Universidad Internacional de La Rioja

Escuela De Ingeniería

Máster universitario en Seguridad Informática

RAMA EXPERIMENTAL

Desarrollo de servicios web REST "inseguros" para auto-aprendizaje en la explotación de vulnerabilidades

Trabajo Fin de Máster

presentado por: CARLOS MICHAEL MEJÍA GRANDA

Director: JOSE JAVIER MARTINEZ HERRAIZ

Ciudad: Quito

Fecha: 26/07/2018

Resumen

El presente Trabajo de Fin de Máster consiste en el desarrollo de servicios web REST inseguros, empleando el lenguaje de programación PHP para auto-aprendizaje en la explotación de vulnerabilidades y, su posterior exposición a distintas herramientas de análisis de vulnerabilidades (Hp Fortify, OWASP ZAP, ACUNETIX) con la finalidad de conocer, evidenciar y mitigar las amenazas encontradas.

Se incluirá un análisis del estado del arte con lo referente a seguridad en servicios Web REST, se someterán los servicios web “inseguros” ante un scanner de vulnerabilidades tomando en cuenta las 5 amenazas más comunes de la lista OWASP 2017 y el REST Security Cheat Sheet, se analizarán y expondrán los resultados para luego presentar las debidas conclusiones; finalmente, se ofrecerán breves consejos, recomendaciones y buenas prácticas que serán útiles para el programador durante la fase de desarrollo de servicios web REST.

Palabras Clave: Vulnerabilidades Web, RESTful, Seguridad aplicaciones online, OWASP 2017, Ciberseguridad.

Abstract

This document refers to the development of insecure REST web services, using PHP as programming language for self-learning in vulnerabilities exploitation and then, Web services will be exposed to different vulnerability analysis tools (Hp Fortify, OWASP ZAP, ACUNETIX) looking for evidencing, knowing and mitigating the founded threats.

Will be included an analysis about the state of art related to security in REST web services. The “insecure” Web services developed will be submitted to a vulnerability scanner taking into account the five most common threats reported in the OWASP 2017 list and the REST Security Cheat Sheet, results will be analyzed, exposed and conclusions will be presented. Finally, readers will find brief tips, suggestions and good practices that will be useful during the development phase of REST web services for programmers.

Keywords: Web vulnerabilities, RESTful, security in online applications, OWASP 2017, cybersecurity.

Índice de contenidos

1. Introducción. 1
2. Estado del arte 3
   2.1. Contexto general de REST 3
   2.2. Características Principales de REST 3
   2.3. Servicios REST vulnerables para estudio 5
      2.3.1. OWASP WebGoat 5
      2.3.2. DVWA 6
      2.3.3. SOA secRT 7
   2.4. Aseguramiento REST 8
      2.4.1. Inyección de SQL 9
      2.4.2. Cross-site scripting (XSS) 10
      2.4.3. Configuración débil 11
      2.4.4. CSRF 11
      2.4.5. Pérdida de autenticación 11
   2.5. Herramientas de análisis de código 14
      2.5.1. OWASP/ZAP 15
      2.5.2. HP FORTIFY (Audit Workbench) 15
      2.5.3. ACUNETIX 16
3. Objetivos y Metodología 18
   3.1. Objetivo general 18
   3.2. Objetivos específicos 18
   3.3. Metodología de trabajo 18
4. Desarrollo específico de la contribución 20
   4.1. Configuración y ejecución de pruebas con OWASP-ZAP (previo asegurar los servicios REST desarrollados) 20
   4.2. Configuración y ejecución de pruebas con HP Fortify (previo asegurar los servicios REST desarrollados) 24

4.3. Configuración y ejecución de pruebas con ACUNETIX (previo asegurar los servicios Rest desarrollados)....27
5. Resultado de las pruebas de análisis de código....31
5.1. Código vulnerable....31
5.1.1. Informe emitido por hp Fortify....31
5.1.2. Informe emitido por Owasp-Zap....40
5.1.3. Informe emitido por ACUNETIX....48
5.2. Breve análisis del código y configuraciones corregidas....51
5.2.1. Mejoras sobre el informe emitido por hp Fortify....51
5.2.2. Mejoras informe emitido por Owasp-Zap....60
5.2.3. Mejoras sobre el informe emitido por ACUNETIX....69
6. Análisis de resultados y breve guía de seguridad para el desarrollo de servicios REST71
6.1. Análisis de resultados....71
6.2. Breve guía de seguridad para el desarrollo de servicios REST....75
6.2.1. Buenas prácticas a nivel de codificación....75
6.2.2. Buenas prácticas a nivel de configuración....79
7. Conclusiones y trabajo futuro....82
7.1. Conclusiones....82
7.2. Líneas de trabajo Futuro....83
8. Referencias....85

Índice de figuras

Figura 1: Aplicación vulnerable Webgoat 6
Figura 2: Aplicación vulnerable DVWA 7
Figura 3: SOA secRT Consola de administración. 8
Figura 4: Autenticación con cookies (github.io, 2015) 12
Figura 5: Owasp ZAP pantalla de trabajo 15
Figura 6: Audit Workbench Pantalla de trabajo 16
Figura 7: Acunetix escáner pantalla general (www.emtdist.com, 2016) 17
Figura 8: Opciones OWASP ZAP 20
Figura 9: Active scan input vectors OWASP ZAP 21
Figura 10: Extensiones OWASP ZAP 1 21
Figura 11: Extensiones escaneo pasivo OWASP ZAP 22
Figura 12: Extensiones OWASP ZAP 2 22
Figura 13: Configuración PROXY de OWASP ZAP 23
Figura 14: Configuración de URL y modo de ataque OWASP ZAP 23
Figura 15: Inicio del scan OWASP ZAP 24
Figura 16: Selección del “Advanced scan” Audit Workbench 24
Figura 17: Selección del directorio a analizar Audit Workbench 25
Figura 18: Aceptación del árbol de directorio a analizar Audit Workbench 25
Figura 19:Configuración de módulos en Audit Workbench 26
Figura 20: Establecimiento del tipo de aplicación Audit Workbench 26
Figura 21: Resultados de análisis Audit Workbench 27
Figura 22: Selección de Nuevo scan ACUNETIX 28
Figura 23: Selección de URL para auditar ACUNETIX 29
Figura 24: Selección de opciones de escaneo ACUNETIX 29
Figura 25: Resumen de configuraciones ACUNETIX 29
Figura 26: Secuencia de autenticación ACUNETIX 30
Figura 27: Finalización de configuración e inicio de escaneo ACUNETIX 30
Figura 28:Diagrama de flujo vulnerabilidad SQLi Audit Workbench 31
Figura 29: Test Ataque manual SQLi Audit Workbench 1 32
Figura 30: Ataque manual SQLi Audit Workbench 2 32
Figura 31: Ataque manual SQLi Audit Workbench 3 32
Figura 32: Ataque manual SQLi Audit Workbench 4 33
Figura 33: Vulnerabilidad SQLi método insert 33
Figura 34: Diagrama de flujo de la inyección SQL para el método insert 34
Figura 35: Vulnerabilidad SQLi método update 34

Figura 36: Diagrama de flujo para la realización de SQLi del método update.....34
Figura 37 Vulnerabilidad SQLi método Delete.....35
Figura 38: Diagrama de flujo para la realización de SQLi del método Delete 35
Figura 39: Diagrama de flujo para XSS almacenado.....36
Figura 40: Ataque manual XSS Almacenado.....36
Figura 41: Código fuente vulnerable XSS Almacenado para el método getPeoples().....37
Figura 42: Diagrama de flujo para XSS almacenado en el método getPeoples().....37
Figura 43: Código vulnerable para XSS DOM 38
Figura 44: Diagrama de flujo de XSS DOM para view.php 38
Figura 45: Fuente de datos no validada para JSON Injection método savePeople.....39
Figura 46: Diagrama de flujo de JSON Injection para el método savepeople 39
Figura 47: Fuente de datos no validada para JSON Injection método updatePeople().....40
Figura 48: Diagrama de flujo de JSON Injection para updatePeople 40
Figura 49: Script view.php vulnerable a sobre-escritura de action.....41
Figura 50: Código js que permitiría la modificación del atributo action en view.php.....41
Figura 51: Script create.php vulnerable a sobre-escritura de action 41
Figura 52: Código js que permitiría la modificación del atributo action en create.php.....42
Figura 53: Script update.php vulnerable a sobre-escritura de action 42
Figura 54: Código js que permitiría la modificación del atributo action en update.php.....42
Figura 55: Script delete.php vulnerable a sobre-escritura de action 42
Figura 56: Código js que permitiría la modificación del atributo action en delete.php 43
Figura 57: Evidencia de Cross-Domain JavaScript Source File Inclusion 44
Figura 58: Ausencia cabecera X-Frame-Options Header 45
Figura 59: Ausencia encabezado CSP.....45
Figura 60: Divulgación de información de versión del servidor 46
Figura 61: Ausencia cabecera Web Browser XSS Protection.....46
Figura 62: Ausencia cabecera X-Content-Type-Options.....47
Figura 63: Cabeceras para contenido no almacenable/almacenable.....48
Figura 64: ausencia de las cabeceras para X-Frame-Options header 49
Figura 65: Método Options habilitado.....50
Figura 66:Método trace habilitado 50
Figura 67: Corrección SQLi método getPeople 52
Figura 68: Corrección SQLi método insert.....52
Figura 69: Corrección SQLi método delete.....53
Figura 70: Corrección SQLi método update 53
Figura 71: Corrección SQLi método checkID.....54
Figura 72: Corrección método getPeoples contra XSS almacenado 56

Figura 73: Corrección contra XSS DOM en JavaScript en el script view.php.....57
Figura 74: Corrección de XSS DOM en HTML para view.php 57
Figura 75: Corrección método savePeople contra JSON Injection 59
Figura 76: Corrección método updatePeople contra JSON Injection 60
Figura 77: Corrección de sobre-escritura de parámetros HTTP script view.php 61
Figura 78: Corrección de sobre-escritura de parámetros HTTP script create.php (HTML)....61
Figura 79: Corrección de sobre-escritura de parámetros HTTP script create.php (JavaScript) 62
Figura 80: Corrección de sobre-escritura de parámetros HTTP script update.php (HTML)...62
Figura 81: Corrección de sobre-escritura de parámetros HTTP script update.php (JavaScript) 63
Figura 82: Corrección de sobre-escritura de parámetros HTTP script delete.php (HTML)....63
Figura 83: Corrección de sobre-escritura de parámetros HTTP script delete.php (JavaScript) 64
Figura 84: Corrección JavaScript Source File Inclusion.....66
Figura 85: Cabeceras de seguridad para solventar problemas de falta de configuración 1. .69
Figura 86: Figura 85: Cabeceras de seguridad para solventar problemas de falta de configuración 2. 70
Figura 87: Reporte Audit Workbench luego de mitigar las vulnerabilidades reportadas.....74
Figura 88: Reporte Owasp ZAP luego de mitigar las vulnerabilidades reportadas.....74
Figura 89: Reporte ACUNETIX luego de mitigar las vulnerabilidades reportadas.....74

Índice de Tablas

Tabla 1: Vulnerabilidades a través de Audit Workbench.....71
Tabla 2: Vulnerabilidades reportadas a través de OWASP-ZAP.....72
Tabla 3: Vulnerabilidades reportadas a través de ACUNETIX.....73
Tabla 4 Indicadores de riesgo 73

1. Introducción.

En la actualidad el apoyarse en una arquitectura orientada a servicios (SOA) en el desarrollo de aplicaciones es indispensable, esto en virtud del compartimiento de información entre sistemas heterogéneos que desean consumir y realizar operaciones mediante servicios web. En el año 2000, Roy Fielding define a REST como una serie de principios para un modelo de arquitectura a seguir en el desarrollo de aplicaciones apoyada en HTTP que no se encuentra obligado y atado a la utilización de protocolos que usen patrones de intercambio de mensajes como lo hace el protocolo de acceso simple a objetos (SOAP).

Con la base de REST nace RESTful, que es un estilo para la construcción de servicios web apoyado en HTTP y exponiendo URIS a manera de directorios, mismos que no almacenan estado, finalmente como resultado se tiene una transferencia de XML, JSON más simple que SOAP. Sin embargo, el uso de RESTful está asociado a vulnerabilidades con respecto a seguridad ya que al estar basado en HTTP hereda todos sus posibles fallos.

Sumado a las vulnerabilidades propias del protocolo HTTP tenemos que REST no es más que un estilo de arquitectura, a diferencia de SOAP que es un estándar reconocido por la W3C, esto significa que en la práctica no hay patrones de diseño y aseguramiento establecidos y, las falencias o problemas presentados por el factor humano en su manera de codificar las soluciones informáticas dependen enteramente de los creadores del servicio web (WS) en cuestión. La falta de conocimiento por parte de los desarrolladores en materia de seguridad y prácticamente la ausencia de guías de desarrollo seguro para REST, hacen posible la implementación de servicios débiles y vulnerables.

Tomando en cuenta que los servicios web REST hoy por hoy son los más difundidos y usados, incluso por grandes empresas como Google, Facebook, Twitter, etc., como solución al problema de seguridad en REST se propone el desarrollo de servicios web basados en REST “vulnerables” para autoaprendizaje, bajo el lenguaje PHP, y la aportación de una breve guía con consejos útiles para mitigar vulnerabilidades en fase de implementación, previo a la publicación y puesta en producción.

En el capítulo uno se presenta de manera general el contenido del aporte a realizar a través de este Trabajo de Fin de Master (TFM).

En el capítulo dos se muestra un “estado del arte” en lo referente a REST en materia de seguridad, los conceptos básicos necesarios para la comprensión del presente trabajo, los tipos de vulnerabilidades para evitar el desarrollo de servicios web REST inseguros y

herramientas que se han desarrollado y publicado para conocer, concienciar, realizar algunas pruebas, e incluso mitigar las amenazas que se contemplarán más adelante.

En el capítulo tres se expone el objetivo general y los objetivos específicos; además se detalla, en el apartado de metodología, las acciones realizadas desde el desarrollo de los servicios web REST vulnerables, hasta su aseguramiento contra inyección de SQL, cross-site scripting (XSS), configuración débil, revelación de datos sensibles, pérdida de autenticación.

En el capítulo cuatro se encuentra el desarrollo de los test de intrusión y el sometimiento de los servicios REST vulnerables al pentesting ya sea manual, o automatizado con OWASP ZAP, ACUNETIX, y hp Fortify, con la finalidad de obtener la mayor cantidad de información sobre malos hábitos de programación, las vulnerabilidades y problemas que ocasionan al exponerlos en la WEB.

En el capítulo cinco se expone los resultados de los test de intrusión de los servicios web REST “vulnerables” que fueron desarrollados y sometidos a las herramientas de pentesting y el test de intrusión manual.

En el capítulo seis se indica lo referente al análisis de resultados de los test de intrusión, por ende, una comparativa entre los servicios web REST desarrollados inicialmente y los servicios REST resultantes que ya han sido asegurados contra las vulnerabilidades tratadas en este TFM, además de una breve guía con lo referente a consideraciones necesarias en el desarrollo de servicios web REST de cara a Producción.

En el capítulo siete se presentan las conclusiones y líneas de trabajo a futuro, que pueden ser aplicadas a partir de esta investigación.

En el capítulo ocho se encuentra las referencias empleadas durante el desarrollo de este TFM.

2. Estado del arte

2.1. Contexto general de REST

El estado representacional de transferencia (REST) es un tipo de arquitectura para desarrollo web de sistemas distribuidos que emplea el protocolo HTTP y que fue mencionado por primera vez en el 2000 por Roy Fielding en una disertación sobre su trabajo de tesis (Fielding, 2000), REST es mucho más simple que SOAP al punto que muchas empresas importantes alrededor del mundo como Google y Facebook han optado por implementarla (expansión.mx, 2011), fomentando así un fácil consumo de sus interfaces de programación de aplicaciones (APIs) liberadas en cualquier dispositivo que soporte el protocolo HTTP, esto permite la intercomunicación entre sistemas heterogéneos. Se debe tener en cuenta que REST no se encuentra definido como un estándar y por lo tanto no es reconocido como tal por la W3C, por otra parte SOAP si lo está (Macías, 2016).

REST es una de las arquitecturas más empleadas al implementar servicios web ligeros y de fácil consumo entre clientes y el servidor. El modo de comunicación de REST es muy sencillo, básicamente se trata de varios clientes que se encuentran enganchados a la red, de los cuales no importa la naturaleza que sean, simplemente con el hecho de soportar el protocolo HTTP están listos para realizar peticiones. A continuación, se envían las peticiones a un proxy el cuál buscará en sus recursos REST la manera de atenderlos, en primera instancia verifica si posee la respuesta de la petición en caché, si no la posee procede a pedir al servidor el recurso solicitado, finalmente la respuesta será devuelta como XML o JSON, El manejo de los mensajes de error se los hace mediante los propios mensajes que están soportados en el protocolo HTTP.

2.2. Características Principales de REST

Los servicios Web REST al momento de ser consumidos por un cliente tienen una cuota de bajo acoplamiento por el uso del protocolo HTTP, esto facilita una gran escalabilidad, simplicidad y robustez en sus implementaciones; por otra parte, al realizar el envío de datos, su tipo es definido en las cabeceras y básicamente puede ser cualquier tipo como XML, JSON, Binarios (imágenes, documentos), textos, etc. (Blancarte, 2017). El manejo y envío de datos en formato JSON es mucho más ligero que el XML de SOAP y es de común interpretación para los lenguajes JAVA, JavaScript (JS), AngularJS, entre otros (Caules, 2014). A manera general los servicios REST presentan las siguientes características:

- ✓ **Tecnología cliente-servidor. -** Se debe contemplar un servidor que almacene los recursos para ser accedidos y otra parte que desea hacer uso de esos recursos para consumirlos (Cliente), en esta función ni el cliente ni el servidor deben recordar estados dado que en cada petición HTTP realizada deben constar todos los datos necesarios para ser atendida.

- ✓ **Permiten Información en caché. –** Una de las principales características que posee REST es la velocidad en sus respuestas, esto lo hace posible mejorando la eficiencia del tráfico de red mediante el almacenamiento de la información recibida del servidor tomando en cuenta una fecha del último cambio y la posibilidad de conocer si ha cambiado o no de estado dicha información, al recibir una confirmación o negación en cada consulta por parte del servidor se decide si la información guardada en caché debe ser actualizada o no.

- ✓ **Uniformidad en la interfaz. –** Los servicios web hacen uso explícito de los propios métodos de HTTP que a su vez son incluidos en la cabecera para cada petición (academiaandroid, 2015) y estos son:
  - ➢ **Put. –** Se utiliza para modificar o actualizar el estado de un recurso, similar al Update de CRUD.
  - ➢ **Post. –** Se utiliza para crear un estado de un recurso; similar al Create de CRUD.
  - ➢ **Delete. –** Se utiliza para eliminar un recurso en el servidor, similar al Delete en CRUD
  - ➢ **Get. –** Se emplea para leer la información de un recurso desde el servidor, es similar al método Read de CRUD
  - ➢ **Head. -** Comprueba si ha cambiado la información del contenido que ha sido enviado por el servidor.

  Los métodos definidos simplifican la manera en la cual una interfaz uniforme conocida como URI, define y ejecuta el proceso de información que es requerido por el cliente. Por otra parte, generalmente la URL identifica información relacionada a un recurso por ejemplo un usuario determinado sería algo así http://www.carlosmejia/tfm/usuarios/1. La URL puede estar configurada de manera que el servidor pueda filtrar los datos de una determinada entidad o clase por un criterio de selección mediante la añadidura de parámetros en la petición HTTP, un caso de añadidura de parámetros sería por ejemplo

http://www.carlosmejia/tfm/usuarios?estado=1 donde se estaría solicitando los usuarios del sistema con estado 1.

- ✓ **Sistema por capas. –** Considera que se deben implementar servidores intermedios para mejorar la escalabilidad del sistema y estos funcionarían como balanceadores de carga y proxy, ofertando compartición en el almacenamiento y de esta manera mejorando algunas características de seguridad como pueden ser la disponibilidad y accesibilidad.

- ✓ **Código Bajo demanda**. – Si bien no es una característica obligatoria como el caso de las anteriores, los servidores de cierto modo pueden "delegar" algunas porciones de código y funcionalidad basada en JavaScript con la finalidad de que el cliente ejecute algunas acciones que liberarían la carga del servidor.

2.3. Servicios REST vulnerables para estudio

La seguridad informática ha jugado un papel muy importante en los últimos años por la infinidad de ataques que se han perpetuado a diferentes sitios de todo el ciberespacio y han ido evolucionando a través de la red, por esta razón hay muchas organizaciones y empresas que promueven y se dedican a brindarle relevante atención a la seguridad informática para el desarrollo seguro de software en cada una de sus etapas, estas empresas han publicado proyectos especializados para entrenamiento y evidencia de las consecuencias por la falta de aseguramiento en los servicios y aplicaciones web, dado el caso tomaremos para referencia tres de las herramientas más comunes y famosas del mercado a continuación:

2.3.1. OWASP WebGoat

En su afán por concientizar a las empresas con lo referente a seguridad informática, OWASP lleva el proyecto de código abierto WebGoat. De acuerdo a su definición, "WebGoat es una aplicación web J2EE deliberadamente insegura, mantenida por OWASP y diseñada para enseñar lecciones de seguridad en aplicaciones Web. En cada lección, los usuarios deben demostrar su entendimiento de los problemas de seguridad al explotar la vulnerabilidad real en la aplicación WebGoat. Por ejemplo, en una de las lecciones el usuario debe usar SQL Injection para robar números de tarjeta de crédito ficticios. La aplicación es un ambiente realista de enseñanza, que provee a los usuarios con pistas y código para explicar mejor la lección" (owasp, 2017). En la Figura 1, se presenta la página principal de la aplicación WebGoat que se tomó como referencia.

Figura 1: Aplicación vulnerable Webgoat

2.3.2. DVWA

Por definición, “Damn Vulnerable Web App (DVWA) es una aplicación hecha en PHP y MySQL para el entrenamiento de explotación de vulnerabilidades web, perfecto para poner a prueba nuestras habilidades en el tema e igualmente para aprender nuevas técnicas. DVWA está divido en tres niveles: Low, medium y hight, cada uno respectivamente va aumentando su nivel de dificultad…” (redinfocol, 2011). Esta herramienta de apoyo en el análisis y entrenamiento de vulnerabilidades web, es mantenida por “dvwa.co.uk”. En la Figura 2 se indica la página principal de este aplicativo.

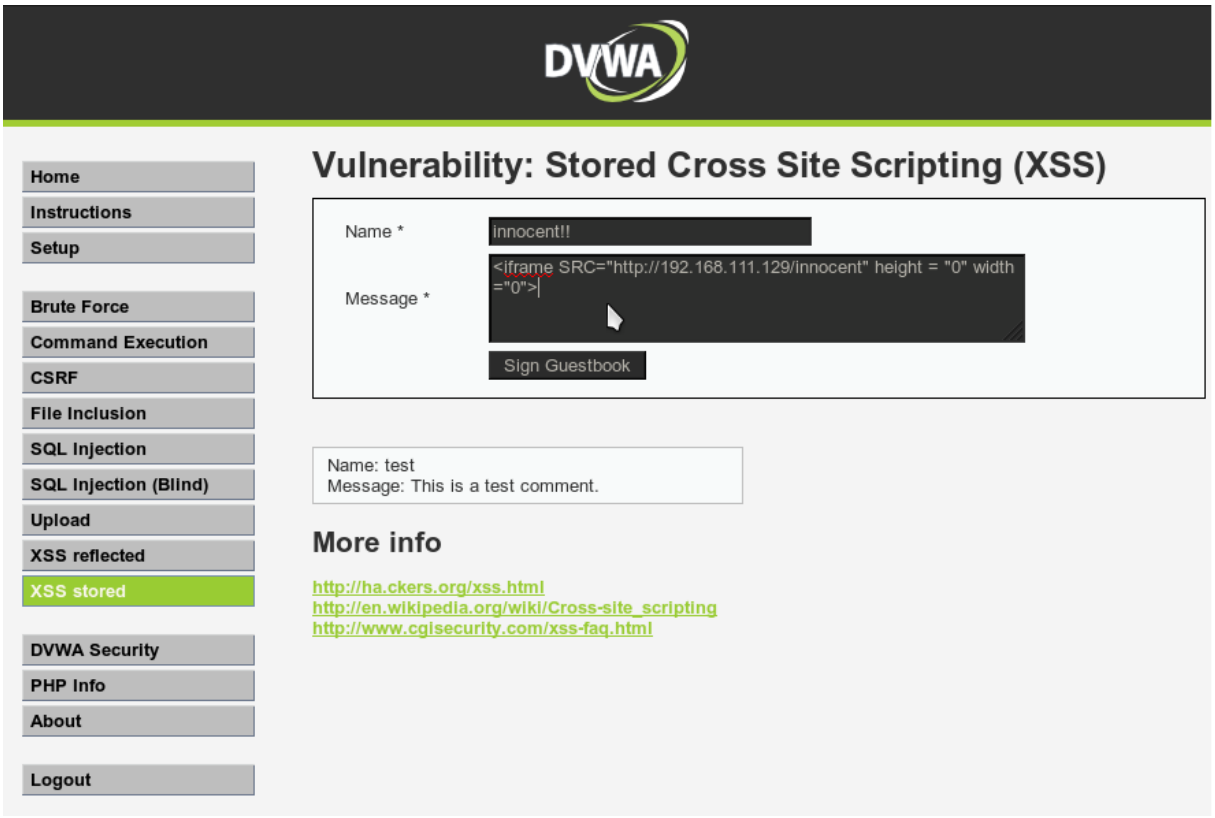


Figura 2: Aplicación vulnerable DVWA

2.3.3. SOA secRT

Según la página oficial, secRT "es una plataforma de seguridad de código abierto, desarrollada por CORISECIO en cooperación con la Oficina Federal Alemana de Seguridad de la Información (BSI). El secRT es la base de soluciones de seguridad integrales. Proporciona un marco de desarrollo que contiene un conjunto completo de funcionalidades de seguridad. La función de modelado secRT permite la creación de procesos de seguridad con poco esfuerzo de programación. Toda la funcionalidad también está disponible a través de una API WSDL, que permite una manera fácil de integrar sus propios proyectos de seguridad" (corisecio, 2014).

El marco secRT ofrece características tales como:

- ✓ Entity Management: que permite la integración en los meta-directorios existentes
- ✓ Gestión de claves: una infraestructura completa de PKI para claves y certificados
- ✓ Funciones criptográficas: donde se elige el algoritmo
- ✓ Motor de flujo de trabajo: para procesos de seguridad automatizados

Una de las principales ventajas de los proyectos enumerados anteriormente es que se tiene libre acceso al código de los scripts o clases implementadas y por consecuencia puede ser modificado conforme se vayan realizando las pruebas respectivas; sin embargo, esto no es lo

mismo que desarrollar servicios web inseguros de manera “artesanal” (ad-oc) con la finalidad de explotar vulnerabilidades y mitigarlas a nivel de código y configuración de servidores previo a su publicación. Otra desventaja que supone el uso de estas herramientas es que la mayoría emplea un firewall o un proxy como XML Gateway que, prácticamente hace transparente para el usuario o desarrollador algunas opciones que deberían ser implementadas manualmente al momento de codificar los servicios web como son: el cifrado de mensajes enviados, la configuración de roles y directivas de seguridad en servidores, aseguramiento de los canales de comunicación, etc., y de esta manera añadir una capa más según el modelo de seguridad en profundidad que ayudará a evitar ataques tanto del tipo activo como pasivo (IBM, 2016). A continuación, en la Figura 3 se presenta la consola de administración de SOA secRT.

