Un lenguaje para especificar pruebas de seguridad de caja negra automatizadas para sistemas Web

Ricardo Yepes Guevara

Trabajo presentado como requisito para optar al título de
Magister en Ingeniería

Asesor:
John Freddy Duitama Muñoz

Universidad de Antioquia
Facultad de Ingeniería
Maestría en Ingeniería
Grupo de Investigación Ingeniería y Software
Línea Seguridad Informática

Medellín, Colombia
2017

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

INFORME RESUMIDO

1.1. Descripción de la investigación

El correcto funcionamiento de las plataformas de cómputo que soportan tareas industriales esenciales y actividades estratégicas de gobierno, dependen de la calidad y de la estandarización del proceso de prueba utilizado por los analistas de seguridad. Este proyecto propone una solución al problema de lograr estandarizar el proceso de pruebas de seguridad sobre un TOE[1] ("objetivo de evaluación" o sistema sobre el cual se está realizando la prueba). La solución propuesta está orientada a la realización de pruebas en sistemas Web por ser una necesidad común en la industria y se considera únicamente la técnica de pruebas de caja negra, porque este tipo de escenario es con frecuencia el único disponible cuando el analista de seguridad no tiene acceso al código fuente de la aplicación.

Nuestro enfoque está dirigido a soportar un diseño de pruebas basadas en modelos; es decir, el analista de seguridad define el modelo de la prueba y un framework ejecuta una transformación desde el modelo para obtener un conjunto de comandos ejecutables para controlar escaneadores de vulnerabilidades que interactuarán con el TOE para encontrar sus fallos de seguridad. De esta manera, la prueba se hace reutilizable y se obtienen resultados menos dependientes de aspectos subjetivos relacionados a la persona que ejecuta la prueba. En la actualidad es difícil mejorar sistemáticamente un proceso de pruebas de seguridad porque cada analista incide en gran medida en los resultados obtenidos. En cambio, cuando se tiene un proceso más estándar, cada nueva técnica incluida en la herramienta podrá incrementar la calidad de cualquier prueba realizada posteriormente, independientemente del usuario de la herramienta.

Como resultado del proyecto, realizamos un análisis comparativo de los trabajos anteriores que abordan técnicas de "pruebas basadas en modelos" que han sido aplicadas en el contexto de la seguridad. Además, desarrollamos un prototipo que soporta múltiples analizadores de vulnerabilidades Web y sugerimos algunas ideas que podrían mejorar su nivel de adopción en la industria. La principal contribución teórica es la definición de un lenguaje visual y textual para modelar pruebas de seguridad. La especificación precisa usando un metamodelo y una gramática

---

1. *"Target of Evaluation" u "objetivo de evaluación", según lo define el estándar de seguridad Common Criteria. El término se refiere al sistema objetivo de la prueba sobre el cual se está realizando la búsqueda de vulnerabilidades.*

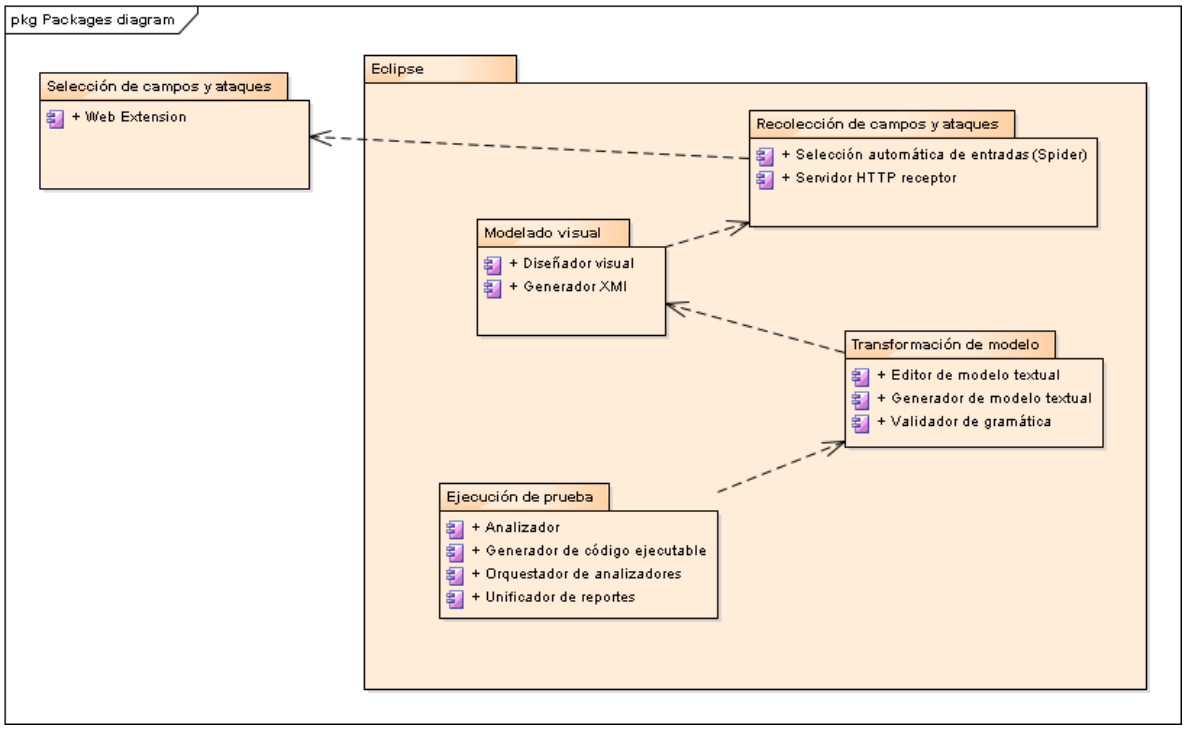
permite que los modelos expresados en este lenguaje puedan ser transformados a instrucciones específicas de ejecución para analizadores de vulnerabilidades.

Palabras clave: Pruebas dirigidas por modelos, pruebas basadas en modelos, automatización de pruebas de seguridad, pruebas de caja negra, analizadores de vulnerabilidades, pruebas de aplicaciones Web.

1.2. Descripción del desarrollo

Plataforma para ejecución de pruebas de seguridad de caja negra en sistemas Web con soporte para pruebas basadas en modelos. La aplicación está desarrollada como un complemento de Eclipse, utilizando los lenguajes de programación Java y Clojure. También cuenta con un componente que actúa como plugin en el navegador Web, encargado de detectar los campos y ofrecer una interfaz para selección de ataques por entrada.

La arquitectura general de la herramienta desarrollada se muestra en el siguiente diagrama:



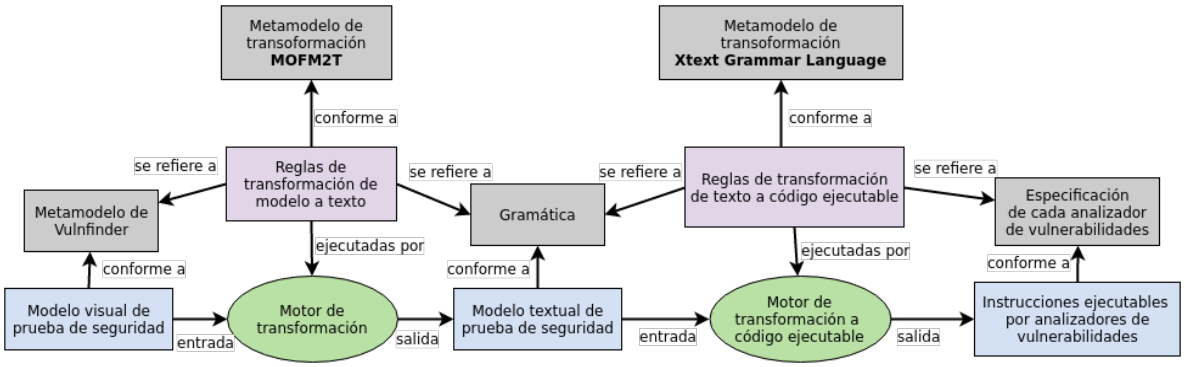
Se observa cómo la mayoría de los componentes dependen de la plataforma Eclipse. Es una decisión producto de la revisión del estado del arte en técnicas visuales de

modelado. Eclipse una de las alternativas de software libre más completa para modelado visual basado en metamodelos, permitiendo la creación de un entorno de desarrollo integrado para el usuario final. El código fuente de la herramienta se encuentra disponible en: https://gitlab.com/ryepesg/vulnfinder

1.3. Descripción de la mejora o novedad

La herramienta utiliza analizadores de vulnerabilidades existentes, de un modo extendible, utilizando metodologías de pruebas basadas en modelos, con representación textual (gramática) y modelado visual (metamodelo). Está en capacidad de generar modelos a partir de sistemas ya construidos, de modo que ahorra tiempo modelado, característica fundamental en entornos industriales.

A continuación se presentan los elementos de transformación de modelos utilizados por la herramienta desarrollada:



Se requiere soportar dos transformaciones, una de modelo a texto y la otra de texto a código ejecutable para analizador de vulnerabilidades específico. La primera utiliza un metamodelo general como origen mientras que la segunda una gramática. Los metamodelos de las transformaciones son directamente provistos por las herramientas de Eclipse (Acceleo y Xtext). La transformación de modelo a texto sigue el estándar MOFM2T.

Para pasar desde un modelo de alto nivel hacia un código ejecutable para cada una de los analizadores, se decidió dividir el proceso en dos etapas:

- **Transformación de modelo a texto**: En esta fase se convierte el modelo de alto nivel y visual especificado por el usuario, en un modelo textual que representa la misma información pero en un formato que es más fácilmente validable por una gramática formal. La representación textual es un punto de partida más cercano al resultado final esperado por cada herramienta, lo que facilita que en etapas posteriores la lógica específica de cada analizador de vulnerabilidades pueda estar encapsulada en módulos individuales.

- **Transformación de texto a código ejecutable para herramienta específica**: Cada herramienta usa su propio lenguaje, es por ello que se requiere una transformación especial por cada una. Por cada analizador de vulnerabilidades o fuzzer se diseña un adaptador que traduce el modelo textual en el código ejecutable.

1.4. Aplicaciones

Detección de vulnerabilidades en sistemas Web, para que los desarrolladores puedan corregirlas antes de desplegar el producto en producción y que sea accesible por atacantes. Los usuarios potenciales son empresas de seguridad informática y organizaciones que ofrecen servicios críticos mediante infraestructura Web.

Protección de la propiedad intelectual: Se cuenta con un Registro ante la Dirección Nacional de Derechos de autor, radicado 1-2017-26092.

Aportes para el fortalecimiento tecnológico y científico de la universidad:

1. Formación de estudiantes. Se formó un estudiante de maestría y participó un estudiante de pregrado como auxiliar de investigación.
2. Desarrollo de capacidades en el grupo. Se desarrolló un proyecto CODI con la participación de profesores investigadores, estudiantes de posgrado y

pregrado. Además el software se probó en sitios Web desarrollados por terceros, incluyendo el de universidades vinculadas al CERN.

3. Implementación de nuevos servicios o productos. Un software que se puede usar para desarrollar pruebas de seguridad de caja negra en aplicaciones Web.
4. El desarrollo permite contar con módulos reutilizables en futuros proyectos. El software es extendible. Es decir, la arquitectura permite agregar nuevos analizadores y nuevos fuzzer para desarrollar pruebas de software. Además es una implementación independiente de una plataforma específica.

INTRODUCCIÓN

En esta sección se explica la problemática alrededor de las pruebas de seguridad y se definen los conceptos fundamentales relacionados a las pruebas basadas en modelos, especialmente la pruebas de seguridad de caja negra en aplicaciones Web.

Los sistemas informáticos se desarrollan para cumplir una función específica y satisfacer una necesidad de un conjunto de usuarios. En ocasiones, las necesidades satisfechas son básicas, por ejemplo las empresas del sector de servicios públicos de energía usan software para distribuir la energía eléctrica que llega a sus usuarios en Colombia (Zuluaga, 2012). Sin embargo, es común que existan errores de programación que den origen a problemas de seguridad o vulnerabilidades. Una vulnerabilidad es una falla o debilidad en los procedimientos del sistema de seguridad, en el diseño, la ejecución, o los controles internos que podrían ser ejercidos y dan lugar a un fallo de seguridad o una violación de la política de seguridad del sistema (NIST, 2002). Desde hace más de una década la mayoría de incidentes de seguridad en software son causados intencionalmente por atacantes que explotan fallas de seguridad según estadísticas del SEI/CERT mencionadas en (Jan, 2004).

El costo de la ciberdelincuencia hasta 2015 alcanzó 594 mil millones de dolares en el mundo (Symantec, 2016) y ocurrieron 348 millones de robos de identidad durante 2014 en incidentes de seguridad en organizaciones reconocidas (Symantec, 2015). Fueron comprometidos más de 868 millones de registros con información sensible entre 2005 y 2014 (PCI Security Standards Council, 2015). El Managed Services Team de Cenzic (ahora adquirida por Trustwave) ha revisado la seguridad de miles de aplicaciones con clientes principalmente en Estados Unidos e informa que el 96% de las aplicaciones analizadas durante 2013 tuvieron por lo menos una vulnerabilidad crítica y 14 vulnerabilidades en promedio (Cenzic, 2014), aun cuando las organizaciones de desarrollo de software emplean en la fase de pruebas y depuración entre el 30% y el 40% del esfuerzo total del desarrollo del software (Pressman, 2010).

Se estima que para la década de 2010-2020 el enfoque reactivo frente a eventos de seguridad resultará ser más costoso que invertir previamente en aseguramiento (PriceWaterhouseCoopers, 2010). Aunque se calcula que en 2012 se destinaron 60 mil millones de dólares en seguridad a nivel global y que la cifra seguirá en aumento hasta 86 mil millones a finales de 2016, el costo promedio debido a delitos informáticos durante 2012 fue de 8.9 millones de dólares por organización. Hubo un incremento constante de la cantidad y nivel de sofisticación de los ataques, produciendo como resultado una cifra de 356 millones de incidentes entre 2005 y 2012, con el agravante de que en 2012 los ataques exitosos fueron el doble que en 2011 (Hewlett Packard, 2012).

Los fallos de seguridad pueden afectar vidas humanas y causar daños en infraestructura industrial y social, poniendo en riesgo la confidencialidad y privacidad de la información y socavando la viabilidad de sectores completos de negocio, tal como lo anuncia la asociación europea ITEA para la investigación y desarrollo en el área de sistemas y servicios intensivos en software (ITEA, 2014). La existencia de vulnerabilidades en las aplicaciones representa riesgos que afectan los objetivos de negocio de las organizaciones. Cuando existe una vulnerabilidad que permite que un atacante (amenaza) comprometa un activo de información (información valiosa para el negocio), se habla de riesgo, o en otros términos, la probabilidad de ocurrencia de un evento adverso y el impacto asociado en caso de que ocurra. Una vulnerabilidad crítica en un sistema podría permitir, por ejemplo, que un grupo terrorista controlara el sistema de suministro de recursos vitales para un país, afectando con ello la necesidad de subsistencia de muchos habitantes. Este es un escenario perfectamente probable, y de hecho, las principales potencias mundiales ya han tenido incidentes de seguridad asociados a ataques de esa naturaleza y las brechas han conducido a ciberterrorismo y espionaje entre países (Kerr, Rollins, & Theohary, 2010).

