archivo llamado `passwords.txt`, podría introducir `; cat passwords.txt` y leer el contenido de ese archivo (figura 1.6).

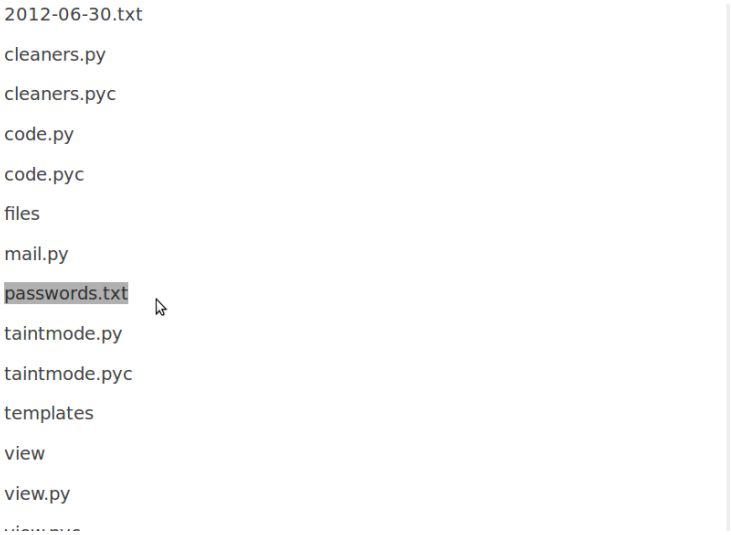

Figura 1.5: Uno de los archivos tiene un nombre interesante

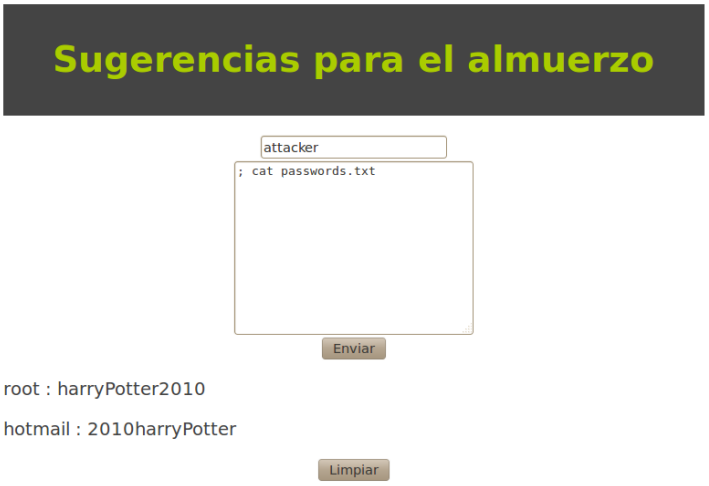

Figura 1.6: «Sugerencias para el almuerzo» usado para ver el contenido de un archivo

Incluso más, podría leer el contenido de archivos sensibles del sistema. Enviando por ejemplo `; cat /etc/passwd` puede obtener la lista de usuarios del sistema (figura 1.7).

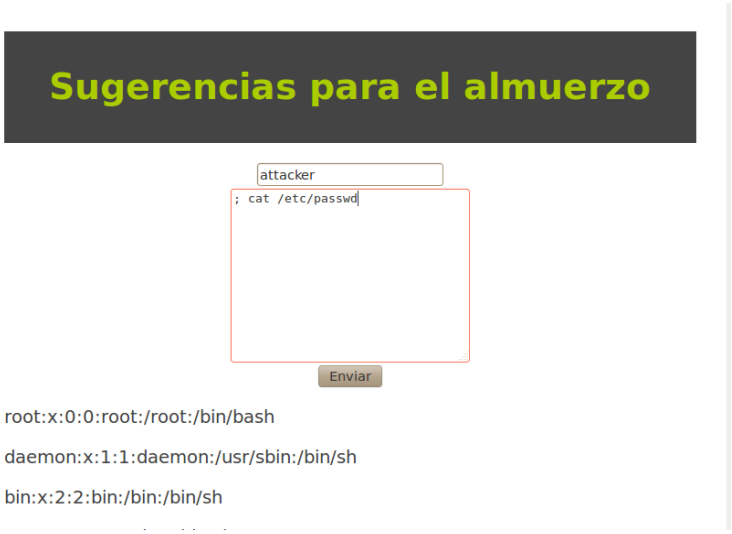

Figura 1.7: «Sugerencias para el almuerzo» usado para ver el contenido de un archivo del sistema

Estos ataques demuestran cómo, lo que se tenía como una simple aplicación web, puede convertirse en un navegador de archivos web o una terminal. Para evitar este tipo

de vulnerabilidades, las aplicaciones necesitan revisar que no haya datos maliciosos provistos por los usuarios o cualquier otro actor no confiable.

El análisis de manchas ayuda a detectar datos no sanitizados antes de que sean usados en operaciones críticas para la seguridad de un sistema. En el capítulo 3 muestro cómo asegurar la aplicación «Sugerencias para el almuerzo» a fin de evitar las vulnerabilidades mostradas en esta sección.

1.4. Objetivo

Comenté que la forma en que se suele implementar análisis de manchas dinámico para un lenguaje de programación es modificando el intérprete. En lugar de hacer esto, tengo como objetivo proveer un análisis de manchas dinámico para Python mediante una biblioteca escrita enteramente en este lenguaje de programación.

En la biblioteca se hace uso de características avanzadas del lenguaje con el fin de lograr una implementación elegante, compacta e independiente de la versión específica del intérprete con el que se la use. Esto permitirá que hagan uso de ella un amplio rango de programadores interesados en contar con una herramienta que les permita atender a la necesidad de detectar posibles vulnerabilidades en su código durante el desarrollo.

La biblioteca provee un método general para ampliar las clases de Python agregándoles la propiedad de poder saber si una instancia está manchada o no y qué tipo de manchas tiene. Para demostrar la flexibilidad del enfoque, la biblioteca le agrega esta capacidad a las clases `str`, `unicode`, `int` y `float`. En general, el análisis de manchas tiende a considerar sólo strings o caracteres [3, 26, 17, 20, 16, 34]. En contraste, esta biblioteca puede ser fácilmente adaptada para tener en cuenta distintas clases y así proveer un análisis de manchas para un conjunto de tipos de datos más amplio. Solo considerando strings manchados, la biblioteca provee un análisis similar al de Kozlov y Petukhov [20], pero sin modificar el intérprete. En la investigación bibliográfica del problema no encontré evidencias de que alguna vez se haya usado un enfoque como este.

1.5. El lenguaje de programación Python

Como lo explica su sitio web [4], Python es un lenguaje de programación que le permite a los programadores trabajar de forma rápida e integrar sistemas de forma efectiva. Una vez aprendido el lenguaje, se nota un aumento en la productividad y una reducción de los tiempos de mantenimiento. Es un lenguaje de programación interpretado, interactivo y orientado a objetos. Incorpora módulos, excepciones, tipado dinámico, tipos de datos de muy alto nivel y clases. Python combina un sorprendente poder con una sintaxis muy limpia. Tiene interfaces con llamadas al sistema y bibliotecas, así como varios sistemas de ventanas y es extensible con C y C++. También puede usarse como un lenguaje para extender aplicaciones que necesiten proveer de una interfaz programable. Por ejemplo, un procesador de textos podría embeber el intérprete de Python y permitirle a los usuarios escribir plug-ins utilizando este lenguaje. Finalmente, Python es multiplataforma: corre en muchas variantes de UNIX, en Mac y en PCs con MS-DOS o Windows, por mencionar algunas plataformas.

El uso de Python se está expandiendo rápidamente en el desarrollo web [2]. Por ejemplo, el conocido software para administrar wikis MoinMoin (usado en los foros de Ubuntu) y sitios como Youtube están escritos mayormente en Python. Junto a otras compañías, se puede mencionar a Google, quien usa el lenguaje en los mecanismos internos de servicios como Google Groups, Gmail y Google Maps. También existen muchos frameworks de desarrollo web escritos en Python; Django [1], el más popular, es ampliamente usado en la industria y fue adoptado como uno de los primeros soportados en la plataforma de cloud computing Google App Engine.

Además de su amplio uso, Python presenta algunas abstracciones que hacen posible proveer un análisis de manchas mediante una biblioteca. Por ejemplo, los decoradores [24], un mecanismo simple y no invasivo que utilizo para declarar fuentes de datos manchados, sumideros sensibles y funciones de sanitización. La orientación a objetos de Python y sus mecanismos de ejecución dinámica permiten realizar el análisis de manchas casi sin modificaciones en el código fuente.

1.6. Organización

El resto de la tesis está organizada de la siguiente forma. El capítulo 2 describe las ideas detrás de la biblioteca, enfoques existentes para análisis de manchas y la idea

general del mismo. El capítulo 3 introduce la biblioteca (su API) y muestra cómo usarla para asegurar la aplicación de ejemplo «Sugerencias para el almuerzo». El capítulo 4 presenta los detalles de implementación de la biblioteca, con especial atención en las características de Python que hacen que sea fácil construir esta tecnología de forma no invasiva desde el punto de vista del programador. El capítulo 5 explica las limitaciones del enfoque original y discute alternativas. El capítulo 6 presenta conclusiones e ideas para futuros trabajos. El código fuente de la biblioteca, así como el de la aplicación de ejemplo y casos de prueba se incluyen en los apéndices.

Capítulo 2

Análisis de manchas

En este capítulo repaso los conceptos principales del análisis de manchas y comento enfoques existentes que se han utilizando para implementar el mismo a la vez que los comparo con mi implementación.

2.1. Conceptos del análisis de manchas

Los tres conceptos principales del análisis de manchas son:

- Fuentes no confiables
- Funciones limpiadoras
- Sumideros sensibles

2.1.1. Fuentes no confiables

Como dije en la introducción, consideramos que los datos recibidos enviados por un usuario u otro sistema son no confiables ya que no sabemos quién los generó. Puede ser un usuario real o un usuario atacante o, incluso, un programa malicioso. Los datos no confiables deben ser marcados como manchados.

Son ejemplos de fuentes no confiables:

- Parámetros en solicitudes HTTP de tipo POST o GET
- Encabezados HTTP

- Solicitudes AJAX

También se puede considerar marcar como manchados datos provenientes de la capa de persistencia. Los datos en una base de datos pueden haber sido escritos mediante una aplicación externa o haber sido alterados antes de almacenarse.

Los datos no confiables pueden transformarse en confiables a través de un proceso de sanitización.

2.1.2. Funciones limpiadoras

Las funciones limpiadoras o de sanitización son las que permiten escapar, codificar o validar datos de entrada para volverlos aptos para enviarlos a algún sumidero.

Un ejemplo de función de sanitización es la función de Python `cgi.escape` cuyo uso se muestra en la figura 2.1. La etiqueta `<script>` se utiliza para definir un script en el cliente, por ejemplo en el lenguaje JavaScript. Este elemento puede contener sentencias o apuntar a un archivo externo. Escapando los caracteres < y > se elimina la etiqueta y se evita que su contenido se ejecute en el navegador en el caso de incluir el string en una página web.

```
1 >>> import cgi
2 >>> cgi.escape("<script>alert('this is an attack')</script>")
3 "&lt;script&gt;alert('this is an attack')&lt;/script&gt;"
```

Figura 2.1: La función `cgi.scape`

2.1.3. Sumideros sensibles

Los sumideros sensibles son aquellas partes del sistema a las que no queremos que lleguen datos sin haber sido apropiadamente sanitizados.

Ejemplos de sumideros sensibles son:

- El navegador o un sistema de templates HTML
- Un motor de base de datos SQL
- El intérprete de comandos del sistema operativo
- El intérprete de Python

Los datos que van a terminar en diferente sumideros, necesitan diferentes tipos de sanitización. Necesitan ser apropiadamente limpiados para ser aptos para ese tipo de sumidero. No es la misma función la que usaremos para proteger al sumidero «generador de páginas HTML» de un ataque de XSS, que la función que usaremos para codificar datos antes de usarlo para generar una consulta SQL que terminará ejecutándose en el sumidero «motor de base de datos» para evitar un ataque de SQL injection.

2.1.4. El análisis

El análisis de manchas es un enfoque automático para encontrar vulnerabilidades. Se puede comprender de forma intuitiva que el análisis de manchas restringe cómo los datos manchados (o no confiables) fluyen dentro del programa. Específicamente, impone la restricción de que los datos sean no manchados (confiables) o hayan sido previamente sanitizados para poder alcanzar un sumidero sensible.

2.2. Antecedentes

Perl y Ruby

Perl fue el primer lenguaje de programación web en proveer análisis de manchas como un modo especial de ejecución del intérprete. Este modo se llama *taint mode* [9].

Perl se ejecuta en este modo si se llama con la opción `-T`. Tiene fuentes no confiables y sumideros sensibles predefinidos. Marca los strings originados en el exterior del programa como manchados (por ejemplo, datos ingresados por usuarios, variables de entorno y archivos). La sanitización se realiza mediante expresiones regulares. Considera como sumideros sensibles las siguientes operaciones: escribir en archivos, ejecutar comandos en el intérprete y enviar información a través de la red.

A diferencia de este enfoque, la biblioteca que presento le permite a los desarrolladores definir sus propias fuentes de datos no confiables, funciones limpiadoras y sumideros sensibles.

De forma similar a Perl, Ruby [35] provee soporte para análisis de manchas. Sin embargo, lo provee a nivel de objetos en lugar de sólo strings. Ambos, Perl y Ruby, utilizan técnicas dinámicas en su análisis.

PHP

Se han desarrollado muchos análisis de manchas para el popular lenguaje de programación web PHP.

Por ejemplo, Huang y otros [18] presentan una implementación que tiene como objetivo que el proceso no requiera intervención del usuario. Lo hacen combinando técnicas estáticas y dinámicas para reparar automáticamente vulnerabilidades en el código PHP.