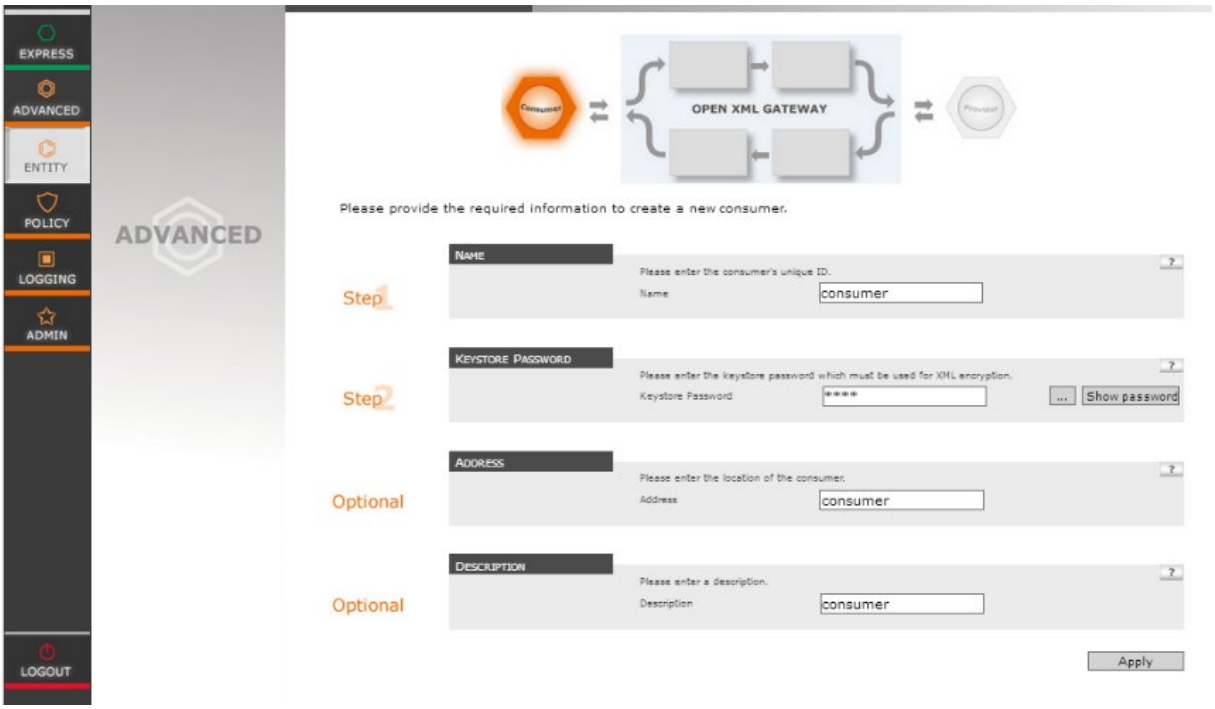


Figura 3: SOA secRT Consola de administración.

2.4. Aseguramiento REST

En la actualidad no basta con satisfacer las necesidades de una empresa u organización mediante la funcionalidad para la que se diseñan los servicios web, sino más bien se trata de ofertar una solución robusta, segura y sencilla que cumpla con los principios básicos de seguridad de la información: autenticidad, confidencialidad, integridad y disponibilidad (ACID) ante el consumo de una determinada API por parte de los clientes. En los últimos años, el crecimiento de las vulnerabilidades web, ciberdelito, y ciberdelincuencia han jugado un papel importante al punto de preocuparnos por asegurar nuestros desarrollos informáticos y

configuraciones previo a la puesta en producción; como es de esperarse una API REST se encuentra sujeta a ciertas vulnerabilidades inherentes al protocolo HTTP y riesgos de seguridad que son propios de la naturaleza de su arquitectura. Sin embargo, no todo está perdido, algunas organizaciones como OWASP, han venido elaborando y colaborando con la lista del top 10 de las vulnerabilidades encontradas y explotadas en todo el mundo en un cierto período de tiempo, para ser exactos, cada 3 años (owasp, 2017).

La mayor parte de vulnerabilidades son la consecuencia de una falta de implementación de un ciclo de vida de desarrollo de software seguro (SSDLC) que asegure las fases en cada una de sus etapas, siendo el principal factor de riesgo los problemas en la fase de diseño y consecuentemente en la etapa de implementación o codificación. La mejor forma de evitar vulnerabilidades de seguridad es una política de concientización sobre la codificación segura que garantizará soluciones relativamente rápidas y con la menor repercusión tanto económica como de planificación para un determinado proyecto. La falta de una adecuada información sobre métodos de aseguramiento para los servicios web REST que no dejan de pasar por las vulnerabilidades inherentes y propias a las aplicaciones web, propicia la investigación personal por parte de cada uno de los desarrolladores en búsqueda de mecanismos que les permita mitigar vulnerabilidades y robustecer sus servicios Web expuestos hacia el mundo, esto generalmente desencadena la comisión de muchos fallos durante la implementación del aseguramiento. Por esta razón, se han tomado en cuenta las siguientes vulnerabilidades para su estudio y ciertas sugerencias de mitigación que en la mayoría de veces son aportadas por OWASP y que serán tratadas en el presente TFM:

2.4.1. Inyección de SQL

La inyección de SQL (SQLi) es uno de los tipos de ataques de inyección de código más comunes y peligrosos, aprovechados por los atacantes con la intención de obtener información no autorizada o en sí generar problemas en los servidores de base de datos y comportamiento de aplicaciones; según OWASP el SQLi “consiste en la inserción o ’inyección’ de una consulta SQL a través de los datos de entrada del cliente a la aplicación…” (Owasp.org, 2016).

Los datos de entrada que son tomados desde la vista de la aplicación o desde las peticiones que pueden ser interceptadas y modificadas a través de un proxy como BURP, son manipulados, de tal modo que se emplean ciertos caracteres especiales por los atacantes como pueden ser: guiones (--) o comillas simples (‘ ’), y la famosa ejecución de ‘1’=’1’ (VERACODE, 2017).

Ejemplo:

"SELECT * FROM Users WHERE Username='$username' AND Password='$password'"
Donde la variable $username puede venir dada de la siguiente manera: **$username** = 1' or '1' = '1 o **la variable $password** = 1' or '1' = '1

Teniendo como resultado la siguiente consulta:

SELECT * FROM Users WHERE Username='1' OR '1' = '1' AND Password='1' OR '1' = '1'

2.4.2. Cross-site scripting (XSS)

Como su nombre lo indica son ataques mediante scripting cruzado, es decir del lado del cliente, realizando inyección de código malicioso del tipo JS, pueden ejecutarse infinidad de tareas como: la redirección a sitios no confiables, el dibujado de formularios HTML maliciosos, robo de cookies de sesión, etc. Según OWASP "son un tipo de inyección en la que se inyectan scripts maliciosos en sitios web benignos y de confianza. Los ataques XSS ocurren cuando un atacante usa una aplicación web para enviar código malicioso, generalmente en la forma de un script del lado del navegador, a un usuario final diferente. Los fallos que permiten que estos ataques tengan éxito son bastante comunes y ocurren en cualquier lugar en el que una aplicación web utiliza la información de entrada de un usuario dentro de la salida que genera sin validarla ni codificarla."

Existen 3 tipos de XSS:

- ✓ **Reflejado. –** El código que es presentado al cliente es contaminado al hacer la petición a un servidor, este puede ser interceptado y modificado sus valores que son pasados por URL, finalmente ese código es ejecutado por el servidor y a continuación el código inyectado se presenta del lado del cliente (hostalia.com, 2015).

- ✓ **Almacenado. –** Consiste primordialmente en la inserción de código HTML malicioso en las entradas de sitios web vulnerables, como pueden ser las entradas a comentarios de un blog para que de este modo se encuentren disponibles cada que el usuario recargue y solicite la información de determinada página (Pérez, 2015)

- ✓ **DOM. –** Su objetivo es la modificación de los métodos de objetos del documento **(DOM)** del lado del cliente, mediante la ejecución de código inesperado, generalmente es JS, el código del lado del cliente es ejecutado de una manera diferente a pesar de no cambiar, debido a las modificaciones maliciosas en el entorno DOM (OWASP, 2015).

2.4.3. Configuración débil

Se pueden tomar en cuenta varios aspectos en esta sección de seguridad y uno de ellos es la posibilidad que presentan los servicios basados en REST para la intercepción de los paquetes y datagramas que son transportados por el protocolo HTTP y de este modo tener acceso a información no autorizada, modificación de la información perteneciente al mensaje de HTTP o el posible reenvío de datagramas con data contaminada que genere acciones perjudiciales y maliciosas para el sistema. (maframaran, 2015). Por otra parte, hay problemas de seguridad asociados a la fuga de información delicada en las respuestas que el servidor da para cada request hecha por los clientes; esto se puede evidenciar en el encabezado, en el cual se incluyen por ejemplo la versión de PHP ejecutado por el servidor, si se utiliza o no autenticación, etc. De este modo un atacante puede beneficiarse de esta vulnerabilidad y aprovechar bugs de seguridad al conocer la versión específica de PHP.

2.4.4. CSRF

Básicamente este tipo de vulnerabilidad permite que un atacante ejecute acciones o solicitudes a sitios en los cuales ya se encuentra autenticado el usuario legítimo haciendo uso del token perteneciente al formulario o petición secuestrada; por lo general el método de materialización del ataque es mediante el uso de la ingeniería social, al enviar a la víctima un mensaje de correo electrónico invitándole a hacer clic sobre un enlace. Si la víctima es un usuario con alto grado de privilegios como un administrador; el atacante puede crear grandes estragos que pueden ir desde la creación de un usuario con súper poderes en el sistema, hasta la eliminación de los datos de una infraestructura crítica. Esto es posible ya que el servidor que va a ser atacado hace una validación únicamente de la ip del cliente quien estaba autorizado a realizar la petición sin asegurarse previamente del origen. La validación de seguridad puede realizarse en varios niveles, uno de ellos es el uso y revisión de la cabecera HTTP-referer que se encargará de verificar el lugar de donde se originó el acceso y obviamente se podrá hacer un seguimiento de la transacción solicitada (Kyrnin, 2018). Sin embargo, se puede no implementar esta etiqueta por cuestiones de privacidad hacia el usuario, debido a que algunos sistemas de seguridad como antivirus acostumbran a deshabilitarla, de ser este el caso se puede emplear la etiqueta http: origin.

2.4.5. Pérdida de autenticación

La pérdida de autenticación en las aplicaciones es un fallo muy común con el cual un atacante pude intentar adivinar las contraseñas para acceso a diferentes recursos mediante ataques de fuerza bruta, diccionarios, sniffing de paquetes, etc. El atacante puede lograr con este método la suplantación y usurpación de su usuario víctima. La pérdida de autenticación se debe a varios factores como: la falta de robustez en contraseñas y mecanismos de seguridad

débiles. Generalmente la autenticación HTTP sobre la que se basan los servicios REST, pueden ser:

- ✓ **Autenticación básica. –** Consiste en el envío de credenciales para cada petición ya que el protocolo HTTP no posee estado. Esta autenticación HTTP consiste en enviar un usuario y un password codificados en base 64 sobre la cabecera "Authorization"; no se debe confundir el hecho de estar codificado con el hecho de estar cifrado. HTTP Basic authentication basa sus respuestas en los estatus del propio protocolo HTTP, de tal manera que si se intenta acceder a un recurso que no estemos autorizados o a su vez no se posea la cabecera de Authorization; se obtendrá un código de error 401. Se puede considerar como una mejora de este mecanismo de autenticación al método Digest que usa https y un hash md5.

- ✓ **Autenticación con cookies. –** Si bien el protocolo HTTP no maneja estados, compensa esta falta de control con un mecanismo de sesiones en el cual se guardan ciertos datos de importancia para que se pueda autenticar el usuario mientras continúa su navegación y el envío de peticiones por el sitio. "Las cookies son una extensión del protocolo http que permite al cliente almacenar datos persistentes entre los ciclos de petición y respuesta" (ottocol, 2016). Las cookies son gestionadas por el navegador, el éxito o fracaso de este tipo de autenticación consiste en generar un id de cookie de sesión lo suficientemente aleatorio como para evitar ser descubiertos o adivinados por un atacante. En la Figura 4 se presenta el mecanismo de autenticación basada en cookies.

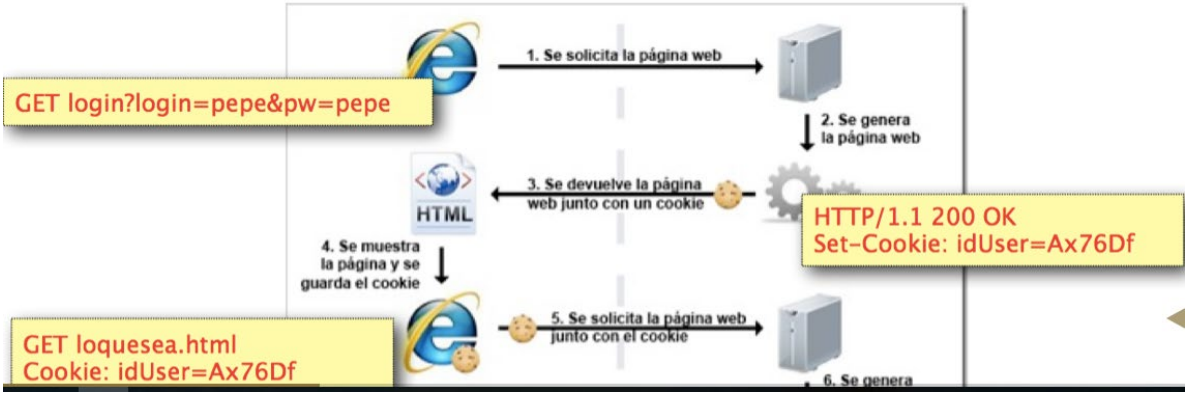


Figura 4: Autenticación con cookies *(github.io, 2015)*

- ✓ **Autenticación con tokens. –** El servidor posee un servicio de tokens en el cual al momento que un usuario realiza un login correcto, se le devuelve un identificador

único con el suficiente grado de entropía y fortaleza criptográfica. El token será utilizado a partir de ese momento para cada una de las peticiones que se realicen contra el servidor por parte del usuario del sistema, dotando a la comunicación de autenticidad y seguridad. Básicamente consta de tres componentes en el caso de JASON web tokens (JWT), concatenados con '.' y que son:

a. **Cabecera. –** Se encarga de establecer el tipo de token y el algoritmo de cifrado que se empleará; por lo general va codificado en base 64:

   {"typ"=>"JWT", "alg"=>"HS256"}

b. **Payload. –** Contiene los datos que se van a almacenar en el token, mismos que deben estar expresados en formato JSON y que irán codificados en BASE64URL.

   {"login"=>"Carlos"}

c. **Firma. –** Es el resultado de aplicar un algoritmo de hash sobre la cabecera, el Payload y una clave secreta que será enviada en BASE64URL

La seguridad en los sistemas basados en token recae en que a partir de un token generado genuinamente no se pude adivinar la clave secreta y de este modo tampoco se podrán generar tokens falsos al no conocer dicha clave.

En la necesidad vital de compartir información referente a soluciones para las vulnerabilidades en aplicaciones web y servicios REST que ayuden a los desarrolladores en su mitigación; muchas empresas comprometidas con la seguridad informática han implementado analizadores de vulnerabilidades de código automatizadas que pueden ser de tipo estático, dinámico o simplemente en modo híbrido. El personal que se encargará del aseguramiento de los servicios web no debe obviar o dejar en segundo plano el escaneo de vulnerabilidades de código de aplicaciones a nivel manual, con la finalidad de detectar y verificar el informe de falsos positivos arrojados por las herramientas utilizadas. Para su respectivo estudio y apoyo en el presente TFM se mencionarán las herramientas en el siguiente numeral.

2.5. Herramientas de análisis de código

Para asegurar la implementación de servicios web REST eficientes, simples y seguros, como parte de un SSDLC, se recomiendan las pruebas de código para cubrir toda la superficie de ataque al servicio REST; mismas que pueden realizarse mediante el uso de herramientas semi-automatizadas o también pruebas del tipo manual. Las herramientas de análisis automatizadas producirán los respectivos informes que deberán ser auditados y comprobados en la mayoría de casos con la finalidad de evitar falsos positivos, esto permitirá alcanzar la calidad, funcionalidad y seguridad del software establecida desde los requerimientos iniciales. Las herramientas de análisis de código las tenemos de tipo estático, dinámico e híbridas.

- ✓ **Herramientas de análisis de código estático (STAT). –** Son más conocidas como herramientas de análisis y detección temprana de errores para la etapa de implementación del código fuente. El escaneo con este tipo de herramientas, es considerado como uno de los más importantes en virtud de que su principal característica es el análisis de todo el código del servicio REST cubriendo la mayor parte de ataques y bugs de seguridad que podrían originar problemas y costes superiores para una organización a futuro si no son mitigados a tiempo; consecuentemente con este tipo de herramienta se encuentran más vulnerabilidades que con las herramientas dinámicas e híbridas.

- ✓ **Herramientas de análisis de código Dinámico (DAST).** – Este tipo de análisis se realiza contra una aplicación o servicio que ya se encuentra en ejecución; a diferencia de las de análisis estático de código. Si bien no cubre todo el espectro como lo hace una SAST por las limitaciones y controles de acceso que tiene en referencia a los roles de usuario de un sitio web y sus niveles de restricción y accesibilidad. El análisis DAST envía código y peticiones maliciosas que son tomadas de su base de conocimiento y pueden ser implementadas por medio del lenguaje PQL. Estas herramientas son excelentes en la detección de vulnerabilidades de XSS, SQLi, en defectos de configuración, etc.

- ✓ **Herramientas Híbridas.** – Se tratan nada más y nada menos que una combinación entre las herramientas de escáner de código estático y dinámico, en algunos casos se suele compaginar incluso con herramientas de análisis de código en tiempo real que son capaces de monitorizar, bloquear, permitir o denegar las peticiones realizadas al servidor mediante la aplicación y la recuperación del comportamiento “normal” del aplicativo que desea ser vulnerado.

Para la realización de pruebas y análisis desarrollados en los siguientes capítulos del presente TFM se emplearán herramientas de análisis de código del tipo estático y dinámico que serán mencionadas a continuación:

2.5.1. OWASP/ZAP

El Zed Attack Proxy (ZAP) de OWASP es una de las herramientas de software para análisis dinámico de aplicaciones que es mantenida y distribuida por la organización OWASP. Su principal objetivo es el análisis de seguridades en aplicaciones web orientados a empresas, se caracteriza por ser de código abierto y totalmente gratuita. Actualmente está potente herramienta se encuentra en su versión 2.7 (Velasco, 2015). En la Figura 5 se indica la interfaz principal de la herramienta OWASP-ZAP.

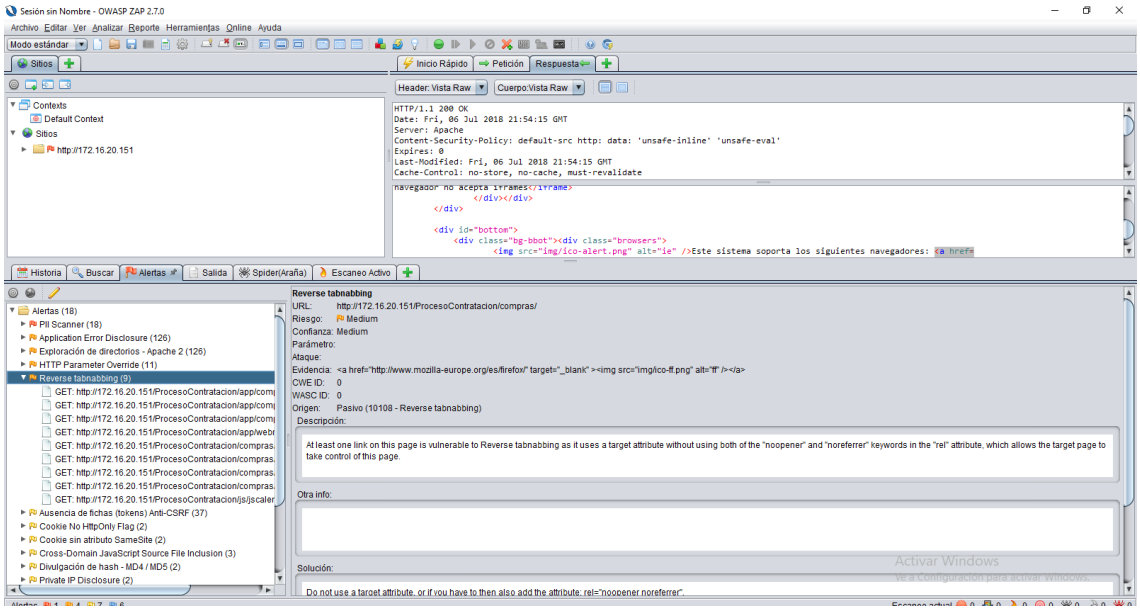


Figura 5: OWASP ZAP pantalla de trabajo

2.5.2. HP FORTIFY (Audit Workbench)

Audit Workbench como un complemento de interfaz gráfica de usuario para la herramienta de análisis de código estático de HP Fortify que un profesional de la seguridad puede manejar para escanear proyectos de software, permitiendo a la organización realizar investigación y priorización de los resultados de análisis para que su equipo pueda solucionar los problemas de seguridad de forma rápida y efectiva.

Audit Workbench cuenta con plantillas de análisis que lo ayudan a ordenar los resultados de escaneos grandes de una manera que funcione para su negocio y flujos de trabajo

(Development, 2017). En la Figura 6 se presenta la interfaz principal de trabajo de la herramienta Audit Workbench.

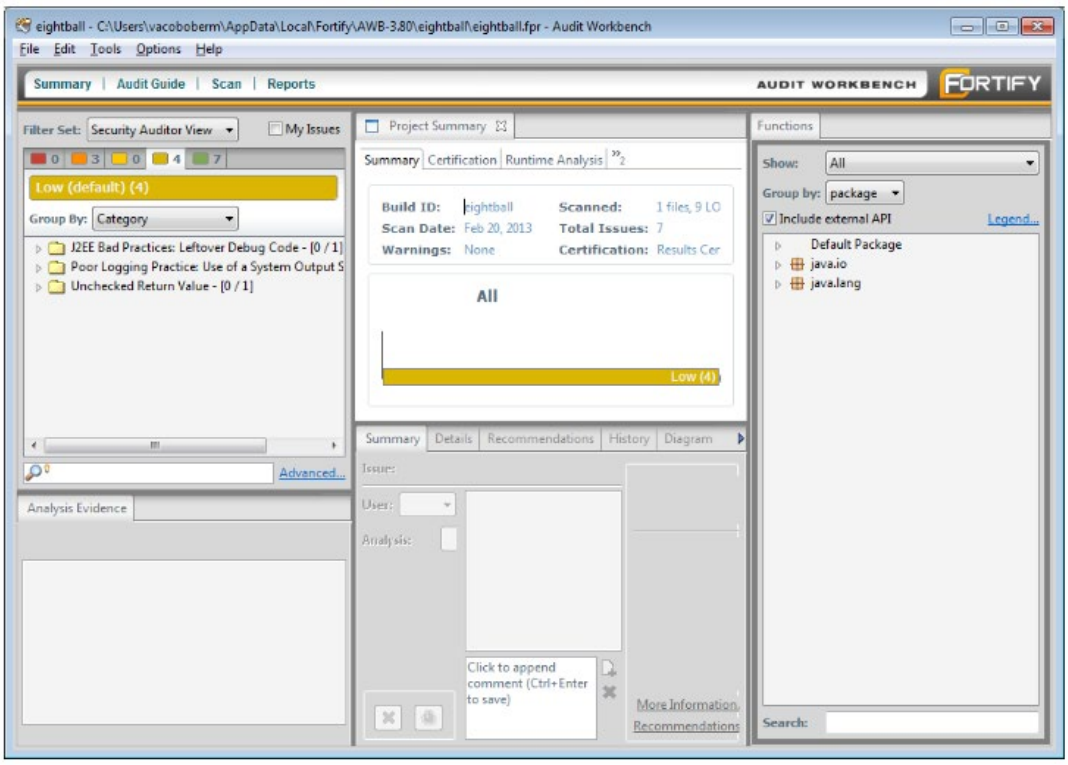


Figura 6: Audit Workbench Pantalla de trabajo

2.5.3. ACUNETIX

“Acunetix es el escáner de vulnerabilidades web líder utilizado por las empresas serias Fortune 500 y ampliamente aclamado por incluir la inyección SQL más avanzada y la tecnología de escaneo de caja negra XSS. Automáticamente rastrea sus sitios web y realiza técnicas de piratería de caja negra y caja gris que detecta vulnerabilidades peligrosas que pueden comprometer su sitio web y sus datos.

Acunetix realiza pruebas para inyección SQL, XSS, XXE, SSRF, inyección de encabezado de host y más de 4500 vulnerabilidades web. Tiene las técnicas de escaneo más avanzadas que generan la menor cantidad de falsos positivos posibles. Simplifica el proceso de seguridad de la aplicación web a través de sus características integradas de administración de vulnerabilidades que lo ayudan a priorizar y administrar la resolución de vulnerabilidades.” (acunetix, 2018). Cuenta con las siguientes características:

- ✓ Rastreo y análisis en profundidad: escanea automáticamente todos los sitios web
- ✓ Mayor tasa de detección de vulnerabilidades con bajos falsos positivos
- ✓ Gestión integrada de vulnerabilidades: priorizando y controlando amenazas
- ✓ Integración con WAF (web application firewall) populares y Trackers de Issues

- ✓ Escaneo gratuito de seguridad de red y herramientas de prueba manual
- ✓ Disponible en el local y en línea

En la Figura 7 se ilustra la interfaz de trabajo de ACUNETIX:

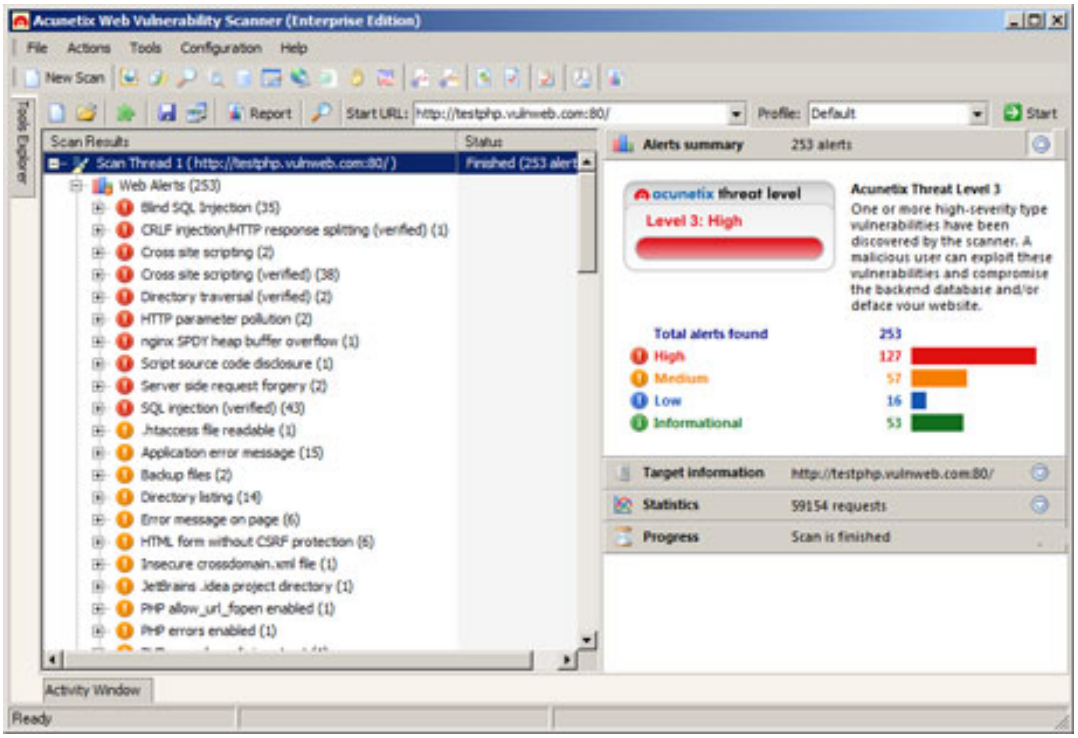


Figura 7: Interfaz de trabajo ACUNETIX *(www.emtdist.com, 2016)*

3. Objetivos y Metodología

3.1. Objetivo general

Desarrollar servicios web REST “inseguros” en lenguaje de programación PHP para auto-aprendizaje en la explotación de vulnerabilidades y su mitigación.