Cada vez más tareas críticas son realizadas desde plataformas Web, considerado el medio más común para realizar ataques informáticos (Trustwave, 2013). Más de la mitad de los hallazgos públicos ocurren en aplicaciones Web (IBM, 2010). Desafortunadamente, 86% de los sitios poseen por lo menos una vulnerabilidades crítica (WhiteHat, 2013). Consecuencias derivadas de las fallas de seguridad incluyen estafas virtuales

multimillonarias; durante 2012, prácticamente todos los tipos de industrias, países y tipos de datos fueron comprometidos de algún modo (Trustwave, 2013) implicando la exposición de datos personales (Homeland Security ICS Cyber Emergency Response Team, 2012). En total, se calculan pérdidas económicas por un millón de millones de dólares al año (Cenzic, 2014).

En la actualidad la industria tiene experticia en el uso de herramientas que facilitan la automatización de las pruebas de seguridad. Sin embargo, el ajustado tiempo de los proyectos obliga a que cada analista[2] deba escoger solo unas pocas herramientas entre los cientos que existen. Además, su experiencia incide en la forma de configurar cada herramienta, lo que inevitablemente lleva a una diferencia importante en los resultados obtenidos por diferentes personas que ejecuten una prueba aún cuando sea el mismo TOE (lo que en otros contextos se conoce como SUT o System Under Test). Tratar de lograr resultados similares en pruebas realizadas por personas diferentes es lo que denominamos en este proyecto como estandarización de la prueba. El enfoque de estandarización abordado durante esta investigación cumple también con la propiedad de la repetibilidad, es decir, que una vez especificada la prueba, pueda ejecutarse más de una vez contra el mismo TOE.

Muchas veces los arquitectos de seguridad encargados de revisar la calidad de las pruebas deben limitarse a revisar los resultados obtenidos y no el proceso. Un lenguaje intermedio simplificado con la información del diseño de la prueba (no con los detalles de ejecución) podría facilitar la tarea del arquitecto. En nuestro conocimiento no existe ese lenguaje estandarizado y de libre uso para especificación de pruebas de seguridad de caja negra basadas en modelos para sistemas Web. Si bien gran parte de lo discutido en este documento aplica a las pruebas de seguridad en general, el alcance de la investigación está limitado solo a pruebas de seguridad en Web, de modo que en adelante, siempre que se emplee el término “pruebas de seguridad” realmente se estará haciendo referencia a ese tipo de pruebas aplicadas en sistemas Web.

2. *Un analista es la persona competente en ciertas tareas necesarias durante la creación de aplicaciones, p.e. es el analista de seguridad quien realiza las pruebas de seguridad.*

Las pruebas de seguridad (“vulnerability testing” o "security testing”) son el filtro final para evitar que las vulnerabilidades se desplieguen luego en sistemas de producción y sean accesibles para los atacantes. Una de las formas de clasificar las pruebas de seguridad es según el grado de conocimiento sobre el comportamiento interno de la aplicación que posea el analista.

Cuando la información es completa (por ejemplo se tiene acceso al código fuente) se denominan "pruebas de caja blanca", de las cuales el enfoque más común es "Static Application Security Testing” (SAST), que consiste en revisar el código de la aplicación sin ejecutarla. En contraste, las "pruebas de caja negra" ignoran el funcionamiento interno del sistema y se concentran solo en las entradas y salidas, como ocurre por ejemplo en la técnica "Dynamic Application Security Testing” (DAST), donde se interactúa con la aplicación en ejecución para analizar las salidas producidas y encontrar sus vulnerabilidades. La diferencia principal frente a SAST es que en vez de revisar los componentes internos del sistema se examina es el comportamiento del TOE.

Las pruebas “de caja blanca” tienen sentido principalmente cuando el analista de seguridad tiene acceso al código fuente, un insumo que en manos equivocadas permitiría el robo de propiedad intelectual, por ello la dificultad para que los propietarios permitan que sea accedido incluso bajo cláusulas de confidencialidad. El proyecto se enfoca en los escenarios en los que no se dispone dicho código fuente, es decir, en las pruebas “de caja negra”.

Una forma de aumentar el nivel de estandarización de las pruebas es utilizando pruebas basadas o dirigidas por modelos, conocidas también como “Model-Based Testing” o MBT. Si bien existe una importante diferencia entre “desarrollo dirigido por modelos” y “desarrollo basado en modelos”, en el caso de las pruebas no se hace dicha distinción y se espera que los modelos no sean solo un simple soporte dentro del proceso sino un insumo fundamental del que se derivan otros artefactos. Una prueba basada en modelos es aquella que se diseña a partir de un modelo de alto nivel de abstracción en donde se especifican los aspectos que se desean cubrir y se deja que los detalles específicos de

realización sean configurados automáticamente en el momento en que se realice la prueba (Bau, Bursztein, Gupta, & Mitchell, 2010).

La idea central es que a partir de un modelo de alto nivel de abstracción que describe la prueba se generen los comandos específicos necesarios para aplicar dicha prueba en un TOE[3].

Los requerimientos de la industria respecto a las pruebas de seguridad piden que se garantice un nivel de calidad estándar. Hoy en día la solución empleada es utilizar procedimientos escritos que desafortunadamente se desactualizan con frecuencia y las personas no los siguen rigurosamente. La industria también exige que se empleen casos de prueba exhaustivos para considerar todos los escenarios que luego un atacante desarrollará contra el sistema. Sin embargo, el analista que realiza la prueba cuenta con menos tiempo que el atacante real, pues las pruebas de seguridad generalmente se enmarcan en proyectos con un tiempo definido, generalmente más corto que la suma del tiempo libre de todos los posibles atacantes. La industria ha empleado como solución la actividad de automatizar, la cual implica buscar vulnerabilidades conocidas y descubrir nuevas vulnerabilidades mediante Fuzzing[4]. Sin embargo, la experiencia de cada analista es única, y si bien en términos generales todos ellos realizan el mismo tipo de tareas, los detalles específicos de cómo ejecutarlas pueden variar, incluyendo su preferencia personal de herramientas de seguridad a utilizar, pues ninguna de ellas es claramente mejor que las demás y utilizar más de una o dos por prueba está generalmente prohibido por los ajustados tiempos de un proyecto en una empresa.

La estandarización del proceso de pruebas se ve afectado por la subjetividad de la persona que sigue el proceso de búsqueda de vulnerabilidades. Además, las herramientas no cuentan con la inteligencia suficiente para considerar muchos aspectos del contexto de un sistema, a diferencia de un analista de seguridad quien sí podría correlacionarla con facilidad. La Tabla 1 resume la problemática descrita.

---

3. *También conocido como SUT o "System Under Testing". Ese término es más utilizado en pruebas para requisitos funcionales, pero para seguridad es más común denominarlo TOE.*
4. *Probar entradas no esperadas por la aplicación, usualmente generadas con valores aleatorios, para encontrar vulnerabilidades desconocidas.*

Condición o requerimiento	Enfoque actual de solución	Problema actual
Garantizar un nivel de calidad estándar.	Procedimientos escritos.	El procedimiento no contiene casos de prueba exhaustivos, se desactualiza con frecuencia y las personas no lo siguen rigurosamente. Resultados altamente dependientes del analista que ejecuta la prueba.
El analista que realiza la prueba cuenta con menos tiempo que el atacante real.	Automatizar: • Escanear vulnerabilidades conocidas. • Descubrir vulnerabilidades mediante Fuzzing.	Muchas herramientas entre las cuales escoger, ninguna absolutamente mejor que todas las demás.

Tabla 1: Problemática y falencias de los enfoques actuales de solución.

Una posible solución es permitir que las personas diseñen las pruebas en lenguajes de alto nivel usando MBT (esto ayuda a comunicar entre personas el diseño de una prueba) y dejar los detalles específicos de ejecución a una herramienta que genere los casos de prueba (esto hace que la prueba sea más reutilizable). Este enfoque podría permitir beneficios como el de su ejecución en diferentes escenarios; por ejemplo cuando se desarrollan versiones del mismo software para diferentes plataformas. También facilita repetir una misma prueba en diferentes contextos, por ejemplo en ambiente de desarrollo y luego en ambiente de producción (Dai, 2006). Es natural que los sistemas informáticos evolucionen con el tiempo, y mientras más reuso de las pruebas pueda lograrse, mayor ahorro de recursos económicos y de tiempo se obtiene.

El modo de implementar técnicas de MBT para ayudar a aumentar la estandarización de las pruebas de seguridad es aún un tema de investigación que tiene el potencial de reducir fugas de vulnerabilidades[5]. Las pruebas de caja negra basadas en modelos podrían disminuir los efectos individuales dependientes del analista que ejecuta la prueba, ayudando así a garantizar un nivel de calidad estándar. Un nivel de detección de

5. *Una fuga de vulnerabilidad es cuando una prueba de seguridad no detecta cierta debilidad del sistema.*

vulnerabilidades mínimo independiente de quien ejecute la prueba permite que un arquitecto de mayor experiencia pueda revisar el proceso y sugerir mejoras que impacten cada una de las pruebas futuras.

El enfoque tradicional usado por la industria para medir el valor de la inversión en temas de seguridad informática (ROSI), consiste en estimar cuál es la expectativa de pérdida anual (ALE) por causas de incidentes de seguridad. Es decir que se da por hecho que las organizaciones pierden dinero a causa de incidentes informáticos, por lo que se hace preferible invertir en seguridad que asumir las pérdidas económicas esperadas.

Investigar en el área de la seguridad informática presenta un interrogante ético sobre en qué medida se está ayudando a que los delincuentes informáticos adquieran más conocimiento y cuenten con mejores herramientas para lograr sus propósitos. Cuestiones similares aparecen cuando se considera la publicación de vulnerabilidades en aplicaciones, que así como sirven al fabricante para que repare su producto, sirven al atacante para comprometer sistemas afectados que no hayan sido actualizados. Algunos autores piensan que el rol de un investigador ético en seguridad informática incluye cerrar la brecha de conocimiento entre los atacantes y el resto de las partes sobre las vulnerabilidades de un sistema (Ford & Frincke, 2010). El desconocimiento de eventuales fallas de seguridad por parte de usuarios y desarrolladores de software, puede evitar una adecuada toma de decisiones para proteger la propiedad intelectual y privacidad de la información. Aumentar el conocimiento sobre de los sistemas usados por las partes afectadas puede ayudar a proteger los sistemas computacionales que a menudo son críticos en la infraestructura gubernamental y el bienestar de la comunidad.

Uno de los objetivos del proyecto desarrollado es ofrecer una nueva herramienta. Como en cualquier área, esta podrá ser usada con diferentes propósitos. Sin embargo, el producto desarrollado es un aporte positivo que puede ayudar a mejorar el nivel de la seguridad en aplicaciones Web más que ocasionar daños para los intereses de los propietarios de las plataformas que lleguen a ser evaluadas. La herramienta puede ser usada antes de la salida a producción del TOE, lo que implica que los propietarios tendrán la oportunidad de conocer sus vulnerabilidades mucho antes que un potencial atacante.

Otro aspecto en el que se puede contribuir, está relacionado con el bajo nivel de preparación de los responsables de la seguridad en las organizaciones. El informe Forrester Wave del primer trimestre de 2013 hace énfasis en la carencia de habilidades y medios suficientes para enfrentar amenazas informáticas (Need et al., 2013). Podría considerarse un aporte valioso el trabajar en desarrollar un mecanismo estándar de pruebas, que pueda ser utilizado por un mayor número de analistas de seguridad para obtener resultados acertados y minimizar con ello el número de vulnerabilidades que quedan expuestas en la etapa de producción.

Aún cuando se desea ofrecer un mecanismo que ayude a hacer las pruebas más estándar y repetibles, no es objetivo de este trabajo de investigación el reemplazar la capacidad humana para tomar decisiones complejas basadas en la experiencia que solo años de práctica pueden desarrollar en un analista. Un análisis de seguridad riguroso a menudo requiere ideas creativas, por lo cual sigue siendo necesaria la intervención humana. No debe generarse un falso sentido de seguridad al utilizar la herramienta de modo automático, en vez de ello, se espera que un analista la utilice para explorar rápidamente aspectos repetitivos de la prueba y pueda concentrar su destreza manual en los lugares donde su intuición y experiencia le indican mayor probabilidad de identificar vulnerabilidades.

El documento está estructurado de la siguiente manera: el *Capítulo 1: Alcance de la investigación* presenta la pregunta de investigación, hipótesis y los objetivos del proyecto. El *Capítulo 2: Marco teórico* presenta conceptos de MBT y del área de la seguridad. El *Capítulo 3: Revisión de la literatura* reporta un análisis de los modelos que han sido utilizados para la especificación de pruebas de seguridad de caja negra. El *Capítulo 4: Solución propuesta* muestra un lenguaje de modelado para especificar pruebas de seguridad y presenta la arquitectura de Vulnfinder, un prototipo de software usado para especificar las pruebas de seguridad explicadas en capítulos anteriores. El *Capítulo 5: Validación* reporta un diseño experimental sobre detección de vulnerabilidades para validar la hipótesis respecto al uso de analizadores de vulnerabilidades Web por medio de MBT. Finalmente el *Capítulo 6: Conclusiones y trabajo*

futuro resume los principales hallazgos de la investigación y presenta observaciones sobre una experiencia de usar técnicas basadas en modelos por parte de la industria en proyectos reales para detectar vulnerabilidades.

CAPÍTULO 2. ALCANCE DE LA INVESTIGACIÓN

2.1. PREGUNTA DE INVESTIGACIÓN

Uno de los problemas principales de la industria del software es el cómo lograr industrializar los procesos de desarrollo. Industrializar significa tener procesos estándar y la reutilización de componentes. En el área de las pruebas de seguridad, un proceso estándar significa que unos patrones de ataque que han sido identificados por iniciativas como CAPEC[6] puedan ser ejecutados sin que el analista que realiza la prueba tenga que pensar en exactamente cómo se ejecutan. De este modo, el diseño de una sola prueba puede utilizarse por ejemplo en una misma aplicación que corre en dos plataformas diferentes.

Los patrones no son el único mecanismo para lograr procesos estándar. También existen modelos, plantillas y reglas entre otros. Al ayudar a separar el diseño de una prueba de su implementación, estos mecanismos promueven la estandarización de la prueba. Además podrán usarse simultáneamente varios analizadores de vulnerabilidades en vez de unos pocos como es lo habitual en pruebas de la industria.

Dentro de nuestro conocimiento, no existe una herramienta que mediante un enfoque de pruebas basadas en modelos, use varios de los analizadores de vulnerabilidades que la industria de la seguridad emplea habitualmente. Por ello es común que las investigaciones actuales se enfoquen en el modelado de un número reducido de tipo de vulnerabilidades. Así que la pregunta de investigación del proyecto puede resumirse en: *¿Cómo emplear MBT en pruebas de seguridad que utilicen herramientas automatizadas de detección actualmente empleadas por la industria?*

2.2. HIPÓTESIS

La hipótesis que se pretende explorar es la siguiente: *Las pruebas de seguridad basadas en modelos pueden usar, de modo repetible, analizadores de vulnerabilidades Web y*

6. *Common Attack Pattern Enumeration and Classification, https://capec.mitre.org/*

aprovechando las conocidas ventajas del MBT sobre la disminución de la influencia sobre los resultados de la prueba que pueda ejercer la persona que la ejecuta.