Proponen usar un sistema de tipos (análisis estático) para detectar dónde insertar funciones de sanitizacion predeterminadas en el punto anterior a que un valor manchado alcance un sumidero sensible. Las funciones de sanitización son ejecutadas durante la corrida del programa.

Esto tiene la desventaja de que la semántica del programa podría cambiar al insertar llamadas a las funciones de sanitización, las cuales conforman la parte dinámica de esta implementación.

En cambio, mi enfoque no implementa un sistema de tipos, sólo reporta las vulnerabilidades. Es decir que depende de los desarrolladores decidir dónde, o cómo, llamar a procesos de sanitización.

Nguyen-Toung y otros [26] adaptaron el intérprete de PHP para proveer un análisis de manchas dinámico al nivel de caracteres, lo cual es llamado por los autores *precise tainting*.

Argumentan que esta técnica gana precisión en relación a los análisis de manchas tradicionales para strings y así pueden reducir el número de falsos positivos.

Para lograrlo, los autores debieron hacer modificaciones específicas para distintas funciones de forma de poder mantener la traza de los caracteres modificados.

En la biblioteca que presento se utiliza siempre el mismo mecanismo para manejar todos los valores manchados independientemente de la función que los procese. Esta uniformidad me permite extender automáticamente el análisis a distintos tipos de datos aunque sin ser tan preciso como el trabajo de Nguyen-Toung.

Un trabajo similar al de Nguyen-Toung y otros es el de Futoransky y otros [16] que provee un análisis de manchas dinámico y preciso para PHP.

Pietraszek y Berghe [27] modificaron el entorno de ejecución de PHP para asignarle meta-datos a las entradas provistas por usuarios y hacer que las operaciones de manipulación de strings conserven esos meta-datos. También se modificaron las ope-

raciones críticas para la seguridad a fin de evaluar, cuando se reciben strings como parámetros, el riesgo de ejecutar esa operación basándose en los meta-datos recibidos.

Jovanovic y otros [19] propusieron combinar los tradicionales análisis de flujo de datos y de alias para incrementar la precisión de su análisis de manchas estático para PHP. Obtuvieron una relación de un falso positivo por cada vulnerabilidad.

Los trabajos de Saner y Monga [8, 25] también son ejemplos de trabajos que combinan técnicas estáticas y dinámicas. Las técnicas estáticas son usadas para reducir el número de variables en el programa, para las cuales se deben seguir las manchas en tiempo de ejecución.

Java

El de Haldar y otros [17] es un análisis de manchas para Java. Sobreescribe la clase `java.lang.String` a la vez que implementa mecanismos para manejar fuentes no confiables y sumideros sensibles.

Otro análisis de manchas para Java es el llamado TAJ [36], que se centra en requerimientos de escalabilidad y desempeño para aplicaciones de nivel industrial. Para responder a las demandas de la industria, TAJ usa técnicas estáticas para realizar análisis de punteros y construir grafos de llamadas.

De forma similar, Livshits y Lam [23] proponen un análisis estático para Java que se centre en lograr precisión y escalabilidad.

Android

TaintDroid [14], es un análisis de manchas para smartphones Android. Para propagar información de manchas dentro de los programas, TaintDroid necesita que se modifique el intérprete de la máquina virtual de Android.

Haskell

Una serie de trabajos [22, 11, 28, 29, 30] proponen implementar seguridad mediante análisis de flujo de información utilizando bibliotecas para Haskell. Estas bibliotecas manejan flujos explícitos e implícitos pero demandan que los desarrolladores escriban sus programas utilizando una API de propósito específico.

En la biblioteca que presento, el análisis no implementa flujos implícitos (ver sección 3.2), solo explícitos, y los programas no necesitan ser escritos respetando una API especial.

Python

Entre los trabajos más cercanos al mío se puede mencionar el de Kozlov y Petukhov [20], y el de Seo y Lam [34].

En el primero, los autores modifican el intérprete de Python para proveer un análisis de manchas dinámico. Más específicamente, extienden la clase `str` para que incluya una bandera booleana que indica si el string está manchado o no. Otra característica de esta implementación es que las fuentes no confiables, funciones limpiadoras y sumideros sensibles deben declararse en archivos de configuración.

El análisis presentado en esta tesis es similar pero sin la necesidad de modificar el intérprete ni de realizar definiciones en archivos externos.

El trabajo de Seo y Lam [34], llamado InvisiType, tiene como objetivo agregar chequeos de seguridad sin modificar el código analizado. Su enfoque está diseñado para un modelo de ataque más fuerte: el atacante tiene acceso a modificar el código fuente.

InvisiType es más general que mi enfoque. De hecho, los autores muestran como implementar análisis de manchas y chequeos de control de acceso para programas Python usándolo. Sin embargo, InvisiType necesita varias modificaciones al intérprete para poder realizar los chequeos de seguridad en los puntos adecuados. Por ejemplo, cuando se llama a métodos nativos, el entorno de ejecución primero necesita llamar al método de propósito específico `__nativecall__` para propagar la información de las manchas.

La forma que se tiene de especificar políticas de seguridad es extendiendo la clase `InvisiType`. El enfoque se basa en el uso de herencia múltiple para extender clases existentes con chequeos de seguridad. Por ejemplo, se puede definir una clase `SafeStr` que extienda a `str` y a alguna subclase de `InvisiType`. Para agregar o quitar chequeos de seguridad de los objetos, los programadores necesitan llamar explícitamente a las funciones *promote* y *demote*.

La biblioteca que presento es menos invasiva ya que utiliza decoradores en lugar de llamadas explícitas a funciones para manchar o desmanchar objetos. Tampoco requiere

hacer uso de herencia múltiple.

Finalmente, no puedo dejar de mencionar el reciente trabajo de Bello y Russo [21] que toman las ideas originales de la biblioteca presentada aquí con el objetivo de aplicar el análisis a la plataforma de cloud computing Google App Engine.

Desventajas de modificar intérpretes

Adaptar un intérprete para proveer análisis de manchas presenta dos problemas que impactan directamente en la adopción de esta tecnología.

Primero, incorporar el análisis dentro del intérprete puede ser una tarea grande y difícil de por sí. Segundo, es muy probable que sea necesario adaptar repetidamente el intérprete por cada nueva versión del mismo que esté disponible.

Capítulo 3

Una biblioteca para análisis de manchas en Python

En este capítulo analizo `taintmode.py`, una biblioteca cuya primera versión fue presentada en un paper que escribí con Alejandro Russo [12]. También describo la API de la biblioteca y su uso mediante ejemplos y muestro cómo hacer que «Sugerencias para el almuerzo», la aplicación de ejemplo presentada en el capítulo 1, sea más segura sanitizando apropiadamente sus datos de entrada.

3.1. Uso de la biblioteca

Etiquetas

Por defecto, las etiquetas pueden tomar los valores `XSS`, `SQLI`, `OSI` (Inyección en el Sistema Operativo) e `II` (Inyección en el Intérprete). La biblioteca utiliza etiquetas para indicar vulnerabilidades específicas que pueden ser explotadas por los datos manchados. Por ejemplo, datos manchados asociados a la etiqueta `SQLI` son propensos a explotar vulnerabilidades de inyección de SQL.

Ejemplos de la API

Esta sección incluye ejemplos de las principales entidades manejadas por la biblioteca, los cuales son los conceptos claves del análisis de manchas:

1. Fuentes no confiables: desde las que llegan valores no confiables al programa.

2. Funciones limpiadoras: usadas para sanitizar los valores no confiables, volviéndolos confiables.

3. Sumideros sensibles: lugares a los que no queremos que lleguen valores no confiables.

Los ejemplos (de la figura 3.1 a la figura 3.10) son capturas de una sesión interactiva en la consola REPL[1] de Python

Las primitivas `taint` y `tainted`

La primitiva `taint` (cuya definición es `taint(o, v=None)`) es una función auxiliar que mancha el valor `o` con la etiqueta `v`.

Si `v` no se provee, `taint` marca a `o` con todos los tipos de manchas.

La figura 3.1 muestra un ejemplo de esta primitiva. La línea 1 asocia todas las etiquetas al valor `42`. La figura 3.2 muestra otro ejemplo de esta primitiva. En este caso al valor `42` se le asocia solo la etiqueta `II`.

```
1 >>> t = taint(42)
2 >>> tainted(t, XSS)
3 True
4 >>> tainted(t, OSI)
5 True
6 >>> tainted(t, SQLI)
7 True
8 >>> tainted(t, II)
9 True
```

Figura 3.1: Datos manchados con todas las etiquetas

[1] Read-Eval-Print-Loop

```
>>> taint(42, II)
42
>>> t = taint(42, II)
>>> tainted(t, II)
True
>>> tainted(t, OSI)
False
```

Figura 3.2: Datos manchados sólo con la etiqueta II (Inyección en el Intérprete)

La primitiva `tainted` (cuya definición es `tainted(o, v=None)`) dice si un valor `o` está manchado con una etiqueta `v` dada.

Si `v` no se provee, `tainted` chequea si el valor está manchado con al menos una etiqueta.

La figura 3.3 muestra más ejemplos de esta primitiva.

```
>>> t1 = taint(42, OSI)
>>> t2 = taint(42)
>>> tainted(t1)
True
>>> tainted(t1, OSI)
True
>>> tainted(t1, II)
False
>>> tainted(t2, II)
True
>>> tainted(t2, SQLI)
True
>>> tainted(t2)
True
```

Figura 3.3: ¿Está un valor manchado?

Fuentes de datos manchados

La forma más simple de definir una entrada como no confiable es usando el decorador `untrusted`. Su definición es `untrusted(f)`. Una forma de usarlo es mediante el azúcar sintáctica que provee Python para los decoradores.

En la figura 3.4, el decorador se aplica a la función `values_from_outside` cuando ésta es definida. Como consecuencia, cada vez que se llama a la función, el valor retornado se mancha con todas las etiquetas.

```
>>> @untrusted
... def values_from_outside(stream):
...     return stream.read()
...
>>> v = values_from_outside(f)
>>> tainted(v)
True
```

Figura 3.4: Uso del decorador `untrusted` en una fuente no confiable

Alternativamente, se puede usar el decorador `untrusted` mediante una llamada a función, lo que es extremadamente útil para marcar una función para la que no tenemos acceso a su definición como fuente de datos manchados. Por ejemplo, funciones en bibliotecas de terceros. La figura 3.5 reescribe el ejemplo de la figura 3.4 sin utilizar la sintaxis de decoradores.

```
>>> def values_from_outside(stream):
...     return stream.read()
...
>>> values_from_outside = untrusted(values_from_outside)
>>> v = values_from_outside(f)
>>> tainted(v)
True
```

Figura 3.5: Uso del decorador `untrusted` como una llamada a función en una fuente no confiable

Hay una forma más elaborada de definir fuentes de datos manchados que principalmente tiene lugar al utilizar frameworks de programación. Por ejemplo, puede darse el caso en el que se nos pide definir una función o método que será llamado cada vez que se reciba un valor desde el exterior [6]. Para manejar estos casos, la biblioteca provee un decorador llamado `untrusted_args`.

El decorador `untrusted_args` tiene dos argumentos. El primero es una lista de posiciones y el segundo es una lista de palabras clave. Los argumentos posicionales en la lista de posiciones y los argumentos de palabra clave en la lista de palabras clave son los marcados como no confiables, no así el valor retornado.

La figura 3.6 muestra cómo los argumentos de `untrusted_args` expresan que las posiciones 1 y 2 así como la palabra clave `file` son marcados como manchados cuando la función es ejecutada. El código en las líneas 5 a 20 hace una llamada a función y captura el resultado en dos variables sobre las cuales se itera para mostrar qué valores están manchados y cúales no.

```
>>> @untrusted_args([1,2], ['file'])
... def values_from_outside(*args, **kwargs):
...     return args, kwargs
...
>>> vals, keys = values_from_outside(42, 88, 7, file='help.txt'
    , sev=1)
>>> vals
(42, 88, 7)
>>> keys
{'sev': 1, 'file': 'help.txt'}
>>> for x in vals:
...     print x, tainted(x)
...
42 False
88 True
7 True
>>> for x in keys.values():
...     print x, tainted(x)
...
1 False
help.txt True
```

Figura 3.6: Una fuente no confiable usando el decorador `untrusted_args`

Funciones de sanitización

El decorador `cleaner` (cuya definición es `cleaner(v)`) se usa para decir que un método o función es capaz de limpiar manchas de un valor, esto es, borra cierta etiqueta correspondiente a una vulnerabilidad (argumento `v`).

Como antes, el decorador puede ser aplicado usando el azúcar sintáctica provista por el lenguaje o una llamada a función.

En la figura 3.7 se indica que la función `plain_text` es capaz de remover vulnerabilidades `XSS` de sus argumentos. Las líneas 9 a 11 muestran cómo la etiqueta `XSS` es quitada del valor `t`.

```
>>> @cleaner(XSS)
... def plain_text(input):
...     # some code
...     return input
...
>>> t = taint("bad string", XSS)
>>> tainted(t, XSS)
True
>>> u = plain_text(t)
>>> tainted(u)
False
```

Figura 3.7: Una función limpiadora

Existe también un decorador similar con una semántica diferente llamado `validator`. Su definición es `validator(v, cond=True, nargs=[], nkwargs=[])`.

Este decorador marca una función o método como capaz de validar su entrada. Cuando se tiene funciones que determinan si un dato tiene la forma esperada, se pueden utilizar dichas funciones para sanitizar datos cuando dichas funciones determinan que los datos están estructurados adecuadamete.

El argumento `nargs` es una lista de posiciones. Los argumentos posicionales en esas posiciones son los que se validan. `nkwargs` es una lista de palabras clave. Los argumentos de palabra clave para esas claves son los que se validan.

El argumento `cond` puede ser `True` o `False`. Si la función decorada retorna `cond`, la etiqueta `v` será quitada de las entradas validadas.