3.2. Objetivos específicos

- ✓ Recolectar información sobre las diferentes vulnerabilidades a las que son propensos los servicios web REST.
- ✓ Conocer y emplear las diferentes herramientas de análisis de código para detectar vulnerabilidades en los servicios web REST desarrollados.
- ✓ Analizar los servicios web REST desarrollados para detectar vulnerabilidades.
- ✓ Solventar las vulnerabilidades encontradas en los servicios web REST desarrollados.
- ✓ Elaborar una guía de consideraciones a tomar en cuenta de cara a desarrollar servicios web REST seguros.

3.3. Metodología de trabajo

Para el desarrollo de los servicios web inseguros en búsqueda del autoaprendizaje se recabó información relevante acerca de las amenazas a las que se encuentra expuesta la sociedad de la información y la comunicación, y que están permitiendo ataques por el aprovechamiento de una falta de cultura de seguridad. Se seleccionaron las vulnerabilidades y errores encontrados en el código fuente, mismos que, personalmente como programador considero son graves, de común ocurrencia y que no son tomados en cuenta en el día a día.

Se implementó una API REST con los métodos y operaciones tradicionales para: crear, leer, actualizar, eliminar (CRUD), mismos que obtuvieron información a partir de una base datos de PostgreSQL 9.4; además se codificó y desplegó un cliente que consumía los métodos expuestos por el servicio web REST.

El consumo del servicio web REST se lo hizo por medio de jQuery del lado del cliente debido a que es la manera más habitual, sencilla y común en la que es realizada, hoy por hoy, por una gran cantidad de desarrolladores que implementan sus soluciones en todo el mundo valiéndose del envío de datos en formato JSON.

Luego de implementar el cliente y el servicio web “vulnerable”, estos fueron sometidos a diferentes pruebas de análisis de código (estático y dinámico) usando las herramientas mencionadas en el capítulo anterior (OWASP-ZAP, Hp Fortify, Acunetix), se realizó una auditoría de las vulnerabilidades encontradas y en algunos casos fue posible una revisión manual.

Se mitigaron los resultados obtenidos de los analizadores de código y pruebas realizadas en nuestra implementación de servicios web vulnerables siguiendo las sugerencias establecidas por OWASP y otros sitios de seguridad; acto seguido se realizó un nuevo análisis con las herramientas anteriormente descritas para obtener información del nivel de seguridad de los servicios Web desarrollados, ya asegurados y funcionales.

Se generó una breve guía de las experiencias adquiridas y buenas prácticas a tomar en cuenta durante la etapa de desarrollo y mitigación de vulnerabilidades encontradas en los servicios codificados y desplegados, misma que servirá de cimiento para futuras prácticas en implementaciones de servicios web REST seguros, de cara a evitar las vulnerabilidades y posibles explotaciones que se podrían originar y, que son detalladas durante el presente TFM.

4. Desarrollo específico de la contribución

Una de las fases más importantes que posee el ciclo de vida de desarrollo seguro de software (SSDLC) es el apoyarse en pruebas, mismas que ayudarán a verificar problemas de funcionalidad o seguridad en nuestro caso. Para realizar las pruebas en virtud de su demasiada cantidad, el equipo de pruebas hace uso de herramientas automatizadas que le permitan optimizar tiempo y eficacia en los resultados. A continuación, se detallan las pruebas realizadas.

4.1. Configuración y ejecución de pruebas con OWASP-ZAP (previo asegurar los servicios REST desarrollados)

4.1.1. La ip del servidor donde se encuentra desplegado nuestro servicio web es la 172.16.20.151; por tanto: debemos abrir la interfaz para análisis de OWASP-ZAP y acceder a las opciones del menú herramientas. En la Figura 8 se presentan el menú opciones de OWASP ZAP

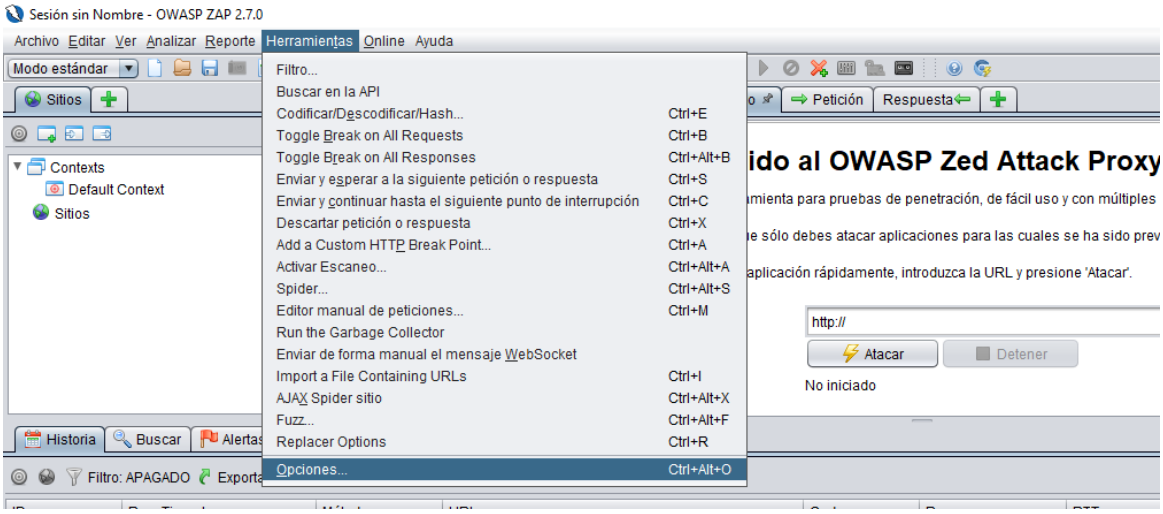


Figura 8: Opciones OWASP ZAP

4.1.2. En el apartado “active scan input vectors” seleccione todas las opciones como se ilustra en la Figura 9 sobre los “Active scan input vectors” OWASP ZAP

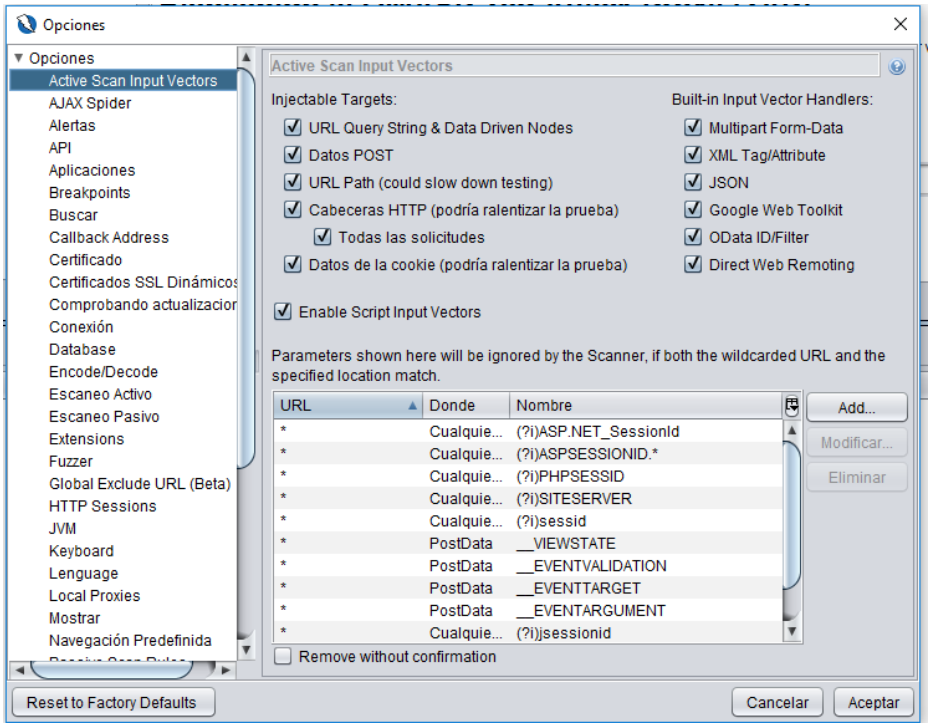


Figura 9: Active scan input vectors OWASP ZAP

4.1.3. Asegurarse que todas las extensiones se encuentren habilitadas como se presenta en la Figura 10

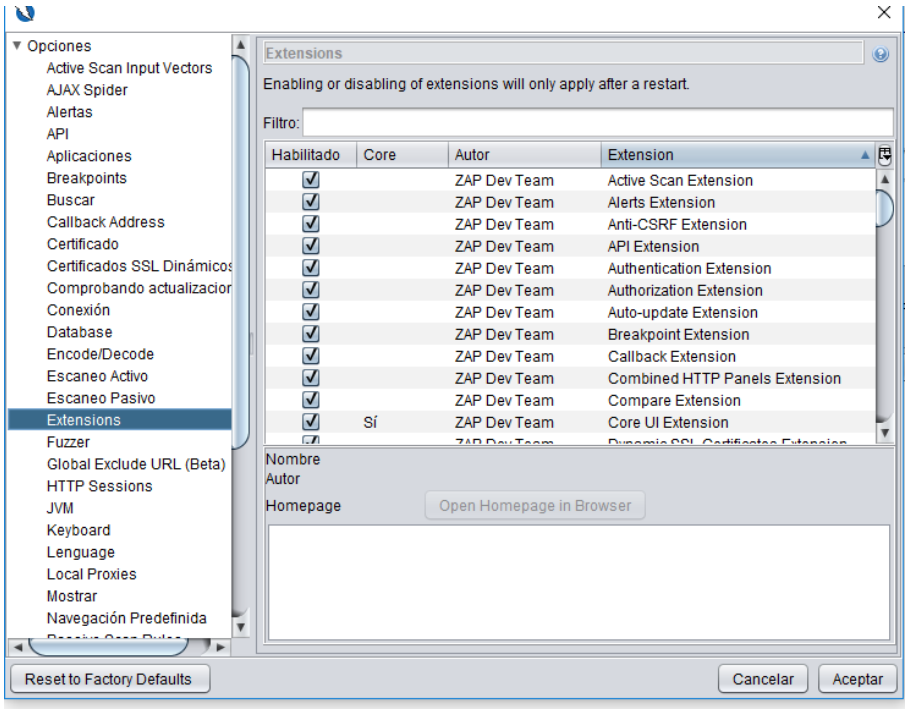


Figura 10: Extensiones OWASP ZAP 1

4.1.4. Del apartado escaneo pasivo, seleccionar todas las extensiones como se expone en la Figura 11 referente a “Extensiones escaneo pasivo OWASP ZAP”.

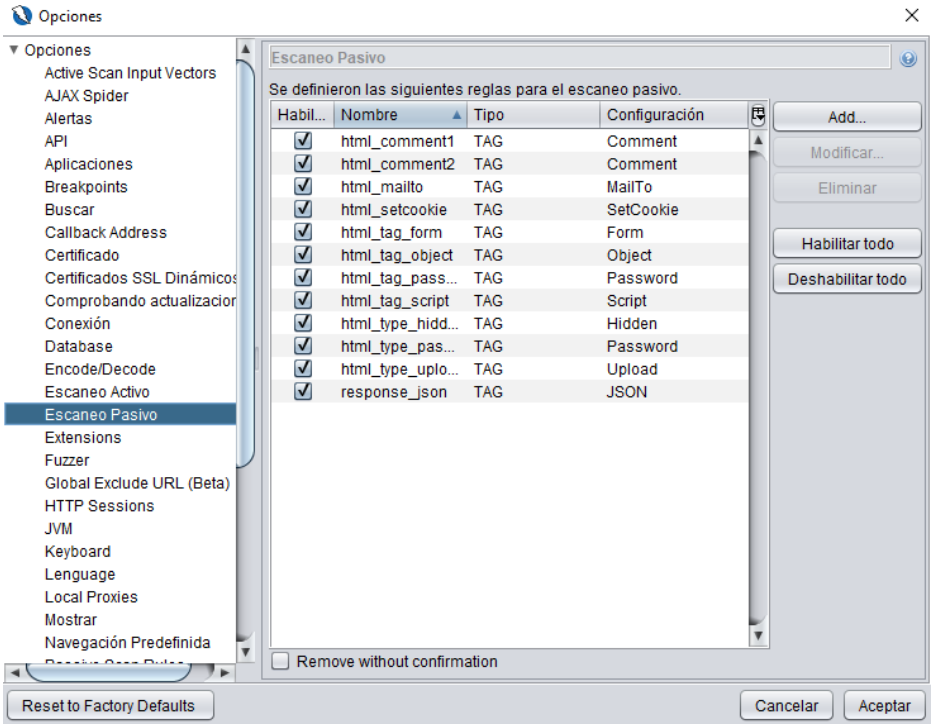


Figura 11: Extensiones escaneo pasivo OWASP ZAP

4.1.5. Verifique las extensiones que se emplearán para el escaneo como se muestra en la Figura 11 de “Extensiones OWASP ZAP revisión”.

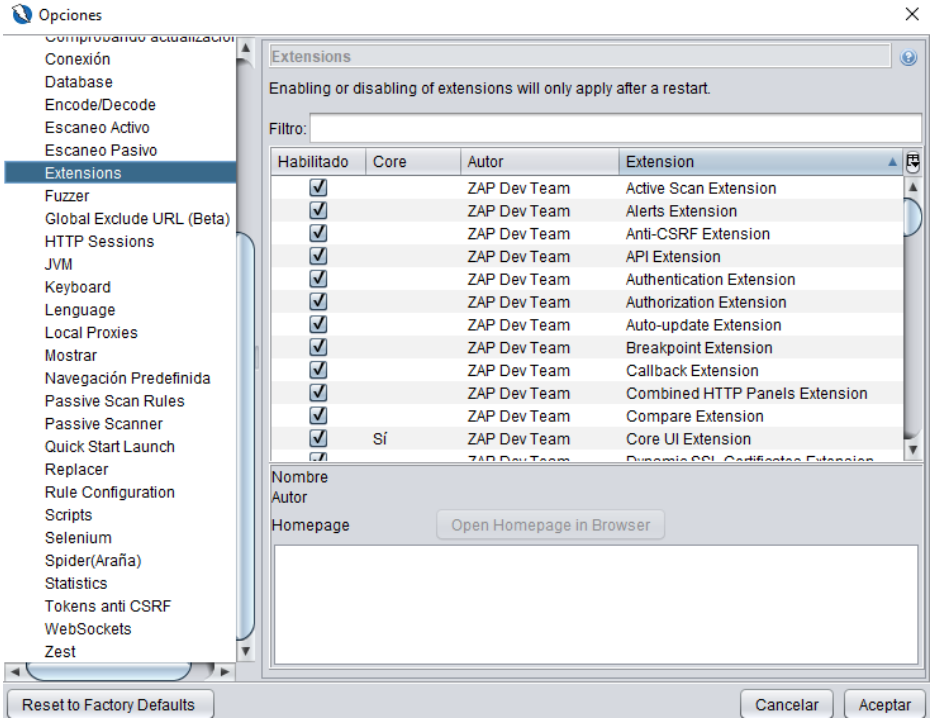


Figura 12: Extensiones OWASP ZAP revisión

4.1.6. Configurar local proxies de manera que consuma el puerto 8080. La Figura 13 no ilustrará sobre la configuración proxy expuesta.

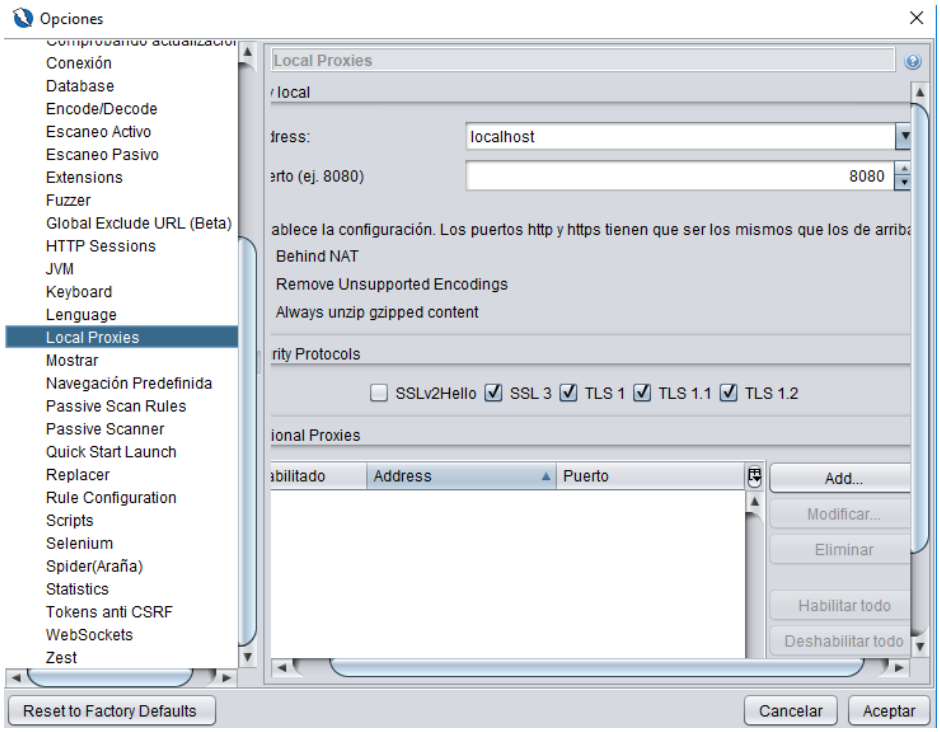


Figura 13: Configuración proxy de OWASP ZAP

4.1.7. Ingresar en el campo “URL a atacar” la dirección “http://172.16.20.151/cliente/” y seleccionar el modo de ataque como se indica en la Figura 14.

Figura 14: Configuración de URL y modo de ataque OWASP ZAP

4.1.8. Clic en el botón atacar y comenzará el análisis de las vulnerabilidades presentes en el servicio web y el cliente como se presenta en la Figura 15.

Figura 15: Inicio del scan OWASP ZAP

4.2. Configuración y ejecución de pruebas con HP Fortify (previo asegurar los servicios REST desarrollados)

4.2.1. Abrir el Audit Workbench de hp Fortify y clic en la opción escaneo avanzado como se muestra en la Figura 16.

Figura 16: Selección del "Advanced scan" Audit Workbench

4.2.2. Seleccionar el directorio que contiene el código fuente a analizar, ilustrado en la Figura 17.

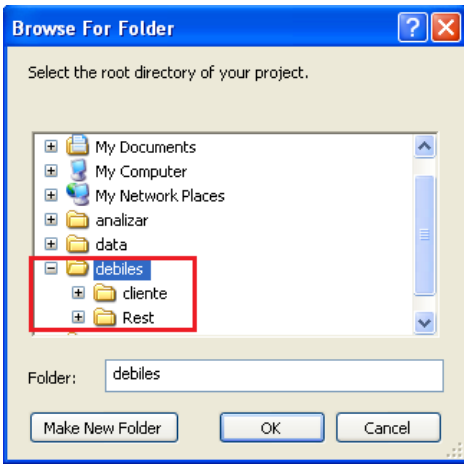


Figura 17: Selección del directorio a analizar Audit Workbench

4.2.3. Se desplegará el árbol de directorios a analizar el código fuente y clic en next. La configuración realizada es presentada en la Figura 18.

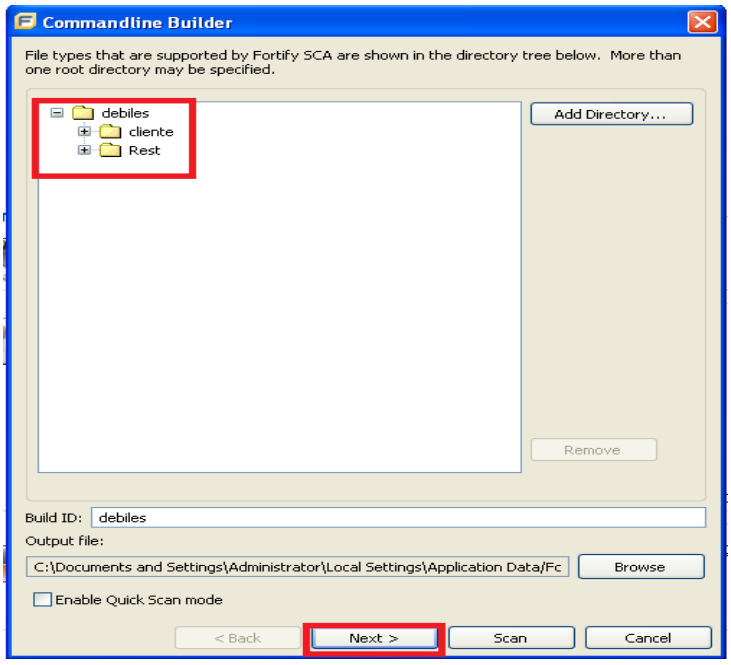


Figura 18: Selección del árbol de directorios a analizar Audit Workbench

4.2.4. Habilitar la limpieza, traducción y módulo de escaneo, finalmente se debe dar clic en next como muestra en la Figura 19.

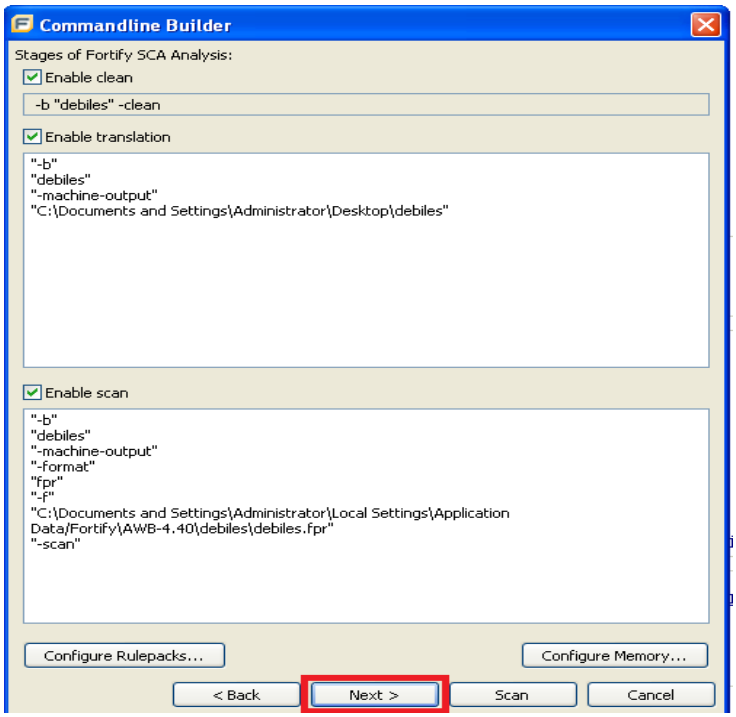


Figura 19:Configuración de módulos en Audit Workbench

4.2.5. Seleccionar que no es una aplicación java, en virtud que es código PHP. Clic en scan y dejar la configuración presentada en la Figura 20.

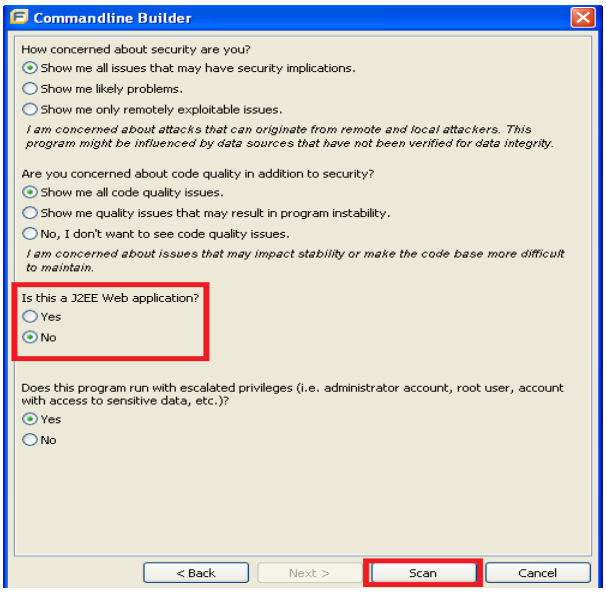


Figura 20: Establecimiento del tipo de aplicación Audit Workbench

4.2.6. Se obtendrán los siguientes resultados del análisis del servicio web REST vulnerable como se ilustra en la Figura 21.

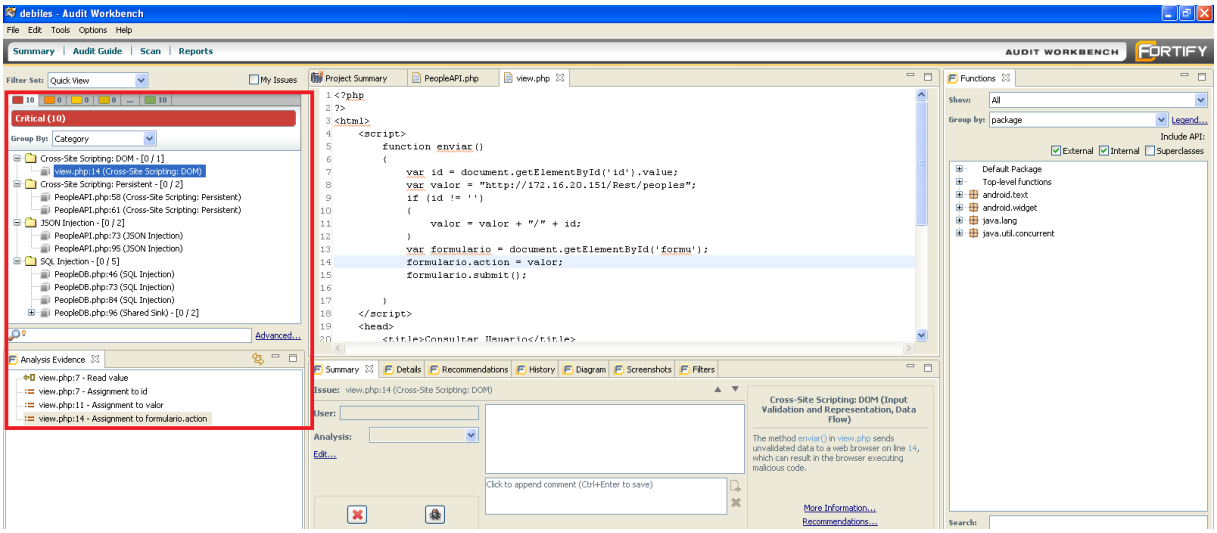


Figura 21: Resultados de análisis Audit Workbench

4.3. Configuración y ejecución de pruebas con ACUNETIX (previo asegurar los servicios Rest desarrollados)

La potente herramienta de ACUNETIX viene en su modo default garantizado para realizar un scan de alto performance a pesar que se lo puede personalizar con credenciales de acceso, modos de profundidad en su ejecución, etc. A continuación, se describe la configuración y pasos seguidos para la ejecución del escaneo de vulnerabilidades del servicio web REST y el cliente vulnerable desarrollados.