Esta proyecto aporta en la búsqueda de una respuesta para esa hipótesis, enfocándose en la propuesta del lenguaje de modelado y en validar la efectividad de las pruebas dirigidas por modelos por medio de la construcción de una herramienta.

Tal herramienta será entonces el resultado de la investigación y así mismo el mecanismo principal que ayudará a validar la hipótesis en el proyecto marco. Una de las aplicaciones de ese software es permitir que un experto en seguridad pueda analizar la especificación (y no solo el resultado) de una prueba efectuada por otro analista (generalmente de menor experiencia).

2.3. OBJETIVOS DEL PROYECTO

2.3.1. Objetivo general

Definir un lenguaje (un metamodelo para los aspectos visuales y una gramática para el aspecto textual) para especificación de pruebas de seguridad automatizadas de caja negra basadas en modelos y validar su funcionalidad en una herramienta de código abierto para detección de vulnerabilidades en aplicaciones Web.

2.3.2. Objetivos específicos

1. Definir un lenguaje que soporte la especificación de pruebas de seguridad Web de caja negra basadas en modelos.
2. Construir una herramienta de código abierto que soporte el lenguaje definido y que se integre con herramientas automatizadas para detección de vulnerabilidades: dos (2) analizadores de vulnerabilidades Web y un (1) fuzzer.
3. Validar que un modelo de prueba se transforme en la sintaxis aceptada por las herramientas automatizadas para la detección de vulnerabilidades y que los casos de prueba se ejecutan exitosamente cubriendo la mayoría de la superficie de ataque[7].

7. *La superficie de ataque corresponde a la totalidad de entradas que componen al TOE.*

4. Validar que la herramienta ejecute diferentes tipos de ataques Web incluidos en el TOP 10 OWASP[8].

2.3.3. Características de la solución buscada

- Herramienta de código abierto.
- Generación semiautomática de un modelo inicial a partir de una aplicación Web en ejecución.
- Especificación de la prueba por medio de modelado de alto nivel.
- Ejecución automática de casos de prueba.
- Compendio de patrones de ataque seleccionables desde la herramienta.
- Soporte de modelado visual restringido por un metamodelo.
- Lenguaje textual definido por una gramática.
- Prueba de concepto de integración con analizadores de vulnerabilidades y/o fuzzers.

8. *The Open Web Application Security Project, proyecto dedicado a promover la seguridad en las aplicaciones Web por medio de documentación y herramientas. Al momento de realizar el trabajo de investigación, la última versión de OWASP publicada es la de 2013, aunque han anunciado una actualización para 2017. Evoluciones futuras de la investigación pueden llegar a tener en cuenta el nuevo listado que se publique para determinar si la herramienta Vulnfinder puede adaptarse a nuevos riesgos identificados.*

CAPÍTULO 3. MARCO TEÓRICO

Este capítulo presenta conceptos de seguridad y MBT. Primero explica una clasificación de las pruebas de seguridad en general y luego se enfoca en técnicas de automatización. También da a conocer aspectos esenciales de MBT incluyendo aspectos de un lenguaje de modelado como la sintaxis y la semántica. Concluye mostrando las relaciones entre modelado y generación de código por medio de las transformaciones de modelos. La Figura 1 presenta un esquema que resume los principales temas abordados en el capítulo, relacionando las pruebas de seguridad con aspectos de automatización y estandarización. Ese marco conceptual ayudará a analizar la revisión de la literatura presentada en el Capítulo 3 y a seleccionar las técnicas soportadas por el lenguaje propuesto para especificación de pruebas del Capítulo 4.

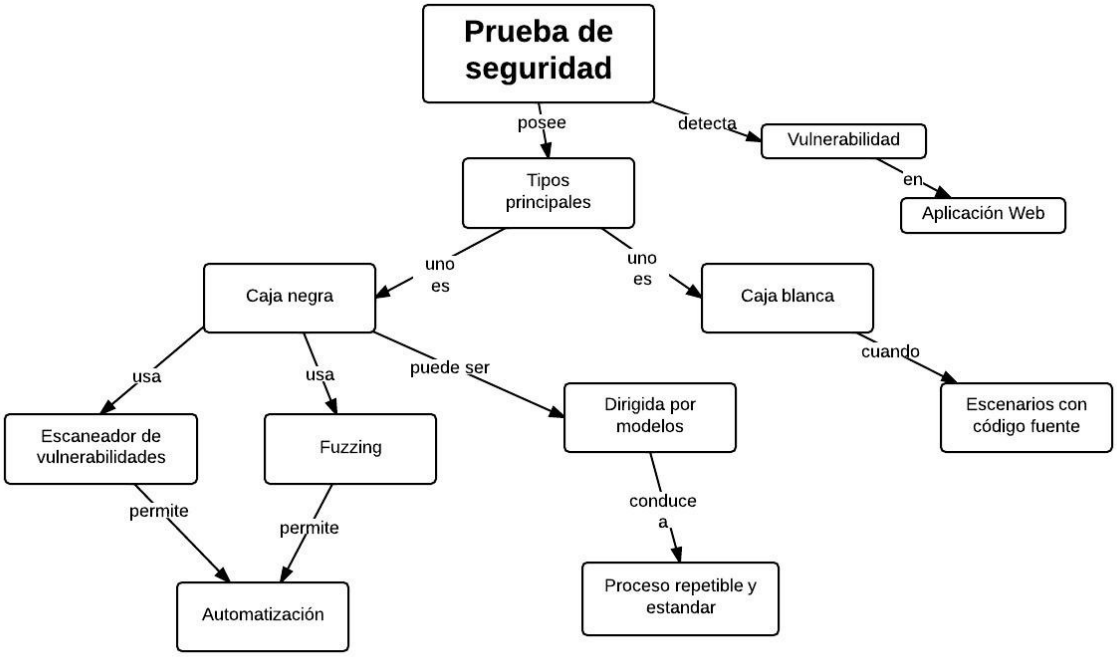


Figura 1: Aspectos en pruebas de seguridad que conducen a la automatización y estandarización.

3.1.1. Tipos de pruebas

En esta sección describimos los tipos de pruebas a partir de la clasificación realizada por ITEA DIAMONDS. ITEA es un programa para apoyar proyectos de investigación; uno de sus proyectos fue DIAMONDS[9] que trata de atender la creciente necesidad de métodos de pruebas de seguridad sistemáticos ofreciendo técnicas y herramientas que pueden ser utilizadas para asegurar de manera eficiente aplicaciones en red en diferentes dominios de negocio. El proyecto terminó en 2013 y como resultado dejó una serie de publicaciones y herramientas privadas y abiertas. Entre sus documentos se encuentra el paquete *WP2 – Técnicas,* referenciados en (Richier, 2011) y (Maag, 2013). A continuación se resume la clasificación identificada:

- *Pruebas formales*: Usan generalmente métodos matemáticos o algorítmicos para detectar contradicciones a partir de unas premisas iniciales. Son difíciles de llevar a la práctica debido a la complejidad de modelar los sistemas y su relación con el entorno (Wotawa, 2012).
- *Pruebas exploratorias*: Centran su efectividad en las habilidades del analista, por lo cual son intrínsecamente no estandarizadas.
- *Pruebas basadas en el riesgo*: Identifican amenazas e impactos pero a muy alto nivel, no desde la perspectiva de la aplicación.
- *Pruebas de seguridad en aplicaciones Web*: Siguen las mejores prácticas de proyectos como OWASP[10], donde se documenta una metodología para pruebas de seguridad con casos de prueba para aplicaciones Web. La metodología completa no ha sido automatizada ni describe sus etapas por medio de modelos, sino mediante pasos concretos a ser seguidos por una persona.
- *SAST (Static Application Security Testing)*. Incluye enfoques de caja blanca que requieren como insumo el código con el que fue creada la aplicación a ser probada o TOE. Se espera contar con el código fuente o formas cercanas que según la implementación del lenguaje de programación pueden ser instrucciones de máquina virtual (bytecode en Java) o directamente código de máquina (opcodes para la arquitectura de hardware). En un despliegue convencional de un sistema

9. *Development and Industrial Application of Multi-Domain Security Testing Technologies http://www.itea2-diamonds.org/Overview/index.html*
10. *The Open Web Application Security Project*

Web, solo el código enviado al navegador está disponible para el usuario final de la aplicación. Habitualmente este código no representa un riesgo de seguridad porque corresponde únicamente a la interfaz de usuario y no a una lógica de negocio especializada.

- *DAST (Dynamic Application Security Testing)*: Consiste en interactuar con la aplicación en ejecución para analizar las salidas que produce. Hace parte de las pruebas de caja negra y habitualmente incluye el uso de analizadores de vulnerabilidades. Teniendo en cuenta que no es común que los atacantes puedan realizar SAST en los sistemas Web convencionales y que aún así existe una alta cantidad de reportes de seguridad en estos sistemas, puede inferirse que DAST es altamente efectiva.
- *Pruebas de caja gris:* No requieren el código fuente como las pruebas de caja blanca, pero sí requieren un acceso parcial a la aplicación, ya sea el entorno de ejecución o el código binario, escenarios que generalmente no ocurren en pruebas para aplicaciones Web.
    - *Casos de prueba seleccionados en función de cambios en el código*: A partir de grafos de dependencias entre funciones de código (visibles no solo en código fuente sino en código binario), se determina la propagación de un cambio en la aplicación para ejecutar casos de prueba intensivos solo en las porciones modificadas.
    - *Análisis estático con variables marcadas (Taintflow)*: Examina cómo se mueve una entrada seleccionada en los flujos de la aplicación, generalmente modelados mediante grafos.
- *Pruebas basadas en inyección de fallas y basadas en propiedades*: Consisten en introducir errores intencionados en los sistemas con el objetivo de evaluar su robustez. En particular la técnica de *Detección de fallas* consiste en realizar cambios en los componentes de software de los que una aplicación depende (API o interfaces de programación) para analizar la estabilidad del sistema.
- *Automatización de intrusión*: Usa herramientas que incluyen casos de prueba específicos para ciertos tipos de vulnerabilidades conocidas. Dentro de la categoría se encuentran los escaneadores y analizadores de vulnerabilidades[11], programas

11. *Por analizadores de vulnerabilidades nos referimos a herramientas para automatizar la detección de vulnerabilidades. Muchas de las herramientas existentes incluyen simultáneamente funcionalidad de fuzzers,*

que automáticamente detectan fallos de seguridad, por medio de patrones de ataques incorporados en su funcionalidad.

Los tipos de pruebas que más se alinean con los objetivos de la presente investigación son DAST y Automatización de intrusión, aplicados a sistemas Web. En la siguiente sección, “Tipos de técnicas”, se explica la razón de utilizarlas en conjunto con técnicas de Fuzzing y pruebas basadas en modelos.

3.1.2. Tipos de técnicas

Los diferentes tipos de pruebas mencionados no necesariamente son excluyentes. Algunas vulnerabilidades se encuentran con mayor facilidad en uno de los tipos. DAST por ejemplo favorece la detección de ataques como *XSS*[12] y *CSRF*[13] mientras que SAST lo hace con *Buffer Overflow y Format Strings* según (Lebeau, Legeard, Peureux, & Vernotte, 2013). Por ello, se seleccionó para ser exploradas con mayor profundidad un conjunto de tipos de pruebas que puedan complementarse entre sí. A continuación el aporte principal de cada uno de ellos:

- Pruebas de seguridad en aplicaciones Web*:* Los sistemas Web son uno de los tipos de sistemas más comunes.
- DAST: En las pruebas de seguridad realizadas por terceros es común que no se proporcionen detalles sobre el funcionamiento interno de las aplicaciones, y menos aún su código fuente, para así evitar fuga de información propia del negocio. En (Henry, Zenteno, & X-t, 2008) adoptaron el enfoque de pruebas de caja negra precisamente para evitar esa dificultad.
- Automatización de intrusión*:* Son las técnicas típicas utilizadas por la industria de la seguridad para hacer pruebas de caja negra.

---

escaneadores y analizadores de vulnerabilidades. La diferenciación precisa no será necesaria durante el proyecto dado que el lenguaje a definir se concentra en términos de más alto nivel, por lo que seguiremos utilizando únicamente del término “analizador de vulnerabilidades” en adelante.

12*. “Cross Site Scripting”, ataque a sistemas Web en el que código malicioso es ejecutado en el navegador de la víctima.*

13*. “Cross Site Request Forgery”, ataque en el que la víctima resulta enviando peticiones prefabricadas por el atacante. Estas resultan en ejecución de acciones en la aplicación Web a nombre de la víctima.*

Esos tres tipos de pruebas pueden complementarse con la técnica de Fuzzing, que consiste en probar entradas no esperadas por la aplicación, usualmente generadas con valores aleatorios, muestras o datos derivados a partir de una especificación. Su importancia reside en que ayudan a encontrar vulnerabilidades desconocidas. Incluye los tipos:

- *Fuzzing activo*: El fuzzer genera entradas aleatorias para los distintos componentes del SUT y el analista monitorea su comportamiento para detectar fallos técnicos.
- *Fuzzing basado en modelos de comportamiento*: Utiliza fuzzing activo, pero compara las salidas obtenidas con el comportamiento esperado del SUT. Es decir, que no solo encuentra fallos técnicos de la aplicación, sino también problemas en la lógica.
- *Fuzzing evolutivo asistido por inferencia de modelos*: Infiere el modelo de procesamiento de entradas al interactuar con la aplicación. Este tipo de fuzzing consiste en utilizar algoritmos evolutivos como por ejemplo algoritmos genéticos.

Los tipos de pruebas DAST, Web y Automatización de intrusión también pueden ser complementados con pruebas basadas en modelos (Model-Based Testing o MBT), para ofrecer un nivel de abstracción que permita separar el diseño de la prueba de la ejecución de la misma, de modo que diferentes pasos de la implementación puedan ser generados de un modo más estándar. Los dos principales tipos de MBT son:

- *Ataques basados en patrones*: MBT que usa casos de prueba de ataques conocidos.
- *MBT de seguridad y modelos de comportamiento*: Además de los patrones de ataque, utiliza los modelos de comportamiento del sistema, como por ejemplo diagramas de secuencias para generar los casos de prueba.