Por ejemplo, en una función llamada `is_not_digit` (ver figura 3.8), le daremos a `cond` el valor `False`. Si `is_not_digit` retorna `False`, entonces el valor *es* válido y no tiene datos alterados.

```
>>> @validator(XSS, False, [0])
... def is_not_digit(n):
...     if len(n) != 1:
...         return True
...     return n not in "0123456789"
...
>>> t1 = taint('1', XSS)
>>> t2 = taint('a', XSS)
>>> tainted(t1)
True
>>> tainted(t2)
True
>>> is_not_digit(t1)
False
>>> tainted(t1)
False
>>> is_not_digit(t2)
True
>>> tainted(t2)
True
```

Figura 3.8: El validador `is_not_digit`

Para una función llamada `is_digit` (ver figura 3.9), a `cond` le daremos el valor `True`.

```
>>> @validator(XSS, True, [0])
... def is_digit(n):
...     if len(n) != 1:
...         return False
...     return n in "0123456789"
...
>>> t1 = taint('1', XSS)
>>> t2 = taint('a', XSS)
>>> tainted(t1)
True
>>> tainted(t2)
True
>>> is_digit(t1)
True
>>> tainted(t1)
False
>>> is_digit(t2)
False
>>> tainted(t2)
True
```

Figura 3.9: El validador `is_digit`

Sumideros sensibles

El decorador `ssink` (cuya definición es `ssink(v=None, reached=reached)`) marca una función o método como sensible a datos manchados. Si es llamado con un valor con la etiqueta `v` (o cualquier etiqueta si `v` es `None`), la función decorada no se ejecuta y en su lugar se ejecuta `reached` que reporta la existencia de una posible vulnerabilidad.

Estos sumideros son sensibles a un tipo de vulnerabilidad y ésta debe ser especificada cuando se usa el decorador (ver figura 3.10).

```
>>> @ssink(SQLI)
... def write_database(connector, data):
...     connector.write(data)
...
>>> t1 = taint(42, OSI)
>>> t2 = taint(42, SQLI)
>>> write_database(c, t1)
>>> # writes with no problem
>>> write_database(c, t2)

Violation in line 1 from file <stdin>
Tainted value: 42
```

Figura 3.10: Un sumidero sensible

Si se llama a la función sensible con un valor manchado con la misma etiqueta a la que el sumidero es sensible, entonces la ejecución se puede detener y se muestra una advertencia explicando qué pasó.

3.2. Flujos implícitos

En la mayoría de los casos [3, 5], el análisis de manchas propaga la información de las manchas en las asignaciones. De forma intuitiva, podemos ver que cuando el lado derecho de la asignación usa un valor manchado, la variable que aparece a la izquierda será manchada. El análisis de manchas se puede ver como un mecanismo de seguimiento de flujos de información, con el objetivo de lograr integridad [32]. De hecho, el análisis de manchas es un mecanismo para seguir flujos de información (esto es, flujos directos de información de una variable a otra). El análisis de manchas tiende a ignorar los flujos implícitos [13] (esto es, flujos a través de las estructuras de control del lenguaje).

La figura 3.11 muestra un flujo implícito. Las variables `t` y `u` están manchadas y no manchadas, respectivamente. Observemos que la variable `u` no está manchada luego de la ejecución porque se le asigna un valor no manchado (`'a'` o `''`). El valor de la variable manchada `t` se copia a una variable no manchada `u` cuando `t == 'a'`. No es difícil

imaginar programas que esquiven el análisis de manchas copiando el contenido de strings manchados en variables no manchadas usando flujos implícitos [31].

```
1 if t == 'a':
2     u = 'a'
3 else:
4     u = ''
```

Figura 3.11: Un flujo implícito

En escenarios donde los atacantes tienen control total sobre el código ejecutado (es decir que el código es potencialmente malicioso), los flujos implícitos presentan una forma efectiva de saltar la protección de un análisis de manchas. En este caso, el objetivo del atacante es alterar código y datos de entrada para evitar el mecanismo de seguridad. Hay mucha literatura en el área de seguridad basada en lenguajes referente a cómo controlar flujos implícitos [32].

Por otro lado, hay escenarios donde el código es no malicioso (esto es, escrito sin maldad). A pesar de las buenas intenciones y la experiencia de los programadores, el código aún puede contener vulnerabilidades. El objetivo del atacante consiste en manipular los datos de entrada para poder explotar vulnerabilidades y/o corromper datos. En este escenario, el análisis de manchas ciertamente ayuda a encontrar vulnerabilidades.

¿Qué tan peligrosos son los flujos implícitos en código no malicioso? Generalmente son inofensivos [31]. La razón de esto recae en que programadores no maliciosos necesitan escribir código muy rebuscado para, por ejemplo, copiar un string manchado en una variable no manchada. En contraste, para producir flujos explícitos, los programadores sólo tienen que olvidarse de llamar a una función de sanitización. En este trabajo, considero escenarios donde el código analizado es no malicioso.

3.3. Versión segura de «Sugerencias para el almuerzo»

En la sección 1.3 mostré «Sugerencias para el almuerzo», una aplicación web usada en una oficina para organizar lo que los empleados almuerzan cada día. También mostré algunos problemas de seguridad que tenía la aplicación.

En esta sección mostraré cómo utilizar la biblioteca `taintmode.py` para asegurar la aplicación.

Para desplegar el análisis de manchas lo único que tenemos que agregar son unas pocas líneas al inicio del programa original (ver figura 3.12).

```
1 from taintmode import *
2 web.input = untrusted(web.input)
3 os.system = ssink(OSI)(os.system)
4 import taintmode
5 taintmode.ends_execution()
```

Figura 3.12: Versión segura de «Sugerencias para el almuerzo»

Más específicamente, importamos la API del módulo `taintmode`, marcamos `web.input` como una fuente no confiable y `os.system` como un sumidero sensible a Inyecciones en el Sistema Operativo (`OSI`). Finalmente, ejecutamos `taintmode.ends_execution()` para decir explícitamente que queremos que la ejecución se detenga si ocurre un problema. De lo contrario el flujo de ejecución del programa sigue.

Si ahora intentamos enviar algún dato usando el formulario de la aplicación, veremos que nada sucede. ¿Nada? No vemos nada en la página web porque la ejecución se cortó. Si vemos el archivo de log de la aplicación, veremos algo como la salida capturada en la figura 3.13.

```
============================================================
Violation in line 39 from file /home/juanjo/python/taintmode/
    webdemo/code.py
Tainted value: echo Rice and Chiken. >> 2010 10 10.txt
============================================================
        meal = web.input().meal
        # save it to the file of the day
        dayfile = datetime.today().strftime('%Y%m%d') + '.txt'
==>     os.system("echo " + meal + " >> " + dayfile)
        raise web.seeother('/')



============================================================
```

Figura 3.13: Ejecución cortada en «Sugerencias para el almuerzo»

A pesar de que los datos de entrada no eran dañinos la ejecución se cortó. Un valor con la mancha `OSI` alcanzó un sumidero sensible a `OSI`. ¿Cómo arreglamos la aplicación para que sea útil a la vez que detiene a los atacantes? La aplicación necesita usar una función limpiadora. En la figura 3.14 se puede ver que la importamos y marcamos como capaz de limpiar datos contra `OSI`.

```
1 from cleaners import clean_osi
2 clean_osi = cleaner(OSI)(clean_osi)
```

Figura 3.14: Importar una función limpiadora

El paso final se ve en la figura 3.15 y es *usar* `osi_cleaner` en el código.

```
class add:
    def POST(self):
        user = web.input().user
        meal = clean_osi(web.input().meal)
        meal = web.input().meal
        # save it to the file of the day
        dayfile = datetime.today().strftime('%Y%m%d') + '.txt
            '
        os.system("echo " + meal + " >> " + dayfile)
        raise web.seeother('/')
```

Figura 3.15: Uso de la función limpiadora

Se observa que el código añadido es mínimo y tan poco intrusivo como se pudo. Al usar la biblioteca, depende del programador decidir dónde se hace la sanitización.

Capítulo 4

Implementación

En este capítulo abordo los detalles internos de implementación de la biblioteca explicando qué técnicas y características propias del lenguaje Python se utilizaron en su implementación.

La siguiente descripción intentará ser lo más exhaustiva posible.

4.1. Mecanismos dinámicos

Una de las partes principales de la biblioteca consiste en los mecanismos necesarios para seguir la traza de la información de las manchas en los objetos.

La biblioteca define subclases de los tipos primitivos que vienen con el lenguaje (como `str` o `int`) para agregarles la capacidad de determinar si una instancia de esa clase está manchada o no. Un objeto de estas clases tiene un atributo (`taints`) cuyo valor es un conjunto (tipo de dato `set` en Python) de etiquetas asociadas a él. Cuando el conjunto de etiquetas está vacío, podemos decir que el objeto no está manchado.

En el contexto de la biblioteca, estas clases que extienden a las provistas originalmente por el lenguaje son llamadas *taint-aware*. Además de agregar este atributo, estas clases se encargan de sobreescribir automáticamente los métodos de sus clases padre para que tengan en cuenta al atributo `taints`. Esto es, propagar la información de las manchas a medida que se realizan operaciones entre objetos. La propagación de la información de las manchas se logra actualizando el valor de `taints` según corresponda.

Por ejemplo, supongamos que `a` tiene un valor de `taints` igual a `s1` y `b` tiene un valor de `taints` igual a `s2`. Si realizamos la operación `a + b`, es de esperar que el objeto resultante tenga para `taints` un valor igual a la unión de `s1` y `s2`. La figura 4.1 muestra este ejemplo.

```
>>> a = taint('a', XSS)
>>> b = taint('b', SQLI)
>>> a.taints
set([1])
>>> b.taints
set([2])
>>> c = a + b
>>> c.taints
set([1, 2])
```

Figura 4.1: Propagación de manchas

La figura 4.1 también sirve para ver un detalle de implementación de la biblioteca y es que las etiquetas son representadas con números enteros y que hay una serie de nombres definidos que hacen referencia a estos valores. La definición de las etiquetas que maneja por defecto la biblioteca puede verse en la figura 4.2.

```
KEYS  = [XSS, SQLI, OSI, II] = range(1, 5)
TAGS = set(KEYS)
```

Figura 4.2: Definición de etiquetas

Volviendo al mecanismo de propagación, lo que sucede a nivel implementación es que los métodos de las clases taint-aware retornan un objeto cuyo atributo `taints` es la unión de las etiquetas encontradas en los argumentos del método y en el propio objeto sobre el que se llamó al método.

En Python, los mecanismos de ejecución dinámica (cuyas reglas se explican en la sección 5.4) garantizan, por ejemplo, que al concatenar un objeto manchado con un objeto no manchado el proceso se lleve a cabo ejecutando un método del objeto manchado (instancia de la clase taint-aware), lo que permite propagar adecuadamente

la información de las manchas.

4.2. Generación de clases taint-aware

`taint_class`

La figura 4.3 presenta una función para generar clases taint-aware. La función toma como argumentos una clase del lenguaje (`klass`) y la lista de métodos (`methods`) en los que se debe llevar a cabo propagación de manchas.

```
1  def taint_class(klass, methods):
2      class tklass(klass):
3        def __new__(cls, *args, **kwargs):
4            self = super(tklass, cls).__new__(cls, *args, **
                 kwargs)
5            self.taints = set()
6            return self
7      d = klass.__dict__
8      for name, attr in [(m, d[m]) for m in methods]:
9          if inspect.ismethod(attr) or
10            inspect.ismethoddescriptor(attr):
11               setattr(tklass, name, propagate_method(attr))
12     if '__add__' in methods and '__radd__' not in methods:
13         setattr(tklass, '__radd__',
14                 lambda self, other: tklass.__add__(tklass(other
                       ),
15                  self))
16     return tklass
```

Figura 4.3: Función para generar clases taint-aware

La línea 2 define una nueva clase, `tklass`, como subclase de `klass`. Se sobrecarga el método `__new__` para agregar a las instancias de `tklass` el atributo `taints` con el conjunto vacío (`set()`) como valor (líneas 3 a 6).

En la línea 7 se asigna a `d` el atributo `__dict__` que contiene, entre otras cosas, todos los métodos de la clase (introspección).

Por cada método que esté en la clase original y en el argumento `methods` (líneas 8 a 10) se agrega a la clase `tklass` un método con el mismo nombre, mismos argumentos y que retorna el mismo resultado pero que también propaga la información de las manchas (línea 11). El argumento `methods` es necesario para tener control sobre qué métodos son sobrecargados para solucionar algunos problemas de implementación (ver sección 5.3)

La función `propagate_method` se explica en la siguiente sección.

La generalidad de la función es interrumpida de las líneas 12 a 15 con un detalle de implementación necesario a los fines prácticos. Se incluye el método `__radd__` en las clases taint-aware donde la clase original tenía el método `__add__` pero no este otro. El método `__radd__` es llamado para implementar la operación binaria con operandos intercambiables[1].

En un ejemplo inverso al de la introducción, para evaluar la expresión `b + a`, dónde `b` es un string y `a` es un string taint-aware, Python llama al método `__radd__` de `a`. Lo que realmente se ejecuta es `a.__radd__(b)`. De esta forma, la información de las manchas de `a` es propagada en la expresión. Si se ejecutara `b.__add__(a)`, no se podría realizar la propagación de las manchas y el resultado sería un string sin manchar.

Finalmente se retorna la clase taint-aware (línea 16).

`propagate_method`

En la figura 4.4 se muestra la implementación de la función `propagate_method`. El objetivo de la función es implementar el mecanismo de propagación de la información de las manchas dentro de cada método. La función toma como argumento un método y retorna uno nuevo que recibe los mismos argumentos y calcula el mismo resultado pero además propaga la información de las manchas.