4.3.1. Abrir ACUNETIX y seleccionar la opción “New scan” como se presenta en la Figura 22.

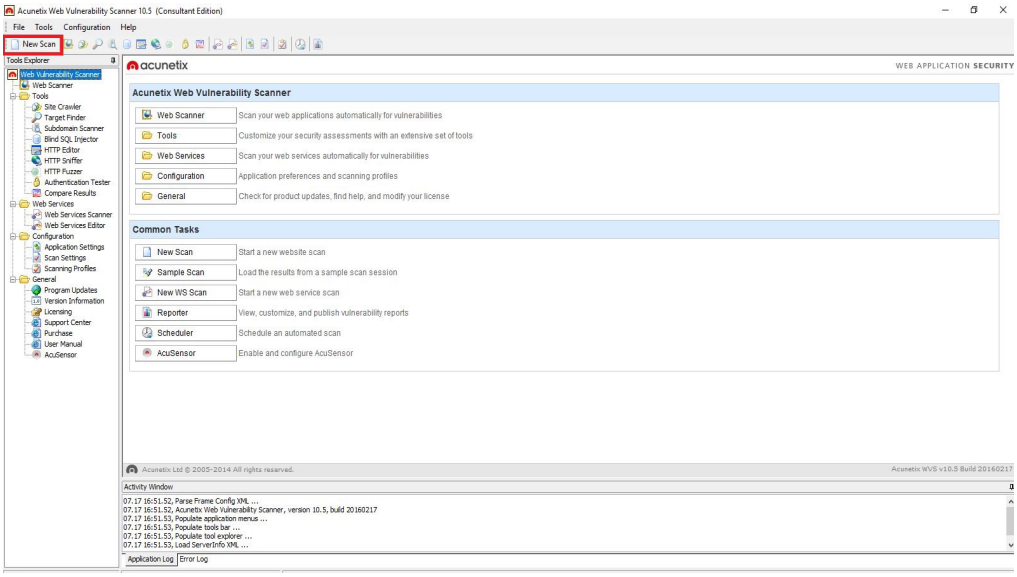


Figura 22: Selección de Nuevo scan ACUNETIX

4.3.2. Ingresar en el campo “Website URL” la dirección objetivo a analizar y clic en el botón “Next” como se indica en la Figura 23.

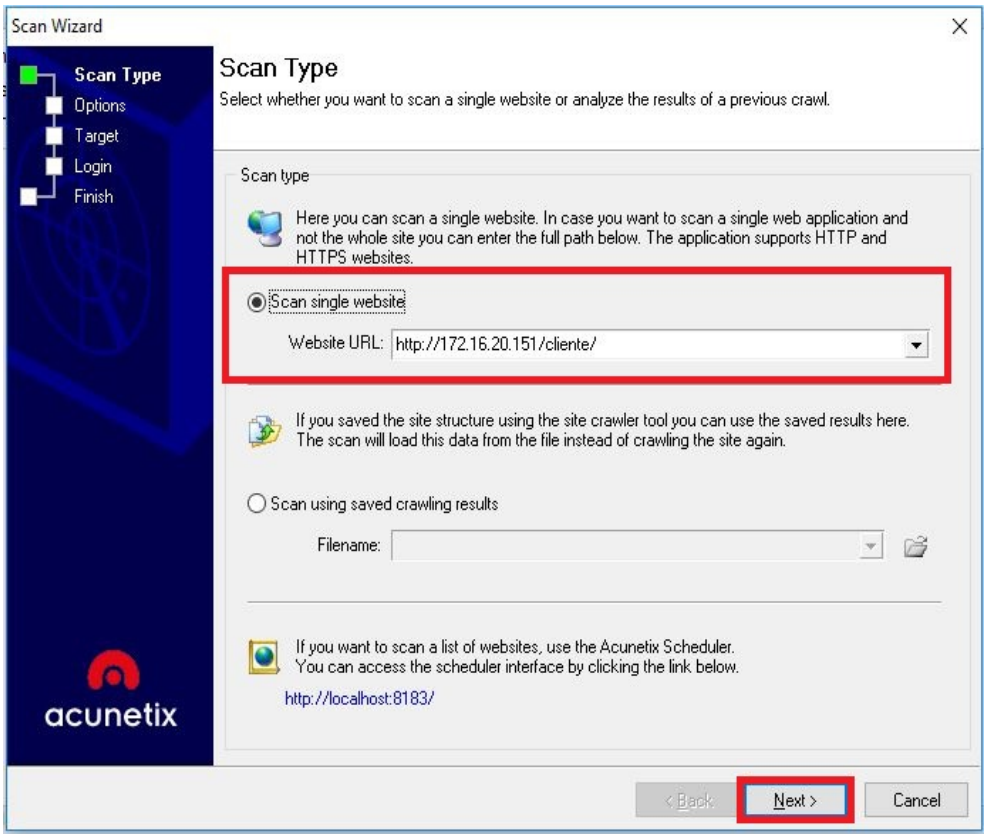


Figura 23: Selección de URL para auditar ACUNETIX

4.3.3. Seleccionar el modo por defecto para el escaneo y las configuraciones; Finalmente clic en el botón Next como se evidencia en la Figura 24.

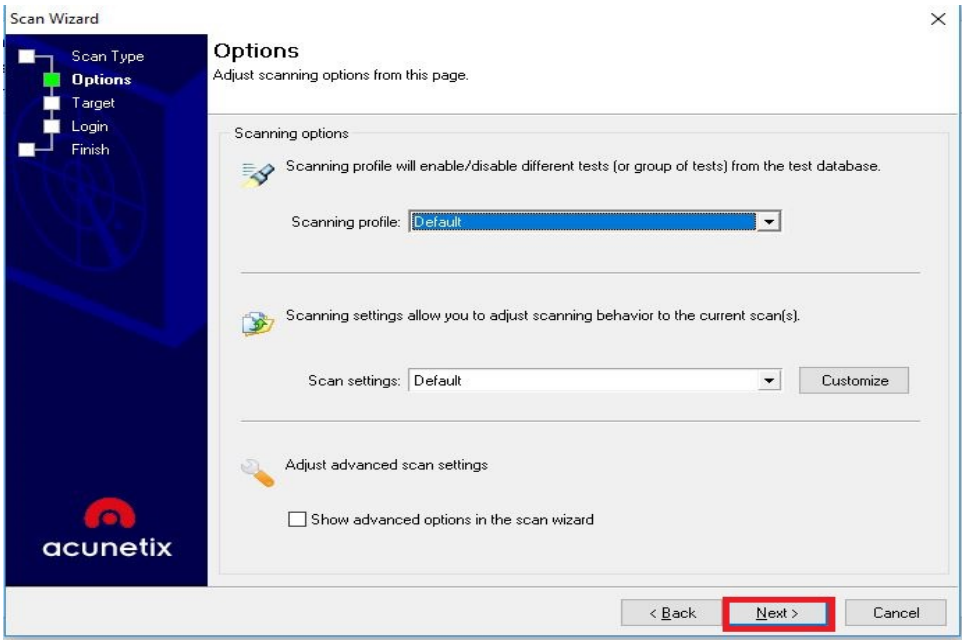


Figura 24: Selección de opciones de escaneo ACUNETIX

4.3.4. Se presentará un breve resumen de los sitios a analizar y se deberá dar clic sobre el botón Next como se ilustra en la Figura 25.

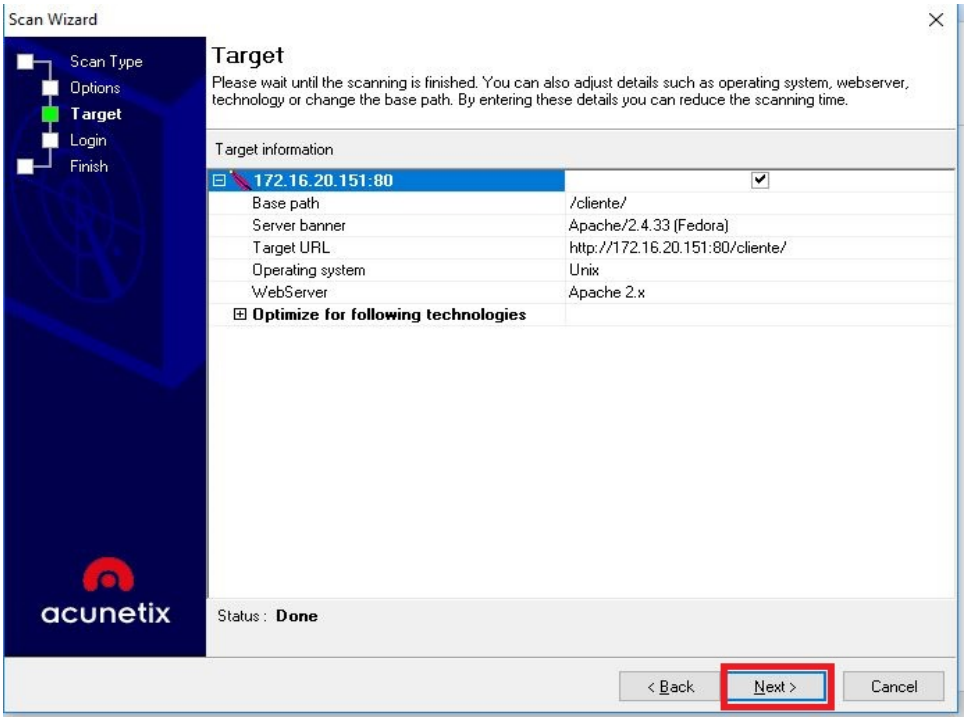


Figura 25: Resumen de configuraciones ACUNETIX

4.3.5. Establecer no login sequence y dar clic en el botón Next como se muestra en la Figura 26.

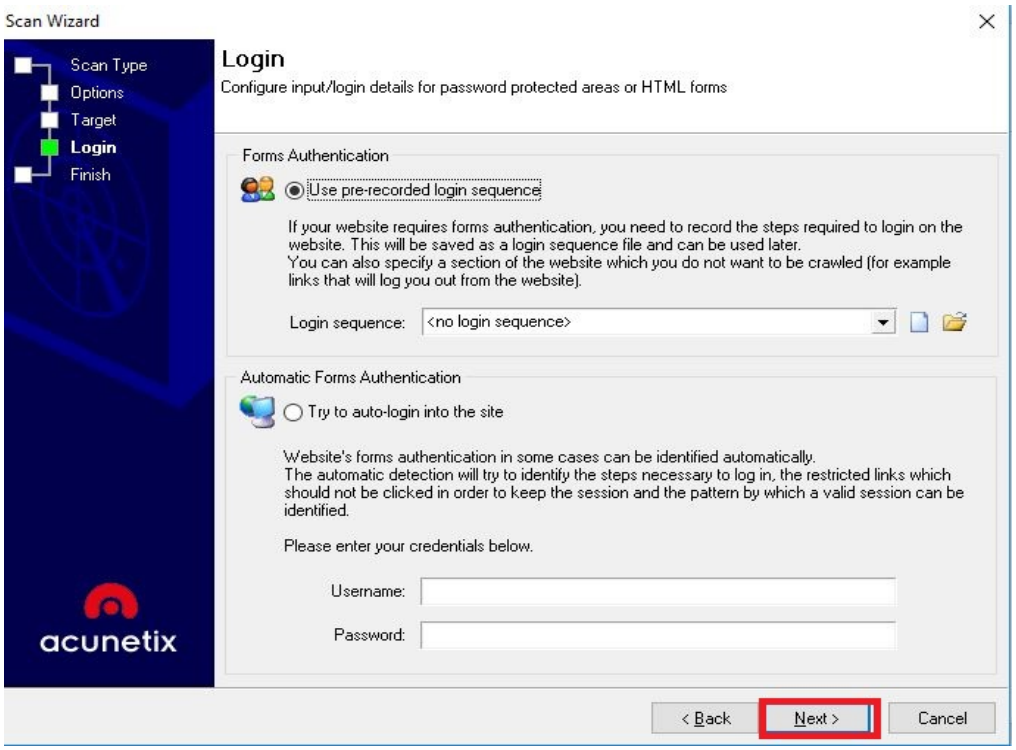


Figura 26: Secuencia de autenticación ACUNETIX

4.3.6. Dar clic en el botón finalizar para finalizar la configuración e iniciar el escaneo del servicio web REST como se presente en la Figura 27.

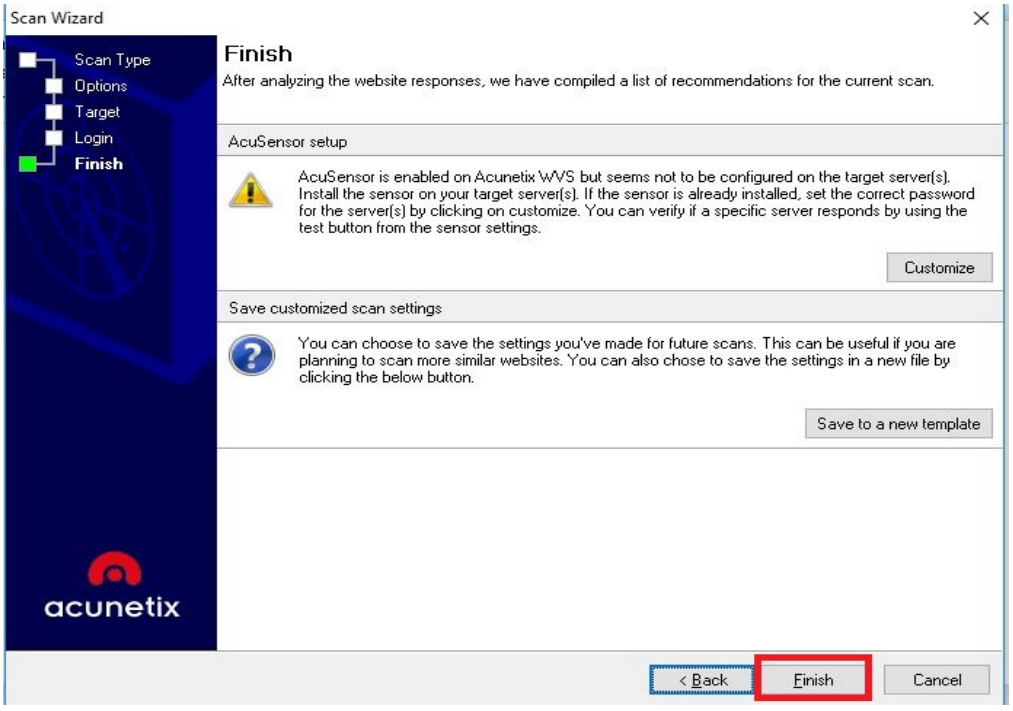


Figura 27: Finalización de configuración e inicio de escaneo ACUNETIX

5. Resultado de las pruebas de análisis de código

5.1. Código vulnerable

5.1.1. Informe emitido por hp Fortify

Según el analizador de código estático Hp Fortify se presenta las siguientes vulnerabilidades en el código desarrollado para los servicios web REST:

5.1.1.1. SQL Injection (SQLi)

- ✓ **Descripción de la vulnerabilidad. -** El método getPeople de la clase PeopleDB.php recibe directamente un parámetro pasado desde la clase PeopleAPI.php en la línea 57 y crea una consulta sin realizar el tratamiento adecuado de los parámetro e entrada que son recibidos según se muestra en la Figura 28 tomado de la herramienta hp Fortify:

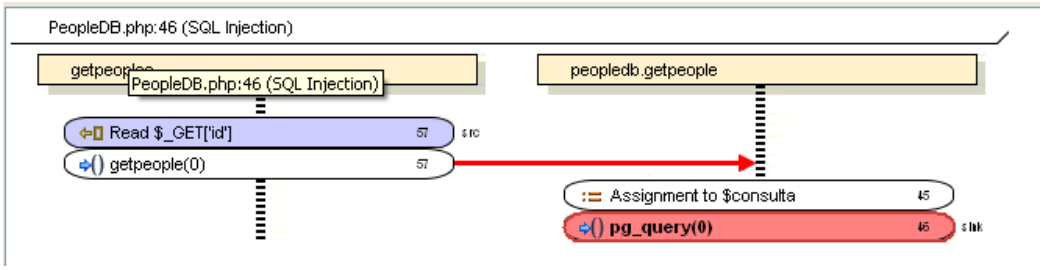


Figura 28:Diagrama de flujo vulnerabilidad SQLi Audit Workbench

En este tipo de ataques bastaría con enviar un parámetro tal como: **"1; select * from people"** en la entrada de datos no validada para obtener toda la data de la tabla sin haber sido autorizado, en otro caso con deseos de mayor daño se podría enviar por ejemplo un **"1; Drop database dbTest"**

- ✓ **Prueba de ataque manual.-** Haremos uso de la url http://172.16.20.151/Rest/peoples y le enviaremos un parámetro 75 que nos daría la información relacionada al usuario en cuestión como se presenta en la Figura 29.

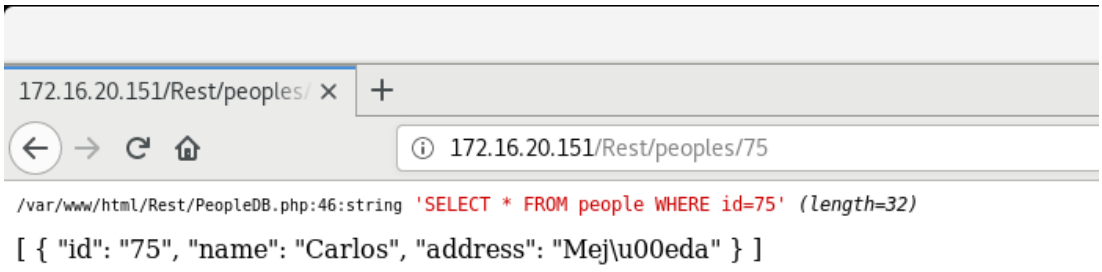


Figura 29: Test Ataque manual SQLi Audit Workbench 1

Se obtiene la respuesta retornada por el servicio web al presentarse una ausencia de registros en la tabla referentes al id solicitado, en este caso será el id 10 para la Url **http://172.16.20.151/Rest/peoples/10** como se ilustra en la Figura 30.

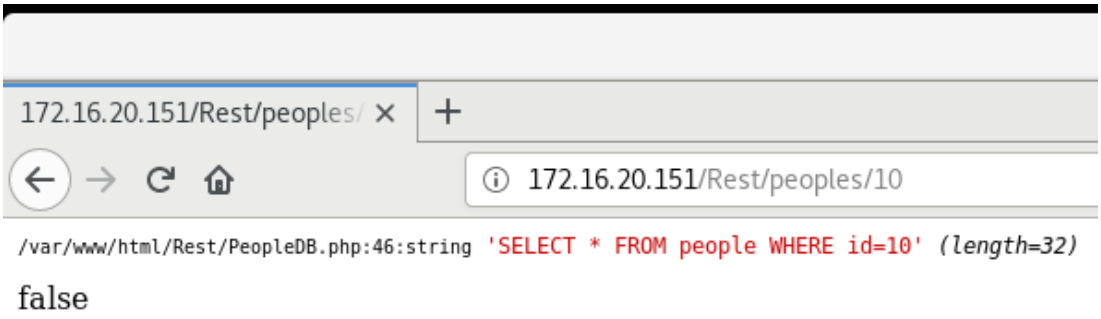


Figura 30: Ataque manual SQLi Audit Workbench 2

Se procede a inyectar código SQL de manera indiscriminada, enviándolo a modo de parámetros de petición para una URL que no posee los mecanismos de protección ni validación adecuados para el tratamiento de datos de entrada en la URL "**http://172.16.20.151/Rest/peoples/10; select * from people**" como se evidencia en la Figura 31.

Figura 31: Ataque manual SQLi Audit Workbench 3

En la Figura 31, el resultado aparenta no ser tan alarmante, sin embargo, un atacante puede ejecutar código SQL por parámetros de URL para eliminar incluso la base de datos. Los atributos referentes a la base de datos pueden ser fácilmente descubiertos usando diferentes mecanismos y herramientas como sqlmap, consecuentemente se obtendría el nombre "dbTest", acto seguido se procederá a realizar la petición por URL con la dirección "172.16.20.151/Rest/peoples/10; Drop database dbTest; select * from people" y cuyo resultado se presenta en la Figura 32.

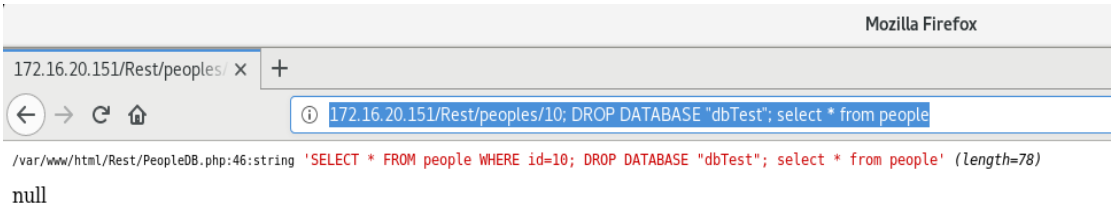
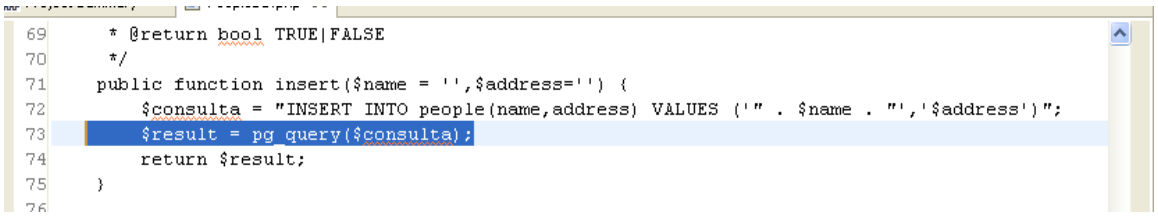


Figura 32: Ataque manual SQLi Audit Workbench 4

Como se muestra en la Figura 32, se ha borrado la base de datos y, por lo tanto, no se tienen datos asociados al id 75. Las consultas SQL que han sido mostradas en cada una de las peticiones por URL realizadas en esta comprobación manual de SQLi, se obtienen mediante una ejecución previa del var_dump () de la consulta SQL resultante que va a ser ejecutada por nuestro DBMS.

✓ Vulnerabilidades SQLi para los demás métodos CRUD

- Se permite SQLi en el método insertar del servicio web REST, esto debido a que no se da ningún tratamiento a las entradas; puede ser en cualquiera de los 2 parámetros que son enviados, tanto en address y name; se puede inyectar consultas como: "Insert into people (name, address) values ('**Carlos',''); Drop database dbTest; --**,'mejía')". El código vulnerable se presenta en la Figura 33 y, el diagrama para SQLi se ilustra en la Figura 34.



```
69      * @return bool TRUE|FALSE
70      */
71     public function insert($name = '',$address='') {
72         $consulta = "INSERT INTO people(name,address) VALUES ('" . $name . "','$address')";
73         $result = pg_query($consulta);
74         return $result;
75     }
```

Figura 33: Vulnerabilidad SQLi método insert

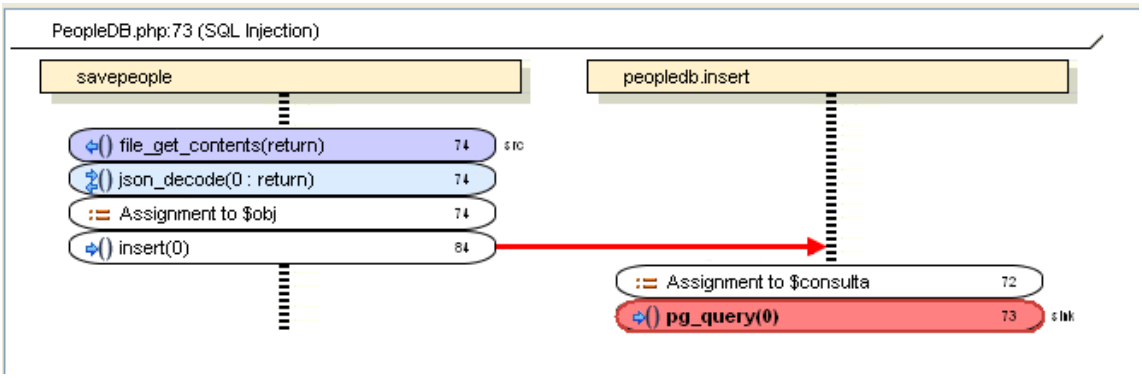


Figura 34: Diagrama de flujo de la inyección SQL para el método insert

- Se permite SQLi en el método **actualizar** del servicio web REST, esto debido a que no se da ningún tratamiento a las entradas; puede ser en cualquiera de los 3 parámetros que son enviados, tanto en: address, id y name; se puede inyectar consultas como: "update people set name = 'Carlos', address = 'dirección' where id= 75**; Drop database dbTest; --**". El código vulnerable es expuesto en la Figura 35 y su respectivo diagrama de flujo es ilustrado en la Figura 36.

```
89 /**
90  * Actualiza registro dado su ID
91  * @param int $id Description
92  */
93 public function update($id, $name,$address) {
94     // if ($this->checkID($id)) {
95     $consulta = "UPDATE people SET name='".$name."', address='".$address."' WHERE id=" . $id;
96     $result = pg_query($consulta);
97     return $result;
98 }
99
```

Figura 35: Vulnerabilidad SQLi método update

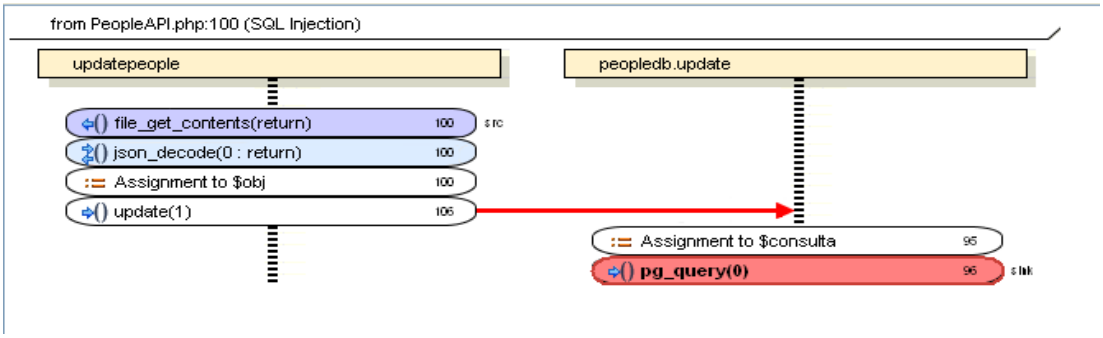


Figura 36: Diagrama de flujo para la realización de SQLi del método update

- Se permite SQLi en el método para eliminar del servicio web REST, esto debido a que no se da ningún tratamiento a las entradas y puede ser en el parámetro id que se ejecuta sin ningún tratamiento previo; se puede inyectar consultas como: “delete from people where id= 75**; Drop database dbTest; --**”. En la Figura 37 se ilustra el código vulnerable y el diagrama de flujo para SQLi es presentado en la Figura 38.

```
80      * @return Bool TRUE|FALSE
81      */
82     public function delete($id = 0) {
83         $consulta = "DELETE FROM people WHERE id=" . $id;
84         $result = pg_query($consulta);
85         return $result;
86     }
87
```

Figura 37 Vulnerabilidad SQLi método Delete

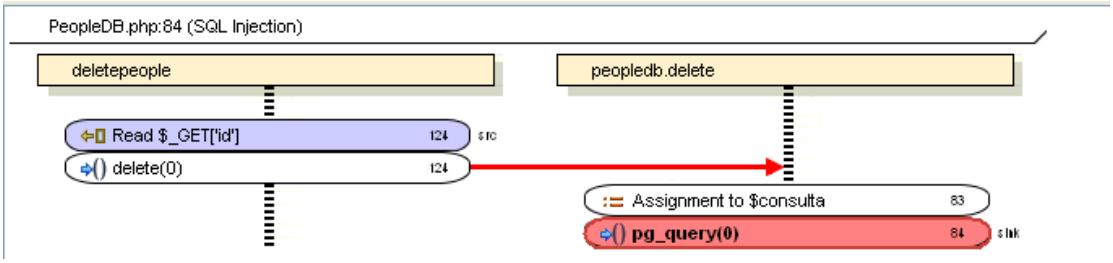


Figura 38: Diagrama de flujo para la realización de SQLi del método Delete

5.1.1.2. XSS (Almacenado/DOM)

- ✓ **Descripción de la vulnerabilidad. -** El método getPeople de la clase PeopleAPI.php realiza una consulta a la base de datos sobre un registro, sus datos son remitidos al mismo método con la finalidad de ser encodificados posteriormente en un JSON mediante el método Json_encode; el problema radica en la falta de tratamiento y validación de los datos que pueden provenir de la base de datos; pudiendo haber sido contaminados anteriormente con un xss almacenado mediante alguna de las entradas al guardar la información sin las debidas precauciones en el DBMS según se reporta en el diagrama de flujo de la herramienta Audit Workbench en la Figura 39.