MBT puede generar automáticamente casos de prueba (ataques) a partir de modelos del sistema. Es una prueba diseñada a partir de un modelo de alto nivel en donde se especifican los aspectos que se desean cubrir y se deja que los detalles específicos de realización sean generados automáticamente en el momento en que se ejecute (Mallouli, 2011). Incluso aproximaciones de MBT que no soportan un modelado visual, han sido

aplicadas con éxito en sistemas Web proporcionando un enfoque repetible y sistemático (Stepien & Peyton, 2012) (Calvi & Viganò, 2016). Se clasifica como un tipo de prueba de caja negra dado que los casos de prueba son derivados de modelos en vez de código fuente. Incluye los tipos "*MBT con ataques basados en patrones"* que usa casos de prueba de ataques conocidos y "*MBT de seguridad y modelos de comportamiento"* que además de los patrones de ataque, utiliza los modelos de comportamiento del sistema, como por ejemplo diagramas de secuencias o navegación para generar los casos de prueba. Los conceptos principales de MBT se muestran en la Figura 2 tomada de (Microsoft, 2016): *System Under Test (SUT) o TOE:* sistema sobre el cual se realizará la prueba. *Secuencia de pruebas*: controla al TOE, conduciéndolo a diferentes condiciones bajo las cuales este puede ser probado. *Oráculo de la prueba:* observa el progreso de la implementación y escenarios de éxito o falla de cada prueba. *Transformación de modelos*: proceso para obtener automáticamente código a partir de un modelo, es una técnica usada con éxito en construcción de software con enfoques como Model-driven engineering o ingeniería dirigida por modelos (MDE), metodología que busca derivar software a partir de modelos conceptuales que representan el conocimiento del dominio de un problema, en vez de conceptos de computación como lo son los algoritmos. Es similar a la arquitectura dirigida por modelos (MDA) pero en un contexto más amplio que solo lo definido por OMG[14].

14. *"Object Management Group", organización creadora de estándares como UML.*

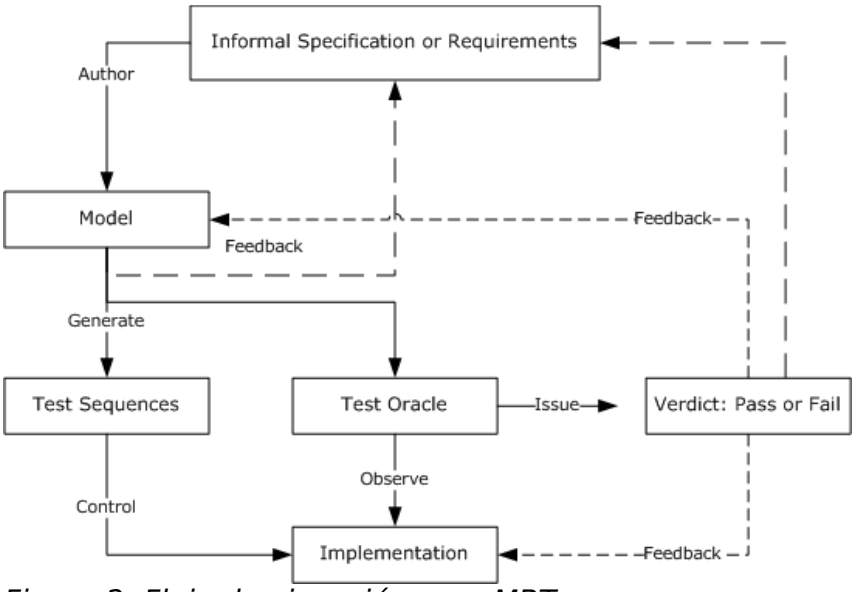


Figura 2: Flujo de ejecución para MBT

Si bien dentro de MDE (Ingeniería dirigida por modelos) existe una distinción frente a la ingeniería *basada* en modelos, en (*Model-Based Testing for Embedded Systems*, 2011) se explica que no hay diferencia entre MBT (pruebas basadas en modelos) y MDT (pruebas dirigidas por modelos). Incluso algunos autores utilizan ambos términos en un mismo artículo de modo intercambiable (Mohamed Mussa, Samir Ouchani, Waseem Al Sammane, 2009). A falta de una terminología precisa en la literatura para el concepto de pruebas de seguridad de caja negra basadas en modelos, en el resto del artículo se utilizará simplemente la sigla MBST (Model Based Security Testing). Se utilizará el término “MBT tradicional” para hacer mención a las pruebas que no son de seguridad.

MBST es una clase de MBT enfocado en detectar fallos de seguridad. En el contexto de este documento el término se refiere a pruebas de seguridad de caja negra automatizadas para sistemas Web que sean basadas en modelos

3.1.3. Tipos de lenguajes

Ya se ha mencionado la importancia de MBT para desacoplar el diseño de una prueba de su ejecución. Pero un aspecto esencial de MBT es el lenguaje a ser utilizado para

especificar las pruebas. En esta sección se explican las características principales de ese tipo de lenguajes.

Un lenguaje está compuesto por una sintaxis concreta y otra abstracta. La sintaxis concreta define la representación del lenguaje. Ejemplos incluyen definir que una secuencia de elementos se agrupe usando los caracteres [] (corchetes) o que un ataque de seguridad se represente con el ícono de una bomba. La sintaxis abstracta consiste en la estructura del lenguaje descrita como tipos de datos, sin una representación o codificación específica. Cuando se hace el análisis semántico de un modelo expresado usando el lenguaje, se construye un árbol de sintaxis abstracta para facilitar el análisis semántico, es decir entender el significado de las relaciones entre los elementos usados.

Un lenguaje puede ser utilizado para programación o para modelado. Puede ser de propósito general o de propósito específico y tener una notación textual o visual. La Tabla 2 muestra algunos lenguajes categorizados por esos tres aspectos.

Lenguaje	**Nivel de abstracción**	**Tipo de representación**	**Alcance**	**Usado en**
Python	Programación	Textual	Propósito general	Youtube.com
Emacs Lisp	Programación	Textual	DSL	Editor de texto GNU Emacs
Logo	Programación	Visual	DSL	Entorno de aprendizaje UCBLogo
Gellish Formal English	Modelado	Textual	Propósito general	CROW, una empresa alemana de construcción de infraestructura física
TTCN-3	Modelado	Textual	DSL	Titan TTCN-3 Toolset
UML	Modelado	Visual	Propósito general	Enterprise Architect
Perfiles UML	Modelado	Visual	DSL	Papyrus

Tabla 2: Ejemplos de diferentes tipos de lenguajes categorizados por nivel de abstracción, tipo de representación y alcance.

Los lenguajes de propósito general tienen la ventaja de poder representar conceptos de una variedad muy amplia de temáticas, a diferencia de un DSL (Domain Specific Language) que no puede expresar términos de más de un dominio. La ventaja de este último es la alta expresividad dentro del dominio, es decir que logran representar la misma información haciendo uso de menor cantidad de construcciones del lenguaje. En el caso de las pruebas de seguridad, un DSL es una solución más práctica que un lenguaje de propósito general. Los perfiles UML (Unified Modeling Language) permiten restringir la semántica de UML para dotarlo de significados específicos dentro de contextos más concretos. De ese modo los perfiles UML son una de las formas de especificar DSL, que ofrece la ventaja de poder usarse en la gran cantidad de herramientas con soporte para UML. Otros DSL, al no estar restringidos por la sintaxis y semántica de UML, pueden tener una flexibilidad mayor con una sintaxis concreta más rica. En general los DSL incrementan la productividad y aumentan la base de usuarios al permitir que personas que conocen del dominio pero no son programadores puedan de

igual forma especificar modelos que serán transformados a código. Los DSL además facilitan el análisis, verificación, optimización y paralelización de las instrucciones a ser ejecutadas, dado que el usuario indica el “qué” pero no el “cómo”, dando así la libertad a que en fases posteriores puedan tomarse decisiones de implementación.

La sintaxis abstracta de un lenguaje puede ser especificada mediante un metamodelo (modelo que explica la semántica y sintaxis permitida, usado generalmente para describir lenguajes visuales) o mediante una gramática (cuando el lenguaje es textual). Un lenguaje de modelado está compuesto por notación, estructura y semántica (Fischer, Scheidgen, & Blunk, 2014). Para trabajar con la notación, hay dos tipos de herramientas, el analizador sintáctico (compuesto de un lexer y un parser) y el editor. Este último proporciona los elementos visuales para el diseño del modelo. El analizador sintáctico puede procesar gramáticas basadas en texto y en grafos, así como también la representación textual un modelo gráfico, de modo que se verifique la coincidencia entre el modelo de entrada con los elementos definidos en la sintaxis abstracta. La estructura del lenguaje de modelado define cómo se interrelacionan las construcciones del lenguaje. La forma típica de proporcionar esa estructura es utilizando metamodelos y definiciones de restricciones. Finalmente la semántica es el significado de cierto arreglo de construcciones del lenguaje usando una notación específica. Herramientas para tratar la semántica incluyen compiladores, generadores de código, motores de transformación (transformation engines), intérpretes, simuladores y comprobadores de modelos (model checkers).

3.1.4. Transformación de modelos

La transformación de modelos es una parte esencial de cualquier MBT, tanto del tradicional enfocado en pruebas funcionales como el MBST. En esta sección se explican algunas definiciones que fundamentan la solución que se propone en capítulos posteriores.

La ingeniería del software ha dado gran importancia al uso de lenguajes visuales de modelado, no solo para documentar software sino también para generar código ejecutable automáticamente a partir de los modelos. Esa generación de código se puede realizar con una transformación del tipo modelo a texto (M2T). Si la transformación no genera texto como resultado, se denomina transformación de modelo a modelo (M2M). Este tipo de transformaciones son usadas para bajar el nivel de abstracción, por ejemplo llevando una representación abstracta a un modelo de una tecnología específica. Existen diferentes formas para especificar las reglas de validación de modelos para corroborar que estos cumplan con las restricciones de la sintaxis abstracta, así como maneras de definir el cómo realizar las transformaciones.

La definición de las transformaciones es posible utilizar un lenguaje de propósito general como Java. Sin embargo, muchas de las necesidades que aparecen durante la implementación se resuelven más fácilmente con otros métodos más especializados. En particular es posible utilizar Eclipse para proveer a los usuarios final todas las ventajas de un entorno integrado de desarrollo moderno con soporte para modelado visual (con revisión cumplimiento frente a las restricciones del metamodelo), modelado textual (validando cumplimiento contra una gramática) y transformación de modelo a código ejecutable.

Para revisar que el modelo cumple con todas las restricciones del metamodelo, el software debe hacer una gran cantidad de cálculos que pueden tomar mucho tiempo y recursos de procesamiento. Una estrategia para lograr alta eficiencia es utilizar algoritmos especializados en ese tipo de tareas, por ejemplo SMT solvers[15].

Un SMT Solver cuenta con diferentes mecanismos para determinar si el conjunto de restricciones entregadas se cumplen. Por ejemplo incluye un SAT Solver para restricciones booleanas. Cuando se es capaz de expresar todas las restricciones en términos de OR, NOT y AND, el SAT lo resuelve muy rápidamente. Cuando las

15. *Para los modelos utilizados durante el proyecto de investigación no se encontró la necesidad de optimizar esa validación de restricciones, pues los algoritmos por defecto ofrecidos por Eclipse producían tiempos razonables (5 segundos en promedio) de chequeo en todos los casos. Los párrafos siguiente incluyen una explicación de cómo funcionan los STM Solvers solo por referencia, pero esos conceptos no son necesarios para entender las secciones siguientes sobre la arquitectura de la solución.*

restricciones son aritméticas (por ejemplo x < 5), se usa la programación por restricciones, pero se debe seguir cumpliendo que cada variable es de tipo booleana (es decir que x < 5 implica definir las variables, x0, x1...x4 para luego hacer XOR entre ellas). Apenas algunos SMT Solvers como Z3 están empezando a resolver directamente restricciones aritméticas sin necesidad de que las variables sean booleanas, pero ese tipo de problemas ya lo resolvía años atrás herramientas como GNU Prolog, que también usa programación por restricciones. Es menos eficiente que un SMT Solver, pero permite expresar fácilmente restricciones como x+y=y/50, lo cual aunque es posible en un SMT Solver, en la mayoría de ellos lograrlo es muy tedioso. Dado que los modelos habitualmente se representan en formato XMI (forma especializada de XML), luego de que se ha revisado que el modelo corresponde al metamodelo, puede transformarse el modelo en texto utilizando XPath[16], lenguaje estándar para recorrer archivos XML.

Los SMT Solver y Xpath tienen como principal ventaja el alto nivel de eficiencia para detectar cumplimiento un gran número de restricciones por unidad de tiempo y transformar modelos. La plataforma Eclipse ofrece mecanismos comparables en funcionalidad pero no en rendimiento.

Hay escenarios en los que se hace verificación de modelos (model checking) fuera de línea, es decir, sin interacciones externas, así que el procesador se dedica completamente a hacer cálculos hasta encontrar las respuestas. Tal es el caso de verificación de modelos para especificación de protocolos y sistemas de almacenamiento de alta criticidad (Pierce, 2016). Sin embargo, en el caso de las pruebas de seguridad en sistemas Web, la limitación de rendimiento está en las velocidades de transferencia de información entre cliente y servidor HTTP (I/O bound en vez de CPU bound). Es decir, que hay muchos momentos en los que el procesador está inactivo y puede utilizarse para verificar restricciones. Así que en este contexto, no es tan evidente para el usuario final la diferencia de rendimiento entre plataformas como Eclipse comparada a los SMT Solvers y Xpath. En cambio, otros diferenciadores pasan a tener más relevancia, como por ejemplo la capacidad de Eclipse para ofrecer un editor con resaltado de sintaxis y autocompletado para la gramática definida para los modelos textuales o editores gráficos para expresar modelos visuales.

16. *https://www.w3.org/TR/xpath/*

CAPÍTULO 4. REVISIÓN DE LA LITERATURA

En este capítulo se realiza un análisis de los modelos que han sido utilizados para la especificación de pruebas de seguridad de caja negra y revisa cómo ese enfoque puede ayudar en la solución efectiva de las dificultades relacionadas a la automatización y estandarización de las pruebas de seguridad. Inicialmente se explica la metodología de revisión de la literatura empleada y se hace un recorrido por los modelos visuales usados en MBT para sistemas Web. Luego se enfoca la búsqueda hacia MBT aplicado a seguridad y especificación de requisitos de seguridad. Se concluye el capítulo con un cuadro comparativo del estado del arte y se listan las oportunidades de mejora identificadas en el estado del arte actual.

4.1.1. Metodología de revisión de la literatura

El método de búsqueda seguido para encontrar los antecedentes fue inspeccionar publicaciones entre 2009 y 2016 en conferencias internacionales como *A-MOST 2015-2016*[17] y *ACM/IEEE MODELS 2012-2015*[18]*.* Se consultaron revistas científicas como *IJSEA*[19]*, JATT*[20] y publicaciones de *ITEA DIAMONDS project*[21]*.* A partir de allí se continuó, usando una selección de palabras clave, con una búsqueda en las bases de datos *IEEE Xplore, ScienceDirect, Springer Link, ACM Digital Library y Google Scholar*. Además de estas fuentes académicas, también se examinó las herramientas incluidas en el sistema operativo de código abierto para pruebas de seguridad más utilizado por la industria: Kali Linux[22].