[1] La clase `str` implementa todas las versiones reflexivas de sus operadores excepto la de `__add__`.

```
1  def propagate_method(method):
2      def inner(self, *args, **kwargs):
3          r = method(self, *args, **kwargs)
4          t = set()
5          for a in args:
6              collect_tags(a, t)
7          for v in kwargs.values():
8              collect_tags(v, t)
9          t.update(self.taints)
10         return taint_aware(r,t)
11     return inner
```

Figura 4.4: Propagación de la información de las manchas

En la línea 2 comienza la definición de `inner`, el nuevo método que se retornará. En la línea 3 se llama al método recibido como argumento y se almacena su resultado en la variable `r`. De las líneas 4 a 9 se colectan en la variable `t` las etiquetas del objeto cuyo método se está ejecutando y de los argumentos del método. El resultado `r` puede ser un objeto de una clase no taint-aware y por lo tanto no tener el atributo `taints`. Por esta razón, la función `taint_aware` está diseñada para transformar objetos de clases del lenguaje en objetos de clases taint-aware.

Por ejemplo, si `r` es un objeto de la clase `str`, la función `taint_aware` retorna un objeto instancia de una clase taint-aware derivada de `str`. Más aún, si `r` es una lista de objetos de la clase `str`, la función `taint_aware` retorna una lista de objetos instancia de esta clase taint-aware.

La función `taint_aware` se implementó de forma tal que realiza un mapeo en las estructuras de datos contenedoras: listas, tuplas, conjuntos y diccionarios. La biblioteca no mancha a las estructuras contenedoras, sino a sus elementos. Esta es una decisión de diseño basada en la presunción de que no existe código no malicioso que, haciendo uso de contenedores, pueda evitar el análisis de las manchas. Por ejemplo, codificar en la longitud de listas el valor de un entero manchado. Si no fuera así, la biblioteca puede ser fácilmente adaptada.

En la línea 11 se retorna la versión taint-aware de `r` con las etiquetas colectadas en `t`.

Ejemplo

Para demostrar el uso de la función `taint_class`, la figura 4.5 crea clases taint-aware para enteros y strings. `str_methods` e `int_methods` son listas de nombres de métodos para las clases `str` e `int`, respectivamente. Veamos cómo el código presentado en las figuras 4.3 y 4.4 es suficientemente genérico como para aplicarse a distintas clases que vienen con el lenguaje.

```
STR = taint_class(str, str_methods)
INT = taint_class(int, int_methods)
```

Figura 4.5: Clases taint-aware para enteros y strings

4.3. Decoradores

A excepción de `taint` y `tainted`, el resto de la API es implementado como decoradores. En Python, los decoradores son funciones de alto nivel [10], esto es, funciones que toman otras como parámetros y devuelven funciones.

`untrusted`

La figura 4.6 muestra el código de `untrusted`.

```
1 def untrusted(f):
2     def inner(*args, **kwargs):
3         r = f(*args, **kwargs)
4         return taint_aware(r, TAGS)
5     return inner
```

Figura 4.6: Código de `untrusted`

En la línea 1, la función argumento `f` es la la función que retorna resultados no confiables.

La función `untrusted` retorna una función (`inner`) que llama a la función `f` (línea 3) y mancha los valores retornados por ésta (línea 4).

Como se vio en la introducción, `TAGS` es el conjunto de todas las etiquetas soportadas por la biblioteca.

`untrusted_args`

En la figura 4.7 se muestra el código del decorador `untrusted_args`.

```
1  def untrusted_args(nargs=[], nkwargs=[]):
2      def _untrusted_args(f):
3          def inner(*args, **kwargs):
4              args = list(args)
5              for n in nargs:
6                  args[n] = mapt(args[n], taint)
7              for n in nkwargs:
8                  kwargs[n] = mapt(kwargs[n], taint)
9              r = f(*args, **kwargs)
10             return r
11         return inner
12     return _untrusted_args
```

Figura 4.7: Código de `untrusted_args`

A diferencia del decorador anterior, al ser éste un decorador con argumentos, se necesita un nivel extra en la definición de funciones de orden superior.

En la línea 1 se reciben los parámetros que configurarán al decorador. `nargs` es una lista de posiciones. Los argumentos posicionales en estas posiciones serán manchados para todos los tipos de manchas. `nkwargs` es una lista de strings. Los argumentos de palabra clave para esas claves serán manchados para todos los tipos de manchas.

La función `untrusted_args` retorna una función (`_untrusted_args`) que recibe como parámetro la función a decorar `f`.

`inner` es la función que se terminará llamando cuando ejecutemos la función que hayamos decorado. Lo primero que hace es convertir `args`, una tupla (estructura de datos inmutable en Python), en una lista para poder manipularla (línea 4). Luego se utiliza la función `mapt` para manchar cada uno de los argumentos cuyas posiciones o palabras clave coinciden con los especificados en los parámetros de la

función `untrusted_args` (líneas 5 a 8). Finalmente se ejecuta la función decorada con los parámetros correspondientes manchados y el resultado es retornado.

`cleaner`

El código del decorador `cleaner` se lista en la figura 4.8. Al igual que `untrusted_args` es un decorador con argumentos.

```
1 def cleaner(v):
2     def _cleaner(f):
3         def inner(*args, **kwargs):
4             r = f(*args, **kwargs)
5             remove_tags(r, v)
6             return r
7         return inner
8     return _cleaner
```

Figura 4.8: Código de `cleaner`

En la línea 1, `v` es la etiqueta que la función limpiadora decorada será capaz de limpiar. En la línea 2, `f` es la función limpiadora a decorar.

La función `_cleaner` retorna una función (`inner`) que llama a la función `f` (línea 4) y elimina la etiqueta `v` del resultado (línea 5).

`validator`

El código del decorador `validator` (figura 4.9) es similar al de `cleaner` pero con una complejidad extra. En la línea 1, además del argumento `v`, vemos el argumento `cond` (explicado a continuación) y los argumentos `nargs` y `nkwargs` (similares a los de `untrusted_args`).

```
1  def validator(v, cond=True, nargs=[], nkwargs=[]):
2      def _validator(f):
3          def inner(*args, **kwargs):
4              r = f(*args, **kwargs)
5              if r == cond:
6                  tovalid = set(args[n] for n in nargs)
7                  tovalid.update(kwargs[n] for n in nkwargs)
8                  for a in tovalid:
9                      remove_tags(a, v)
10             return r
11         return inner
12     return _validator
```

Figura 4.9: Código de `validator`

`nargs` y `nkwargs` definen los argumentos que la función validadora está revisando. Si el resultado de la función validadora es igual a `cond`, entonces la etiqueta `v` debe borrarse de los argumentos validados (líneas a 5 a 9).

ssink

La figura 4.10 muestra el código del decorador `ssink`.

```
def ssink(v=None, reached=reached):
    def _solve(a, f, args, kwargs):
        if ENDS:
            if RAISES:
                reached(a)
                raise TaintException()
            else:
                return reached(a)
        else:
            reached(a)
            return f(*args, **kwargs)

    def _ssink(f):
        def inner(*args, **kwargs):
            allargs = chain(args, kwargs.itervalues())
            if v is None:   # sensitive to ALL
                for a in allargs:
                    t = set()
                    collect_tags(a, t)
                    if t:
                        return _solve(a, f, args, kwargs)
            else:
                for a in allargs:
                    t = set()
                    collect_tags(a, t)
                    if v in t:
                        return _solve(a, f, args, kwargs)
            return f(*args, **kwargs)
        return inner
    return _ssink
```

Figura 4.10: Código de `ssink`

De la línea 2 a 11 se define la función auxiliar `_solve` de la que nos ocuparemos más tarde.

En la línea 13, la función argumento `f` es la la función sumidero sensible a la que no queremos que lleguen valores manchados con cierta etiqueta (`v`).

La función `_ssink` retorna una función (`inner`) que primero arma un iterador (`allargs`) con todos los argumentos del sumidero (línea 15). Si `v` es `None` revisa si alguno de los argumentos tiene alguna mancha (cualquiera); si esta condición se da, se llama a `_solve`. Si, caso contrario, `v` tiene especificado un valor de etiqueta, entonces se llama a `_solve` sólo si esta etiqueta está presente en alguno de los argumentos de la función sumidero (líneas 16 a 27). Si nada de lo anterior sucede, se ejecuta el sumidero normalmente (línea 28).

La función `_solve` se comporta de la siguiente manera. Utiliza las configuraciones globales `ENDS` y `RAISES` para determinar si corta o no la ejecución y, en caso de cortarla, si se debe lanzar una excepción. En cualquiera de los 3 casos, siempre se ejecuta la función `reached` que por defecto imprime en la salida estándard un mensaje explicando qué pasó, similar al de la figura 3.13 en el capítulo anterior. Esta función es configurable mediante el argumento `reached` del decorador `ssink`.

4.4. Funciones taint-aware

Muchos de los análisis de manchas mencionados en el capítulo 2 [3, 26, 19, 20, 16, 34] no propagan la información de las manchas cuando el valor resultante de una operación que involucra algún valor manchado no es un string. Por ejemplo, el cálculo de la longitud de un string manchado resulta en un entero no manchado.

Esta decisión de diseño puede afectar la habilidad del análisis de manchas para detectar vulnerabilidades. Por ejemplo, podría no considerar patrones peligrosos como el de codificar strings como listas de números.

Una forma común de resolver este problema es marcar las funciones que realizan codificaciones sobre strings como sumideros sensibles. De esta forma, se fuerza que la sanitización ocurra antes de que los strings sean representados en otro formato.

De todas formas, este enfoque no es satisfactorio: se pierde el significado intrínseco de lo que es un sumidero sensible. Los sumideros sensibles son operaciones críticas en seguridad, no funciones que realizan codificación de strings.

La biblioteca propone una forma de solucionar este problema. La figura 4.11 muestra el uso de una función genérica que sirve para sobreescribir funciones del lenguaje con funciones taint-aware, esto es: funciones que devuelven objetos taint-aware si su argumento es taint-aware y conservan las manchas.

```
1 len = propagate_func(len)
2 ord = propagate_func(ord)
3 chr = propagate_func(chr)
```

Figura 4.11: Funciones taint-aware para strings y enteros

Por ejemplo, la función `len` de Python se utiliza para obtener la longitud de una secuencia, como ser un string. Si el string en cuestión está manchado, queremos que su longitud, es decir, el entero resultante de llamar a la función `len`, también esté manchado.

En el ejemplo se redefinen las funciones que vienen con el lenguaje para computar: la longitud de una secuencia (`len`), el código ASCII de un carácter (`ord`) y su inversa (`chr`). Como resultado, `len(taint('string'))` retornará el entero manchado `6`.

Está en manos de los usuarios de la biblioteca decidir qué funciones deben ser taint-aware dependiendo del escenario de uso.

La figura 4.12 muestra el código de `propagate_func`. Al igual que el código mostrado en la figura 4.4, vemos una función de orden superior. Toma como argumento una función `f` y retorna otra función (`inner`) capaz de propagar la información de las manchas de los argumentos al resultado.

```
1  def propagate_func(original):
2      def inner(*args, **kwargs):
3          t = set()
4          for a in args:
5              collect_tags(a,t)
6          for v in kwargs.values():
7              collect_tags(v,t)
8          r = original(*args,**kwargs)
9          if t == set([]):
10             return r
11         return taint_aware(r,t)
12     return inner
```

Figura 4.12: Propagación de la información de las manchas a través de diferentes objetos taint-aware

De las líneas 3 a 7 se obtienen las etiquetas de los argumentos. Si el conjunto de las etiquetas recolectadas está vacío, no hay valores manchados involucrados y por lo tanto no se realiza propagación de manchas (líneas 9 y 10). Caso contrario, se retorna una versión taint-aware del resultado con las etiquetas recolectadas en los argumentos (línea 11).

4.5. Más detalles de implementación

Para más detalles sobre la implementación se puede consultar el código fuente de la biblioteca.

El mismo está disponible en Internet bajo la licencia GNU General Public License (ver http://www.juanjoconti.com.ar/taint/) y en el apéndice A.

Capítulo 5

Limitaciones

En este capítulo analizo el modo de abordar ciertas limitaciones que surgen del enfoque propuesto y de la tecnología utilizada para la implementación. Se han detectado durante el desarrollo y uso de la biblioteca.

5.1. Limitaciones de ámbito

Depende de los usuarios de la biblioteca decidir qué funciones y clases del lenguaje serán taint-aware. Esto se debe al objetivo de que la biblioteca sea flexible y no afecte el desempeño del código ejecutado a menos que sea necesario. Si un usuario está interesado en realizar una análisis de manchas solo sobre strings, no tiene por qué pagar el precio de que la información de las manchas también se lleve para enteros.

Cabe remarcar que la biblioteca sólo mantiene la información de las manchas en el código fuente que se está desarrollando. La información de las manchas se podría perder si, por ejemplo, los valores manchados se pasan a bibliotecas externas (o bibliotecas escritas en otros lenguajes) que no son taint-aware. Una forma de solucionar este problema es convertir las funciones de bibliotecas externas aplicándoles la función `propagate_func`. Esto funcionará solo si la función no tiene efectos colaterales.

5.2. La clase `bool`

En la figura 4.3, el método que produce automáticamente clases taint-aware no funciona con la clase `bool`. La razón es que esta clase no puede ser extendida en Python[1].

Si bien la restricción es arbitraria y no es justificada por una razón técnica, Guido van Rossum, el creador del lenguaje, ha manifestado en la lista de correos del desarrollo del lenguaje que no se permite extender la clase `bool` por que esto sólo sería útil si también se le permitiera tener instancias, pero éstas también serían instancias de `bool` y esto rompería la invariante de que `True` y `False` son las únicas instancias de `bool`[2].