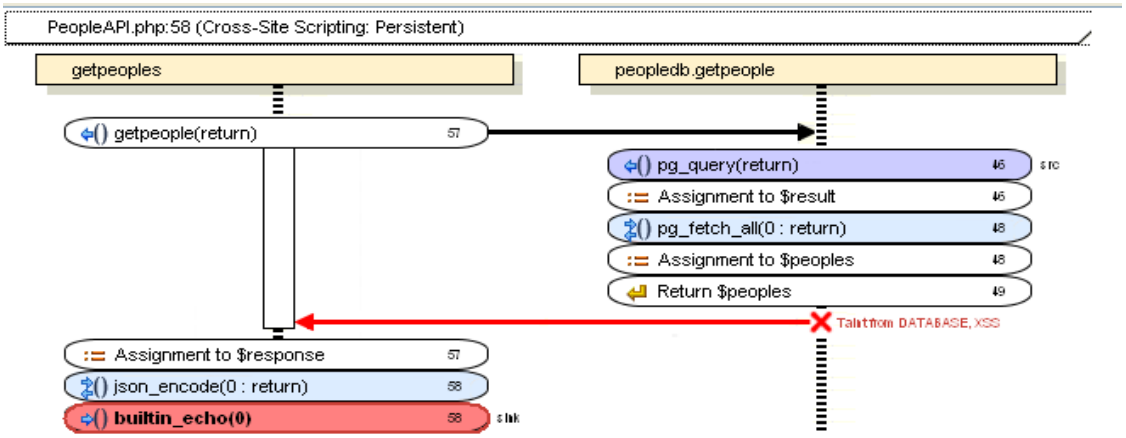


Figura 39: Diagrama de flujo para XSS almacenado

En este tipo de ataques bastaría con almacenar en la base de datos algo tan simple como un **"<script>alert('hackeado'); </script>"** que sería capaz de mostrar ese mensaje molestosamente cada vez que devuelva el resultado de una consulta solicitado por el servicio web al cliente, esto sería en el caso más amable; si nos vamos a un deseo del atacante para realizar acciones maliciosas; se podría almacenar una redirección a un sitio malicioso para comprometer la seguridad de los usuarios.

- ✓ **Prueba de ataque manual.-** Haremos uso de la url http://172.16.20.151/Rest/peoples que nos dará la información relacionada a los usuarios en cuestión como se ilustra en la Figura 40:

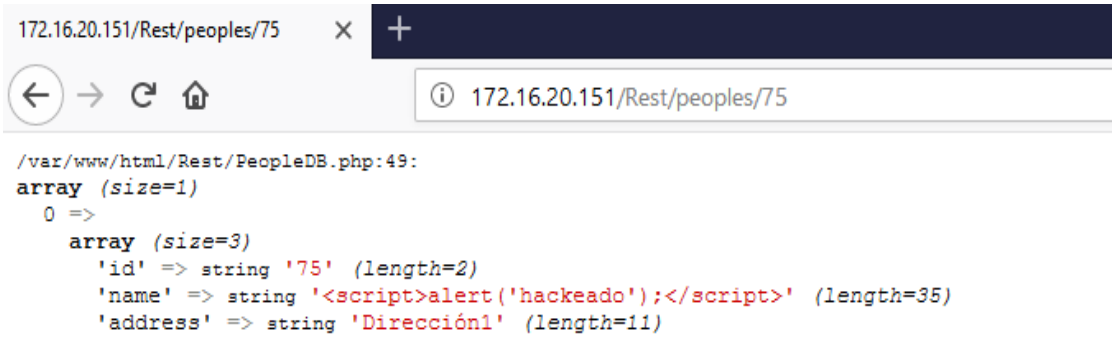


Figura 40: Ataque manual XSS Almacenado

Si observamos más detalladamente, en el lado del cliente, al dar tratamiento a un JSON con el método Json_decode (); se obtendría en "sin tratamiento" los valores maliciosos obtenidos de la base de datos; lo que originaría un molestoso mensaje de "hackeado" o en el peor de los casos redirecciones a otros sitios y posibles acciones maliciosas que comprometerían la seguridad de nuestro servicio Web y sus clientes.

✓ Vulnerabilidad XSS almacenado presentes en el servicio WEB

❖ Se permite XSS almacenado en el método getPeople al mostrar todos los registros de las personas almacenadas en la base de datos. De igual modo que en el caso anterior el problema se produce por un exceso de confianza por parte del programador al fiarse falsamente de que los datos provenientes de su base de datos van a venir siempre sin contaminación. En la Figura 41 se expone el código que permite la vulnerabilidad y su respectivo diagrama de flujo es presentado en la Figura 42.

```
54    if ($_GET['action'] == 'peoples') {
55        $db = new PeopleDB();
56        if (isset($_GET['id'])) {//muestra 1 solo registro si es que existiera ID
57            $response = $db->getPeople($_GET['id']);
58            echo json_encode($response, JSON_PRETTY_PRINT);
59        } else { //muestra todos los registros
60            $response = $db->getPeoples();
61            echo json_encode($response, JSON_PRETTY_PRINT);
62        }
63    } else {
64        response(400);
65    }
```

Figura 41: Código fuente vulnerable XSS Almacenado para el método getPeoples()

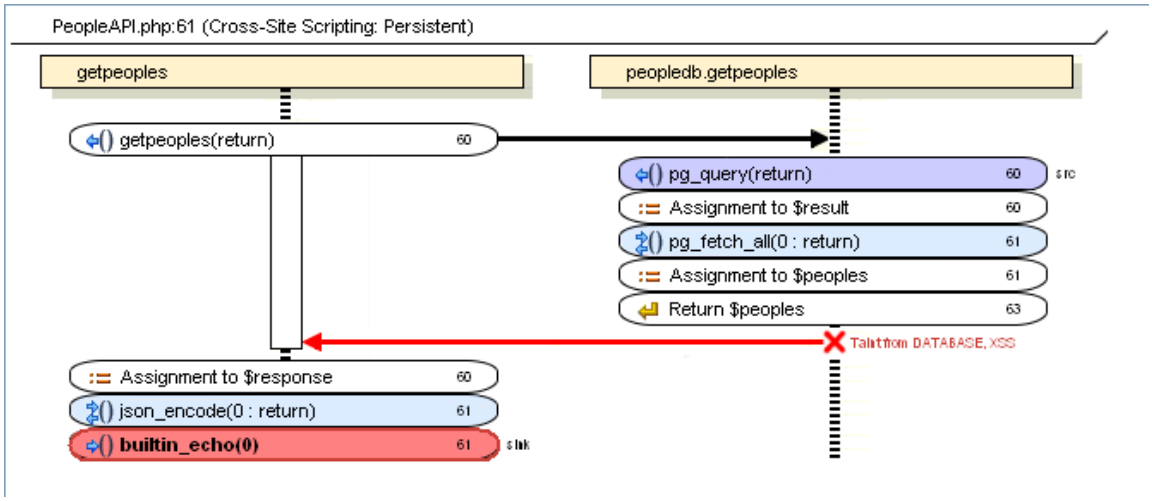


Figura 42: Diagrama de flujo para XSS almacenado en el método getPeoples()

✓ Vulnerabilidad XSS DOM presente en el servicio WEB

Se permite XSS DOM en el método enviar del script **view.php** en el cliente REST, básicamente el problema se origina por la falta de validación una vez más; al preguntar pobremente si sólo es diferente de vacío el parámetro que se recibe y no tomar en cuenta: si tiene caracteres especiales, controlar el máximo/mínimo de longitud,

controlar si es entero, etc. Esto puede originar re-direcciones a consultar datos no permitidos o peor aún realizar una redirección a un sitio que no sea de fiar. En la figura 43 se muestra la codificación insegura y en la Figura 44 su respectivo diagrama de flujo.

Project Summary | view.php

```
1  <?php
2  ?>
3  <html>
4      <script>
5          function enviar()
6          {
7              var id = document.getElementById('id').value;
8              var valor = "http://172.16.20.151/Rest/peoples";
9              if (id != '')
10             {
11                 valor = valor + "/" + id;
12             }
13             var formulario = document.getElementById('formu');
14             formulario.action = valor;
15             formulario.submit();
16
17         }
18     </script>
19     <head>
20         <title>Consultar Usuario</title>
```

Figura 43: Código vulnerable para XSS DOM

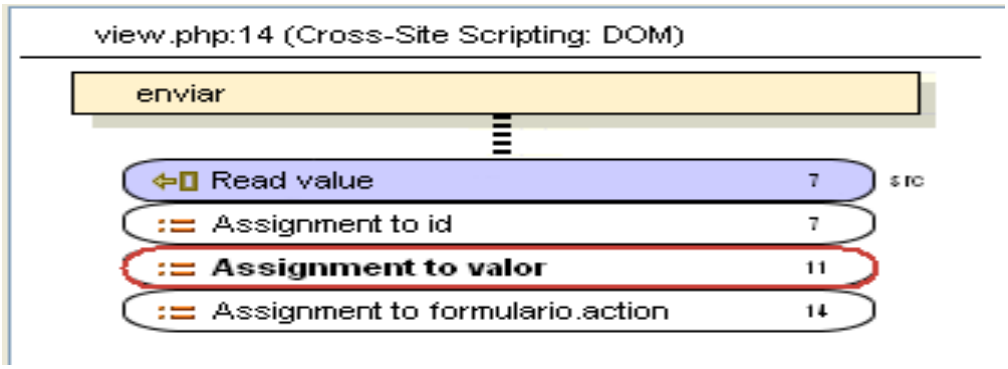


Figura 44: Diagrama de flujo de XSS DOM para view.php

5.1.1.3. JSON Injection

- ✓ **Descripción de la vulnerabilidad. –** La inyección de JSON es producida en este caso particular en el script PeopleAPI.php al tomar los datos del php://input sin ser tratados ni validados previamente, luego se realiza un json_decode para obtener la información en claro, misma que puede ser maliciosa y ocasionar daños al ser tratada como un arreglo de datos que van a ser pasados a los diferentes métodos para guardar los datos de una persona. La implementación débil a nivel de código se presenta en la Figura 45 acompañado de su respectivo diagrama de flujo en la Figura 46.

Project Summary | view.php | PeopleAPI.php

```
68 /* Inserta un objeto en la tabla people */
69
70 function savePeople() {
71     if ($_GET['action'] == 'peoples') {
72         //Decodifica un string de JSON
73          if (is_string(file_get_contents('php://input'))) {
74         $obj = json_decode(file_get_contents('php://input'));
75             $objArr = (array) $obj;
76         } else {
77             $objArr = array();
78         }
```

Figura 45: Fuente de datos no validada para JSON Injection método savePeople

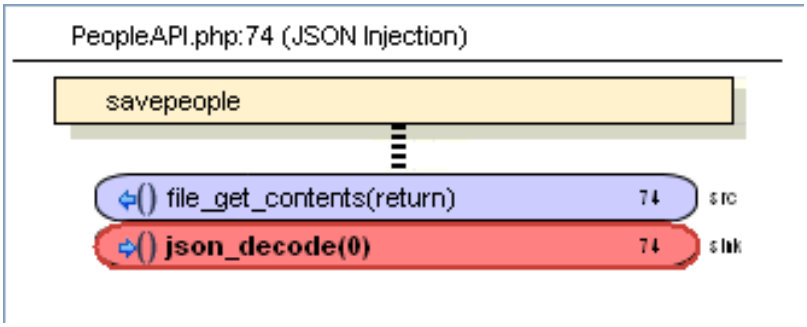


Figura 46: Diagrama de flujo de JSON Injection para el método savepeople

✓ Vulnerabilidad JSON Injection de updatePeople.

❖ Se produce la vulnerabilidad al no tomar en cuenta o dar por sentada la validación de los datos tomados desde “php://input” para luego ser decodificados por el método json_decode en una fase previa a obtener los valores necesarios para realizar la actualización de un registro de la tabla People. En la Figura 47 se muestra la vulnerabilidad a nivel de código acompañado de su respectivo diagrama de flujo en la Figura 48.

Project Summary | view.php | PeopleAPI.php

```
94 /**
95  * Actualiza un recurso
96  */
97 function updatePeople() {
98     if (isset($_GET['action']) && isset($_GET['id'])) {
99         if ($_GET['action'] == 'peoples') {
100             $obj = json_decode(file_get_contents('php://input'));
101             $objArr = (array) $obj;
102             if (empty($objArr)) {
103                 response(422, "error", "Nothing to add. Check json");
104             } else if (isset($obj->name)) {
105                 $db = new PeopleDB();
106                 $db->update($_GET['id'], $obj->name, $obj->address);
107                 response(200, "success", "Record updated");
108             } else {
109                 response(422, "error", "The property is not defined");
110             }
111             exit;
```

Figura 47: Fuente de datos no validada para JSON Injection método updatePeople()

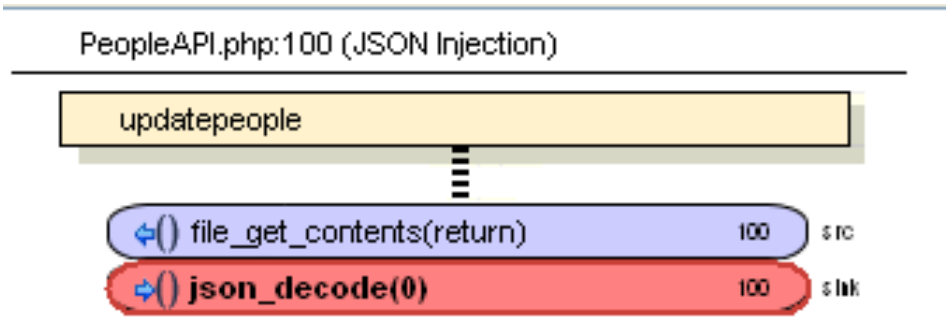


Figura 48: Diagrama de flujo de JSON Injection para updatePeople

5.1.2. Informe emitido por Owasp-Zap

Owasp ZAP al ser un analizador de código de modo dinámico, permite detectar algunos problemas con las peticiones que se pueden enviar en caliente a un servidor desde el lado del cliente; para esto realiza un análisis de cabeceras y configuración del servidor en el que se encuentra desplegado. A continuación, se muestran las siguientes vulnerabilidades detectadas en el servicio web REST desarrollado para auto-aprendizaje:

5.1.2.1. Sobre-escritura de parámetros HTTP

- ✓ **Descripción de la vulnerabilidad. –** Este tipo de vulnerabilidad se presenta cuando no se especifica el action de un formulario y se lo deja en blanco para posteriormente ser llenado desde JS o similares, como una buena práctica se recomienda que todos los formularios deben poseer especificada su action y evitar el

modificarla de modo dinámico mediante código JS o similares; ya que esto puede permitir a un atacante modificar el lugar a donde son enviados los datos del formulario. Esta vulnerabilidad se encuentra presente en el script “view.php” del cliente como se muestra en la Figura 49 (HTML) y la Figura 40 (JS):

```
<form id="formu" action="" method="get">
    <input type="submit" value="Consultar" onclick="enviar()">
</form>
```

Figura 49: Script view.php vulnerable a sobre-escritura de action

```
<script>
    function enviar()
    {
        var id = document.getElementById('id').value;
        var valor = "http://172.16.20.151/Rest/peoples";
        if (id != '')
        {
            valor = valor + "/" + id;
        }
        var formulario = document.getElementById('formu');
        formulario.action = valor;
        formulario.submit();

    }
</script>
```

Figura 50: Código js que permitiría la modificación del atributo action en view.php

En este tipo de ataques bastaría con que el atacante logre reemplazar el JavaScript al que se está apuntando en el formulario y de este modo propender a acciones maliciosas contra la integridad de los datos y la información enviada por el usuario.

- ✓ **Scripts asociados a esta vulnerabilidad. -** En el servicio web REST desarrollado, por la misma naturaleza de programación por parte del programador, se han replicado a los scripts que permiten la creación, eliminación y actualización de registros como operaciones básicas que fueron planteadas para este TFM, a continuación, se muestran los casos de vulnerabilidad para los scripts de crear, actualizar y eliminar que están referenciados en las Figuras de la 51 a la 56.

  - ❖ **Formulario vulnerable para la operación de creación**

```
<form id="formu" action="" method="get">
    <input type="submit" value="Crear" onclick="enviar()">
</form>
```

Figura 51: Script create.php vulnerable a sobre-escritura de action

❖ **Código JavaScript que permite la vulnerabilidad**

```
function enviar()
{
    var formulario = document.getElementById('formu');
    formulario.action="http://172.16.20.151/Rest/peoples";
    crear();
    formulario.submit();
}
```

Figura 52: Código js que permitiría la modificación del atributo action en create.php

❖ **Formulario vulnerable para la operación de actualización**

```
<form id="formu" action="" method="get">
    <input type="submit" value="Editar" onclick="enviar()">
</form>
```

Figura 53: Script update.php vulnerable a sobre-escritura de action

❖ **Código JavaScript que permite la vulnerabilidad**

```
function enviar()
{
    var formulario = document.getElementById('formu');
    formulario.action = "http://172.16.20.151/Rest/peoples";
    actualizar();
    formulario.submit();
}
```

Figura 54: Código js que permitiría la modificación del atributo action en update.php

❖ **Formulario vulnerable para la operación de eliminar**

```
<form id="formu" action="" method="get">
    <input type="submit" value="Eliminar" onclick="enviar()">
</form>
```

Figura 55: Script delete.php vulnerable a sobre-escritura de action

❖ **Código JavaScript que permite la vulnerabilidad**

```
function enviar()
{
    var formulario = document.getElementById('formu');
    formulario.action="http://172.16.20.151/Rest/peoples";
    eliminar();
    formulario.submit();
}
```

Figura 56: Código js que permitiría la modificación del atributo action en delete.php

5.1.2.2. Ausencia de tokens anti CSRF

✓ **Descripción de la vulnerabilidad. –** Este tipo de vulnerabilidad se produce cuando el desarrollador no gestiona un sistema de tokens para validar que las peticiones que son enviadas hacia el servidor son auténticas, esto puede desencadenar un ataque de peticiones http enviadas por el atacante que sin el conocimiento de la víctima pueden realizar acciones en su nombre aprovechado la autenticación de su víctima, Hay algunos métodos de resolver este inconveniente como son el uso de librerías en el diseño de la arquitectura del servicio REST que permita generar tokens robustos para incluirlos en cada petición a realizar. Los scripts php afectados para este caso en específico serán los anteriores (create, update, view, delete) que permiten la gestión básica de las operaciones CRUD. En el capítulo seis se mostrarán las posibles soluciones a este y diversos otros problemas.

5.1.2.3. Cross-Domain JavaScript Source File Inclusion

✓ **Descripción de la vulnerabilidad. –** Este tipo de vulnerabilidad se presenta cuando el desarrollador de software incluye scripts que no se encuentran alojados en su propio servidor, de esta manera al hacer referencia a un dominio que no es propio o verificable; pueden darse cambios en el script remoto que podrían comprometer la funcionalidad de nuestro sitio e inclusive podrían generar un back door por una implementación de funcionalidad no deseada en el script referido; a continuación se muestra una imagen en la que se hace referencia a una llamada errónea de un recurso que no se encuentra alojado en nuestro servidor. La ilustración gráfica del problema de seguridad se expone en la Figura 57.

Figura 57: Evidencia de Cross-Domain JavaScript Source File Inclusion

- ✓ **Scripts asociados a esta vulnerabilidad. –** Este mismo modo de llamar al script de jQuery, se encuentra presente como vulnerabilidad del servicio web REST desarrollado en los scripts para actualizar, eliminar, y crear (update.php, delete.php, create.php). Una solución adecuada para este tipo de vulnerabilidad puede ser el descargar el script necesario en el servidor y hacer referencia directa a él.

5.1.2.4. Falta de configuración adecuada

La ausencia de cabeceras que garanticen la seguridad en las peticiones HTTP es muy frecuente como un defecto de falta de configuración, esto es debido a que los desarrolladores generalmente se centran en la funcionalidad y no se toman un tiempo detallado para analizar si el ambiente en el que va a ser desplegada su solución informática es lo suficientemente robusta y segura. La falta de cultura en materia de seguridad informática por parte de los desarrolladores permite muchos problemas de seguridad, a continuación, se detallan las vulnerabilidades que han sido cometidas en el desarrollo de los servicios web REST para este TFM.

- ✓ **X-Frame-Options Header Not set. –** La ausencia de la definición de esta cabecera puede desencadenar ataques de ClickJacking que es usado generalmente para embeber la funcionalidad de un servicio web en una página que puede ser elaborada por un atacante con la finalidad de generar confusión en las víctimas de este ilícito. Se recomienda el uso de la política DENY que impedirá el ser embebida la funcionalidad en una página web ajena al dominio del servicio web. Su configuración se lo verá en el capítulo sexto. La Figura 58 expone la falta de la cabecera X-Frame-Options.

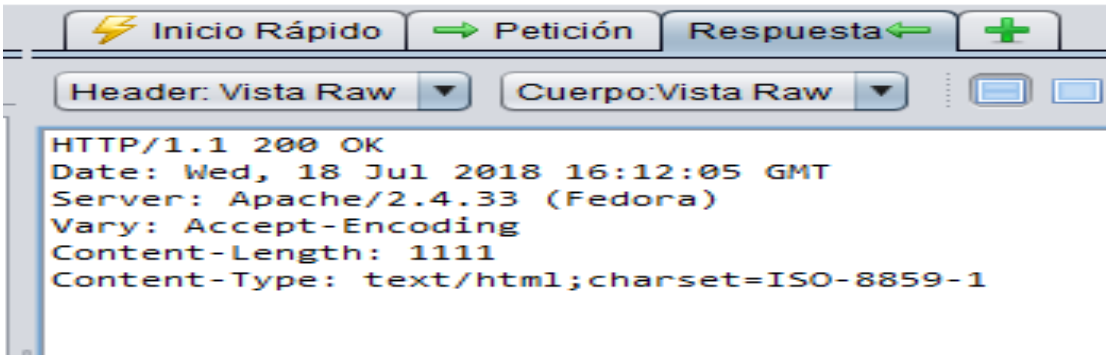


Figura 58: Ausencia cabecera X-Frame-Options Header

- ✓ **Encabezado (CSP) no establecido. -** La definición de esta cabecera ya sea en los archivos de configuración del servidor o en los scripts php de los que se hace uso; es de vital importancia tomando en cuenta que ayuda a detectar y mitigar posibles ataques XSS y de inyección de datos, la definición de CSP proporciona un conjunto de encabezados estándar HTTP que facultan a los propietarios de los sitios web declarar el contenido de tipo de scripts y ficheros que se pueden ejecutar en sus sitios. Su configuración se lo verá en el capítulo sexto. La Figura 59 evidencia la falta de la cabecera CSP.

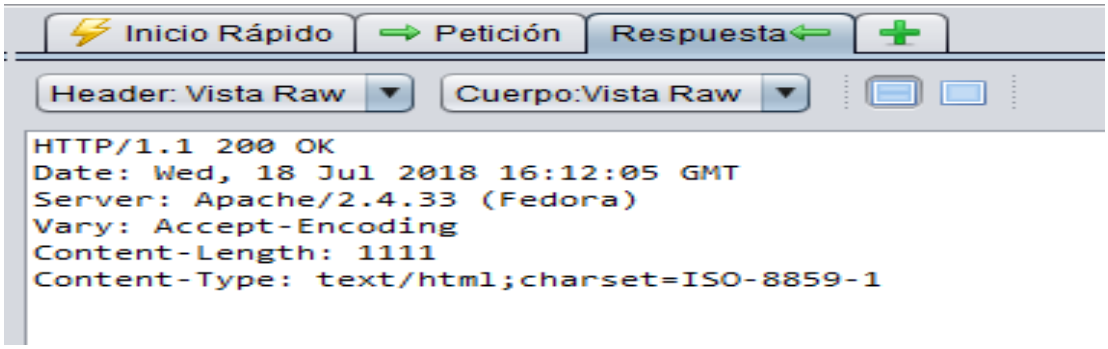


Figura 59: Ausencia encabezado CSP

- ✓ **Divulgación de información de versión del servidor. –** Una vulnerabilidad que causa grandes problemas tomando en cuenta aspectos de seguridad es la divulgación de la información referente al tipo de servidor que está siendo de anfitrión para el servicio web; esto permite al atacante conocer la versión exacta que se está utilizando en un servidor y de este modo buscar vulnerabilidades y exploits que permitan atacar y comprometer el servidor. La acción recomendable es inhabilitar toda información relevante a versiones y tipo de

servicios levantados en un servidor que son mostrados en la respuesta a peticiones realizadas por los clientes; a continuación, se muestra una imagen de divulgación de información de nuestro servidor de aplicaciones web. Su configuración se lo verá en el capítulo sexto. La Figura 60 ilustra la divulgación de la versión del servidor.

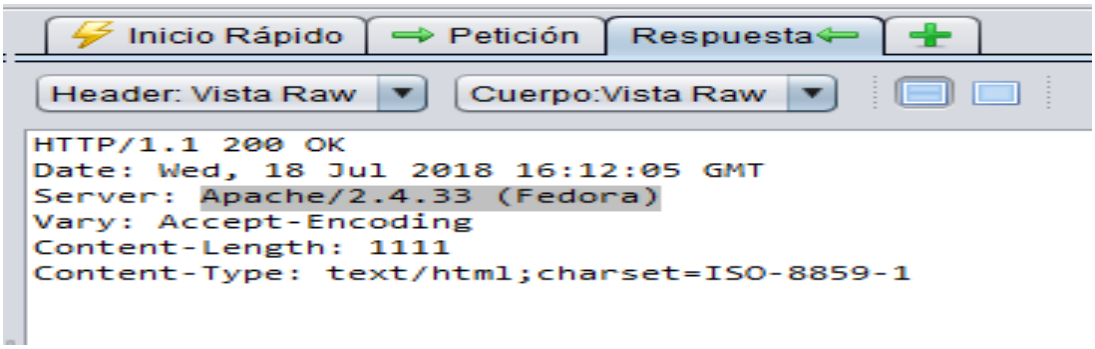


Figura 60: Divulgación de información de versión del servidor

- ✓ **Web Browser XSS Protection Not Enabled. –** La cabecera de XSS Protection permite al navegador conocer si debe o no habilitar su mecanismo embebido para control y protección contra XSS; por defecto los navegadores lo traen habilitado, sin embargo, dada la posibilidad de configuración pueden ser deshabilitados, saltándose esta barrera de seguridad, el uso de este tipo de flags permite al desarrollador asegurarse o forzar que el filtro contra XSS se encuentre siempre habilitado en el lado del cliente por orden del servidor de aplicaciones web. Su configuración se lo verá en el capítulo sexto. La Figura 61 muestra la ausencia de la cabecera Web Browser XSS Protection en la petición realizada.