17. *11th & 10th Workshop on Advances in Model Based Testing*
18. *14th, 15th & 17th Model Driven Engineering Languages and Systems*
19. *International Journal of Software Engineering & Applications*
20. *Journal of applied testing technology*
21. *Development and Industrial Application of Multi-Domain Security Testing Technologies, un proyecto de investigación completo que cuenta con un compendio de técnicas y métodos explicados por medio de varios artículos de diferentes autores*
22. *DistroWatch - http://distrowatch.com/table.php?distribution=kali*

Se utilizó el sistema Journal Citation Reports (JRC) para explorar más a fondo los artículos con mayor nivel de impacto. Los criterios de búsqueda aplicados comprenden MBT, seguridad y modelado visual. Es decir que solo lenguajes de modelado que hayan sido utilizados para pruebas de seguridad de software serán minuciosamente analizados, lo que excluye estándares como QVT (lenguaje estándar de OMG para transformación de modelos) y lenguajes textuales como WebDSL. Si bien los perfiles de UML son de interés, se revisan solo los que soportan el modelado de pruebas de software y no los que describen el software en sí mismo como por ejemplo MARTE. Aproximaciones sobre MBT de seguridad aplicado a hardware criptográfico (Kramer & Legeard, 2016), también fueron intencionalmente excluidas de la revisión de la literatura.

Se encontraron antecedentes que explícitamente abordan el tema de la seguridad mediante MBT, siendo uno de ellos un proyecto de investigación completo que cuenta con un compendio de técnicas y métodos explicados por medio de varios artículos de diferentes autores. Aún así, es una proporción baja de artículos que abordan conjuntamente MBT desde el punto de vista de la seguridad, comparado con otros tipos de pruebas como las de usabilidad. *Model Based Testing* adaptado a seguridad es un tema que, aunque útil, ha sido relativamente poco explorado.

4.1.2. Modelos visuales en MBT para de sistemas Web

En esta sección se explora el uso de modelos visuales en MBT no enfocado en seguridad (MBT tradicional). El término Model-Based Testing o MBT, puede rastrearse en enfoques de pruebas de software tan antiguas como (Jorgensen, 1995). Propuestas para expresar modelos visuales para aplicaciones Web pueden encontrarse desde más de una década atrás (Hennicker, 2001) (Gorshkova & Novikov, 2002) (Schwinger & Koch, 2006) con estándares que empiezan a ser adoptados por OMG, como por ejemplo IFML[23]. Existen propuestas relacionadas que usan lenguajes estándar para definición de servicios web WSDL[24] para generar los casos de pruebas para herramientas como SoapUI (Paydar, 2011); en ella se propone un framework basado en agentes, el cual obtiene información

23. *Interaction Flow Modeling Language, lenguaje de modelado para representar flujos de información (navegación visual)*
24. *Web Service Definition Language, lenguaje para describir la naturaleza de un servicio web*

sobre el SUT a partir de código fuente si está disponible, diagramas de clases UML (usados para representar relaciones entre conceptos), los datos de las sesiones y de la interfaz Web pública. Ese tipo de diagramas UML es utilizado también en (Informatik et al., 2010) para aplicar MBT y evaluar su eficiencia según el nivel de cobertura de código sobre SUT que se ejecuta la prueba.

En la propuesta de modelos navegacionales MDWE[25] presentada por (Henry et al., 2008) se destacan los MBT con enfoque de MDA (Model Driven Architecture), caracterizados principalmente por el uso de los diagramas de clases UML para la definición de metamodelos y la transformación de modelos; además de un interés especial por los modelos navegables que permiten una representación más precisa de las aplicaciones Web, usando diagramas de secuencia UML, usados para representar intercambio de información entre páginas.

La comunidad científica ya cuenta con estados del arte robustos para MBT tradicional, como por ejemplo el presentado dentro de la propuesta de modelos navegacionales MDWE[26] (Henry et al., 2008) y se destaca el modelado con enfoque de MDA (Model Driven Architecture), caracterizados principalmente por el uso de UML para la definición de metamodelos. También se mencionan el uso de modelos navegables que permiten una representación más precisa de las aplicaciones Web, usando diagramas de secuencia UML. En (Neto & Vieira, 2007) se hace una revisión sistemática a partir de 406 artículos sobre MBT tradicional y se sugiere una clasificación de los artículos en aquellos enfocados en pruebas funcionales para requerimientos y aquellos que en cambio describían las pruebas a partir de componentes internos de software (i.e. arquitectura, interfaces, componentes); también hace una división entre los que utilizaban diagramas UML y los que no.

RT-Tester es una herramienta que ha sido usada por 15 años en la industria de sistemas de control aéreos y automovilísticos. Destaca su evolución desde antiguos lenguajes de modelado hacia un nuevo subconjuntos de UML y SysML para usar MBT en sistemas completos (software y hardware). Esos modelos son transformados a una representación

25. *Model-Driven Web Engineering, Ingeniería Web basada en modelos.*
26. *Idem.*

textual intermedia usualmente formato XMI que es un estándard de OMG ampliamente soportado por herramientas de modelado. Un reconocedor sintáctico (parser) de modelos lee el texto del modelo y crea una representación interna (IMR) de la sintaxis abstracta para crear los casos de prueba. Utiliza SMT (módulo de teoría de restricciones) para decidir el mínimo número de casos de prueba necesarios (Peleska, 2013).

4.1.3. Pruebas de seguridad basadas en modelos

UML también se ha intentado utilizar para especificar la seguridad que debe tener un sistema. Los diagramas de secuencia son utilizados por un perfil de *UMLsec RBAC (Matulevicius & Dumas, 2010)(Jürjens, 2008)*, el cual permite adicionar información del rol requerido para completar una interacción, así como sus niveles de acceso, en contraste a SecureUML, donde los diagramas de secuencia son derivados a partir de unos diagramas de actividad previamente construidos. UML es un lenguaje muy utilizado en la especificación de software y por ello muchos autores han tratado de usarlo para el MBT tradicional. Sin embargo, los analistas de seguridad no emplean ese tipo de diagramas en su día a día y conceptos como “restricciones OCL” que son comunes en UML, resultan extraños para ese tipo de público.

UML no fue diseñado para describir pruebas (menos aún enfocadas en seguridad). Aunque existen perfiles como U2TP que ha sido estandarizado por OMG desde 2004, las posibilidades gráficas están limitadas por el metamodelo de UML (Dai, 2006). Por ejemplo, no es posible representar tablas de datos, algo que en seguridad es común para especificar entradas de una aplicación. Incluir aspectos de diseño visual por satisfacer el metamodelo de UML en el contexto de seguridad, aumenta la carga cognitiva con aspectos comunes de la especificación de software que son irrelevantes en la especificación de una prueba de seguridad de caja negra. Algunos autores han identificado problemas adicionales con UML, como su falta de consistencia y de semántica para especificar operaciones (Jagadish. S., Lawrence, C, 2008).

Los antecedentes que explícitamente abordan MBST son una proporción baja comparada con otros tipos de pruebas como las de usabilidad o pruebas funcionales en general. Una razón de esta baja proporción es el nivel tan específico de la temática tratada y el hecho de que las investigaciones más avanzadas están orientadas a ofrecer un producto comercial, y por tanto limitan la información disponible. MBST es un tema que podría revolucionar la forma como se pueden detectar vulnerabilidades antes de exponer sistemas al público general (por ende a los atacantes), sin embargo, apenas desde 2011 surgen investigaciones que realmente abordan el aspecto de la seguridad. A continuación se hará un análisis de métodos encontrados en la literatura, los cuales muestran posibles maneras de aplicar MBST.

Los enfoques utilizados en pruebas de caja negra mencionados anteriormente, Fuzzing, escaneadores de vulnerabilidades y MBT, no son excluyentes sino complementarios (Felderer, Agreiter, Zech, & Breu, 2011). Las MBST han sido usadas con éxito para orquestar los otros tipos de pruebas, en “Model-Based Security Testing (MBST)” (Schieferdecker, Grossmann, & Schneider, 2012) se presenta MBT como el eje central que integra los demás tipos de pruebas: funcionales de seguridad, fuzzing basado en modelos, pruebas basadas en el riesgo y pruebas automatizadas.

El artículo mencionado, “Model-Based Security Testing (MBST)”, es uno más de los productos de DIAMONDS[27] de ITEA[28], un proyecto donde se han obtenido resultados sobre la integración del aspecto seguridad al MBT, en casos de estudio correspondientes a diferentes áreas, como por ejemplo Telecomunicaciones (Ericsson) y Tarjetas inteligentes (itrust). Ericsson contaba con muchas prácticas de seguridad que incluían análisis de riesgo y automatización de pruebas, pero el proyecto DIAMONDS pudo complementar en áreas de identificación de pruebas y Fuzzing. El caso de estudio, así como muchos otros aplicados a otras industrias se encuentra en (FOKUS, 2013). Si bien muchos de los documentos son de acceso público, las herramientas desarrolladas no se ofrecen libre y organizadamente. Por ejemplo, en el sitio oficial del proyecto no hay ningún enlace que permita descargarlas y los enlaces mencionados en los documentos

27. *Development and Industrial Application of Multi-Domain Security Testing Technologies*
28. *http://www.itea2-diamonds.org/Overview/results/index.html*

en muchas ocasiones no se encuentran activos[29] o requieren privilegios especiales de acceso[30]. Otras herramientas se ofrecen solo en modalidad comercial[31]. Los pocos desarrollos que se encuentran en buscadores Web, tienen poca actividad y más de un año sin estar actualizados[32] o no cuentan con el código fuente disponible[33]. De los entregables del proyecto, el contenido más directamente relacionado con esta tesis es *WP2 Técnicas*, en el cual se explica de modo amplio el estado del arte en técnicas fundamentales de MBT, agrupadas en 3 categorías: monitoreo e inspección, pruebas activas de seguridad y pruebas basadas en riesgo. Además ofrece un conjunto de patrones con ataques conocidos para la generación de los casos de pruebas.

Dos tipos de pruebas de caja negra destacan para ser utilizadas en el descubrimiento de vulnerabilidades, siendo el primero el *Fuzzing*, que consiste en enviar gran cantidad de entradas, válidas e inválidas, al TOE con el fin de encontrar vulnerabilidades (Guoxiang, 2012). El segundo enfoque se denomina *automatización de intrusión*, y se centra en el uso de herramientas que contienen múltiples casos de prueba para vulnerabilidades que ya se han hecho públicas. En (Bau et al., 2010) se presenta un completo estado del arte sobre automatización de pruebas de seguridad de caja negra. Desafortunadamente los enfoques anteriores no consideran MBT y se limitan solamente a sugerir que para mejorar el resultado de las herramientas analizadas debe modelarse el dominio del problema de la aplicación de un modo adecuado.

Los métodos utilizados en pruebas de caja negra mencionados anteriormente no son excluyentes sino complementarios. De hecho, las pruebas de caja negra basadas en modelos han sido usadas con éxito para orquestar los otros dos tipos de pruebas, como en el caso del proyecto DIAMONDS de ITEA[34] en donde se han obtenido resultados interesantes respecto a la integración del aspecto seguridad en el MBT; igualmente, en (Schieferdecker, Großmann, & Schneider, 2012) se presentan las pruebas basadas en modelos como el eje central que integra los demás tipos de pruebas: fuzzing basado en

29. *http://www-public.it-sudparis.eu/_mouttapp/TestSym.html*
30. *http://www.guersoy.net/knowledge/crema*
31. *Smartesting Certifyit, herramienta para MBT: http://www.smartesting.com/en/product/certifyit*
32. *Fuzzino, un fuzzer: https://github.com/fraunhoferfokus/Fuzzino*
33. *Montimage, herramienta de monitoreo: http://www.montimage.com/en/download.html*
34. *http://www.itea2-diamonds.org/Overview/results/index.html*

modelos, pruebas basadas en el riesgo (subcategoría de MBT), pruebas funcionales (también llamadas pruebas de sistema) basadas en modelos y automatizadas.

MBT no es una aproximación exclusiva para las pruebas de caja negra, en (Sujithra Sriganesh, 2005) se propone un método de pruebas de caja blanca para MBT, pero se reconoce la escasez de enfoques que automaticen la generación de casos de prueba a ese nivel. De hecho, incluso algunos autores se refieren a MBT solo como un conjunto de técnicas para pruebas de caja negra (El-far & Whittaker, 2001).

Los métodos basados en MBT reseñados en la literatura y aplicados para encontrar vulnerabilidades, no cuentan con una comparación o análisis de resultados porque básicamente son los primeros intentos en su especie que buscan mostrar la viabilidad de estas propuestas.

Los diferentes productos de DIAMONDS se agrupan en entregables organizados según la fecha de publicación, siendo D1 documentos liberados en 2010 y D5 en 2013. Además, el trabajo se organiza en los siguientes Working Packages (WP), es decir la categoría conceptual de un producto (publicación) específico:
WP1 - Requerimientos en técnicas y métodos: Se definen las necesidades y alcances abordados por los interesados del proyecto, para aplicación en dominios como el financiero, automovilístico y telecomunicaciones.
WP2 - Técnicas: Se agrupan en 3 categorías: monitoreo e inspección, pruebas activas de seguridad y análisis de riesgo para pruebas basadas en riesgo. Si bien todos los WP ofrecen material importante, a continuación se hace una exploración del WP2, en el cual se explica de modo amplio el estado del arte en técnicas fundamentales de MBT, cuyo contenido está más directamente relacionado con la presente propuesta. Los documentos que conforman a WP2, disponibles públicamente, corresponden a "*Security Testing Techniques*" (Richier, 2011) donde se hace una revisión del estado del arte en métodos usados para pruebas de seguridad, "*Concepts for Model-Based Security Testing*" (Mallouli, 2011) donde se describen los conceptos principales de MBT, "Initial Model-Based Security Testing" (Wotawa, 2012) donde se brinda una aproximación más concreta a los métodos

para MBT y en "*Final Security Testing Techniques*" (Maag, 2013) se ofrece una inspección más exhaustiva incluyendo métricas de su efectividad en casos de estudio.
WP3 - Plataformas de experimentación y herramientas: Desarrollo de 10 herramientas y 2 plataformas de integración.
WP4 - Metodología: Patrones de seguridad y metodología de pruebas basadas en modelos y en el riesgo. Ofrece guías para aplicar las herramientas y técnicas en la práctica.
WP5 - Aplicación de la investigación: Se enfoca principalmente en la enseñanza de las técnicas desarrolladas, por medio de una plataforma Web. La plataforma es de acceso público[35] pero no todos los contenidos están libremente disponibles.

Entre los modelos propuestos por DIAMONDS, se utilizan diagramas UML de estados para expresar las interacciones del atacante con la aplicación y diagramas de secuencia y de actividades para expresar los casos de prueba. Adicionalmente, las entradas a ser probadas se expresan utilizando una gramática formal y los casos de prueba son ejecutados por el framework do.ATOMS.

DIAMONDS en WP2 propone un método más sistemático que el uso de escaneadores de vulnerabilidades. Considera tres aspectos, siendo el primero el monitoreo, en el que se observan de modo pasivo las entradas y salidas del sistema. El segundo es realizar pruebas que interactúan con el sistema y el tercero es enfocar las pruebas por el criterio del nivel de riesgo que puede representar un tipo de ataque.

La diferencia principal con el MBT estándar consiste en que el modelo de comportamiento puede representar simulaciones no definidas en la especificación del SUT y que el objetivo no es realizar pruebas positivas (encargadas de probar funciones) sino por el contrario buscar fallas. Un ejemplo tomado de (Wotawa, 2012) se ilustra en la Figura 3.