Es por esto que la biblioteca no puede manejar booleanos manchados. Esto no restringe significativamente el uso que se pueda hacer de la biblioteca por dos razones. En primer lugar, los valores booleanos son típicamente usados en condiciones. Como la biblioteca ya ignora a los flujos implícitos (ver sección 3.2), las posibilidades de encontrar vulnerabilidades no se ven reducidas drásticamente al no poder seguir la información de las manchas en booleanos. Segundo, las operaciones booleanas `and` y `or` en Python (por ejemplo `a and b`) son convertidas a una sentencia que utiliza estructuras de control de flujo `if/else` (`b if a else a`) y no se lleva a cabo ninguna conversión al tipo bool. Es decir que si los objetos involucrados son taint-aware, el resultado de esta operación lo será también, reduciendo el número de casos en los que la información de las manchas es perdida.

5.3. Limitaciones en métodos

Al generar la clase taint-aware `STR` (figura 4.5), se encontraron algunos problemas al trabajar con algunos métodos de la clase `str`. El intérprete de Python lanzaba una excepción cuando se quería redefinir algunos métodos. Para otros, como `__new__`, `__init__`, `__getattribute__` y `__repr__` que se redefinían en la creación de las clases, se producía una recursión infinita al llamarlos. Tanto para la generación de la clase `STR` como para `INT` y otras, se notó el mismo comportamiento (métodos

[1] `http://docs.python.org/library/functions.html#bool`

[2] Guido van Rossum sobre extender la clase bool en la lista de correos python-dev: `http://mail.python.org/pipermail/python-dev/2002-March/020822.html`

que no se podían sobreescribir por distintas razones). Estas restricciones no impactan drásticamente en las habilidades de detectar vulnerabilidades.

El método `__new__` se llama al crear objetos. En la figura 4.3, para las clases taint-aware se define este método en la línea 3. El método `__init__` se llama para inicializar el objeto. Python llama a estos dos métodos para crear objetos y los programas no suelen llamarlos explícitamente.

El método `__getattribute__` es usado para acceder a cualquier atributo de un objeto. Este método es heredado de `klass` y funciona como se espera para las clases taint-aware.

Teniendo en cuenta lo anterior, cabe mencionar que el argumento `methods` en la función `taint_class` establece los métodos que serán redefinidos en las clases taint-aware (figura 4.3).

5.4. Problemas específicos

Durante el uso de la biblioteca, se descubrieron algunos problemas específicos. En la batería de pruebas (ver apéndice B) hay 3 tests comentados que los ponen de manifiesto.

En esta sección, a modo de ejemplo, voy a describir uno de ellos: el problema del operador % al combinar objetos de la clase `str` y la clase `unicode`. Si bien la explicación puede ser muy técnica y altamente específica, creo importante incluirla para demostrar el nivel de detalle que se tiene que tener al construir una biblioteca de este tipo.

En la figura 5.1 se ve claramente el problema. Primero se crean dos objetos manchados. `s` es un objeto taint-aware creado a partir de un objeto de la clase `str` y `u` es un objeto taint-aware creado a partir de un objeto de la clase `unicode`.

```
1 >>> s = taint('s')
2 >>> u = taint(u'u')
3 >>> tainted('%s' % s)
4 True
5 >>> tainted(u'%s' % u)
6 True
7 >>> tainted('%s' % u)
8 False
```

Figura 5.1: El problema del operador %

La operación de formateo de la línea 3 es convertida a `s.__rmod__("%s")`. Como `s` es un objeto taint-aware, puede hacer que el resultado de la operación también sea un objeto taint-aware con las manchas correspondientes. En la línea 5 pasa algo parecido. ¿Por qué no pasa lo mismo en la línea 7? La razón está en las reglas de coerción de Python[3].

Cuando se hace `a % b` (independientemente del tipo de las variables):

- Lo primero que se intenta hacer es `a.__mod__(b)` (si existe y no devuelve `NotImplemented`, ese es el resultado).
- El segundo intento es `b.__rmod__(a)`. *Excepto* que `b` sea una instancia de una subclase de la de `a`, en ese caso esta variante se prueba primero.

En los ejemplos de las líneas 3 y 5, como el objeto de la derecha es instancia de una subclase de la clase del objeto de la izquierda, se ejecuta el método `__rmod__` del objeto de la derecha. En el caso de la línea 7, la clase del objeto de la derecha (taint-aware que extiende a `unicode`) no es subclase de la clase del objeto de la izquierda (`str`) por lo que se ejecuta el método `__mod__` del objeto de la izquierda.

[3]Ver ítems 6 y 7 de `http://docs.python.org/reference/datamodel.html#coercion-rules`

Capítulo 6

Conclusiones

Se ha descrito una biblioteca para análisis de manchas escrita enteramente en Python en la cual no se necesitan modificaciones en el intérprete y las modificaciones en el código a analizar son mínimas.

Dentro de las dos alternativas básicas para su implementación, análisis estático y análisis dinámico, se eligió realizar una implementación dinámica ya que estas logran menos falsos positivos.

Al realizar la implementación en forma de una biblioteca, en lugar de modificar el intérprete, se pudo dedicar esfuerzo a desarrollar características innovadoras en lugar de lidiar con detalles de implementación del lenguaje.

Una de las principales características de la biblioteca es que permite rastrear información de manchas en distintos tipos de datos, no sólo strings, sino también otros tipos de datos del lenguaje e incluso tipos de datos definidos por el usuario. Como consecuencia de esta característica, también se ha incluido la posibilidad de tener funciones que reciben objetos manchados de un tipo de datos y retornan objetos manchados de otro tipo; otra característica no presente en la mayoría de las implementaciones.

A diferencia de los enfoques tradicionales, en lugar de distinguir a los objetos manchados con una bandera booleana, cada uno cuenta con un atributo que es un conjunto de manchas que el objeto tiene en un determinado momento. Que un objeto pueda tener distinto tipo de machas, permite que el análisis tenga mayor granularidad con lo que se reducen aún más los falso positivos.

Otra decisión de diseño que ha demostrado ser de mucha utilidad es que las fuentes

no confiables y los sumideros sensibles no estén predefinidos, sino que puedan ser definidos por los usuarios de la biblioteca con mucha libertad.

Para aplicar el análisis de manchas a los programas, sólo se necesita indicar las fuentes no confiables, sumideros sensibles y funciones de sanitización. La biblioteca usa decoradores como una forma no invasiva de marcar el código fuente. Estos y los mecanismos de ejecución dinámica de Python son los que permiten que el análisis se ejecute casi sin modificaciones en el código analizado.

Se ha logrado una implementación de pocas líneas de código y particularmente elegante. La biblioteca tiene alrededor de 350 líneas de código fuente.

La biblioteca se encuentra disponible en Internet para quien quiera extenderla para darle usos no contemplados originalmente.

El hecho de que la biblioteca tenga trazabilidad de manchas a través de distintos tipos de datos, abre las puertas a nuevos escenarios en los cuales realizar trabajos futuros como el de llevar estas herramientas de seguridad a las nuevas plataformas de cloud computing.

Apéndice

A. Código fuente de la biblioteca

```
# * coding: utf 8 *
'''
Taint Mode for Python via a Library

Copyright 2009 Juan Jose Conti
Copyright 2010 Juan Jose Conti & Alejandro Russo
Copyright 2011 Juan Jose Conti & Alejandro Russo
Copyright 2012 Juan Jose Conti & Alejandro Russo

This file is part of taintmode.py

taintmode is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

taintmode.py is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with taintmode.py. If not, see <http://www.gnu.org/licenses/>.

'''
import inspect
import sys
from itertools import chain

__all__ = ['tainted', 'taint', 'untrusted', 'untrusted_args', 'ssink',
           'validator', 'cleaner', 'STR', 'INT', 'FLOAT', 'UNICODE', 'chr',
           'ord', 'len', 'ends_execution', 'XSS', 'SQLI', 'OSI', 'II']

ENDS = False
RAISES = False
KEYS  = [XSS, SQLI, OSI, II] = range(1, 5)
TAGS = set(KEYS)


class TaintException(Exception):
    pass


def ends_execution(b=True):
    global ENDS
    ENDS = b


# ======================== Taint aware functions ============================
```

```
50  def propagate_func(original):
51      def inner (*args, **kwargs):
52          t = set()
53          for a in args:
54              collect_tags(a, t)
55          for v in kwargs.values():
56              collect_tags(v, t)
57          r  = original(*args, **kwargs)
58          if t:
59              r = taint_aware(r, t)
60          return r
61      return inner
62
63  len = propagate_func(len)
64  ord = propagate_func(ord)
65  chr = propagate_func(chr)
66
67  # ======================== Auxiliaries functions ============================
68
69  def mapt(o, f, check=lambda o: type(o) in tclasses):
70      if check(o):
71          return f(o)
72      elif isinstance(o, list):
73          return [mapt(x, f, check) for x in o]
74      elif isinstance(o, tuple):
75          return tuple(mapt(x, f, check) for x in o)
76      elif isinstance(o, set):
77          return  set(mapt(x, f, check) for x in o)
78      elif isinstance(o, dict):
79          klass = type(o) # It's quite common for frameworks to extend dict
80                          # with useful new methdos   i.e. web.py
81          return klass((k, mapt(v, f, check)) for k, v in o.iteritems())
82      else:
83          return o
84
85
86  def remove_taint(v):
87      def _remove(o):
88          if hasattr(o, 'taints'):
89              o.taints.discard(v)
90      return _remove
91
92
93  def remove_tags(r, v):
94      mapt(r, remove_taint(v), lambda o: True)
95
96
97  def collect_tags(s, t):
98      '''Collect tags from a source s into a target t.'''
99      mapt(s, lambda o: t.update(o.taints), lambda o: hasattr(o, 'taints'))
100
101
102 def update_tags(r, t):
103     mapt(r, lambda o: o.taints.update(t), lambda o: hasattr(o, 'taints'))
104
105
106 def taint_aware(r, ts=set()):
107     r = mapt(r, tclass)
108     update_tags(r, ts)
109     return r
110
111
112 # ======================== Decorators ======================================
113
114 def untrusted_args(nargs=[], nkwargs=[]):
115     '''
116     Mark a function or method that would recive untrusted values.
117
118     nargs is a list of positions. Positional arguments in that position will be
119     tainted for all the types of taint.
120
121     nkwargs is a list of strings. Keyword arguments for those keys will be
122     tainted for all the types of taint.
123     '''
```

```
124     def _untrusted_args(f):
125         def inner(*args, **kwargs):
126             args = list(args)
127             for n in nargs:
128                 args[n] = mapt(args[n], taint)
129             for n in nkwargs:
130                 kwargs[n] = mapt(kwargs[n], taint)
131             r = f(*args, **kwargs)
132             return r
133         return inner
134     return _untrusted_args
135
136 def untrusted(f):
137     '''
138     Mark a function or method as untrusted.
139
140     The returned value will be tainted for all the types of taint.
141     '''
142     def inner(*args, **kwargs):
143         r = f(*args, **kwargs)
144         return taint_aware(r, TAGS)
145     return inner
146
147 def validator(v, cond=True, nargs=[], nkwargs=[]):
148     '''
149     Mark a function or method as capable to validate its input.
150
151     nargs is a list of positions. Positional arguments in that positions are
152     the ones validated.
153
154     nkwargs is a list of strings. Keyword arguments for those keys are the ones
155     validated.
156
157     If the function returns cond, v will be removed from the the validated
158     inpunt.
159     '''
160     def _validator(f):
161         def inner(*args, **kwargs):
162             r = f(*args, **kwargs)
163             if r == cond:
164                 tovalid = set(args[n] for n in nargs)
165                 tovalid.update(kwargs[n] for n in nkwargs)
166                 for a in tovalid:
167                     remove_tags(a, v)
168             return r
169         return inner
170     return _validator
171
172
173 def cleaner(v):
174     '''
175     Mark a function or methos as capable to clean its input.
176
177     v tag is removed from the returned value.
178     '''
179     def _cleaner(f):
180         def inner(*args, **kwargs):
181             r = f(*args, **kwargs)
182             remove_tags(r, v)
183             return r
184         return inner
185     return _cleaner
186
187 def reached(t, v=None):
188     '''
189     Execute if a tainted value reaches a sensitive sink
190     for the vulnerability v.
191
192     If the module level variable ENDS is set to True, then the sink is not
193     executed and the reached function is executed instead. If ENDS is set to False
        ,
194     the reached function is executed but the program continues its flow.
195
196     The provided de facto implementation alerts that the violation happened and
```

```
197         information to find the error.
198         '''
199         frame = sys._getframe(3)
200         filename = inspect.getfile(frame)
201         lno = frame.f_lineno
202         print "=" * 79
203         print "Violation in line %d from file %s" %(lno, filename)
204         print "Tainted value: %s" %t
205         print ' ' * 79
206         lineas = inspect.findsource(frame)[0]
207         lineas = ['    %s' %l for l in lineas]
208         lno = lno   1
209         lineas[lno] = '==> ' + lineas[lno][4:]
210         lineas = lineas[lno   3: lno + 3]
211         print "".join(lineas)
212         print "=" * 79
213
214     def ssink(v=None, reached=reached):
215         '''
216         Mark a function or method as sensitive to tainted data.
217
218         If it is called with a value with the v tag
219         (or any tag if v is None),
220         it's not executed and reached is executed instead.
221
222         These sinks are sensitive to a kind of vulnerability, and must be specified
                when
223         the decorator is used.
224         '''
225         def _solve(a, f, args, kwargs):
226             if ENDS:
227                 if RAISES:
228                     reached(a)
229                     raise TaintException()
230                 else:
231                     return reached(a)
232             else:
233                 reached(a)
234                 return f(*args, **kwargs)
235
236         def _ssink(f):
237             def inner(*args, **kwargs):
238                 allargs = chain(args, kwargs.itervalues())
239                 if v is None:   # sensitive to ALL
240                     for a in allargs:
241                         t = set()
242                         collect_tags(a, t)
243                         if t:
244                             return _solve(a, f, args, kwargs)
245                 else:
246                     for a in allargs:
247                         t = set()
248                         collect_tags(a, t)
249                         if v in t:
250                             return _solve(a, f, args, kwargs)
251                 return f(*args, **kwargs)
252             return inner
253         return _ssink
254
255     def tainted(o, v=None):
256         '''
257         Tells if a value o, a tclass instance, is tainted for the given
258         vulnerability v.
259
260         If v is not provided, checks for all taints. If the value is tainted
261         with at least one vulnerability, returns True.
262         '''
263         if not hasattr(o, 'taints'):
264             return False
265         if v is not None:
266             return v in o.taints
267         return bool(o.taints)
268
269     def taint(o, v=None):
```

```
270     '''
271     Helper function for taint the value o with the vulnerability v.
272
273     If v is not provided, taint with all types of taints.
274     '''
275     ts = set()
276     if v is not None:
277         ts.add(v)
278     else:
279         ts.update(TAGS)
280
281     return taint_aware(o, ts)
282
283 # ======================== Taint aware classes ==============================
284
285 def propagate_method(method):
286     def inner(self, *args, **kwargs):
287         r = method(self, *args, **kwargs)
288         t = set()
289         for a in args:
290             collect_tags(a, t)
291         for v in kwargs.values():
292             collect_tags(v, t)
293         t.update(self.taints)
294         return taint_aware(r, t)
295     return inner
296
297
298 def taint_class(klass, methods=None):
299     if not methods:
300         methods = attributes(klass)
301     class tklass(klass):
302         def __new__(cls, *args, **kwargs):
303             self = super(tklass, cls).__new__(cls, *args, **kwargs)
304             self.taints = set()
305
306             # if any of the arguments is tainted, taint the object aswell
307
308             for a in args:
309                 collect_tags(a, self.taints)
310             for v in kwargs.values():
311                 collect_tags(v, self.taints)
312
313             return self
314
315         # support for assigment and taint change in classobj
316
317         def __setattr__(self, name, value):
318             if self.__dict__ and name in self.__dict__ and tainted(self.__dict__[
                    name]):
319                 for t in self.__dict__[name].taints:
320                     # if other field had it, keep it
321                     taintsets = [v.taints for k,v in self.__dict__.items() if not
                            callable(v) and tainted(v) and k != name]
322                     if not any([t in x for x in taintsets]):
323                         self.taints.remove(t)
324
325             if self.__dict__ is not None:
326                 self.__dict__[name] = value
327                 if tainted(value):
328                     self.taints.update(value.taints)
329
330     d = klass.__dict__
331     for name, attr in [(m, d[m]) for m in methods]:
332         if inspect.ismethod(attr) or inspect.ismethoddescriptor(attr):
333             setattr(tklass, name, propagate_method(attr))
334     # str has no __radd__ method
335     if '__add__' in methods and '__radd__' not in methods:
336         setattr(tklass, '__radd__', lambda self, other:
337                                     tklass.__add__(tklass(other), self))
338     # unicode __rmod__ returns NotImplemented
339     if klass == unicode:
340         setattr(tklass, '__rmod__', lambda self, other:
341                                     tklass.__mod__(tklass(other), self))
```

```
342     return tklass
343
344
345 dont_override = set(['__repr__', '__cmp__', '__getattribute__', '__new__',
346                      '__init__', '__reduce_ex__',
347                      '__str__', '__int__', '__float__', '__unicode__'])
348
349
350 # ======= Taint aware classes for strings, integers, floats, and unicode ======
351
352 def attributes(klass):
353     a = set(klass.__dict__.keys())
354     return a   dont_override
355
356 str_methods = attributes(str)
357 unicode_methods = attributes(unicode)
358 int_methods = attributes(int)
359 float_methods = attributes(float)
360
361 STR = taint_class(str, str_methods)
362 UNICODE = taint_class(unicode, unicode_methods)
363 INT = taint_class(int, int_methods)
364 FLOAT = taint_class(float, float_methods)
365
366 tclasses = {str: STR, int: INT, float: FLOAT, unicode: UNICODE}
367
368 def tclass(o):
369     '''Tainted instance factory.'''
370     klass = type(o)
371     if klass in tclasses.keys():
372         return tclasses[klass](o)
373     else:
374         raise KeyError
375
376 if __name__ == "__main__":
377         import doctest
378         doctest.testmod()
```