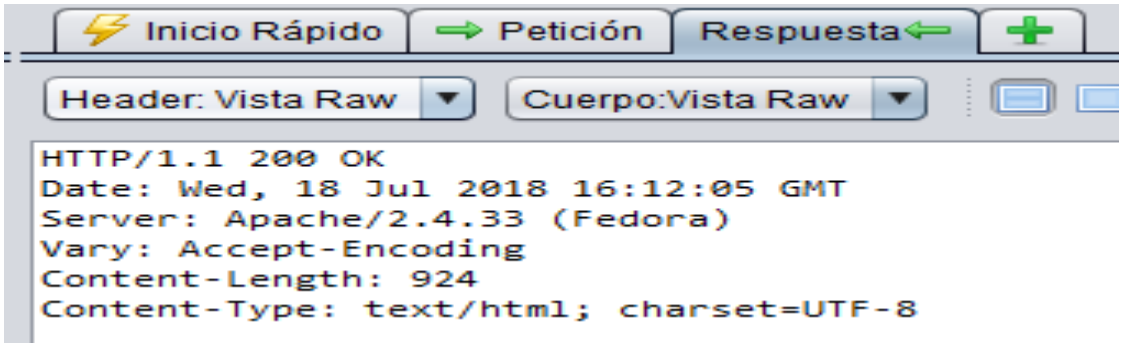


Figura 61: Ausencia cabecera Web Browser XSS Protection

- ✓ **X-Content-Type-Options Header Missing. –** La cabecera de anti sniffing para el contenido MIME, es una medida de seguridad que ayuda a evitar que navegadores como google Chrome e Internet Explorer intenten interpretar el tipo de contenido de MIME de un archivo subido desde el lado del usuario; de este modo respeta el tipo de MIME que se le fue indicado por parte del servidor y es de gran ayuda para evitar posibles XSS ya se controla directamente desde la respuesta del servidor el tipo de MIME que se debe considerar. La Figura 62 ilustra una petición realizada sin el uso de la cabecera X-Content-Type-Options.

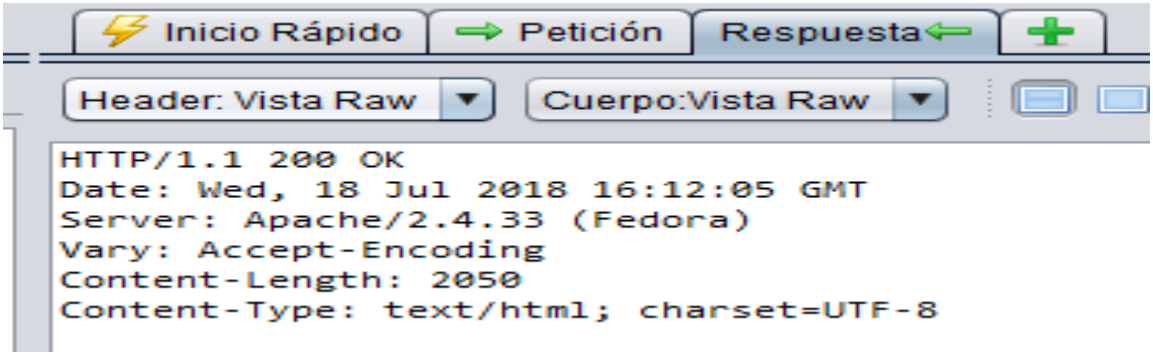


Figura 62: Ausencia cabecera X-Content-Type-Options

- ✓ **Contenido almacenable y almacenable en caché. –** El desarrollador debe tener muy en cuenta que datos se pueden almacenar en caché, debido a que al no personalizar la configuración de caché de la cabecera HTTP, la configuración por defecto permite que los contenidos de las respuestas sean almacenados en caché en puntos intermedios que pueden ser servidores proxy, para agilitar la obtención de datos desde el servidor, esto si bien implica una mejora en el rendimiento de los servicios web REST, puede también suponer un agujero de seguridad al almacenar datos sensibles como passwords o peor aún tokens de sesión en la caché de proxys que pueden ser obtenidos fácilmente incurriendo en un problema de divulgación de información. La Figura 63 expone la falta de una cabecera que controle el tiempo de vida y el atributo caché.

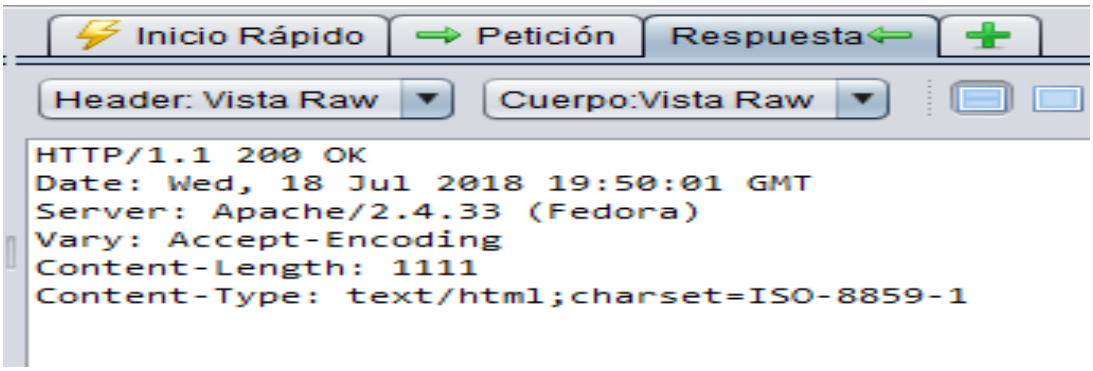


Figura 63: Cabeceras para contenido no almacenable/almacenable

- ✓ **Contenido no almacenable. –** Si la respuesta emitida por el servidor no contiene datos que sean comprometedores y sensibles y si por el contrario contienen datos que pueden acelerar el rendimiento de los servicios web REST, se debe considerar almacenarlos en caché mediante el establecimiento de directivas de cabecera. Se debe considerar el establecimiento de un tiempo máximo de vida de la información almacenada en caché.

5.1.3. Informe emitido por ACUNETIX

ACUNETIX al igual que el analizador de código dinámico de OWASP ZAP, permite detectar vulnerabilidades una vez ya se encuentra desplegado el servicio web REST y, de igual modo el cliente que consumirá ese servicio web REST. Básicamente la principal diferencia es su motor de inferencia basado en sus propias reglas y, que la herramienta ACUNETIX cuenta con la opción de Acusensor, misma que permite detectar una vulnerabilidad en tiempo de ejecución ya que funciona a modo de un IDS, analizando y alertando las consultas que van hacia la base de datos, esto lo hace interponiéndose entre el servicio web REST y la base de datos. A continuación, se listan las vulnerabilidades reportadas por ACUNETIX.

5.1.3.1. Falta de X-Frame-Options header. – Como se revisó en las vulnerabilidades reportadas por la herramienta OWASP ZAP, la ausencia de este tipo de cabecera permite los ataques de ClickJacking, se debe habilitar este tipo de cabecera para evitar el hecho de que la funcionalidad de nuestro servicio web REST sea embebida en páginas ajenas al propio dominio en el cuál se encuentra desplegado. En la Figura 64

se muestra el encabezado de petición y respuesta capturado por ACUNETIX que no tiene configurado X-Frame-Options.

Figura 64: ausencia de las cabeceras para X-Frame-Options header

5.1.3.2. Método options se encuentra habilitado. – La consecuencia de estar habilitada esta bandera implica que se puede conocer las opciones y métodos en forma de una lista que son provistos por el servidor web, básicamente esto puede servir al atacante para conocer información acerca de los métodos que son expuestos por el servidor web y ayudarle a desarrollar mecanismos más avanzados de ataque, la mejor recomendación que se puede dar para este tipo de vulnerabilidad es la deshabilitación del método options en el servidor web mediante algunas configuraciones en su archivo de configuración. A continuación, en la Figura 65 se presenta las cabeceras de petición y respuesta que ha capturado ACUNETIX y evidencia estar activo el método options.

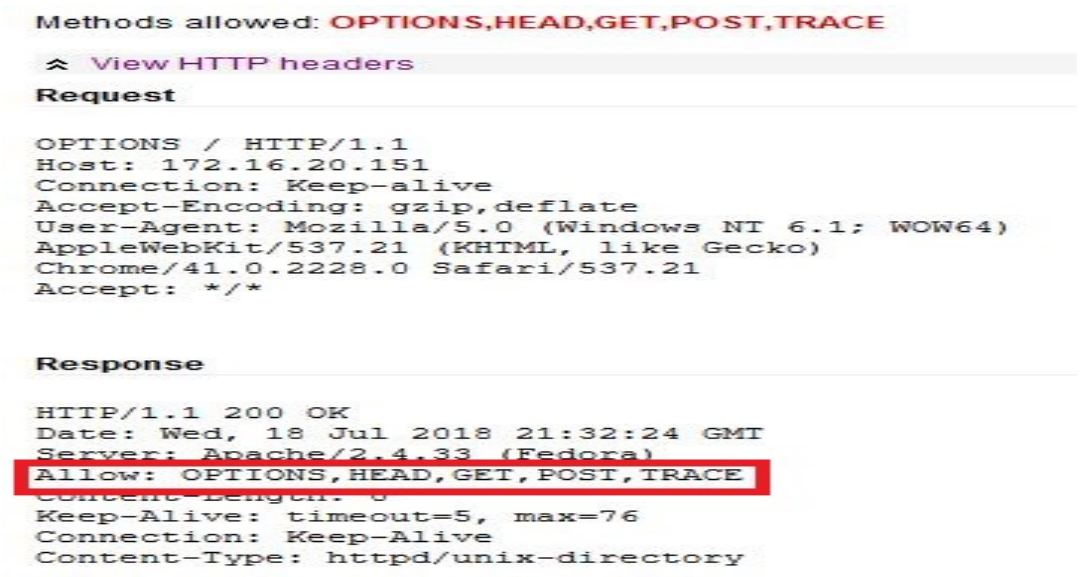


Figura 65: Método Options habilitado

5.1.3.3. Método trace se encuentra habilitado. – Al encontrarse habilitada este método, un atacante pude valerse de la información sensitiva que viaja en las cabeceras de tal modo que, ante un ataque de tipo XSS, cualquier dominio que incluya HTTP TRACE puede habilitar la captura de información importante de las cabeceras que son enviadas en las peticiones de usuario tales como cookies o información de datos de autenticación. La mejor recomendación para este tipo de vulnerabilidad es la deshabilitación del método TRACE, a continuación, en la Figura 66 se muestra la información de la petición capturada por ACUNETIX teniendo el método trace habilitado.

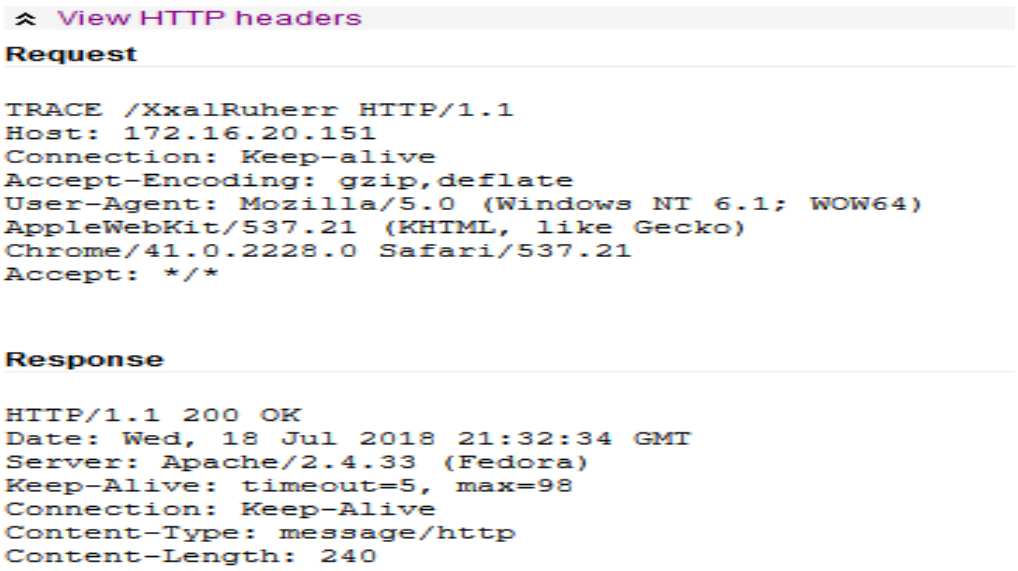


Figura 66: Método trace habilitado

5.2. Breve análisis del código y configuraciones corregidas

Una fase importante de cara a mitigar y procurar solventar las vulnerabilidades reportadas por las distintas herramientas de análisis de código y en general de análisis de la seguridad de los servicios web REST vulnerables desarrollados para este trabajo; es la comprobación manual en los casos que se permitan y más importante aún, es la comprensión de la lógica del programador para de este modo buscar los mecanismos para reemplazar o parchar el código implementado a modo que la seguridad no implique sacrificar funcionalidad del servicio REST en ningún instante.

5.2.1. Mejoras sobre el informe emitido por hp Fortify

5.2.1.1. SQLi

- ✓ **Descripción de la corrección. –** Para evitar los ataques del tipo inyección de código SQL hay principalmente 3 maneras de hacerlo según OWASP:

  - ❖ **Empleando sentencias pre-compiladas**
  - ❖ **Implementando procedimientos almacenados**
  - ❖ **Validando la entrada de los datos ingresados por el usuario.**

  Para el presente Trabajo de servicios Web REST vulnerables se implementará la solución de sentencias pre-compiladas o **prepared statements**; este tipo de método realiza una validación sobre el tipo de parámetros que son ingresados por el usuario, de manera que en el motor de base de datos separa la consulta SQL de los datos que la conforman para ser formada; El DBMS en primer instancia evalúa los parámetros que son enviados por el usuario y verifica si no están relacionados con funciones y operaciones específicas del lenguaje SQL. Si pasa todos los controles, se ejecuta la operación de SQL. En el caso específico de PHP y Postgresql se emplearán los métodos nativos del lenguaje como son**: pg_prepare (), pg_execute (), mysqli->prepare (), bind_param (), execute (), store_result ().** Como una ventaja adicional se tiene que emplear este tipo de métodos mejora el rendimiento de las consultas y optimiza su tiempo de ejecución.

- ✓ **Corrección de las vulnerabilidades en los métodos reportados**

  La clase PeopleDB.php es la encargada de alojar los métodos que interactúan directamente con la base de datos, a modo de modelo en un patrón MVC, por lo que es en esta capa donde se implementarán las soluciones a nivel de código propuestas.

- **Método getPeople ($id = 0). –** Se obtiene una instancia de la conexión a la base de datos (1), se construye la sentencia pre-compilada del select y es evaluada (2), se ejecuta la sentencia pre-compilada en el caso de ser válida (3). Si el resultado de la ejecución del query es true, se retornan todos los elementos (4). Se cierra la conexión a la base de datos permitiendo la liberación de memoria (5). Se retorna null en el caso que la consulta no se haya ejecutado de manera correcta (6). En la Figura 67 se indican las correcciones realizadas sobre el código.

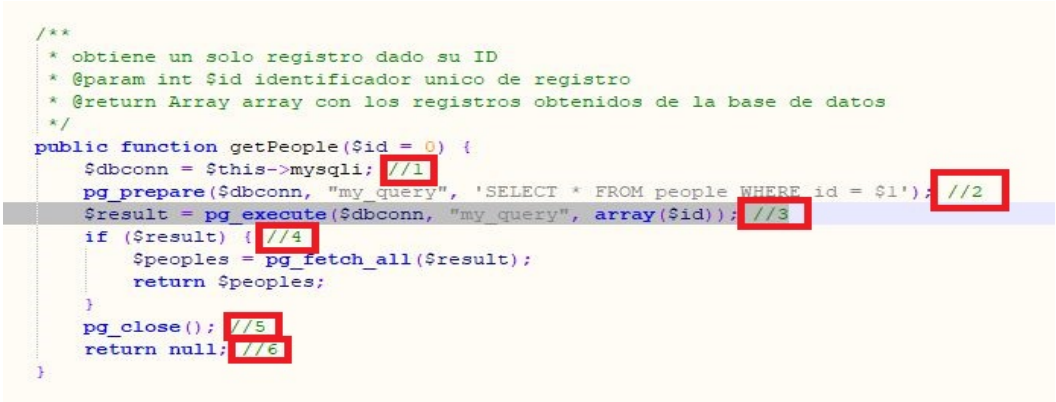


Figura 67: Corrección SQLi método getPeople

- **Método insert ($name = '', $address = ''). -** Se obtiene una instancia de la conexión a la base de datos (1). Se construye la sentencia pre-compilada para el insert (2). Se evalúa la sentencia (3). Se ejecuta la sentencia pre-compilada en el caso de ser válida (4). Se cierra la conexión a la base de datos permitiendo la liberación de recursos (5). Se retorna el resultado de la consulta (6). En la Figura 68, se evidencian las correcciones realizadas sobre el método insert en el código.

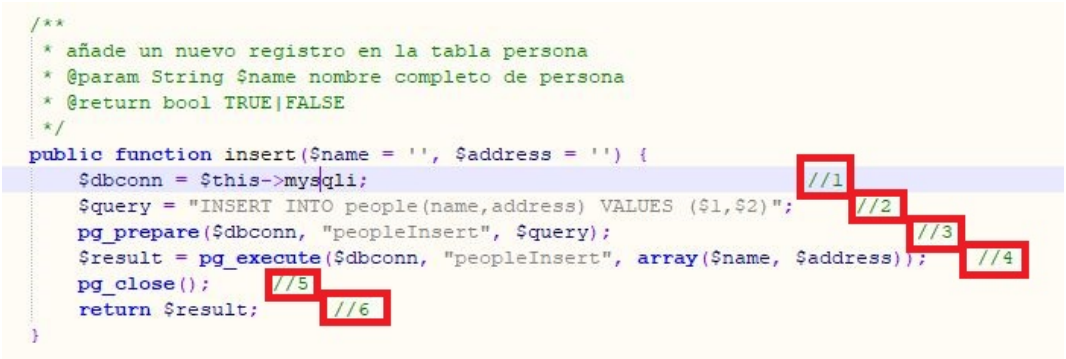


Figura 68: Corrección SQLi método insert

- **Método delete ($id = 0). -** Se obtiene una instancia de la conexión a la base de datos (1). Se construye la sentencia pre-compilada para el delete (2). Se evalúa la sentencia (3). Se ejecuta la sentencia pre-compilada en el caso de ser válida (4). Se cierra la conexión a la base de datos permitiendo la liberación de recursos (5). Se retorna el resultado de la consulta (6). En la Figura 69, se ilustra sobre las correcciones realizadas para el método delete.

```php
/**
 * elimina un registro dado el ID
 * @param int $id Identificador unico de registro
 * @return Bool TRUE|FALSE
 */
public function delete($id = 0) {
    $dbconn = $this->mysqli;                                    //1
    $query = "DELETE FROM people WHERE id=$1";                  //2
    pg_prepare($dbconn, "peopleDelete", $query);                //3
    $result = pg_execute($dbconn, "peopleDelete", array($id));  //4
    pg_close();                                                 //5
    return $result;                                             //6
}
```

Figura 69: Corrección SQLi método delete

- **Método update ($id, $name, $address). -** Se obtiene una instancia de la conexión a la base de datos (1). Se construye la sentencia pre-compilada para el update (2). Se evalúa la sentencia (3). Se ejecuta la sentencia pre-compilada en el caso de ser válida (4). Se cierra la conexión a la base de datos permitiendo la liberación de recursos (5). Se retorna el resultado de la consulta (6). En la Figura 70 se muestra las reformas implementadas sobre el método update para solventar SQLi.

```php
/**
 * Actualiza registro dado su ID
 * @param int $id Description
 */
public function update($id, $name, $address) {
    $dbconn = $this->mysqli;                                                    //1
    $query = "UPDATE people SET name=$1, address=$2 where id=$3";               //2
    pg_prepare($dbconn, "peopleUpdate", $query);                                //3
    $result = pg_execute($dbconn, "peopleUpdate", array($name, $address, $id)); //4
    pg_close();                                                                 //5
    return $result;                                                             //6
}
```

Figura 70: Corrección SQLi método update

- **Método checkID($id). –** Se establece un valor por defecto de false para retorno de respuesta (1). Se obtiene una instancia de la conexión a la base de datos y se construye la sentencia pre-compilada para el select (2). Se establecen parámetros y evalúa la sentencia (3). Se ejecuta la sentencia pre-compilada en el caso de ser válida (4). Se solicita los datos obtenidos de la consulta (5). Se pregunta si hay un solo id del que se encuentra buscando en la colección y se cambia la variable valor a true (6). Se cierra la conexión a la base de datos permitiendo la liberación de recursos (7). Se retorna el resultado de true/false dependiendo si se encontró el id (8). En la Figura 71 se presentan las mejoras realizadas al código para el método checkIS.

```
/**
 * verifica si un ID existe
 * @param int $id Identificador unico de registro
 * @return Bool TRUE|FALSE
 */
public function checkID($id) {
    $valor= false; //1
    $stmt = $this->mysqli->prepare("SELECT * FROM people WHERE ID=?"); //2
    $stmt->bind_param("s", $id); //3
    if ($stmt->execute()) { //4
        $stmt->store_result(); //5
        if ($stmt->num_rows == 1) { //6
            $valor= true;
        }
    }
    pg_close(); //7
    return $valor; //8
}
```

Figura 71: Corrección SQLi método checkID

5.2.1.2. XSS (Almacenado)

✓ **Descripción de la corrección. –** Los modos de evasión de este tipo de ataque se encuentran descritos en el OWASP Cheat Sheet para protección de XSS (Jeff Williams, 2018), de entre los cuales se han destacado los siguientes métodos:

- **Escapar el envío de meta caracteres por JavaScript y HTML**
- **Validar todas las cadenas de entrada usando una lista blanca**
- **Validar los datos que provienen de las salidas**

Para el presente trabajo de servicios Web REST vulnerables, se implementará la solución de validación de los datos que provienen de medios ajenos al procesamiento propio del servicio web REST. En nuestro caso, en el código vulnerable reportado por la herramienta HP Fortify, el programador confía “ciegamente” en los datos que provienen de la base de datos y los usa directamente sin realizar ninguna validación; afortunadamente en el lenguaje de programación PHP existen métodos ya

desarrollados que permiten validar los caracteres especiales que pueden formar parte de cadenas maliciosas, específicamente nos referimos al método **htmlspecialchars ();** este método permite verificar que los datos evaluados no posean meta caracteres que den oportunidad a inclusión de etiquetas y código diferente al que se espera se ejecute en la aplicación en modo normal. Los caracteres que controla y codifica el método son:

- Reemplaza el & por &
- Remplaza la comilla doble por "
- Reemplaza la comilla simple por '/'
- Reemplaza el menor que por <
- Reemplaza el mayor que por >

✓ **Corrección de las vulnerabilidades en los métodos reportados**

La clase PeopleAPI.php es la encargada de alojar los métodos que interactúan con el modelo; en cierto modo podemos decir que es una capa que soporta la lógica de negocio y simula ser un controlador si nos refiriéramos al patrón MVC, por lo que es en esta capa donde se implementarán las soluciones a nivel de código propuestas para mitigar XSS almacenado.

- **Método getPeoples (). –** A diferencia del método original que fue implementado en su primera versión, este método ha unificado el tratamiento de la respuesta en la verificación de un json_encode, obviando el tratamiento de las comillas dobles **(ENT_NOQUOTES)** para evitar fallos en la estructuración del JSON que será devuelto al cliente del servicio web REST. En el análisis de la funcionalidad de código, tenemos que: Luego de verificar si se ha recibido el parámetro id, se pide buscar esa persona en concreto (1). Si el parámetro id no fue enviado por el usuario, se pide buscar todas las personas (2). Se imprime o muestra el resultado de la respuesta obtenida desde la base de datos a manera de un JSON, habiendo previamente sido filtrado su contenido por el método htmlspecialchars, mismo que obviará la encodificación de comilla doble. En caso que no se haya enviado por url los parámetros solicitados o de manera incorrecta, se imprimirá como response 400, que se encuentra asociado a la solicitud incorrecta. En la Figura 72 se puede observar las modificaciones sobre el método getPeoples contra XSS almacenado.

```php
/* Devuelve todos los elementos de la tabla persona */

function getPeoples() {
    if ($_GET['action'] == 'peoples') {
        $response = "";
        $db = new PeopleDB();
        if (isset($_GET['id'])) {//muestra 1 solo registro si es que existiera ID
            $response = $db->getPeople($_GET['id']);  //1
        } else { //muestra todos los registros
            $response = $db->getPeoples();            //2
        }
        echo htmlspecialchars(json_encode($response, JSON_PRETTY_PRINT), ENT_NOQUOTES); //3
    } else {
        response(400);   //4
    }
}
```

Figura 72: Corrección método getPeoples contra XSS almacenado

5.2.1.3. XSS (DOM)

✓ **Descripción de la corrección. –** Los modos de evasión de este tipo de ataque se encuentran descritos en el OWASP Cheat Sheet para protección de XSS (Jeff Williams, 2018), sin embargo, se hace mención a que el evitar todos los defectos de XSS tanto en un aplicación del tipo web como un servicio web REST vulnerable, es prácticamente muy difícil, no obstante, se pueden emplear varios mecanismos para garantizar la seguridad en profundidad, de los cuales se recomiendan los siguientes:

- ❖ **Habilitar la bandera para cookies HTTPOnly**
- ❖ **Implementación de una política de seguridad de contenido (CSP)**
- ❖ **Habilitar la cabecera de protección X-XSS**
- ❖ **Evitar la modificación dinámica del action en formularios.**

Para el presente trabajo de servicios Web REST vulnerables, se implementarán todas las soluciones antes detalladas, sin embargo, las tres primeras serán tratadas más adelante en el apartado de falta de configuración de seguridad de los reportes de OWAP ZAP, en virtud que se trata de configuración de cabeceras que, si bien es cierto pueden ser especificadas en el código fuente de cada script php, se recomienda hacerlo a nivel de servidor para robustecer los mecanismos de seguridad. Con lo referente a evitar la modificación dinámica del action en los formularios, al ser un tema netamente de programación hemos planteado cambiar la lógica de cómo se venía realizando por la que se muestra en la Figura 73 en lo que concierne a JS.

Figura 73: Corrección contra XSS DOM en JavaScript en el script view.php

En la Figura 73 se añade a la variable url el id a buscar en caso que este haya sido diferente de vacío (1). Por AJAX se establece el parámetro URL a donde apuntar (2). Se establece el tipo de dato que procesa (3). Se establece el tipo de operación HTTP a ejecutar para el servicio web REST, en este caso GET (4). Se establece el tipo de dato que espera como respuesta el cliente del servicio web REST (5). Se establece la variable de procesamiento de datos en falso (6). Se define la acción a realizar en caso de éxito (7). Se define la acción a realizar en caso de error (8). En la siguiente figura se muestra cómo ha quedado el cuerpo de HTML para evitar el uso de formularios y las vulnerabilidades que estos ocasionarían.