35. *http://diamonds.virtues.fi/*

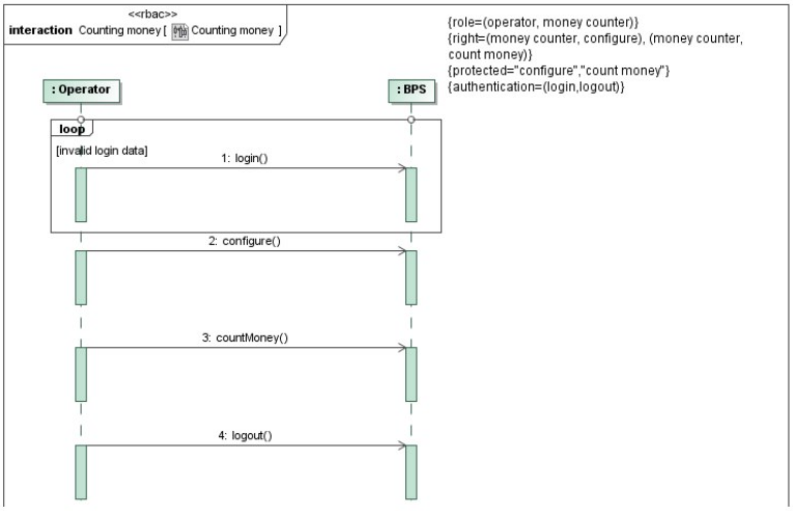


Figura 3: Diagrama de secuencia etiquetado con los roles de los actores.

En el diagrama se aprecia el uso de notación de diagrama de secuencias UML para representar la interacción entre un operador y el sistema. A la derecha puede observarse una notación textual para detallar restricciones de autorización, como por ejemplo el rol requerido para realizar la acción.

Para la generación de los casos de prueba, en WP2 se recomienda el análisis de riesgos utilizando CORAS, una metodología basada en modelos enfocada en la seguridad de sistemas críticos. La ventaja que ofrece sobre otras metodologías de análisis de riesgos, es que ofrece una notación inspirada en UML para representar tres tipos de diagramas: amenazas, riesgos y tratamientos. El analista extrae a partir de este los criterios de riesgo y los incorpora al modelado basado en riesgos. A partir de allí el framework do.Atoms genera los casos de prueba concretos. Entre los diagramas aceptados se incluyen los diagramas de estados, soportados por la herramienta doARTSG2 que facilita el uso del framework. En la investigación referenciada de Wotawa se presenta también un diagrama de actividades en el que se indica que cuando el usuario ingrese un PIN, se aplicará la técnica de *fuzzing* a la entrada para intentar detectar vulnerabilidades en el tratamiento de entradas no esperadas.

En “Model-Based Vulnerability Testing for Web Applications (MBVT)” (Lebeau et al., 2013) se explora un diseño de patrones de ataque basado en diagramas UML de clases, estados y objetos que resultan ejecutables mediante la herramientas propietarias de la empresa Smartesting. Para usar técnicas de MBT en la detección de vulnerabilidades, se requiere no solo adaptar el enfoque de modelado sino también el mecanismo de cómputo de los casos de prueba generados. En el caso de MBVT, el lenguaje de modelado utilizado para describir el comportamiento del SUT es un subconjunto de UML llamado UML4MBT. Como generador de casos de prueba a partir de los modelos se utiliza la herramienta *CertifyIt* de Smartesting. La herramienta misma ejecuta los casos de prueba generados, logrando con ello un proceso repetible a partir de un mismo diseño. Sin embargo, no está orientada a pruebas de seguridad y no es capaz de presentar un modelo a partir de aplicaciones ya construidas, de modo que no sea necesario describir los elementos de la prueba sino solamente los tipos de ataque sobre cada componente.

Para el modelado se utilizan tres tipos de diagramas UML para representar el SUT. Los *diagramas de clases*, al igual que en el MBT tradicional, representan la vista estática (estructura). Los *diagramas de objetos* (anotados con el lenguaje de restricciones OCL) modelan el estado inicial, instanciando los elementos del diagrama de clases para representar información concreta. Los *diagramas de estados* especifican la vista dinámica (comportamiento), modelando la navegación entre las páginas Web. Un ejemplo tomado del artículo de referencia se presenta en la Figura 4.

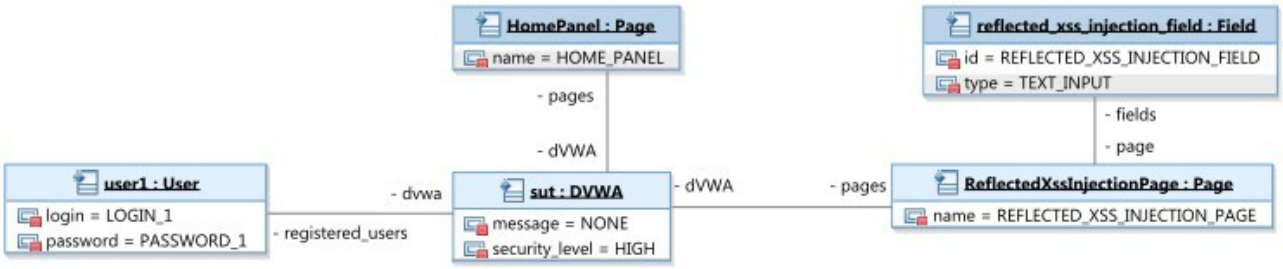


Figura 4: Diagrama de clases, objetos y estados usados en MBVT.

Los modelos poseen un significado preciso y sin ambigüedades y gracias a las restricciones expresadas mediante OCL cuentan con un nivel de formalización esperado

para ser utilizados en MBT, de modo que pueden ser entendidos y procesados por *CertifyIt*. Cada caso de prueba generado contiene una secuencia de acciones de alto nivel, una secuencia de *stimuli,* o instrucciones de bajo nivel a ser ejecutadas y los resultados esperados. Esa metodología guía al analista en el diseño de los casos de prueba permitiéndole razonar a más alto nivel y desacoplando la ejecución de la prueba del diseño de la misma.

(Schieferdecker, Grossmann, et al., 2012) propone utilizar diagramas UML de casos de uso adaptados para modelar amenazas y describe las entradas de la aplicación usadas por el atacante por medio de diagramas de secuencia. Explican modelos de amenazas, fallas y riesgos usando el enfoque CORAS para modelar, pero no llega a ejecutar los casos de prueba. Presenta un conjunto de enfoques sobre técnicas y modelos que pueden ser integradas alrededor de MBT. Incluye las técnicas de pruebas funcionales de seguridad, fuzzing basado en modelos, pruebas basadas en el riesgo y pruebas automatizadas. Se destaca por tener en cuenta patrones con ataques conocidos para la generación de los casos de pruebas. Así mismo, identifica distintos tipos de modelos, arquitectónicos y funcionales, para representar la disposición de los componentes de alto nivel del sistema y las funcionalidades críticas desde el punto de vista de la seguridad. Los modelos basados en riesgo son usados para representar el punto de vista del atacante, las causas y consecuencias de las fallas e ilustrar los aspectos del sistema que requieren controles más robustos. El último tipo de modelos mencionados son los basados en vulnerabilidades, usados para especificar los casos de prueba que deben ser generados según vulnerabilidades públicamente conocidas.

Los diagramas de casos de uso, adaptados para modelado basado en el riesgo, lucen como se muestra en la Figura 5, tomada del artículo referencia. El concepto de modelado basado en riesgo, es análogo a los casos de abuso (*misuse cases*) y modelo de amenazas (*threat modeling*) descritos por otros autores de organizaciones como OWASP y Microsoft (The Open Web Application Security Project, 2013).

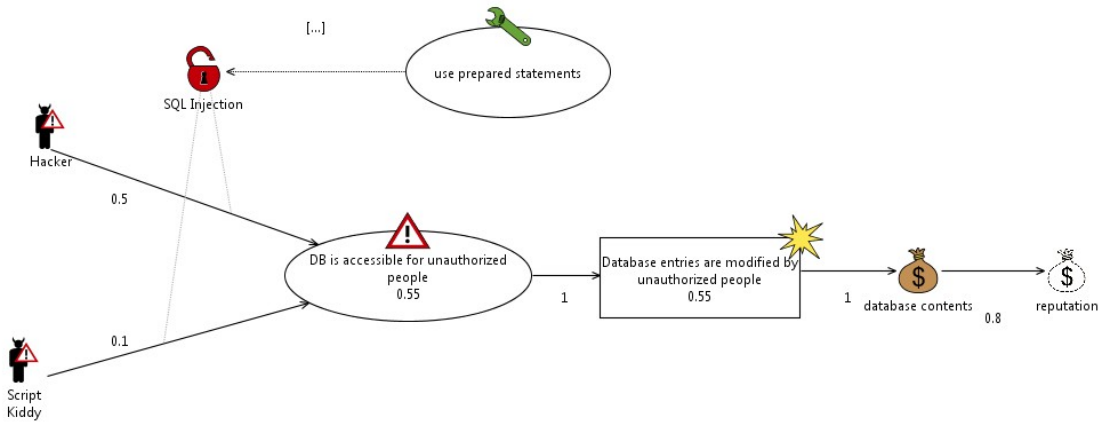


Figura 5: Modelado basado en el riesgo.

El artículo de referencia recomienda también diagramas de secuencia para expresar las entradas a ser mutadas a partir de ataques que aprovechan vulnerabilidades. Las actividades que sigue incluyen la identificación de objetivos y métodos de ataque, el diseño del modelo de pruebas funcionales, la determinación del criterio para generación de casos de prueba, la generación de las pruebas y finalmente la evaluación de las pruebas obtenidas. De acuerdo con el problema de interés, los modelos de amenazas, fallas y riesgos son limitados en el sentido que no emplean una herramienta concreta para generar los casos de prueba.

Los antecedentes revisados exponen una nueva tendencia en la descripción de pruebas de seguridad, la de guiar por medio de modelos la ejecución de los casos de pruebas. Una ventaja que comparten todos esos antecedentes es el alto nivel de abstracción que ofrece el lenguaje de modelado (independientemente de que sea textual o gráfico). También presentan la ventaja sobre los analizadores de vulnerabilidades de proveer un enfoque más sistemático en la concepción de la prueba completa, lo cual contribuye a estandarizar los resultados obtenidos (Stepien & Peyton, 2012). El problema de obtener resultados no uniformes en las pruebas en realidad no es nuevo. En (Suto, 2010b) y su continuación (Suto, 2010a) se muestra la poca uniformidad en términos de resultados entre diferentes escaneadores de vulnerabilidades Web, la principal herramienta de automatización utilizada por los analistas de aplicaciones Web en pruebas de caja negra. Esto se debe a que una sola prueba en MBT puede generar siempre los mismos casos de

prueba y puede ser integrada con más de una herramienta encargada de ejecutarla. En cambio, los escaneadores de vulnerabilidades tienen el problema de no ser compatibles entre sí y de arrojar resultados muy diferentes, tal como lo comprueban investigaciones como (Na Wang, 2011) y proyectos como *sectoolmarket*[36], en el que realizan experimentos que incluyen más de 60 herramientas diferentes. Como resultado importante concluyen que no hay una única herramienta que sea mejor que todas las demás bajo todos los escenarios y que en realidad muchas de ellas se complementan entre sí.

Nótese que los métodos referenciados no ofrecen herramientas libres de fácil acceso. Podría decirse que la orientación de los MBT hacia la seguridad está apenas en sus inicios. En general hacen un buen trabajo en términos de abstraer el nivel sobre el cual el analista puede razonar sobre el sistema, pero carecen de:

- Integración o comparación con las herramientas reconocidas por la industria y usadas por analistas de pruebas de seguridad en su trabajo diario, tal como escaneadores Web de seguridad como OWASP ZAP o W3af. Herramientas como las anteriores (fuzzers y analizadores de vulnerabilidades web), junto con pruebas manuales, es cómo hoy la industria aborda el problema de las pruebas de caja negra, según la experiencia del autor realizando pruebas de seguridad para la empresa FLUID S.A.
- En su gran mayoría, son herramientas de código cerrado (lo cual limita su extensibilidad) o son de difícil acceso (restricciones de autorización para ciertas organizaciones o para clientes).
- Reconstrucción del modelo a partir de la aplicación Web ya desplegada. Actualmente el procedimiento estándar es construir el modelo desde cero.
- El número de ataques soportados no es exhaustivo si se compara con listados de ataques como CAPEC[37] y no sugieren de manera contextual recomendaciones sobre la identificación de patrones de ataque.

La presente propuesta se enfoca en los tres primeros ítems del listado anterior. En la revisión de los antecedentes se encontró información esencial que ahorrará esfuerzo

36. *http://sectoolmarket.com/*
37. *http://capec.mitre.org/*

durante la implementación de la solución propuesta. Se tiene claro que UML puede ser utilizado no solo para documentar arquitecturas de software sino también para modelar las pruebas del software. Puede concluirse que los diagramas más utilizados son máquinas de estado, diagramas de secuencia (Magazine & Testers, 2012) y diagramas de clases. Se han encontrado aplicaciones de MBT al contexto de la seguridad, con resultados incluso de nivel comercial. Sin embargo, con respecto a toda la investigación realizada sobre MBT enfocada en seguridad aplicaciones Web, la proporción de artículos destinados a explorar el aspecto seguridad es pequeña. Los métodos actuales pueden ofrecer ventajas, sobre los escaneadores de vulnerabilidades, en términos de estandarización del proceso de prueba. Sin embargo, no hay evidencia de que detecten al menos el mismo nivel de vulnerabilidades, métrica fundamental si quiere llevarse MBT a la industria. Existen criterios de comparación que ninguno de los métodos antecedentes cumple a cabalidad, lo cual perfila un posible camino para mejoras a futuro. Una hipótesis que surge del estudio es que podría alcanzarse simultáneamente un proceso más repetible de prueba si se usan las pruebas de seguridad basadas en modelos para dirigir de modo consistente el uso de *fuzzers* y escaneadores de vulnerabilidades Web y lograr con ello disminuir la influencia sobre los resultados de la prueba que pueda ejercer el analista que realice dicha prueba, además de facilitar la adopción por parte de usuarios sin mucha experiencia previa. Aunque ha habido varias propuestas para modelar características de seguridad, aún hace falta modelos que se enfoquen en aspectos de seguridad de alto nivel sin obligar a los diseñadores a inmediatamente definir mecanismos específicos de seguridad (F Massacci, Mylopoulos, & Zannone, 2007).

4.1.4. Requisitos de seguridad

Una prueba de seguridad busca faltas de cumplimiento respecto a unos requisitos de seguridad previamente definidos. Entender cuál requisito de seguridad es el incumplido cuando se encuentra una vulnerabilidad en un software ayuda a que el negocio tome decisiones más acertadas respecto a la posibles alternativas de acción frente al riesgo (Landoll & Landoll, 2005):

- Aceptarlo: Asumir el impacto negativo en caso de que ocurra, usualmente cuando el riesgo representado por la vulnerabilidad no es tan crítico comparado al costo de establecer el control que lo mitiga.
- Transferirlo o compartirlo: Por ejemplo comprando seguros.
- Mitigarlo: Establecer controles que reduzcan o anulen por completo el riesgo.
- Evitarlo: Cambiar los planes para no continuar en una vía que conduce a exponerse al riesgo.