B. Casos de prueba

```
from taintmode import *
import unittest

ends_execution()

def reached(f):
    return False

@untrusted
def some_input(value="some input from the outside"):
    '''Some random input from the 'outside'.'''
    return value

@cleaner(SQLI)
def cleanSQLI(s):
    '''Dummy SQL injection cleaner.'''
    return s.replace(" ", "")

@cleaner(XSS)
def cleanXSS(s):
    '''Dummy XSS cleaner.'''
    return s.replace("<", "&lt;")

@cleaner(II)
def cleanII(s):
    '''Dummy II cleaner.'''
    return s.replace("os", "")

@cleaner(OSI)
def cleanOSI(s):
    '''Dummy OSI cleaner.'''
    return s.replace(";", "")

@ssink(reached=reached)
def saveDB1(valor):
    '''Dummy save in database function. Sensitive to all vulnerabilities.'''
    return True

@ssink(v=SQLI, reached=reached)
def saveDB2(valor):
    '''Dummy save in database function. Only sensitive to SQL injection.'''
    return True

@ssink(v=XSS, reached=reached)
def saveDB3(valor):
    '''Dummy save in database function. Only sensitive to SQL injection.'''
    return True

class TestTaintFlow(unittest.TestCase):

    def test_tainted(self):
        '''a tainted value reaches a sensitive sink.'''

        i = some_input('a r v r a s s')
        self.assertFalse(saveDB1(i))

    def test_tainted_not_clean_anough(self):
        '''a partial tainted value reaches a full sensitive sink.'''

        i = some_input('a p t v r a f s s')
        self.assertFalse(saveDB1(cleanSQLI(i)))

    def test_not_tainted(self):
        '''an SQLI cleaned value reaches a SQLI sensitive sink.
        It's all right.'''

        i = some_input('a s c v r a s s s i a r')
        self.assertTrue(saveDB2(cleanSQLI(i)))


class TestSTR(unittest.TestCase):
```

```
72
73      def test_right_concatenation_not_cleaned(self):
74          '''a tainted value is right concatenated with a non tainted value.
75          The result is tainted. If not cleaned, the taint reaches the sink.'''
76
77          i = some_input('right concatenation')
78          self.assertFalse(saveDB2(i + "hohoho"))
79
80      def test_left_concatenation_not_cleaned(self):
81          '''a tainted value is left concatenated with a non tainted value.
82          The result is tainted. If not cleaned, the taint reaches the sink.'''
83
84          i = some_input('left concatenation')
85          self.assertFalse(saveDB2("hohoho" + i))
86
87      def test_right_concatenation(self):
88          '''a tainted value is right concatenated with a non tainted value.
89          The result is tainted.'''
90
91          i = some_input('clean right concatenation')
92          self.assertTrue(saveDB2(cleanSQLI(i + "hohoho")))
93
94      def test_left_concatenation(self):
95          '''a tainted value is left concatenated with a non tainted value.
96          The result is tainted.'''
97
98          i = some_input('clean left concatenation')
99          self.assertTrue(saveDB2(cleanSQLI("hohoho" + i)))
100
101     def test_indexing_not_cleaned(self):
102         '''if you get an item from a tainted value, the item is also tainted.'''
103
104         i = some_input('indexing')
105         self.assertFalse(saveDB2(i[4]))
106
107     def test_indexing(self):
108         '''if you get an item from a tainted value, the item is also tainted.'''
109
110         i = some_input('clean indexing')
111         self.assertTrue(saveDB2(cleanSQLI(i[4])))
112
113     def test_mul_not_cleaned(self):
114         '''if s is tainted, s * n is also tainted.'''
115
116         i = some_input('multi')
117         self.assertFalse(saveDB2(i * 8))
118
119     def test_mul(self):
120         '''if s is tainted, s * n is also tainted.'''
121
122         i = some_input('clean multi')
123         self.assertTrue(saveDB2(cleanSQLI(i * 8)))
124
125     def test_left_mul_not_cleaned(self):
126         '''if s is tainted, n * s is also tainted.'''
127
128         i = some_input('left multi')
129         self.assertFalse(saveDB2(8 * i))
130
131     def test_left_mul(self):
132         '''if s is tainted, n * s is also tainted.'''
133
134         i = some_input('clean left multi')
135         self.assertTrue(saveDB2(cleanSQLI(8 * i)))
136
137     def test_slice_not_cleaned(self):
138         '''if  you slice a tainted value, the slice also tainted.'''
139
140         i = some_input('take a slice')
141         self.assertFalse(saveDB2(i[2:5]))
142
143     def test_slice(self):
144         '''if  you slice a tainted value, the slice also tainted.'''
145
```

```
146         i = some_input('clean teke a slice')
147         self.assertTrue(saveDB2(cleanSQLI(i[2:5] )))
148 
149     def test_mod_not_cleaned(self):
150         '''if s is tainted, s %a is also tainted.'''
151 
152         i = some_input("fomat %s this 1")
153         self.assertFalse(saveDB2(i %'a'))
154 
155     def test_mod(self):
156         '''if s is tainted, s %a is also tainted.'''
157 
158         i = some_input("fomat %s this 2")
159         self.assertTrue(saveDB2(cleanSQLI(i %'b')))
160 
161     def test_rmod_not_cleaned(self):
162         '''if s is tainted, a %s is also tainted.'''
163 
164         i = some_input("ar1")
165         self.assertFalse(saveDB2("%s" %i))
166 
167     def test_rmod(self):
168         '''if s is tainted, a %s is also tainted.'''
169 
170         i = some_input("ar2")
171         self.assertTrue(saveDB2(cleanSQLI("%s" %i)))
172 
173     # tests for public str methdos
174 
175     def test_join_not_cleaned(self):
176         '''if s is tainted, s.join(aLista) is also tainted.'''
177 
178         i = some_input('join')
179         self.assertFalse(saveDB2(i.join(['_', '_', '_'])))
180 
181     def test_join(self):
182         '''if s is tainted, s.join(aLista) is also tainted.'''
183 
184         i = some_input('clean join')
185         self.assertTrue(saveDB2(cleanSQLI(i.join(['_', '_', '_']))))
186 
187     #def test_join_taint_argument(self):
188     #    '''if s is tainted, .join(list cotaining s) is also tainted.'''
189     #
190     #    i = some_input('clean join')
191     #    self.assertFalse(saveDB2(",".join([i, '_', '_'])))
192 
193     def test_capitalize_not_cleaned(self):
194         '''if s is tainted. s.capitalize() is also tainted.'''
195 
196         i = some_input('capitalize')
197         self.assertFalse(saveDB2(i.capitalize()))
198 
199     def test_capitalize(self):
200         '''if s is tainted. s.capitalize() is also tainted.'''
201 
202         i = some_input('clean capitalize')
203         self.assertTrue(saveDB2(cleanSQLI(i.capitalize())))
204 
205     def test_center_not_cleaned(self):
206         '''if s is tainted. s.center(n) is also tainted.'''
207 
208         i = some_input('center')
209         self.assertFalse(saveDB2(i.center(6)))
210 
211     def test_center(self):
212         '''if s is tainted. s.center(n) is also tainted.'''
213 
214         i = some_input('clean center')
215         self.assertTrue(saveDB2(cleanSQLI(i.center(6))))
216 
217     def test_expandtabs_not_cleaned(self):
218         '''if s is tainted. s.expandtabs(n) is also tainted.'''
219 
```

```
220         i = some_input('\t')
221         self.assertFalse(saveDB2(i.expandtabs(4)))
222
223     def test_expandtabs(self):
224         '''if s is tainted. s.expandtabs(n) is also tainted.'''
225
226         i = some_input('\tclean\t')
227         self.assertTrue(saveDB2(cleanSQLI(i.expandtabs(4))))
228
229     def test_ljust_not_cleaned(self):
230         '''if s is tainted. s.ljust(n) is also tainted.'''
231
232         i = some_input('left just')
233         self.assertFalse(saveDB2(i.ljust(42)))
234
235     def test_ljust(self):
236         '''if s is tainted. s.ljust(n) is also tainted.'''
237
238         i = some_input('clean left just')
239         self.assertTrue(saveDB2(cleanSQLI(i.ljust(42))))
240
241     def test_lower_not_cleaned(self):
242         '''if s is tainted. s.lower() is also tainted.'''
243
244         i = some_input("NOT LOWER")
245         self.assertFalse(saveDB2(i.lower()))
246
247     def test_lower(self):
248         '''if s is tainted. s.lower() is also tainted.'''
249
250         i = some_input("CLEAN NOT LOWER")
251         self.assertTrue(saveDB2(cleanSQLI(i.lower())))
252
253     def test_lstrip_not_cleaned(self):
254         '''if s is tainted. s.lstrip([chars]) is also tainted.'''
255
256         i = some_input("       left spaces")
257         self.assertFalse(saveDB2(i.lstrip()))
258
259     def test_lstrip(self):
260         '''if s is tainted. s.lstrip([chars]) is also tainted.'''
261
262         i = some_input("       left spaces and clean")
263         self.assertTrue(saveDB2(cleanSQLI(i.lstrip())))
264
265     def test_partition_not_cleaned(self):
266         '''s.partition(sep) > head, sep, tail. If s is tainted,
267         head, sep and tail are also tainted.'''
268
269         i = some_input("sepa/rated")
270         h, s, t = i.partition('/')
271         self.assertFalse(saveDB2(h))
272         self.assertFalse(saveDB2(s))
273         self.assertFalse(saveDB2(t))
274
275     def test_partition(self):
276         '''s.partition(sep) > head, sep, tail. If s is tainted,
277         head, sep and tail are also tainted.'''
278
279         i = some_input("clean sepa/rated")
280         h, s, t = i.partition('/')
281         self.assertTrue(saveDB2(cleanSQLI(h)))
282         self.assertTrue(saveDB2(cleanSQLI(s)))
283         self.assertTrue(saveDB2(cleanSQLI(t)))
284
285     def test_replace_not_cleaned(self):
286         '''if s is tainted. s.replace(old, new[, count]) is also tainted.'''
287
288         i = some_input("a_a_a_a_a")
289         self.assertFalse(saveDB2(i.replace('_', ' ')))
290
291     def test_replace(self):
292         '''if s is tainted. s.replace(old, new[, count]) is also tainted.'''
293
```

```
294         i = some_input("clean_a_a_a_a_a")
295         self.assertTrue(saveDB2(cleanSQLI(i.replace('_', ' '))))
296 
297     def test_replace_with_count_not_cleaned(self):
298         '''if s is tainted. s.replace(old, new[, count]) is also tainted.'''
299 
300         i = some_input("a_a_a_a_a_count")
301         self.assertFalse(saveDB2(i.replace('_', ' ', 2)))
302 
303     def test_replace_with_count(self):
304         '''if s is tainted. s.replace(old, new[, count]) is also tainted.'''
305 
306         i = some_input("clean_a_a_a_a_a_count")
307         self.assertTrue(saveDB2(cleanSQLI(i.replace('_', ' ', 2))))
308 
309     #def test_replace_taint_argument(self):
310     #    '''if s is tainted. x.replace(s, new[, count]) is also tainted.'''
311     #
312     #    i = some_input("aaa")
313     #    self.assertFalse(saveDB2("aaaaaaa".replace(i, '_', 2)))
314 
315     def test_rjust_not_cleaned(self):
316         '''if s is tainted. s.rjust(n) is also tainted.'''
317 
318         i = some_input('right just')
319         self.assertFalse(saveDB2(i.rjust(42)))
320 
321     def test_rjust(self):
322         '''if s is tainted. s.rjust(n) is also tainted.'''
323 
324         i = some_input('clean right just')
325         self.assertTrue(saveDB2(cleanSQLI(i.rjust(42))))
326 
327     def test_rpartition_not_cleaned(self):
328         '''s.rpartition(sep) > head, sep, tail. If s is tainted,
329         head, sep and tail are also tainted.'''
330 
331         i = some_input("rsepa/rated")
332         h, s, t = i.rpartition('/')
333         self.assertFalse(saveDB2(h))
334         self.assertFalse(saveDB2(s))
335         self.assertFalse(saveDB2(t))
336 
337     def test_rpartition(self):
338         '''s.rpartition(sep) > head, sep, tail. If s is tainted,
339         head, sep and tail are also tainted.'''
340 
341         i = some_input("clean rsepa/rated")
342         h, s, t = i.rpartition('/')
343         self.assertTrue(saveDB2(cleanSQLI(h)))
344         self.assertTrue(saveDB2(cleanSQLI(s)))
345         self.assertTrue(saveDB2(cleanSQLI(t)))
346 
347     def test_rsplit_not_cleaned(self):
348         '''s.rsplit(sep) > list of strings. If s is tainted,
349         strings in the list are also tainted.'''
350 
351         i = some_input("right/sepa/rated")
352         aList = i.rsplit('/')
353         for l in aList:
354             self.assertFalse(saveDB2(l))
355 
356     def test_rsplit(self):
357         '''s.rsplit(sep) > list of strings. If s is tainted,
358         strings in the list are also tainted.'''
359 
360         i = some_input("clean/right/sepa/rated")
361         aList = i.rsplit('/')
362         self.assertTrue(len(aList) == 4)
363         for l in aList:
364             self.assertTrue(saveDB2(cleanSQLI(l)))
365 
366     def test_rsplit_max(self):
367         '''s.rsplit(sep [, maxsplit]) > list of strings. If s is tainted,
```

```
368             strings in the list are also tainted.'''
369
370             i = some_input("max/clean/right/sepa/rated")
371             aList = i.rsplit('/', 1)
372             self.assertTrue(len(aList) == 2)
373             for l in aList:
374                 self.assertTrue(saveDB2(cleanSQLI(l)))
375
376         def test_rstrip_not_cleaned(self):
377             '''If s is tainted, s.rstrip([chars]) is also tainted.'''
378
379             i = some_input("right strip it     ")
380             self.assertFalse(saveDB2(i.rstrip()))
381
382         def test_rstrip(self):
383             '''If s is tainted, s.rstrip([chars]) is also tainted.'''
384
385             i = some_input("clean right strip it     ")
386             self.assertTrue(saveDB2(cleanSQLI(i.rstrip())))
387
388         def test_split_not_cleaned(self):
389             '''s.split(sep) > list of strings. If s is tainted,
390             strings in the list are also tainted.'''
391
392             i = some_input("split/sepa/rated")
393             aList = i.split('/')
394             for l in aList:
395                 self.assertFalse(saveDB2(l))
396
397         def test_split(self):
398             '''s.split(sep) > list of strings. If s is tainted,
399             strings in the list are also tainted.'''
400
401             i = some_input("clean/split/sepa/rated")
402             aList = i.split('/')
403             self.assertTrue(len(aList) == 4)
404             for l in aList:
405                 self.assertTrue(saveDB2(cleanSQLI(l)))
406
407         def test_split_max(self):
408             '''s.split(sep [, maxsplit]) > list of strings. If s is tainted,
409             strings in the list are also tainted.'''
410
411             i = some_input("max/clean/split/sepa/rated")
412             aList = i.split('/', 1)
413             self.assertTrue(len(aList) == 2)
414             for l in aList:
415                 self.assertTrue(saveDB2(cleanSQLI(l)))
416
417         def test_splitlines_not_cleaned(self):
418             '''s.splitlines([keepends]) > list of strings. If s is tainted,
419             strings in the list are also tainted.'''
420
421             i = some_input("line\nline\nline")
422             aList = i.splitlines()
423             for l in aList:
424                 self.assertFalse(saveDB2(l))
425
426         def test_splitlines(self):
427             '''s.splitlines([keepends]) > list of strings. If s is tainted,