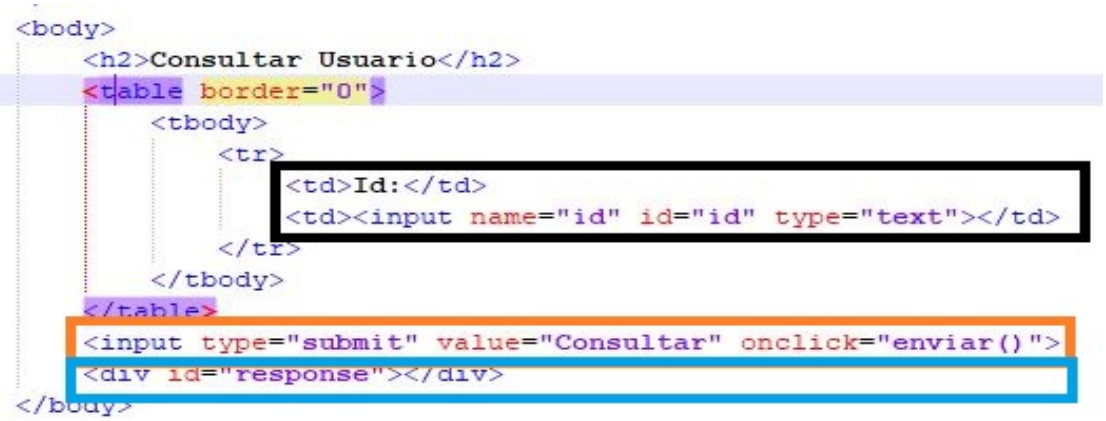


Figura 74: Corrección de XSS DOM en HTML para view.php

En la Figura 74 se dedica un campo del tipo input para recoger el id del recurso a buscar, a continuación, se establece un botón del tipo submit que consumirá el código JavaScript antes explicado, finalmente se reserva una sección denominada response que será la encargada de presentar los resultados de las consultas devueltas por el servicio web REST.

5.2.1.4. JSON Injection

- ✓ **Descripción de la corrección. –** Según el reporte acerca de esta vulnerabilidad emitido por HP Fortify, el problema se produce al momento de convertir un JSON en un objeto, mediante el uso del método json_decode y, que al obtener la información de una entrada que no ha sido validada “php://input”, podría generar conflictos de seguridad. La solución a este tipo de inconvenientes es muy simple y recae en la validación de la entrada misma sobre la cual se va a decodificar un JSON como paso previo para el establecimiento de esos datos a un arreglo que será empleado posteriormente para realizar operaciones de guardado y actualización de registros.

  - ❖ **Corrección del método savePeople ().**

  Para sanear la vulnerabilidad del método obtienen los datos validados desde el común input de php “php://input” estableciendo el parámetro ENT_NOQUOTES para que no codifique las comillas dobles (1), el punto más importante de esta operación es el establecimiento de un patrón de caracteres que serán permitidos; en este caso se establece para la llave name solo números, solo letras, acentos y un máximo de 40 caracteres; la misma regla se encuentra aplicada para el atributo address (2). Se pregunta si los datos que han sido recibidos desde el input de php coinciden con el patrón establecido (3). Sólo si es una cadena verificada con el whitelist previamente establecido, se procede a decodificar el objeto JSON y se lo almacena en la variable “obj” (4). Si no coincide con el patrón de lista blanca, se asigna un arreglo en vacío a la variable “obj” (5). Lo que viene después son simples verificaciones previo a la persistencia del objeto y mensajes para control del tipo de responses para las peticiones HTTP. En la Figura 75 se muestran las correcciones con lo referente a inyección JSON para el método savePeople.

```php
/* Inserta un objeto en la tabla people */

function savePeople() {
    if ($_GET['action'] == 'peoples') {
        //Decodifica un string de JSON
        $datos = htmlspecialchars(file_get_contents('php://input'), ENT_NOQUOTES); //1
        $patron='/^{"name":"+[0-9a-zA-ZñÑáéíóúÁÉÍÓÚ\s]{0,40}","address":"+[0-9a-zA-ZñÑáéíóúÁÉÍÓÚ\-\s]{0,40}"}/'; //2
        if (preg_match($patron, $datos) == 1) { //3
            $obj = json_decode($datos, JSON_UNESCAPED_UNICODE); //4
        } else {
            $obj = array(); //5
        }
        if (empty($obj)) {
            response(422, "error", "Nothing to add. Check json");
        } else if (isset($obj['name'])) {
            $people = new PeopleDB();
            $people->insert($obj['name'], $obj['address']);
            response(200, "success", "new record added");
        } else {
            response(422, "error", "The property is not defined");
        }
    } else {
        response(400);
    }
}
```

Figura 75: Corrección método savePeople contra JSON Injection

❖ Corrección del método updatePeople ().

Para sanear la vulnerabilidad del método obtienen los datos validados desde el común input de php “php://input” estableciendo el parámetro ENT_NOQUOTES para que no codifique las comillas dobles (1), el punto más importante de esta operación es el establecimiento de un patrón de caracteres que serán permitidos; en este caso se establece para la llave name solo números, solo letras, acentos y un máximo de 40 caracteres; la misma regla se encuentra aplicada para el atributo address (2). Se pregunta si los datos que han sido recibidos desde el input de php coinciden con el patrón establecido (3). Sólo si es una cadena verificada con el whitelist previamente establecido, se procede a decodificar el objeto JSON y se lo almacena en la variable “obj” (4). Si no coincide con el patrón de lista blanca, se asigna un arreglo en vacío a la variable “obj” (5). Lo que viene después son simples verificaciones previo a la actualización del objeto y mensajes para control del tipo de responses para las peticiones HTTP. En la Figura 76 se indican las correcciones con lo referente a inyección JSON para el método updatePeople.

```php
/**
 * Actualiza un recurso
 */
function updatePeople() {
    if (isset($_GET['action']) && isset($_GET['id'])) {
        if ($_GET['action'] == 'peoples') {
            $datos = htmlspecialchars(file_get_contents('php://input'), ENT_NOQUOTES); //1
            $patron='/^{"name":"+[0-9a-zA-ZñÑáéíóúÁÉÍÓÚ\s]{0,40}","address":"+[0-9a-zA-ZñÑáéíóúÁÉÍÓÚ\-\s]{0,40}"}/'; //2
            if (preg_match($patron, $datos) == 1) { //3
                $obj = json_decode($datos, JSON_UNESCAPED_UNICODE); //4
            } else {
                $obj = array(); //5
            }
            if (empty($obj)) {
                response(422, "error", "Nothing to add. Check json");
            } else if (isset($obj['name'])) {
                $db = new PeopleDB();
                $db->update($_GET['id'], $obj['name'], $obj['address']);
                response(200, "success", "Record updated");
            } else {
                response(422, "error", "The property is not defined");
            }
            exit;
        }
    }
    response(400);
}
```

Figura 76: Corrección método updatePeople contra JSON Injection

5.2.2. Mejoras informe emitido por Owasp-Zap

5.2.2.1. Sobre-escritura de parámetros HTTP

✓ **Descripción de la corrección. –** Una de las formas más eficientes de evitar este tipo de ataques es estableciendo de una manera estática la acción que va a realizar cada formulario, otro tipo de mitigación de esta vulnerabilidad en virtud de la lógica de programación orientada a los servicios web REST de la actualidad que comúnmente realizan su consumo empleando tecnologías como JQuery, puede ser el evitar el uso de formularios.

Para el presente trabajo de servicios web REST vulnerables, se implementará la solución de omisión de formularios, de este modo la seguridad será tomada en cuenta desde el lado del cliente que consume el servicio web REST y tendrá su afectación en los diferentes scripts que actúan a manera de vista para el usuario.

✓ **Corrección de las vulnerabilidades en los métodos reportados**

La capa del cliente se encarga de consumir el servicio web REST expuesto, en tal virtud se modificará la lógica de programación para los scripts php CRUD (create.php, update.php, delete.php, view.php)

❖ **Script view.php. –** Se encuentra compuesto de una parte de código HTML y JavaScript; para el código HTML observaremos que en el cuerpo ya no se hace uso de un formulario, sino más bien se establece un campo input para

recibir el id de registro a buscar. Se establece un botón submit para ejecutar código JavaScript presente en la misma página. Se declara una sección "response" para mostrar los resultados de la consulta realizada. En la Figura 77 se muestran las correcciones realizadas a nivel de HTML.

```
<body>
    <h2>Consultar Usuario</h2>
    <table border="0">
        <tbody>
            <tr>
                <td>Id:</td>
                <td><input name="id" id="id" type="text"></td>
            </tr>
        </tbody>
    </table>
    <input type="submit" value="Consultar" onclick="enviar()">

    <div id="response"></div>
</body>
```

Figura 77: Corrección de sobre-escritura de parámetros HTTP script view.php

Con lo referente a la parte del código JavaScript queda tal y como fue mostrado para la prevención de ataques XSS DOM en el apartado anterior (5.2.13).

- **Script create.php. –** Se encuentra compuesto de una parte de código HTML y JavaScript. Para el código HTML observaremos en la Figura 78 que para el cuerpo sin usar un formulario se han añadido dos etiquetas input para recolectar el nombre y la dirección a ser tratados. Se declara un botón submit para ejecutar código JavaScript presente en la misma página.

```
<body>
    <h2>Crear Usuario</h2>
    <table border="0">
        <tbody>
            <tr>
                <td>Name:</td>
                <td> <input name="name" id="name" type="text"></td>
            </tr>
            <tr>
                <td>Address:</td>
                <td> <textarea cols="25" name="address" id="address"></textarea></td>
            </tr>
        </tbody>
    </table>
    <input type="submit" value="Crear" onclick="enviar()">
</body>
```

Figura 78: Corrección de sobre-escritura de parámetros HTTP script create.php (HTML)

- **Código JavaScript corregido para la operación de creación (create.php). –** Se define básicamente el tipo de operación HTTP, en este caso es POST. En la variable data se envía la información en formato JSON recolectada desde HTML a través de JavaScript. Y se dará el respectivo tratamiento en el caso de éxito como se muestra en la Figura 79.

```
<script>
    function enviar() {
        var url = "http://172.16.20.151/Rest/peoples";
        var name = document.getElementById('name').value;
        var address = document.getElementById('address').value;
        $.ajax({
            url: url,
            dataType: 'json',
            type: 'POST',
            contentType: 'application/json',
            data: JSON.stringify({"name": name, "address": address}),
            processData: false,
            success: function (data, textStatus, jQxhr) {
                $('#response pre').html(JSON.stringify(data));
            },
            error: function (jqXhr, textStatus, errorThrown) {
                console.log(errorThrown);
            }
        });
    }
</script>
```

Figura 79: Corrección de sobre-escritura de parámetros HTTP script create.php (JavaScript)

- **Script update.php. –** Se encuentra compuesto de una parte de código HTML y JavaScript. Para el código HTML ilustrado en la Figura 80, observaremos que en el cuerpo sin usar un formulario se han añadido tres etiquetas input para recolectar el nombre, la dirección y por último el id del registro a ser editado. Se define un botón submit para ejecutar código JavaScript presente en la misma página.

```
<body>
    <h2>Editar Usuario</h2>
    <table border="0">
        <tbody>
            <tr>
                <td>Id:</td>
                <td> <input name="id" id="id" type="text"></td>
            </tr>
            <tr>
                <td>Name:</td>
                <td> <input name="name" id="name" type="text"></td>
            </tr>
            <tr>
                <td>Address:</td>
                <td> <textarea cols="25" name="address" id="address"></textarea></td>
            </tr>
        </tbody>
    </table>
    <input type="submit" value="Editar" onclick="enviar()">
</body>
```

Figura 80: Corrección de sobre-escritura de parámetros HTTP script update.php (HTML)

- **Código JavaScript corregido para la operación de actualización (update.php). –** Se define básicamente el tipo de operación HTTP, en este caso es PUT. En la variable data se envía la información en formato JSON recolectada desde HTML a través de JavaScript. El id relacionado al registro a actualizar se pasa por URL y se da el respectivo tratamiento que se encuentra establecido en el método success mediante JQuery como se expone en la Figura 81.

```
<script>
    function enviar() {
        var url = "http://172.16.20.151/Rest/peoples";
        var name = document.getElementById('name').value;
        var address = document.getElementById('address').value;
        var id = document.getElementById('id').value;
        url = url + "/" + id;
        $.ajax({
            url: url,
            dataType: 'json',
            type: 'put',
            contentType: 'application/json',
            data: JSON.stringify({"name": name, "address": address}),
            processData: false,
            success: function (data, textStatus, jQxhr) {
                $('#response pre').html(JSON.stringify(data));
            },
            error: function (jqXhr, textStatus, errorThrown) {
                console.log(errorThrown);
            }
        });
    }
</script>
```

Figura 81: Corrección de sobre-escritura de parámetros HTTP script update.php (JavaScript)

- **Script delete.php. -** Se encuentra compuesto de una parte de código HTML y JavaScript; para el código HTML observaremos que, en el cuerpo, sin usar un formulario se ha hecho uso de una etiqueta input para recolectar el id del registro a ser eliminado. Se define un botón submit para ejecutar código JavaScript presente en la misma página como se indica en la Figura 82.

```
<body>
    <h2>Eliminar Usuario</h2>
    <table border="0">
        <tbody>
            <tr>
                <td>Id:</td>
                <td><input name="id" id="id" type="text"></td>
            </tr>
        </tbody>
    </table>
    <input type="submit" value="Eliminar" onclick="enviar()">
</body>
```

Figura 82: Corrección de sobre-escritura de parámetros HTTP script delete.php (HTML)

- **Código JavaScript corregido para la operación de actualización (delete.php). –** Se define el tipo de operación HTTP, en este caso es DELETE. No existe la variable data dado que el id del registro a eliminar es pasado por URL. Finalmente se da el respectivo tratamiento definido en el método success apoyándonos en JQuery como se describe en la Figura 83.

```
<script>
    function enviar() {
        var url = "http://172.16.20.151/Rest/peoples";
        var id = document.getElementById('id').value;
        url=url+"/"+id;
        $.ajax({
            url: url,
            dataType: 'json',
            type: 'DELETE',
            contentType: 'application/json',
            processData: false,
            success: function (textStatus, jQxhr) {
                $('#response pre').html(JSON.stringify());
            },
            error: function (jqXhr, textStatus, errorThrown) {
                console.log(errorThrown);
            }
        });
    }
</script>
```

Figura 83: Corrección de sobre-escritura de parámetros HTTP script delete.php (JavaScript)

5.2.2.2. Ausencia de tokens anti CSRF

- ✓ **Descripción de la corrección. –** Según el OWASP Cheat Sheet para mitigación de ataques del tipo CSRF (Dave Wichers, 2018), se pueden emplear diversos métodos:

  - ❖ **Utilizar tokens sincronizados (Necesario el establecimiento de sesión). –** Se propone el uso de un campo del tipo hidden el cual será añadido en cada formulario y será enviado en conjunto a cada petición para ser validado. Para facilitar este tipo de implementaciones ya existen librerías desarrolladas para lenguajes específicos, para nuestro caso en particular, si hablamos de PHP tenemos el **proyecto "CSRFProtector Project".**

  - ❖ **Defensa de doble petición usando cookies. –** Se envía el valor aleatorio generado criptográficamente en cada petición y se almacena temporalmente en una cookie. El éxito de este tipo de mecanismo se basa en la verificación del valor aleatorio entre la cookie y la petición; finalmente ser debe validar a nivel de cabeceras el origen y el destino de las peticiones que son realizadas al servidor.

- **Patrón de tokens cifrados. -** Fundamentalmente se basa en el hecho de generación de tokens que son cifrados del lado del servidor y entregados al cliente para realizar las respectivas request luego de haberse autenticado. En el token cifrado se incluye un atributo nonce, de tal modo que los datos cifrados por el servidor tengan una fecha de caducidad. Cualquier intento de descifrado fallido será considerado como un posible ataque y se negará la atención a la request realizada. Sólo el servidor posee la clave privada para cifrar y generar los token.

- **Establecimiento de Cabeceras personalizadas. –** Este tipo de mecanismo se basa en la implementación de la política que verifica el mismo origen en peticiones, se recomienda la implementación de cabeceras en las peticiones HTTP realizadas al servidor con el afán de verificar y validar que las transacciones solicitadas al servidor, provienen del lugar esperado y son ejecutadas por el usuario deseado. Los tipos de cabeceras a implementar, permitirán dar seguimiento del origen del objetivo, la referencia del lugar de procedencia, determinación del origen del objetivo cuando se encuentra tras de un proxy, etc.

Para este trabajo se ha optado por implementar la solución de establecimiento de cabeceras personalizadas ya que, por la naturaleza de la programación modificada para los scripts del lado del cliente mediante la eliminación de formularios, ya no se presentaría esta vulnerabilidad, sin embargo, para mayor aseguramiento de las transacciones y operaciones permitidas desde el cliente al servidor se detallará el uso de cabeceras en el apartado de **falta de configuración de seguridad.**

5.2.2.3. Cross-Domain JavaScript Source File Inclusion

- **Descripción de la corrección. –** Una de las maneras simples de evitar este tipo de vulnerabilidad es mediante la descarga de los scripts a los que se hace referencia en nuestro ambiente local, luego de esto se debería realizar una auditoría de código de lo que se ha bajado con la finalidad de evitar backdooring y posibles vulnerabilidades. De cierto modo esto nos permitiría “confiar” en el script que se encuentra en nuestro servidor y de igual modo de la funcionalidad que este script permitiría.

Para usos de estudio en el presente trabajo se ha optado por descargar el fichero al que se hace referencia desde la etiqueta script. Concretamente es el script que permite el uso de jQuery para el consumo de servicios web REST.

- ✓ **Corrección de las vulnerabilidades en los scripts PHP reportados**

  Los scripts php que emplean la librería de jQuery son los asociados a la vista del lado del cliente que permiten CRUD, básicamente son: (view.php, update.php, delete.php, create.php). Para los cuatro ficheros CRUD. Se debería descargar "jquery.min.js" en local y hacerlo referencia en las etiquetas script como se muestra a continuación en la Figura 84.

```
<html>
    <script src="jquery.min.js"></script>
    <script>
        function enviar() {
            var url = "http://172.16.20.151/Rest/peoples";
            var name = document.getElementById('name').value;
            var address = document.getElementById('address').value;
            var id = document.getElementById('id').value;
            url = url + "/" + id;
            $.ajax({
                url: url,
                dataType: 'json',
                type: 'put',
                contentType: 'application/json',
                data: JSON.stringify({"name": name, "address": address}),
                processData: false,
                success: function (data, textStatus, jQxhr) {
                    $('#response pre').html(JSON.stringify(data));
                },
                error: function (jqXhr, textStatus, errorThrown) {
                    console.log(errorThrown);
                }
            });
        }
    </script>
```

Figura 84: Corrección JavaScript Source File Inclusion

5.2.2.4. Falta de configuración adecuada

- ✓ **Descripción de la corrección. –** La lista de cabeceras que no debemos olvidar tomar en cuenta de cara a poner en producción nuestros desarrollos informáticos, se encuentran detalladas en el Cheat Sheet de OWASP (owasp.org, 2014), sin embargo, cabe mencionar que no son las únicas que se implementarán en este apartado.

- **X-Frame-Options: SAMEORIGIN**
- **X-XSS-Protection: 1; mode=block**
- **X-Content-Type-Options: nosniff**
- **Content-Type: text/html; charset=utf-8**

✓ **Corrección de las vulnerabilidades por falta de configuración reportada. –** Si bien el uso de las cabeceras puede ser especificado en cada script PHP, se recomienda por OWASP el establecimiento de una política por defecto implementada del lado del servidor, con la finalidad de robustecer los mecanismos de seguridad; prácticamente serán añadidas las cabeceras en el fichero de configuración de httpd. Para usos del presente trabajo, se tomarán en cuenta las vulnerabilidades reportadas por la herramienta OWASP ZAP y de igual modo las cabeceras detalladas anteriormente.

- **X-Frame-Options Header Not set. –** En el fichero de configuración de httpd, en nuestro caso como usamos una distribución fedora basada en red hat, se encontrará en el directorio **"/etc/httpd/conf/httpd.conf"**; añadiremos el siguiente contenido **"Header set X-Frame-Options SAMEORIGIN"** para asegurar que las peticiones vengan del mismo origen y evitar la inclusión de scripts maliciosos, consecuentemente se evitará **"ClickJacking".**

- **Encabezado (CSP) no establecido. -** En el fichero de configuración de httpd, en nuestro caso como usamos una distribución fedora basada en red hat, se encontrará en el directorio **"/etc/httpd/conf/httpd.conf"**; añadiremos el siguiente contenido **(Header always set Content-Security-Policy "default-src http: data: 'unsafe-inline' 'unsafe-eval'")** para asegurar que no sea embebida la funcionalidad de nuestro cliente REST como parte de una página ajena a nuestro dominio.

- **Divulgación de información de versión del servidor. –** En el fichero de configuración de httpd, en nuestro caso como usamos una distribución fedora basada en red hat, se encontrará en el directorio **"/etc/httpd/conf/httpd.conf"**; añadiremos los siguientes contenidos: **"ServerSignature Off"," ServerTokens Prod", (Header unset "X-Powered-By")** con el afán de ocultar la información de la versión Apache en las peticiones, esto dificultaría aún más las posibilidades del atacante para conseguir información y desarrollar ataques más sofisticados.

- **Web Browser XSS Protection Not Enabled. –** En el fichero de configuración de httpd, en nuestro caso como usamos una distribución fedora basada en red hat, se encontrará en el directorio **"/etc/httpd/conf/httpd.conf"**; añadiremos el siguiente contenido: **(Header set X-XSS-Protection "1; mode=block")** mismo que permitirá habilitar el mecanismo de defensa integrado en los navegadores para contra ataques XSS, además especificamos en el modo bloc la directiva para bloquear cualquier intento de XSS.

- **X-Content-Type-Options Header Missing. –** En el fichero de configuración de httpd, en nuestro caso como usamos una distribución fedora basada en red hat, se encontrará en el directorio **"/etc/httpd/conf/httpd.conf"**; añadiremos el siguiente contenido: **"Header set X-Content-Type-Options nosniff"** mismo que permitirá deshabilitar la opción de sniffing para los navegadores web referentes al contenido MIME.

- **Contenido almacenable y almacenable en caché. –** En el fichero de configuración de httpd, en nuestro caso como usamos una distribución fedora basada en red hat, se encontrará en el directorio **"/etc/httpd/conf/httpd.conf"**; añadiremos el siguiente contenido: **"Header set Pragma "no-cache"", "Header set Expires "0""**. De esta manera se desactivará la opción de guardar en caché cierta información que puede ser sensible, cabe dar a notar que el desarrollador puede habilitar estas cabeceras desde sus scripts PHP en el caso que desee guardar en caché alguna información que considere pueda mejorar el rendimiento del servicio WEB y no afecte la seguridad del mismo.

A continuación, se muestra en la Figura 85 las cabeceras de seguridad añadidas para mitigar los inconvenientes reportados por la herramienta ZAP de OWASP:

```
ServerRoot "/etc/httpd"

#cabeceras de seguridad

Header set X-Frame-Options SAMEORIGIN

Header always set Content-Security-Policy "default-src http: data: 'unsafe-inline' 'u
nsafe-eval'"

ServerSignature Off
ServerTokens Prod
Header unset "X-Powered-By"

Header set X-XSS-Protection "1; mode=block"

Header set X-Content-Type-Options nosniff

Header set Pragma "no-cache"
Header set Expires "0"
```

Figura 85: Cabeceras de seguridad para solventar problemas de falta de configuración 1.

5.2.3. Mejoras sobre el informe emitido por ACUNETIX

- ✓ **Descripción de la corrección. –** Las vulnerabilidades que han sido reportadas por ACUNETIX, son mitigables mediante la implementación de cabeceras de seguridad. A continuación, se detallan las configuraciones realizadas para mitigar las vulnerabilidades reportadas.

5.2.3.1. Falta de X-Frame-Options header. – Como se observó en el informe emitido por OWASP ZAP, se ha implementado la solución mediante la configuración de la cabecera establecida para las observaciones detalladas en su momento. Para más información verificar el apartado anterior.

5.2.3.2. Método options se encuentra habilitado. – En el fichero de configuración de httpd, en nuestro caso como usamos una distribución fedora basada en red hat, se encontrará en el directorio **"/etc/httpd/conf/httpd.conf";** añadiremos los siguientes contenidos: **"RewriteEngine On", "RewriteCond %{REQUEST_METHOD} !^(GET|POST|HEAD|PUT|DELETE)", "RewriteRule .* - [R=405,L]";** esto permitirá controlar los métodos que son expuestos por el servidor a los deseados; en las reglas antes mencionadas se ha establecido específicamente el uso de los métodos: **POST, GET, HEAD, PUT, DELETE.**

5.2.3.3. Método trace se encuentra habilitado. – Para deshabilitar este método, básicamente accederemos al archivo de configuración de apache, para nuestro caso como usamos una distribución fedora basada en red hat, se encontrará en el directorio **"/etc/httpd/conf/httpd.conf**"; añadiremos el siguiente contenido: "TraceEnable off". Finalmente se debe reiniciar el servicio de httpd.

A continuación, en la figura 86 se indica la configuración de todas las reglas y cabeceras que han sido implementadas para mejorar la seguridad de los servicios web REST en el archivo de configuración de httpd.

```
ServerRoot "/etc/httpd"

#cabeceras de seguridad

Header set X-Frame-Options SAMEORIGIN


Header always set Content-Security-Policy "default-src http: data: 'unsafe-inline' 'unsafe-eval'"


ServerSignature Off
ServerTokens Prod
Header unset "X-Powered-By"


Header set X-XSS-Protection "1; mode=block"


Header set X-Content-Type-Options nosniff


Header set Pragma "no-cache"
Header set Expires "0"




RewriteEngine On
RewriteCond %{REQUEST_METHOD} !^(GET|POST|HEAD|PUT|DELETE)
RewriteRule .* - [R=405,L]

TraceEnable off

#
```

Figura 86: Figura 85: Cabeceras de seguridad para solventar problemas de falta de configuración 2.