Uno de los mecanismos para formalizar el conocimiento en seguridad de la información es utilizando *ontologías*, que pueden ser del tipo *general*, cuando expresan relaciones entre conceptos de alto nivel, o *de dominio,* cuando se limitan a áreas específicas de conocimiento pero alcanzan una mayor cobertura de detalles específicos.

Ontologías generales de seguridad

En (A Souag, Salinesi, & Comyn-Wattiau, 2012) se comparan entre sí varias ontologías de seguridad y se concluye que estas son una buena fuente para la ingeniería de requisitos de seguridad. En (Fenz & Ekelhart, 2009) se busca modelar formalmente el conocimiento del dominio de seguridad de la información, enfatizando en aquellas fuentes que se pueden utilizar para enriquecer el modelo de conocimiento con el aspectos concretos y ampliamente aceptados de la seguridad de la información. Para ello proponen una ontología, mostrada en Figura 6, basada en el modelo de relaciones descrito en la publicación 800-12 del NIST (National Institute of Standards and Technology) Special Publication (NIST, 1995) y a la vez toma controles de estándares de mejores prácticas como el German IT Grundschutz Manual (BSI, 2004) e ISO/IEC 27001 (ISO/IEC, 2005) para lograr incorporar conocimiento ampliamente aceptado.

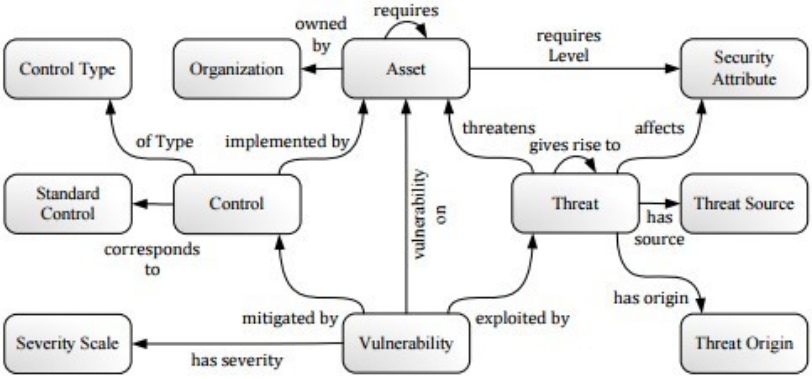


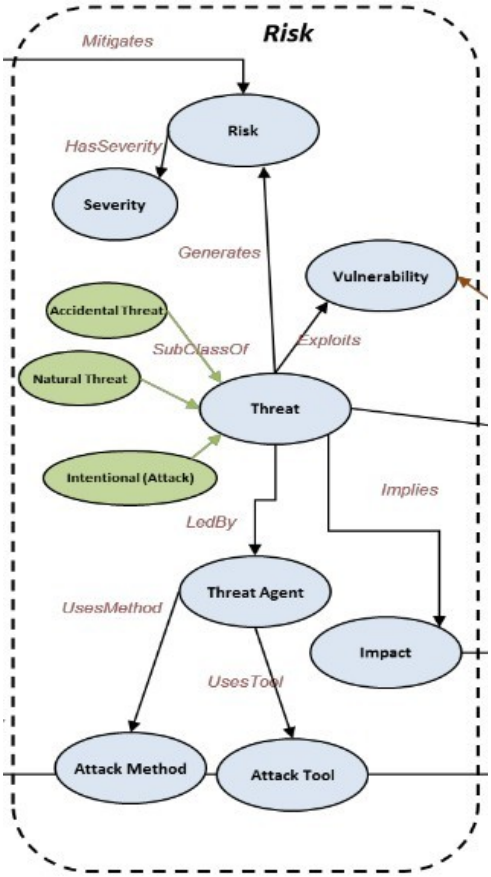
Figura 6: Relaciones y conceptos de una ontología de seguridad basada en modelos del NIST.

En (A Souag, Salinesi, Mazo, & Comyn-Wattiau, 2015) se elicitan requisitos de seguridad a partir de una ontología genérica de seguridad. Las ontologías son una forma conocida de organizar y reutilizar conocimiento, y en el caso de los requisitos de seguridad se hace fundamental dada la dificultad de elicitarlos porque son requisitos no funcionales que generalmente se expresan tácitamente o sin suficiente precisión. La ontología propuesta fue evaluada comparándola con otras en función de su validez y completitud y además se evaluó su usabilidad en un experimento controlado. Uno de los aspectos más interesantes de esta ontología es que ofrece una “meta-vista” para 20 diferentes ontologías de seguridad encontradas en la literatura, logrando establecer una terminología base (validada por 5 expertos) que facilita la comunicación. El artículo citado agrupa los conceptos en tres dimensiones: organizacional, riesgo y tratamiento. Aunque las tres dimensiones son relevantes en la elicitación de requisitos de seguridad, se definen a continuación solo los conceptos de la dimensión de riesgo, que es la más directamente relacionada con las pruebas de seguridad de caja negra.

- *Riesgo (Risk)*: Combinación de una vulnerabilidad y amenaza que causa daño a uno o más de los activos de información. Un riesgo posee una probabilidad de ocurrencia y un impacto asociado.
- *Criticidad (Severity)*: Nivel del riesgo (por ejemplo alto, medio o bajo).
- *Amenaza (Threat)*: Violación de una propiedad de seguridad (por ejemplo confidencialidad, integridad, disponibilidad o trazabilidad). Las propiedades de

seguridad pueden ser también consideradas como restricciones sobre los activos de información. La amenaza puede ser natural, accidental o intencional (ataque).

- *Vulnerabilidad (Vulnerability)*: Debilidad de un activo o grupo de activos que puede ser explotada por una o más amenazas.
- *Impacto (Impact)*: Evento negativo que representa una pérdida para el negocio, por ejemplo sea de imagen ante los clientes, pérdida de disponibilidad o pérdida del control total del sistema de información.
- *Agente de amenaza (Threat agent)*: Actor que lleva a cabo la amenaza. No se usa el término “atacante” porque una amenaza también puede ser involuntaria causada por una persona cualquiera.
- *Método de ataque (Attack method)*: Diferentes métodos utilizados por los agentes de amenaza para lograr sus ataques (por ejemplo suplantación de identidad, método en el que el agente de amenaza se hace pasar por alguien más).
- *Herramienta de ataque (Attack tool)*: Herramienta utilizada para llevar a cabo el método de ataque.



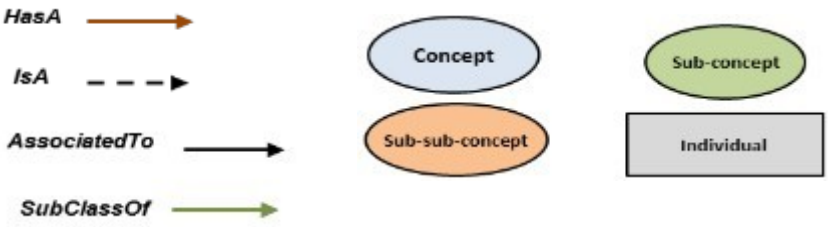


Figura 7: Relación entre conceptos de seguridad de una meta-vista de ontologías de seguridad.

Las relaciones entre los conceptos se clasifican en cuatro tipos: “es un” (IsA), “tiene un” (HasA), “sub-clase de” (SubClassOf) y “asociado a” (AssociatedTo). La Figura 7 tomada del artículo citado muestra cómo se relacionan los conceptos entre sí. Para establecer las conexiones en el artículo usaron como referencia los axiomas definidos en (Fenz & Ekelhart, 2009). Por ejemplo para el concepto de amenaza se tiene la restricción de que “un ataque puede ser intencional o accidental”, que puede expresarse como:

$$\forall x : Amenaza \Rightarrow Amenaza\ \ intencional\ (x) \vee Amenaza\ \ natural\ (x) \vee Amenaza\ \ accidental\ (x)$$

En la Figura 7 puede verse cómo cada uno de los tres sub-conceptos “Amenaza intencional”, “Amenaza natural” y “Amenaza accidental” son una sub-clase de “Amenaza”, lo cual es simplemente una forma alternativa de expresar el axioma anterior.

Ontologías de dominio

En esta sección se presentan ontologías de dominio en el área de la seguridad informática, que al ser más específicas, proveen una aproximación complementaria para el análisis de requisitos de seguridad.

En (Mayer, 2012) se propone un meta-modelo de seguridad para diferentes aspectos de la gestión del riesgo. Se contrastan diferentes lenguajes de modelado, entre ellos *Casos de abuso (Misuse cases)*, *Diagramas de actividad maliciosa (Mal-activity)*, *Marco de abuso (Abuse frames)*, *Extensión de seguridad KAOS (KAOS extended to security)* y *Framework Objetivo-Riesgo Tropos (Tropos Goal-Risk framework)* y *Secure Tropos*. La Figura 8 tomada de (Amina Souag, 2015a) destaca a *Secure Tropos* como uno de los métodos de reuso de conocimiento en seguridad con herramienta de automatización más completos, al incluir reuso con *patrones de seguridad*[38] *(Mouratidis, Giorgini, Schumacher, & Manson, 2003),* ontologías, catálogos y modelos genéricos.

Form of representation: P=Pattern, Tax=Taxonomy, O=Ontology, C=Catalog, GM=Generic Model, T=Template, Pr= Profile, M=Mixed, - =Nul
Reusable knowledge: T=Threats, C=Countermeasures, A=Asset, O= Organization, G=Goal, V=Vulnerabilities, SR=Security Requirements, - =Nul
Technique: FR=Formal Rules, G=Guidelines, P = Process, Q= Queries, - =Null
Automation: N=No, Y=Yes, - =Null

	Methods (re) using patterns			Methods (re) using taxonomies or ontologies								Methods (re) using templates or profiles		Methods (re) using catalogs or generic models						Methods (re) using mixed forms of security knowledge		Methods not (re) using security knowledge								
	KAOS	Secure Tropos	Okubo et al.'s method	GBRAM	Secure Tropos - Si*	RITA	Daramola et al.'s method	Dritsas et al.	Velasco et al.	Salini et al.	Chikh et al.	Zuccato et al.s method	Firesmith	Misuse Cases	Abuse Frames	Security use cases	Saeki & Kaiya Method	SIREN	Secure Tropos	SREP	SQUARE	ISSRM	Secure i*	MSRA	Morda	CRAC++	CORAS	UMLSec	SREF	Secure UML
Form of Representation	P	P	P	Tax	O	O	O & T	P, O	O	O	O	Pr	T	C& GM	GM	C&G M	C&GM	C	C& GM	M	M	-	-	-	-	-	-	-	-	-
Reusable Knowledge	T, C	C, A	A, T, C	G V	G, C, A	T, SR	T, SR	C, A, V, T	A, T, C, V	O, T, SR, A, V	T, O, V, A, SR	-	SR	T, SR	T, A, V	SR	T, O, C	SR	G,T, C	T, SR	T, SR	-	-	-	-	-	-	-	-	-
Technique	FR	G	P	-	FR	G	Q	Q	Q	Q	-	P	P	G	P	G	FR	P	G	-	-	-	-	-	-	-	-	-	-	-
Automation (Reuse tool)	N	N	N	N	N	Y	Y	N	N	N	-	N	N	N	N	N	N	N	Y	N	Y	-	-	-	-	-	-	-	-	-

Figura 8: Comparación de Secure Tropos frente a otras alternativas de reuso de conocimiento en seguridad de la información.

PABRE-Sec es un proyecto en preparación para *Horizon 2020*[39] desarrollado en la Université Paris 1 Panthéon-Sorbonne. El proyecto provee métodos, conocimiento y

38. *Describe un problema de seguridad que se repite, especifica los contextos en los cuales puede existir y presenta un esquema genérico de solución probada.*
39. *Patrocinio de 80 mil millones de euros por parte de la Comisión Europea para fomentar la investigación e innovación entre 2014 y 2020.* https://ec.europa.eu/programmes/horizon2020/en/what-horizon-2020

software para ayudar a las organizaciones a cumplir normas Europeas de seguridad y privacidad, para lograr aumento de la confianza del usuario en los productos y servicios de TIC y proveer infraestructuras críticas más resistentes y servicios. Las tres áreas principales donde se enfoca son:

• Organizacional. Definición de las funciones, tareas y gestión de requisitos y riesgos relacionados con la seguridad.

• Metodológica: Definición de directrices, métodos y estrategias para aplicar la ingeniería de requisitos de seguridad y para gestionar los riesgos relacionados con la seguridad.

• Tecnológico. El despliegue de una plataforma para la permitir la elicitación, estimación de costos, validación y documentación de los requisitos relacionados con la seguridad y la evaluación y gestión de los riesgos asociados.

PABRE-Sec inicialmente ayuda a seleccionar los requisitos de seguridad de la información más relevantes según cada organización teniendo en cuenta las normas de seguridad que le apliquen. En segundo lugar, permite evaluar el nivel de cumplimiento de los requisitos de las normas obligatorias y recomendadas. Utiliza un conjunto de herramientas para identificar posibles vacíos respecto a las mejores prácticas en los requisitos de seguridad de información de los proyectos de una organización específica y a obtener recomendaciones de mejora, así como evaluar el nivel de cumplimiento según la regulación, entregando como resultado un modelo de riesgos de seguridad para el negocio con indicadores y planes de mitigación.

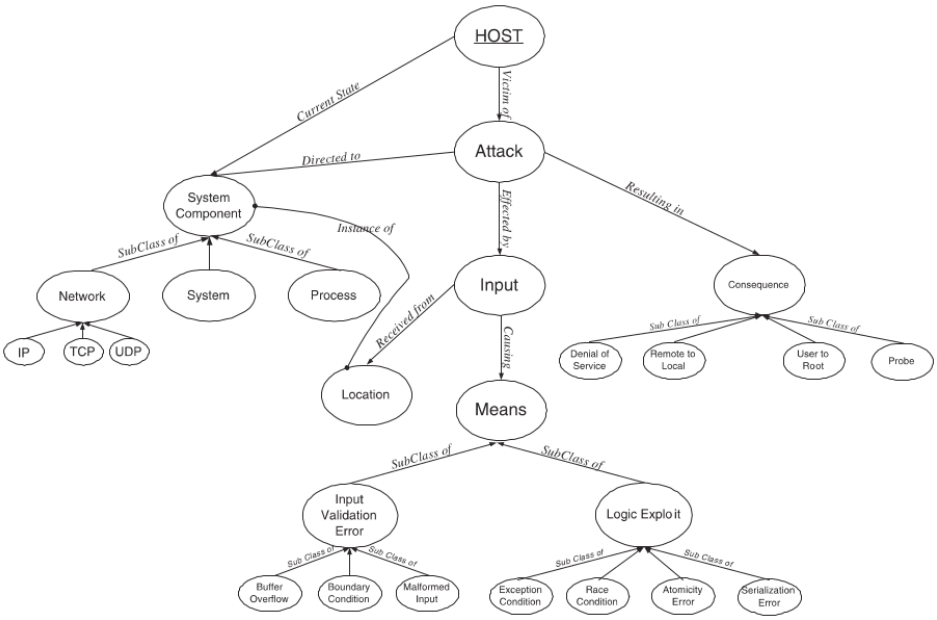


Figura 9: Ontología para ataques informáticos.