428             strings in the list are also tainted.'''
429
430             i = some_input("clean\nline\nline\nline")
431             aList = i.splitlines()
432             for l in aList:
433                 self.assertTrue(saveDB2(cleanSQLI(l)))
434
435         def test_strip_not_cleaned(self):
436             '''if s is tainted. s.strip([chars]) is also tainted.'''
437
438             i = some_input("       leftright spaces       ")
439             self.assertFalse(saveDB2(i.strip()))
440
441         def test_strip(self):
```

```
442         '''if s is tainted. s.strip([chars]) is also tainted.'''
443
444         i = some_input("       leftright spaces and clean       ")
445         self.assertTrue(saveDB2(cleanSQLI(i.strip())))
446
447     def test_swapcase_not_cleaned(self):
448         '''if s is tainted. s.swapcase() is also tainted.'''
449
450         i = some_input('SwApCaSe')
451         self.assertFalse(saveDB2(i.swapcase()))
452
453     def test_swapcase(self):
454         '''if s is tainted. s.swapcase() is also tainted.'''
455
456         i = some_input('cLeAn SwApCaSe')
457         self.assertTrue(saveDB2(cleanSQLI(i.swapcase())))
458
459     def test_title_not_cleaned(self):
460         '''if s is tainted. s.title() is also tainted.'''
461
462         i = some_input('title this')
463         self.assertFalse(saveDB2(i.title()))
464
465     def test_title(self):
466         '''if s is tainted. s.title() is also tainted.'''
467
468         i = some_input('clean title this')
469         self.assertTrue(saveDB2(cleanSQLI(i.title())))
470
471     def test_translate_not_cleaned(self):
472         '''if s is tainted. s.translate(table [, deletechars])
473         is also tainted.'''
474
475         i = some_input('translate it')
476         self.assertFalse(saveDB2(i.translate('o'*256)))
477
478     def test_translate(self):
479         '''if s is tainted. s.translate(table [, deletechars])
480            is also tainted.'''
481
482         i = some_input('clean title this')
483         self.assertTrue(saveDB2(cleanSQLI(i.translate('o'*256))))
484
485     def test_upper_not_cleaned(self):
486         '''if s is tainted. s.upper() is also tainted.'''
487
488         i = some_input("not upper")
489         self.assertFalse(saveDB2(i.upper()))
490
491     def test_upper(self):
492         '''if s is tainted. s.upper() is also tainted.'''
493
494         i = some_input("clean not upper")
495         self.assertTrue(saveDB2(cleanSQLI(i.upper())))
496
497     def test_zfill_not_cleaned(self):
498         '''if s is tainted. s.zfill(width) is also tainted.'''
499
500         i = some_input("9")
501         self.assertFalse(saveDB2(i.zfill(3)))
502
503     def test_zfill(self):
504         '''if s is tainted. s.zfill(width) is also tainted.'''
505
506         i = some_input("8")
507         self.assertTrue(saveDB2(cleanSQLI(i.zfill(3))))
508
509     # Previous tests are for methods returning str or containers
510
511     def test_len(self):
512         '''if s is tainted. len(s) is also tainted.'''
513
514         i = some_input("cinco")
515         self.assertFalse(saveDB2(len(i)))
```

```
516
517 class TestINT(unittest.TestCase):
518
519     def test_abs(self):
520         i = some_input(1)
521         self.assertTrue(tainted(abs(i)))
522
523     def test_add(self):
524         i = some_input(1)
525         self.assertTrue(tainted(i + 2))
526
527     def test_and(self):
528         i = some_input(1)
529         self.assertTrue(tainted(i & 2))
530
531     def test_div(self):
532         i = some_input(1)
533         self.assertTrue(tainted(i / 2))
534
535     def test_divmod(self):
536         i = some_input(1)
537         a,b = divmod(i, 2)
538         self.assertTrue(tainted(a))
539         self.assertTrue(tainted(b))
540
541     def test_floordiv(self):
542         i = some_input(1)
543         d = i // 2
544         self.assertTrue(tainted(d))
545
546     def test_radd(self):
547         i = some_input(1)
548         self.assertTrue(tainted(2 + i))
549
550 class TestFLOAT(unittest.TestCase):
551
552     def test_abs(self):
553         f = some_input(1.0)
554         self.assertTrue(tainted(abs(f)))
555
556     def test_add(self):
557         f = some_input(1.0)
558         self.assertTrue(tainted(f + 2))
559
560     def test_div(self):
561         f = some_input(1.0)
562         self.assertTrue(tainted(f / 2))
563
564     def test_divmod(self):
565         f = some_input(1.0)
566         a,b = divmod(f, 2)
567         self.assertTrue(tainted(a))
568         self.assertTrue(tainted(b))
569
570     def test_floordiv(self):
571         f = some_input(1.0)
572         d = f // 2
573         self.assertTrue(tainted(d))
574
575     def test_radd(self):
576         f = some_input(1.0)
577         self.assertTrue(tainted(2 + f))
578
579 class TestUNICODE(unittest.TestCase):
580
581     def test_add(self):
582         u = some_input(u'Asimov')
583         self.assertTrue(tainted(u + ' books'))
584
585     def test_contains(self):
586         u = some_input(u'Asimov')
587         self.assertTrue(tainted(u + ' books'))
588
589     #def test_rmod_not_cleaned(self):
```

```
590         #     '''if s is tainted, a %s is also tainted.'''
591         #
592         #     i = some_input(u"ar1")
593         #     self.assertFalse(saveDB2("%s" %i))
594
595         def test_rmod_not_cleaned_u(self):
596             '''if s is tainted, a %s is also tainted.'''
597
598             i = some_input(u"ar1")
599             self.assertFalse(saveDB2(u"%s" %i))
600
601         def test_rmod(self):
602             '''if s is tainted, a %s is also tainted.'''
603
604             i = some_input(u"ar2")
605             self.assertTrue(saveDB2(cleanSQLI(u"%s" %i)))
606
607     class TestDict(unittest.TestCase):
608
609         def test_dict(self):
610             @untrusted
611             def retorna_dict():
612                 return dict(a=1)
613
614             d = retorna_dict()
615             self.assertTrue(tainted(d['a']))
616
617         def test_dictkind(self):
618             class myDict(dict):
619                 pass
620
621             @untrusted
622             def retorna_dict():
623                 return myDict(a=1)
624
625             d = retorna_dict()
626             self.assertTrue(tainted(d['a']))
627
628     class TestCHR(unittest.TestCase):
629         '''Test the chr built it function. If the int like argument is tainted,
630          the returned string must be tainted too.'''
631
632         def test_no_tainted_ord(self):
633             c = chr(42)
634             self.assertFalse(tainted(c))
635
636         def test_tainted_ord(self):
637             c = chr(INT(42))
638             self.assertEqual(str, type(c))
639
640         def test_same_taints(self):
641             o = INT(42)
642             o.taints.add(XSS)
643             c = chr(o)
644             self.assertTrue(XSS in c.taints)
645             self.assertEqual(1, len(c.taints))
646
647     class TestORD(unittest.TestCase):
648         '''Test the ord built it function. If the str like argument is tainted,
649          the returned integer must be tainted too.'''
650
651         def test_no_tainted_char(self):
652             c = ord('*')
653             self.assertFalse(tainted(c))
654
655         def test_tainted_char(self):
656             c = ord(STR('*'))
657             self.assertEqual(int, type(c))
658
659         def test_same_taints(self):
660             c = STR('*')
661             c.taints.add(XSS)
662             o = ord(c)
663             self.assertTrue(XSS in o.taints)
```

```
664         self.assertEqual(1, len(o.taints))
665
666 class TestTaints(unittest.TestCase):
667
668     def test_all_set(self):
669         n = some_input('test all set')
670         self.assertTrue(SQLI in n.taints)
671         self.assertTrue(XSS in n.taints)
672
673     def test_in_one_set(self):
674         n = some_input('test in one set')
675         n = cleanSQLI(n)
676         self.assertFalse(SQLI in n.taints)
677         self.assertTrue(XSS in n.taints)
678
679     def test_in_no_set(self):
680         n = some_input('test in no set')
681         n = cleanSQLI(n)
682         n = cleanXSS(n)
683         self.assertFalse(SQLI in n.taints)
684         self.assertFalse(XSS in n.taints)
685
686
687 class TestTainted(unittest.TestCase):
688
689     def test_tainted(self):
690         x = 'taint'
691         self.assertFalse(tainted(x))
692         i = some_input(x)
693         self.assertTrue(tainted(i))
694
695     def test_tainted_vul(self):
696         x = 'taint_vul'
697         self.assertFalse(tainted(x))
698         i = some_input(x)
699         self.assertTrue(tainted(i, v=XSS))
700         self.assertTrue(tainted(i, v=SQLI))
701         i = cleanSQLI(i)
702         self.assertTrue(tainted(i, v=XSS))
703         self.assertFalse(tainted(i, v=SQLI))
704
705     def test_tainted_vul2(self):
706         '''If the givven vul argument is not a valid KEY,
707         return False.'''
708         x = 'taint_vul2'
709         self.assertFalse(tainted(x))
710         i = some_input(x)
711         self.assertFalse(tainted(i, v=100))
712
713     def test_taint_0(self):
714         ''' There was a bug in rev 98 related to 0 as a taint id
715         and its value of truth.
716         '''
717         a = taint('just one taint', 0)
718         self.assertEqual(a.taints, set([0]))
719
720 class TestSink(unittest.TestCase):
721
722     def test_false_all(self):
723         n = some_input('test false all')
724         self.assertFalse(saveDB1(n))
725         self.assertFalse(saveDB2(n))
726         self.assertFalse(saveDB3(n))
727
728     def test_one(self):
729         n = some_input('test one')
730         n = cleanSQLI(n)
731         self.assertFalse(saveDB1(n))
732         self.assertTrue(saveDB2(n))
733         self.assertFalse(saveDB3(n))
734
735     def test_true_all(self):
736         n = some_input('test true all')
737         n = cleanSQLI(n)
```

```
738             n = cleanXSS(n)
739             n = cleanOSI(n)
740             n = cleanII(n)
741             self.assertTrue(saveDB1(n))
742             self.assertTrue(saveDB2(n))
743             self.assertTrue(saveDB3(n))
744
745     class TaintFunction(unittest.TestCase):
746
747         def test_taint_values(self):
748             a = "will be xss tainted"
749             b = "will be sqli tainted"
750             taint(a, XSS)
751             taint(b, SQLI)
752             self.assertTrue(tainted(a, v=XSS))
753             self.assertTrue(tainted(a, v=XSS))
754
755         def test_taint_values(self):
756             a = "will be xss tainted"
757             b = "will be sqli tainted"
758             a = taint(a, XSS)
759             b = taint(b, SQLI)
760             self.assertTrue(tainted(a, v=XSS))
761             self.assertTrue(tainted(a, v=XSS))
762
763     class TaintOperations(unittest.TestCase):
764
765         def test_add_2taints(self):
766             a = "will be xss tainted"
767             b = "will be sqli tainted"
768             a = taint(a, XSS)
769             b = taint(b, SQLI)
770             r = a + b
771             self.assertTrue(tainted(r, v=XSS))
772             self.assertTrue(tainted(r, v=SQLI))
773
774         def test_radd_2taints(self):
775             a = "will be xss tainted"
776             b = "will be sqli tainted"
777             a = taint(a, XSS)
778             b = taint(b, SQLI)
779             r = b + a
780             self.assertTrue(tainted(r, v=XSS))
781             self.assertTrue(tainted(r, v=SQLI))
782
783         def test_mod_2taints(self):
784             a = "will be xss tainted"
785             b = "will be sqli tainted"
786             a = taint(a, XSS)
787             b = taint(b, SQLI)
788             r = b + a
789             self.assertTrue(tainted(r, v=XSS))
790             self.assertTrue(tainted(r, v=SQLI))
791
792     class UnstrustedDecorator(unittest.TestCase):
793
794         def test_untrusted_string(self):
795             @untrusted
796             def uf():
797                 return "untrusted"
798             u = uf()
799             self.assertTrue(tainted(u))
800
801         def test_untrusted_dict(self):
802             @untrusted
803             def uf():
804                 return {0: "untrusted1", 1: "untrusted2"}
805             u = uf()
806             self.assertTrue(tainted(u[0]))
807             self.assertTrue(isinstance(u[0], STR))
808             self.assertTrue(tainted(u[1]))
809             self.assertTrue(isinstance(u[1], STR))
810
811         def test_untrusted_list(self):
```

```
812             @untrusted
813             def uf():
814                 return ["untrustedA", "untrustedB"]
815             u = uf()
816             self.assertTrue(tainted(u[0]))
817             self.assertTrue(isinstance(u[0], STR))
818             self.assertTrue(tainted(u[1]))
819             self.assertTrue(isinstance(u[1], STR))
820
821         def test_untrusted_dict_with_list(self):
822             @untrusted
823             def uf():
824                 return {0: "untrustedC", 1: ["untrustedD"]}
825             u = uf()
826             self.assertTrue(tainted(u[0]))
827             self.assertTrue(isinstance(u[0], STR))
828             self.assertTrue(tainted(u[1][0]))
829             self.assertTrue(isinstance(u[1][0], STR))
830
831         def test_untrusted_list_with_dict(self):
832             @untrusted
833             def uf():
834                 return ["unstrustedE", {0: "untrustedF"}]
835             u = uf()
836             self.assertTrue(tainted(u[0]))
837             self.assertTrue(isinstance(u[0], STR))
838             self.assertTrue(tainted(u[1][0]))
839             self.assertTrue(isinstance(u[1][0], STR))
840
841         def test_untrusted_twisted_structure(self):
842             @untrusted
843             def uf():
844                 return ["unstrustedG", {0: "untrustedH",
845                                         1: ["untrustedI", "untrustedJ"]}]
846             u = uf()
847             self.assertTrue(tainted(u[0]))
848             self.assertTrue(isinstance(u[0], STR))
849             self.assertTrue(tainted(u[1][0]))
850             self.assertTrue(isinstance(u[1][0], STR))
851             self.assertTrue(tainted(u[1][1][0]))
852             self.assertTrue(isinstance(u[1][0][0], STR))
853             self.assertTrue(tainted(u[1][1][1]))
854             self.assertTrue(isinstance(u[1][0][1], STR))
855
856     class CleanerDecorator(unittest.TestCase):
857
858         def test_clener1(self):
859             i = some_input('1')
860             i = cleanOSI(i)
861             self.assertFalse(OSI in i.taints)
862
863     @validator(XSS, nargs=[0])
864     def is_good(a):
865         return True
866
867     class ValidatorDecorator(unittest.TestCase):
868
869         def test_validator(self):
870             i = some_input(1)
871             is_good(i)
872             self.assertFalse(XSS in i.taints)
873             self.assertTrue(SQLI in i.taints)
874
875     if __name__ == '__main__':
876         unittest.main()
```