6. Análisis de resultados y breve guía de seguridad para el desarrollo de servicios REST

6.1. Análisis de resultados

Los resultados del experimento que se ha realizado en el presente TFM, evidencian la falta de formación en materia de seguridad en los desarrolladores para la implementación de servicios web REST. En la mayor parte de sitos que han sido visitados del apartado de referencias, sólo se da una breve explicación de la manera en la cual se pueden implementar los servicios web REST; desafortunadamente ninguno de ellos hace una llamada de atención sobre posibles fallos de seguridad que pueden ser introducidos en la etapa de codificación. A continuación, se muestran las diferentes tablas que resumen las vulnerabilidades encontradas por cada una de las herramientas de análisis de código. En la cabecera se describe el contenido de cada una de las columnas y los diferentes colores se utilizan para diferenciar los tipos de riesgos encontrados

Id	Nombre	Descripción	Objeto comprometido
AW-01	XSS DOM	Se permite la modificación de la acción por JS	✓ ../cliente/view.php
AW-02	XSS STORED (PeopleAPI.php)	Falta de control en las salidas que van a ser transmitidas al cliente	✓ getPeoples()
AW-03	JSON Injection (PeopleAPI.php)	Falta de control de las fuentes que se toman en cuenta para la formación de JSON	✓ savePeople () ✓ updatePeople ()
AW-04	SQL Injection (PeopleDB.php)	Se toman parámetros sin validar en consultas creadas dinámicamente.	✓ getPeople ✓ insert ✓ delete ✓ update

Tabla 1: Vulnerabilidades a través de Audit Workbench

Id	Nombre	Descripción	Objeto comprometido
OZ-01	Cabecera X-Frame-Options no configurada	La ausencia de esta cabecera hace vulnerable al servicio web ante ataques del tipo ClickJacking	✓ ../cliente
OZ-02	Cabecera contra CSP no configurada	Esta cabecera protege al servicio web REST contra ataques XSS, inyección de datos, etc.	✓ ../cliente
OZ-03	Divulgación de Información del servidor	El servidor divulga información de versión mediante el campo de encabezado de respuesta HTTP ""Server"	✓ ../cliente
OZ-04	Cabecera Web Browser XSS Protection no configurada	La protección contra XSS para el navegador se encuentra deshabilitada	✓ ../cliente/
OZ-05	Cabecera X-Content-Type-Options no configurada	No se encuentra habilitad la opción anti sniffing de paquetes para el browser	✓ ../cliente/create.php ✓ ../cliente/update.php ✓ ../cliente/view.php ✓ ../cliente/delete.php
OZ-06	Contenido almacenable y almacenable en caché	Falta de controles de caché para evitar obtención de información sensible	✓ ../cliente/create.php ✓ ../cliente/update.php ✓ ../cliente/view.php ✓ ../cliente/delete.php
OZ-07	Contenido no almacenable	Establecer contenidos de respuesta para almacenar en caché si no poseen información comprometedora.	✓ ../Rest/

Tabla 2: Vulnerabilidades reportadas a través de OWASP-ZAP

Id	Nombre	Descripción	Objeto comprometido
AC-01	ClickJacking: X-Frame-Options header missing	Se puede embeber la funcionalidad del servicio en un formulario ajeno al dominio del sitio.	✓ Servidor apache
AC-02	Método options habilitado	Se está mostrando la lista de métodos que oferta el servicio web REST	✓ Servidor apache
AC-03	Método TRACE se encuentra habilitado	Permite vulnerabilidades del tipo Cross Domain	✓ Servidor apache

***Tabla 3*: Vulnerabilidades reportadas a través de ACUNETIX**

- **Indicadores de riesgo:**

Cada una de las vulnerabilidades reportadas por las herramientas de auditoría de código y resumidas en la Tabla 1, Tabla 2, y Tabla 3, cuentan con un color distintivo de acuerdo a su nivel de riesgo. El nivel de riesgo ha sido establecido conforme el propio ranking que cada herramienta indica. En la Tabla 4 se detallan los colores asociados al riesgo:

***Tabla 4* Indicadores de riesgo**

De las 14 vulnerabilidades reportadas por las herramientas OWASP ZAP, ACUNETIX, Audit Workbench; se han detectado cuatro vulnerabilidades críticas, dos vulnerabilidades medias y ocho vulnerabilidades bajas. A lo largo del capítulo cuatro se han mitigado de diferente forma las vulnerabilidades; sin embargo, en el informe de OWASP ZAP nos muestra que se poseen aún vulnerabilidades que, luego de hacer una prueba manual se comprobaron cómo falsos positivos; de igual manera la herramienta Audit Workbench de Fortify nos deja un falso positivo para JSON Injection luego de su validación. Por otra

parte, las correcciones sobre lo reportado por herramienta ACUNETIX no muestran ya vulnerabilidades.

A continuación, se presentan las Figuras 87, 88, y 89 sobre los escaneos finales luego de las correcciones a nivel de código y servidor.

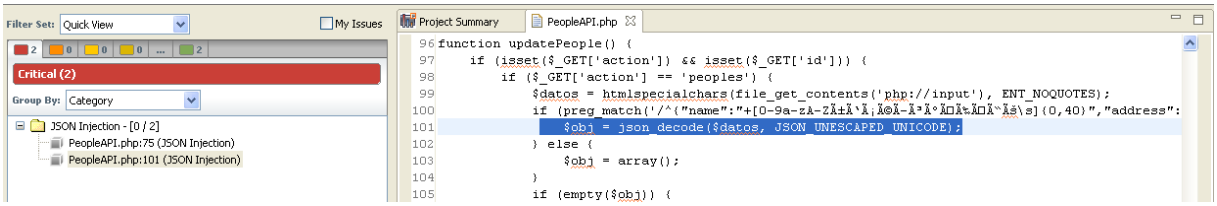


Figura 87: Reporte Audit Workbench luego de mitigar las vulnerabilidades reportadas

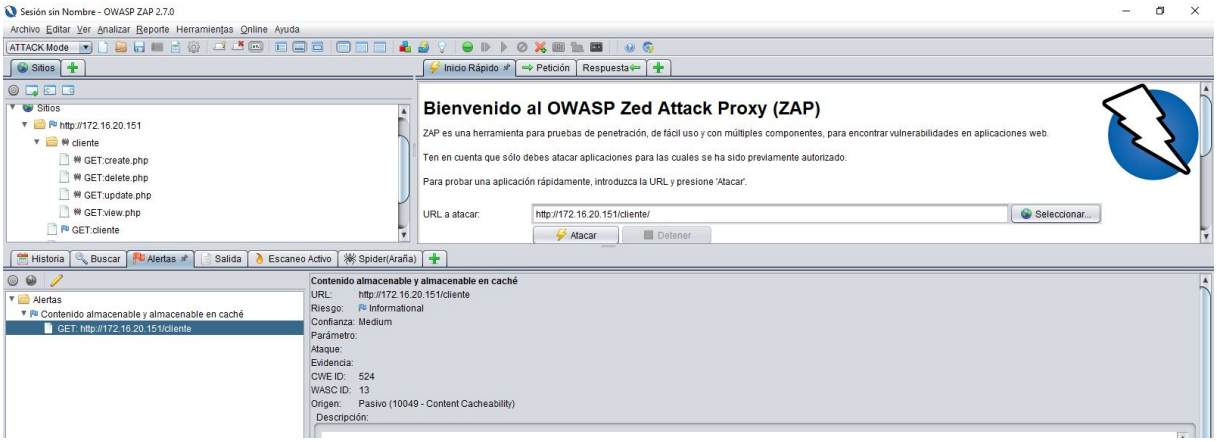


Figura 88: Reporte Owasp ZAP luego de mitigar las vulnerabilidades reportadas

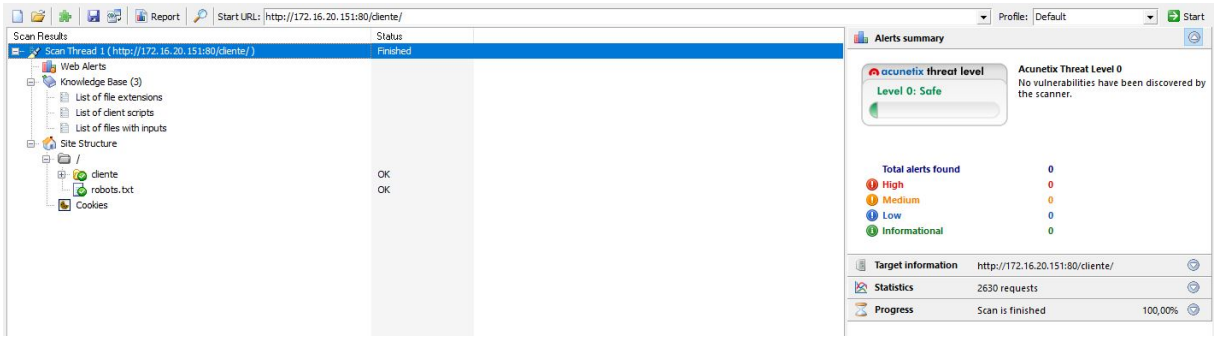


Figura 89: Reporte ACUNETIX luego de mitigar las vulnerabilidades reportadas

6.2. Breve guía de seguridad para el desarrollo de servicios REST

Acorde a la experiencia adquirida durante el proceso de desarrollo de servicios web REST y la mitigación de vulnerabilidades reportadas por las herramientas de análisis de código; se pueden recomendar las siguientes prácticas agrupadas en las categorías de codificación y configuración, mismas que fomentarán la codificación de soluciones seguras y la realización de configuraciones previas a poner en producción los desarrollos implementados.

6.2.1. Buenas prácticas a nivel de codificación

La mayoría de inconvenientes presentados en el desarrollo de soluciones informáticas y por ende en la codificación de servicios web REST, se debe a la validación y tratamiento de datos precaria con respecto a las entradas y salidas con las que interactúa el servicio web. A continuación, se muestran los mecanismos que se han utilizado en este proyecto para mitigación y tratamiento de vulnerabilidades del tipo validación de entradas y salidas.

- **Evitar SQLi**

Una de las maneras más eficientes para evitar SQLi se basa en la implementación de consultas preparadas o tratadas y la implementación de procedimientos almacenados en base de datos para controlar los parámetros que son enviados desde la aplicación a la capa de persistencia de base de datos (BDD). A continuación, en la Figura 90, se presenta un ejemplo del uso de **prepared statement** para la inserción de un registro a la base de datos.

```php
/**
 * añade un nuevo registro en la tabla persona
 * @param String $name nombre completo de persona
 * @return bool TRUE|FALSE
 */
public function insert($name = '', $address = '') {
    $dbconn = $this->mysqli;
    $query = "INSERT INTO people(name,address) VALUES ($1,$2)";
    pg_prepare($dbconn, "peopleInsert", $query);
    $result = pg_execute($dbconn, "peopleInsert", array($name, $address));
    pg_close();
    return $result;
}
```

Figura 90: Método para insertar usando prepared statements

❖ **Evitar XSS (Almacenado)**

Siempre que se obtengan datos de una capa de persistencia se deben dar tratamiento previo a su uso. Una de las maneras más eficaces es la codificación de caracteres especiales previo a su procesamiento. A continuación, en la Figura 91, se indica un ejemplo para la obtención de datos relacionados a un registro con el método **htmlspecialchars**.

```
/* Devuelve todos los elementos de la tabla persona */

function getPeoples() {
    if ($_GET['action'] == 'peoples') {
        $response = "";
        $db = new PeopleDB();
        if (isset($_GET['id'])) {//muestra 1 solo registro si es que existiera ID
            $response = $db->getPeople($_GET['id']);
        } else { //muestra todos los registros
            $response = $db->getPeoples();
        }
        echo htmlspecialchars(json_encode($response, JSON_PRETTY_PRINT), ENT_NOQUOTES);
    } else {
        response(400);
    }
}
```

Figura 91: htmlspecialchars para evitar XSS almacenado

❖ **Evitar XSS (DOM), sobre-escritura de parámetros HTTP y ausencia de tokens anti CSRF**

Se debe siempre definir el **action** a realizar en los formularios, debido a uso de JavaScript se acostumbra a modificarlos con lenguaje DOM. Otro método más actual usado por los desarrolladores es el empleo de alternativas web 2.0 como jQuery que posee AJAX para evitar el uso de envío de datos a través de formularios en el caso de servicios web REST; tomando en cuenta las seguridades necesarias para su uso; al no tener formularios se evita la ausencia de tokens del tipo hidden para controlar CSRF. A continuación, en las Figuras 92 y 93, se muestra un ejemplo del uso eficiente de AJAX presente en jQuery para eliminar un registro.

```
<head>
    <title>Eliminar Usuario</title>
</head>
<body>
    <h2>Eliminar Usuario</h2>
    <table border="0">
        <tbody>
            <tr>
                <td>Id:</td>
                <td><input name="id" id="id" type="text"></td>
            </tr>
        </tbody>
    </table>
    <input type="submit" value="Eliminar" onclick="enviar()">
</body>
```

Figura 92: Código HTML sin uso de formularios

```
<script src="jquery.min.js"></script>
<script>
    function enviar() {
        var url = "http://172.16.20.151/Rest/peoples";
        var id = document.getElementById('id').value;
        url=url+"/"+id;
        $.ajax({
            url: url,
            dataType: 'json',
            type: 'DELETE',
            contentType: 'application/json',
            processData: false,
            success: function (textStatus, jQxhr) {
                $('#response pre').html(JSON.stringify());
            },
            error: function (jqXhr, textStatus, errorThrown) {
                console.log(errorThrown);
            }
        });
    }
</script>
```

Figura 93: Código JavaScript sin uso de formularios

❖ Evitar JSON Injection

Una de las maneras más eficientes de evitar JSON Injection es asegurando la decodificación directa de JSON a partir de fuentes que pueden contener datos no validados, para esto se puede implementar una lista blanca de caracteres que pueden ser permitidos para la población de JSON, mismo que será enviado posteriormente para su procesamiento; en el lenguaje PHP la entrada "**php://input**" es considerada insegura; para eso se valida con el método **htmlspecialchars** que no contenga etiquetas maliciosas y posterior a esto se valida con una lista blanca su contenido. A continuación, en la Figura 94, se muestra el tratamiento de "php://input" para evitar JSON Injection al persistir un registro en la base de datos.

```php
/* Inserta un objeto en la tabla people */

function savePeople() {
    if ($_GET['action'] == 'peoples') {
        //Decodifica un string de JSON
        $datos = htmlspecialchars(file_get_contents('php://input'), ENT_NOQUOTES);
        $patron='/^{"name":"+[0-9a-zA-ZñÑáéíóúÁÉÍÓÚ\s]{0,40}","address":"+[0-9a-zA-ZñÑáéíóúÁÉÍÓÚ\-\s]{0,40}"}/';
        if (preg_match($patron, $datos) == 1) {
            $obj = json_decode($datos, JSON_UNESCAPED_UNICODE);
        } else {
            $obj = array();
        }
        if (empty($obj)) {
            response(422, "error", "Nothing to add. Check json");
        } else if (isset($obj['name'])) {
            $people = new PeopleDB();
            $people->insert($obj['name'], $obj['address']);
            response(200, "success", "new record added");
        } else {
            response(422, "error", "The property is not defined");
        }
    } else {
        response(400);
    }
}
```

Figura 94: Evasión JSON Injection mediante htmlspecialchars y validación de patrones

❖ Cross Domain JavaScript source file inclusion

Muchos programadores hacen referencia a scripts directamente empleados vía URL, lo que se considera una mala práctica de seguridad en virtud de que puede ser adulterado el dominio al cual se apunta, e incluso el contenido del script para realizar acciones maliciosas. La mejor forma de mitigar este tipo de inconvenientes es **NO** ejecutar código desde fuentes que no sean confiables. Se debe descargar el código que se va a usar mientras sea posible en el servidor local para referenciarlo y consumirlo a nivel local. A continuación, en la Figura 95, se detalla un consumo eficiente de **jsquerymin.js** para ser incluidos en scripts PHP.

```html
<script src="jquery.min.js"></script>
<script>
    function enviar() {
        var url = "http://172.16.20.151/Rest/peoples";
        var id = document.getElementById('id').value;
        if (id != "")
            url = url + "/" + document.getElementById('id').value;
        $.ajax({
            url: url,
            dataType: 'json',
            type: 'GET',
            contentType: 'application/json',
            processData: false,
            success: function (textStatus, jQxhr) {
                $('#response').html(JSON.stringify(textStatus));
                },
            error: function (jqXhr, textStatus, errorThrown) {
                console.log(errorThrown);
            }
        });
    }

</script>
```

Figura 95: Referenciación JavaScript para uso local

6.2.2. Buenas prácticas a nivel de configuración

Una de las características importantes de la seguridad informática es que debe ser en profundidad; generalmente esto implica la observancia de muchos factores de seguridad que garanticen la protección de las propiedades de la información. Sin embargo, el desarrollador frecuentemente presta poca atención al momento de verificar si existen las configuraciones adecuadas en su código con respecto al manejo de cabeceras que impidan la divulgación de información delicada y que pueda comprometer la disponibilidad de su servicio web desarrollado. A continuación se muestra una breve manera de configurar las cabeceras a nivel de servidor.

- **Cabecera X-Frame-Options**

Se recomienda habilitar la cabecera X-Frame-Options para evitar ataques del tipo ClickJacking en su afán de enmarcar o embeber la funcionalidad del servicio web REST en una página ajena a nuestro dominio. A continuación, en la Figura 96, se indica la configuración de la cabecera para su uso en el mismo origen

```
Header set X-Frame-Options SAMEORIGIN
```

Figura 96: configuración Cabecera X-Frame-Options

- **Cabecera CSP**

Se debe configurar esta cabecera para evitar ataques de XSS y de inyección de datos evitando la ejecución de scripts o recursos que se consideren no permitidos en el lado del cliente; esto ocurre acto seguido el visitante consume nuestro servicio web. A continuación, en la Figura 97, se muestra la configuración recomendada para CSP.

```
Header always set Content-Security-Policy "default-src http: data: 'unsafe-inline' 'unsafe-eval'"
```

Figura 97: Configuración de Cabecera CSP

- **Evitar divulgación de versión del servidor**

La revelación de información con respecto a la versión de servidores es un aspecto delicado ya que el atacante puede valerse de esto para elaborar ataques más eficientes y elaborados. Se debe configurar las siguientes políticas, ver Figura 98, para evitar la divulgación de información sensible.

```
ServerSignature Off
ServerTokens Prod
Header unset "X-Powered-By"
```

Figura 98: Evitar mostrar la versión del servicio

- **Cabecera web Browser XSS Protection**

El navegador de los clientes que consumen el servicio web tiene un mecanismo de protección contra XSS, se debe habilitar esta protección mediante la configuración de la cabecera XSS Protection como se muestra a continuación en la Figura 99.

```
Header set X-XSS-Protection "1; mode=block"
```

Figura 99: Protección XSS desde el navegador web

- **Cabecera X-Content-Type-Options**

Los navegadores poseen la característica para hacer un sniffing del tipo de MIME que se envían en las peticiones asociadas al tipo de archivo a servir, esto se vuelve peligroso ya que un atacante malicioso puede enviar código JS inyectado y el navegador en su afán de ayudar a la interpretación o verificación de MIME puede comprometer la seguridad del sitio. En la Figura 100 se presenta la configuración para evitar el sniffing MIME.

```
Header set X-Content-Type-Options nosniff
```

Figura 100: Evitar sniffing de contenido MIME

- **Gestionar Contenido almacenable y no almacenable en caché**

Es importante definir qué tipo de scripts e información se pueden almacenar en caché para agilizar la respuesta del servicio web REST sin comprometer la seguridad, esto en virtud que si no se lleva un control eficiente de lo que se va a almacenar en caché, un atacante puede robar información importante como id de sesión, etc. La configuración de esta cabecera se muestra a continuación en la Figura 101.

```
Header set Pragma "no-cache"
Header set Expires "0"
```

Figura 101: Establecimiento de flags para manejo de caché

- **Deshabilitar el método options**

El método options permite conocer los modos de comunicación sobre HTTP que son empleados para consumir un servicio web REST, para evitar la revelación de información importante, se debe deshabilitar esta bandera, la sintaxis para su correcta gestión se muestra a continuación en la Figura 102.

```
RewriteEngine On
RewriteCond %{REQUEST_METHOD} !^(GET|POST|HEAD|PUT|DELETE)
RewriteRule .* - [R=405,L]
```

Figura 102: Gestión Métodos permitidos por HTTP

- **Deshabilitar el método trace**

En sus albores el método trace permití hacer debug de las peticiones realizadas a un servidor, sin embargo, por aspectos de seguridad para evitar la divulgación de la información realizada en cada petición y su posible intercepción, es aconsejable deshabilitarlo, la sintaxis para deshabilitar el método trace se describe a continuación en la Figura 103.

```
TraceEnable off
```

Figura 103: Deshabilitar método Trace

7. Conclusiones y trabajo futuro

7.1. Conclusiones

- ✓ En lo referente al “estado del arte” se puede concluir que en el proceso de búsqueda de información relacionada con el tema del presente TFM es muy general y no se enfoca a una implementación práctica para el desarrollo de servicios web REST seguros; mucha más información se encuentra sobre seguridad en servicios web SOAP.

- ✓ La metodología empleada en el presente TFM contiene los pasos necesarios para seguirlos y poder contar con elementos útiles para la formación personal como desarrollador de software seguro puesto que al realizar una implementación de código propia, como tradicionalmente lo realizamos día a día, permite el auto aprendizaje y corrección de falencias de cada programador

- ✓ Los objetivos planteados para el TFM realizado han sido alcanzados de una manera satisfactoria. Mediante la implementación de la metodología propuesta se ha podido generar un servicio web REST seguro y verificado y validado en cada una de las fases de desarrollo.

- ✓ Si bien las herramientas de análisis de código y vulnerabilidades empleadas para este proyecto pueden ser usadas en la recolección de información acerca de problemas de seguridad en nuestros servicios web REST, no se debe confiar al totalmente en sus resultados puesto que al haber falsos positivos y falsos negativos, se deben realizar las respectivas verificaciones manuales.

- ✓ No es suficiente con emplear una sola herramienta para el análisis de vulnerabilidades, dado que se pueden obviar algunas debilidades entre una y otra; al igual pueden presentarse reportes en los que no se muestren vulnerabilidades de día cero que no constarán en la base de datos de firmas de las herramientas.

- ✓ La configuración de las herramientas de análisis de vulnerabilidades previo un análisis de aplicaciones juegan un papel muy importante, debido a que esto determina la cantidad de falsos positivos y negativos que reportará la herramienta por lo que se ha

descrito en el capítulo cuatro la manera en la cual se han configurado las herramientas para obtener los resultados que fueron tratados.

- ✓ Se debe considerar el uso de pasarelas XML como mecanismo de seguridad en profundidad complementario al aseguramiento de los servicios web REST a nivel de código y configuración del servidor. Las pasarelas estarán encaminadas a proteger la seguridad en el entorno de producción; de igual manera se debe tomar en cuenta firewalls de aplicación que analizarán los paquetes antes de atender cualquier petición.

- ✓ Luego del análisis de los servicios web REST con herramientas de análisis de código, se han detectado un total de 14 vulnerabilidades, habiendo sido mitigadas satisfactoriamente doce, las dos vulnerabilidades restantes fueron analizadas, concluyendo que se tratan de falsos positivos.

- ✓ Se han desarrollado servicios web vulnerables en primera instancia que, luego de ser mitigadas sus deficiencias de seguridad, permitieron la elaboración de una guía de consejos útiles que se presentan en el presente trabajo de fin de master con el afán de servir de apoyo al futuro codificador de soluciones informáticas.

- ✓ Se ha elaborado una breve guía como aporte al desarrollador de aplicaciones y servicios web REST para evitar codificación insegura y asegurar la configuración adecuada en sus servidores o en el propio código fuente manipulando las cabeceras para las peticiones que podrían ser peligrosas e inseguras.

7.2. Líneas de trabajo Futuro

Las conclusiones resultantes de la mitigación de vulnerabilidades en los servicios web REST, sirven de antesala para el desarrollo de algunos trabajos de investigación relacionados con la seguridad informática no solo desde el punto de vista del uso de pasarelas XML o firewall de aplicación sino también relacionados a técnicas de codificación segura orientadas a servicios REST; es así que tenemos:

- ✓ Estudio comparativo de vulnerabilidades encontradas en servicios web REST con código inseguro y código seguro empleando pasarelas XML como filtro de actividades maliciosas. Principalmente se trataría de verificar la eficacia de

pasarelas XML ante diferentes tipos de ataques y vulnerabilidades tomando como variable si hay bugs de seguridad o no en el código fuente desarrollado.

- ✓ Diseño y desarrollo de guías de seguridad para la programación segura, orientadas a concientizar a los desarrolladores en las consecuencias de la implementación de código vulnerable como afectación a la calidad de software.

- ✓ Uso de inteligencia artificial para la detección de vulnerabilidades en servicios WEB en fase de codificación. La idea principal sería que, al momento de finalizar una fase de desarrollo de software, en la etapa de pruebas, el desarrollador ejecute un análisis mediante un motor de inferencia basado en inteligencia artificial para detectar posibles vulnerabilidades presentadas en su código implementado hasta el momento.

- ✓ Propuesta de la guía de desarrollo de servicios web REST seguro ante la comunidad de desarrolladores PHP con la finalidad de homologar sobre un estilo de programación tendiente a evitar fallos de seguridad del software

8. Referencias

academiaandroid. (05 de 11 de 2015). Obtenido de https://academiaandroid.com/servicios-web-arquitectura-rest/

acunetix. (2018). *acunetix*. Obtenido de https://www.acunetix.com/

Blancarte, O. (06 de 03 de 2017). *oscarblancarteblog*. Obtenido de https://www.oscarblancarteblog.com/2017/03/06/soap-vs-rest-2/

Caules, C. Á. (30 de 06 de 2014). *arquitecturajava*. Obtenido de https://www.arquitecturajava.com/rest-JSON-y-java/

corisecio. (04 de 06 de 2014). *opensource.corisecio.com*. Obtenido de http://opensource.corisecio.com/?id=secrt_framework

Dave Wichers, P. P. (02 de 03 de 2018). *Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet OWASP*. Obtenido de https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet

Development, H. P. (15 de 04 de 2017). *www.microfocus.com*. Obtenido de https://www.microfocus.com/documentation/fortify-static-code-analyzer-and-tools/1710/HPE_AWB_Help_17.10/index.htm

expansión.mx. (04 de 11 de 2011). Obtenido de https://expansion.mx/tecnologia/2011/11/03/facebook-vs-google-la-lucha-por-la-web

Fielding, R. (25 de 10 de 2000). *www.ics.uci.edu*. Obtenido de https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

github.io. (25 de 03 de 2015). Obtenido de http://ottocol.github.io/2/seguridad_REST/#/8

hostalia.com. (06 de 04 de 2015). *www.hostalia.com* . Obtenido de https://pressroom.hostalia.com/wp-content/themes/hostalia_pressroom/images/cross-site-scripting-wp-hostalia.pdf

IBM. (12 de 05 de 2016). *IBM Knowledge Center*. Obtenido de https://www.ibm.com/support/knowledgecenter/es/ssw_ibm_i_73/rzaj4/rzaj40a0internetsecurity.htm

Jeff Williams, J. M. (09 de 05 de 2018). *owasp.org*. Obtenido de https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet

Kyrnin, J. (24 de 04 de 2018). *lifewire*. Obtenido de https://www.lifewire.com/how-to-use-http-referer-3471200

Macías, A. D. (25 de 04 de 2016). *Universitad autónoma de Barcelona*. Obtenido de https://ddd.uab.cat/pub/tfg/2016/tfg_48974/TFGArticulo_AlexDuran.pdf

maframaran. (12 de 10 de 2015). *Aqui se programa*. Obtenido de http://aquiseprograma.co/2015/10/como-interceptar-las-peticiones-a-servicios-rest-publicados-con-jersey/

ottocol. (12 de 05 de 2016). *ottocol*. Obtenido de http://ottocol.github.io/2/seguridad_REST/#/7

owasp. (10 de 04 de 2017). *owasp.org*. Obtenido de https://www.owasp.org/index.php/Web_Services

owasp.org. (23 de 02 de 2014). *Owasp Security_Headers*. Obtenido de https://www.owasp.org/index.php/Security_Headers

Owasp.org. (10 de 04 de 2016). Obtenido de https://www.owasp.org/index.php/SQL_Injection

Pérez, I. (29 de 04 de 2015). *welivesecurity.com*. Obtenido de https://www.welivesecurity.com/la-es/2015/04/29/vulnerabilidad-xss-cross-site-scripting-sitios-web/

redinfocol. (28 de 11 de 2011). Obtenido de https://redinfocol.org/dvwa-conociendo-y-explotando-diferentes-vulnerabilidades-level-low/

Velasco, R. (25 de 04 de 2015). *redeszone.net*. Obtenido de https://www.redeszone.net/2015/04/25/seguridad-web-owasp-zap/

VERACODE. (15 de 02 de 2017). *veracode.com*. Obtenido de https://www.veracode.com/security/sql-injection

www.emtdist.com. (24 de 04 de 2016). Obtenido de https://www.emtdist.com/acunetix/