C. Código fuente de «Sugerencias para el almuerzo»

code.py

```
1  import web
2  import view
3  from view import render
4  import os
5  from datetime import datetime
6
7  from taintmode import *
8  web.input = untrusted(web.input)
9  os.system = ssink(OSI)(os.system)
10 import taintmode
11 taintmode.ends_execution()
12
13 from cleaners import clean_osi
14 clean_osi = cleaner(OSI)(clean_osi)
15
16 urls = (
17     '/', 'index',
18     '/add', 'add',
19     '/clean', 'clean'
20 )
21
22 class index:
23     def GET(self):
24         return render.base(view.listing())
25
26 class clean:
27     def POST(self):
28         dayfile = datetime.today().strftime('%Y%m%d') + '.txt'
29         os.system("rm " + dayfile)
30         raise web.seeother('/')
31
32 class add:
33     def POST(self):
34         user = web.input().user
35         meal = clean_osi(web.input().meal)
36         # save it to the file of the day
37         dayfile = datetime.today().strftime('%Y%m%d') + '.txt'
38         os.system("echo " + meal + " >> " + dayfile)
39         raise web.seeother('/')
40
41 if __name__ == "__main__":
42     app = web.application(urls, globals())
43     app.internalerror = web.debugerror
44     app.run()
```

view.py

```
import web
from datetime import datetime

t_globals = dict(
  datestr=web.datestr,
)
render = web.template.render('templates/', globals=t_globals)
render._keywords['globals']['render'] = render

def listing():
    dayfile = datetime.today().strftime('%Y%m%d') + '.txt'
    try:
        f = open(dayfile)
        l = f.readlines()
        f.close()
    except:
        l = []  # no file yet
    return render.listing(l)
```

cleaners.py

```
def clean_osi(s):
    return s.split(';')[0]
```

Bibliografía

[1] Django web framework. `http://www.djangoproject.com`.

[2] List of Python software. `http://en.wikipedia.org/wiki/List_of_Python_software`.

[3] The Perl programming language. `http://www.perl.org/`.

[4] The Python programming language. `http://www.python.org/`.

[5] The Ruby programming language. `http://www.ruby-lang.org/en/`.

[6] Twisted framework. `http://twistedmatrix.com`.

[7] M. Andrews. Guest Editor's Introduction: The State of Web Security. *IEEE Security and Privacy*, 4(4):14–15, 2006.

[8] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 2008. IEEE Computer Society.

[9] S. Bekman and E. Cholet. *Practical mod_perl*. O'Reilly and Associates, 2003.

[10] R. Bird and P. Wadler. *An introduction to functional programming*. Prentice Hall International (UK) Ltd., 1988.

[11] T. chung Tsai, A. Russo, and J. Hughes. A library for secure multi-threaded information flow in Haskell. *Computer Security Foundations Symposium, IEEE*, 0:187–202, 2007.

[12] J. J. Conti and A. Russo. A Taint Mode for Python via a Library. Software release. `http://www.cse.chalmers.se/~russo/juanjo.htm`, Apr. 2010.

[13] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[14] W. Enck, P. Gilbert, B. G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation, OSDI'10*. USENIX Association, 2010.

[15] Federal Aviation Administration (US). Review of Web Applications Security and Intrusion Detection in Air Traffic Control Systems. `http://www.oig.dot.gov/sites/dot/files/pdfdocs/ATC_Web_Report.pdf`, June 2009. Note: thousands of vulnerabilities were discovered.

[16] A. Futoransky, E. Gutesman, and A. Waissbein. A dynamic technique for enhancing the security and privacy of web applications. In *Black Hat USA Briefings*, Aug. 2007.

[17] V. Haldar, D. Chandra, and M. Franz. Dynamic Taint Propagation for Java. In *Proceedings of the 21st Annual Computer Security Applications Conference*, pages 303–311, 2005.

[18] Y. Huang, F. Yu, C. Hang, C. Tsai, D. Lee, and S. Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th International Conference on World Wide Web*, pages 40–52. ACM, 2004.

[19] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *2006 IEEE Symposium on Security and Privacy*, pages 258–263. IEEE Computer Society, 2006.

[20] D. Kozlov and A. Petukhov. Implementation of Tainted Mode approach to finding security vulnerabilities for Python technology. In *Proc. of Young Researchers' Colloquium on Software Engineering (SYRCoSE)*, June 2007.

[21] L.Bello and A. Russo. Towards a taint mode for cloud computing web applications. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, Beijing, China, June 2012*. ACM, 2012.

[22] P. Li and S. Zdancewic. Encoding information flow in Haskell. *Computer Security Foundations Workshop, IEEE*, 0:16, 2006.

[23] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium*, Berkeley, CA, USA, 2005. USENIX Association.

[24] M. Lutz and D. Ascher. *Learning Python*. O'Reilly & Associates, Inc., 1999.

[25] M. Monga, R. Paleari, and E. Passerini. A hybrid analysis framework for detecting web application vulnerabilities. In *IWSESS '09: Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems*, pages 25–32, Washington, DC, USA, 2009. IEEE Computer Society.

[26] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically Hardening Web Applications Using Precise Tainting. In *In 20th IFIP International Information Security Conference*, pages 372–382, 2005.

[27] T. Pietraszek, C. V. Berghe, C. V, and E. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection (RAID)*, 2005.

[28] A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell. In *Proceedings of the first ACM SIGPLAN Symposium on Haskell*, pages 13–24. ACM, 2008.

[29] A. Russo, J. M. D. Stefan, and D. Mazieres. Flexible dynamic information flow control in haskell. In *ACM SIGPLAN Haskell Symposium 2011, Tokyo, Japan, September 2011*. ACM, 2011.

[30] A. Russo and M. Jaskelioff. Secure multi-execution in haskell. In *Proceedings of Andrei Ershov International Conference on Perspectives of System Informatics, Akademgorodok, Novosibirsk, Russia, June 27-July 1, 2011*. LNCS, Springer-Verlag, 2011.

[31] A. Russo, A. Sabelfeld, and K. Li. Implicit flows in malicious and nonmalicious code. *2009 Marktoberdorf Summer School (IOS Press)*, 2009.

[32] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[33] A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, LNCS. Springer-Verlag, June 2009.

[34] J. Seo and M. S. Lam. InvisiType: Object-Oriented Security Policies. In *17th Annual Network and Distributed System Security Symposium*. Internet Society (ISOC), Feb. 2010.

[35] D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby. The Pragmatic Programmer's Guide*. Pragmatic Programmers, 2004.

[36] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: effective taint analysis of web applications. In M. Hind and A. Diwan, editors, *Proc. ACM SIGPLAN Conference on Programming language Design and Implementation*, pages 87–97. ACM Press, 2009.

[37] J. Williams and D. Wichers. OWASP Top 10 2010. `http://www.owasp.org/index.php/Top_10_2010`, 2010.