En (Undercoffer, Joshi, & Pinkston, 2003) se presenta la ontología de dominio mostrada en la Figura 9. Como puede observarse, esta ontología es más detallada en aspectos técnicos como lo es el tipo de protocolo de red (IP, TC, UDP), sin embargo no está orientada a sistemas Web.

En (Amina Souag, 2015a) se hace una revisión sistemática de 158 publicaciones, entre las cuales se hace una selección de 95 artículos para realizar sobre ellos análisis en profundidad y resume la comparación entre ontologías presentada en la Figura 10.

Family	Security Ontology[1]	Security Objective[1]	Assets[1]	Vulnerabilities[1]	Threats[1]	Countermeasures[1]	Organization[1]	**Requirements[2]**
Beginning	(Mylopoulos et al., 1990)	-	-	-		-	•	•
Security taxonomies	(Avizienis et al. 2004)	•••	-	-	••	••••	-	-
	(Landwehr et al. 1994)	-	-	-	••••	-	-	-
General	(Herzog, Shahmehri, et Duma 2007)	••	•••	••	•••	•••	••	-
	(Fenz and Ekelhart 2009)	••••	••	••••	••••	••••	••••	-
Specific	(Undercoffer, Joshi, Pinkston 2003)	-	•	•	••	-	-	-
	(Viljanen 2005)	•	-	-	-	-	-	-
	(Geneiatakis & Lambrinoudakis 2007)	••	-	-	•••	-	-	-
Risk based	(Ekelhart et al. 2007)	••	-	-	••••	•••	•••	-
Web oriented	(Denker et al. 2003) (Denker, Nguyen, Ton 2004) (Denker, Kagal, Finin 2005)	•••	-	-	-	••	-	••
	(Kim, Luo, Kang 2005)	••••	-	-	-	•••	-	••
	(Vorobiev & Han 2006)	-	-	-	••••	-	-	-
For security requirement	(Dobson & Sawyer 2006)	••	-	-	••	-	-	-
	(Tsoumas & Gritzalis 2006)	-	•••	•••	••	•••	•	•
	(Karyda et al. 2006)	••	•••	-	•••	•	••	•
	(Firesmith, 2005)	-	•	-	•	-	-	••
Modelling	(Mouratidis, Giorgini, Manson 2003b)	•	-	-	-	-	•	•••
	(Massacci et al. 2011)	••	••	-	•	-	-	•••

Figura 10: Comparación entre ontologías (mayor número de puntos indica mayor nivel de soporte).

Puede observarse que la ontología de (Vorobiev & Han, 2006) es la única que contempla la perspectiva de ataques en el contexto Web, pero deja de lado la especificación de vulnerabilidades. La Figura 11 muestra su ontología para ataques en aplicaciones.

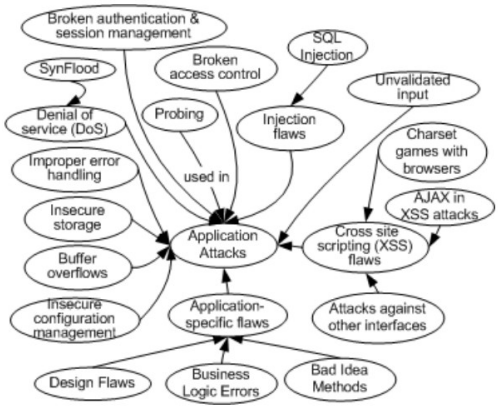


Figura 11: Ontologías para ataques en aplicaciones de (Vorobiev & Han, 2006).

4.1.5. Oportunidades de mejora en el estado del arte actual

La Tabla 3 resume las características de las investigaciones previas que más afinidad tienen con el propósito de este proyecto.

Nombre	**Lenguaje de modelado**	**Transformación a código ejecutable**	**Referencia**
Testing security against chained attacks	Conjunto de conversaciones HTTP. Lenguaje formal ASLan[40].	SATMC (SAT-Based Model Checker for Security-Critical Systems).	(Calvi & Viganò, 2016)
Attack pattern based testing	Especificación de restricciones mediante combinatoria de pruebas (combinatorial testing). Patrones de ataque basado en diagramas UML de estados.	Yakindu, Java, WebScarab.	(Bozic et al., 2015) (Maag, 2013)
Model inference assisted evolutionary fuzzing	Lenguaje textual especificado con gramática BNF.	KameleonFuzz.	(Maag, 2013)
Fault detection	Especificación del comportamiento esperado a nivel de API.	Complemento para GCC.	(Maag, 2013)
Model-Based Vulnerability Testing (MBVT)	Diagramas UML (clases, estados y objetos) para diseño de patrones de ataque.	Smartesting.	(Lebeau et al., 2013)
Model-based security testing from security test purposes and behavioral models	Diagrama UML de secuencia: Entradas del atacante, diagrama UML de estados para interacciones atacante-aplicación, diagramas UML de secuencia y de actividades para los casos de prueba.	MagicDraw, Papyrus, do.ATOMS.	(Schieferdecker & Rennoch, 2013)
TTCN-3 for security testing	TTCN-3 language.	TTCN-3 engine.	(Stepien & Peyton, 2012)
Threat, fault and risk models	CORAS para modelado de riesgos.	-	(Wotawa, 2012)
UMLsec	Especificación UMLsec, una adaptación de diagramas UML de clases y de estados.	UMLsec tool.	(Jürjens, 2008)

Tabla 3: Características del estado del arte

Cada uno de los estudios listados es limitado en al menos dos de los siguientes aspectos:

40. *http://www.avantssar.eu*

- Soporte simultáneo de modelado visual y textual.
- Compatibilidad con estándares como MOFM2T o XMI.
- Generación de modelo inicial a partir del TOE.
- Soporte para tipos de vulnerabilidades adicionales a inyección SQL y XSS.
- Generar casos de prueba avanzados por cada tipo de vulnerabilidad, corroborados en entornos industriales.
- Posibilidad de modelado a nivel de ataques a realizar por entrada.

El aporte novedoso consiste en proponer un lenguaje de modelado, entendible por personas familiarizadas con las pruebas de seguridad, pero que carecen de entrenamiento específico en pruebas basadas en modelos. El lenguaje será validado por medio de una herramienta de código abierto que dirige analizadores de vulnerabilidades de efectividad ya documentada en ambientes industriales.

CAPÍTULO 5. SOLUCIÓN PROPUESTA

Este capítulo explica un lenguaje de modelado para especificar pruebas de seguridad y presenta la arquitectura de Vulnfinder, un prototipo de software usado para especificar las pruebas de seguridad con técnicas de MBT y automatización.

5.1. ESTRATEGIA GENERAL DE LA SOLUCIÓN

El primer paso seguido para implementar la solución fue la *definición del lenguaje de modelado.* Se trata de crear un lenguaje de alto nivel a ser utilizado para modelar las pruebas de seguridad. Esta etapa incluye una *definición del metamodelo* y una e*specificación de la gramática.*

El segundo paso fue definir unas reglas de transformación de modelos que logran producir un código ejecutable con instrucciones específicas para que analizadores de vulnerabilidades interactúen con sitios Web e identifiquen vulnerabilidades en ellos.

En este capítulo se describe una propuesta de metamodelo para pruebas de seguridad y se presenta la descripción de la arquitectura de un prototipo que implementa las ideas discutidas en la investigación. El prototipo será llamado Vulnfinder y aplica técnicas de pruebas dirigidas por modelos y para detectar vulnerabilidades enumeradas en el TOP 10 de OWASP. Vulnfinder será utilizado para ayudar a validar hipótesis respecto a cómo realizar pruebas de seguridad automatizadas y con resultados más estándar que las actuales prácticas de industria.

5.2. LENGUAJE PROPUESTO

El usuario de la solución debe poder diseñar la prueba utilizando un lenguaje de alto nivel. Un lenguaje puede ser textual o visual. En nuestro caso queremos dar soporte a ambos, el visual porque permite acercar la herramienta a usuarios con menos experiencia, y el textual porque permite a usuarios experimentados mayor agilidad en la personalización de sus pruebas.

Para el lenguaje visual, inicialmente se consideró un perfil UML específico para pruebas de seguridad. Sin embargo, se encontró que es limitado para expresar rápidamente relaciones entre cada uno de los ataques realizados a cada campo. Por ejemplo, en la Figura 12 tomada de (Lebeau et al., 2013), se modelan los campos *login* y *password* para el ataque XSS.

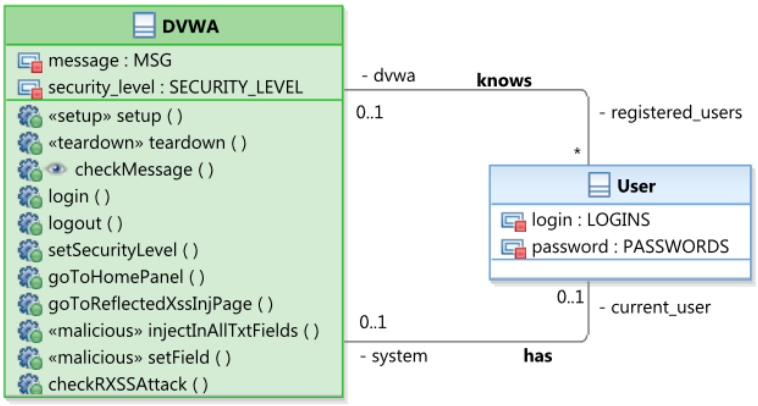


Figura 12: Modelado en UML de ataque XSS para dos entradas

Puede observarse una gran cantidad de detalles presentados en la clase de la izquierda, aún cuando se está modelando un solo ataque. El artículo referenciado, como es habitual en muchos estudios sobre MBST, se enfoca en un solo tipo de ataque, así que no queda claro cómo modelar múltiples simultáneamente. Además el usuario debe especificar muchos detalles de la interacción con el TOE que podrían ser inferidos del propio uso de la aplicación Web. Por ello el lenguaje de modelado propuesto en esta investigación, mostrado en la Figura 13, permite dos tipos de representaciones, uno de clases para

expresar relaciones entre páginas y otro de tablas que permite indicar cuales tipos de

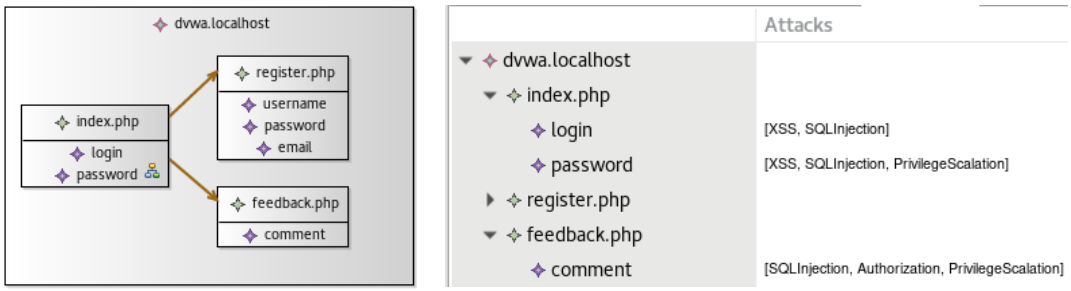


Figura 13: Representación de diagrama de clases y tabla de ataques por campo.

ataques realizar a cada entrada.

La representación en tabla implementada permite ocultar entradas completas para que el usuario ignore las que no sean de su interés en el momento (en la figura las entradas de la página *register.php* están contraídas). Nótese que en la tabla se muestran más campos y más ataques que en la Figura 12, permitiendo que el usuario ingrese solo lo estrictamente necesario para realizar la prueba y no lo que puede ser inferido directamente de la aplicación en ejecución. Además las tablas pueden ser usadas fácilmente para permitir al usuario parametrizar aspectos de la prueba completa, tal como se muestra en la Figura 14.

Figura 14: Configuración de los detalles de conexión con el TOE por medio de una tabla.

Aunque las tablas mostraron su utilidad en la práctica, no son soportadas por UML. Por ello se optó por implementar el lenguaje de modelado visual con metamodelos que son más expresivos que los perfiles tradicionales de UML. Los conceptos a ser modelados se

tomaron de las ontologías revisadas, en particular de (A Souag et al., 2015), (Vorobiev & Han, 2006) y SecureTropos. El metamodelo que permite definir modelos para pruebas de seguridad ha de tener múltiples nombres para conceptos similares (vulnerabilidad, falla de seguridad, debilidad), pues él es una representación abstracta de conceptos. No se enfoca en la terminología o en la forma de representar palabras, sino en el concepto en sí mismo. En el presente documento se hará referencia a las pruebas de seguridad independientes de la plataforma simplemente bajo el término de “modelo” (lo que en otros contextos se conoce como PIT o Platform-Independent Test).

Dentro de las aplicaciones del metamodelo descrito, se encuentra:

a) Crear modelos que no requieren ser pensados desde cero sino que toman como base el modelo navegacional de una aplicación ya desplegada, con ello no solo acelerando la especificación de la prueba sino evitando utilizar documentación que podría estar incompleta o desactualizada.

b) Utilizar el vocabulario común definido en listados de ataques como CAPEC[41] en vez de nombres sin estándar escogidos por el fabricante de cada herramienta.

c) Ofrecer un lenguaje de modelado estándar y abierto que no solo facilita la comunicación entre arquitectos y analistas de seguridad sino que la documentación podrá usarse para probar, con parámetros idénticos, una misma aplicación desplegada en ambientes diferentes (favorece la repetibilidad de una prueba).

d) Una notación y semántica común para expresar conceptos propios de una prueba de seguridad de caja negra también deja abierta la posibilidad a que un mismo modelo pueda orquestar el funcionamiento de más de una herramienta simultáneamente (de aplicación industrial como escaneadores de vulnerabilidades y fuzzers), ganando con ello rigurosidad en los casos de prueba ejecutados y en cobertura de la superficie de ataque probada.

5.2.1. Metamodelo

41. *http://capec.mitre.org/*

Para la especificación del metamodelo se utilizarán los conceptos Criticidad (Severity) para expresar el nivel del riesgo y Método de ataque (Attack method) para hacer referencia a la técnica empleada por el agente de amenaza. Dichos conceptos fueron definidos en ontologías existentes exploradas durante la investigación. Adicionalmente definimos una Prueba de Seguridad (Security Test) como una Prueba (Test) en la que se realiza un conjunto de Ataques (Attack) sobre un grupo de entradas (Input) de la aplicación a ser probada (Target Of Evaluation). Las entradas de una aplicación Web se agrupan en Componentes (Web Component) que pueden ser del tipo Página Web (Web Page) o Servicio Web (Web Service). Las Páginas Web pueden tener enlaces hacia otras páginas Web. Las relaciones entre estos conceptos quedan expresadas en el metamodelo mostrado en la Figura 15.

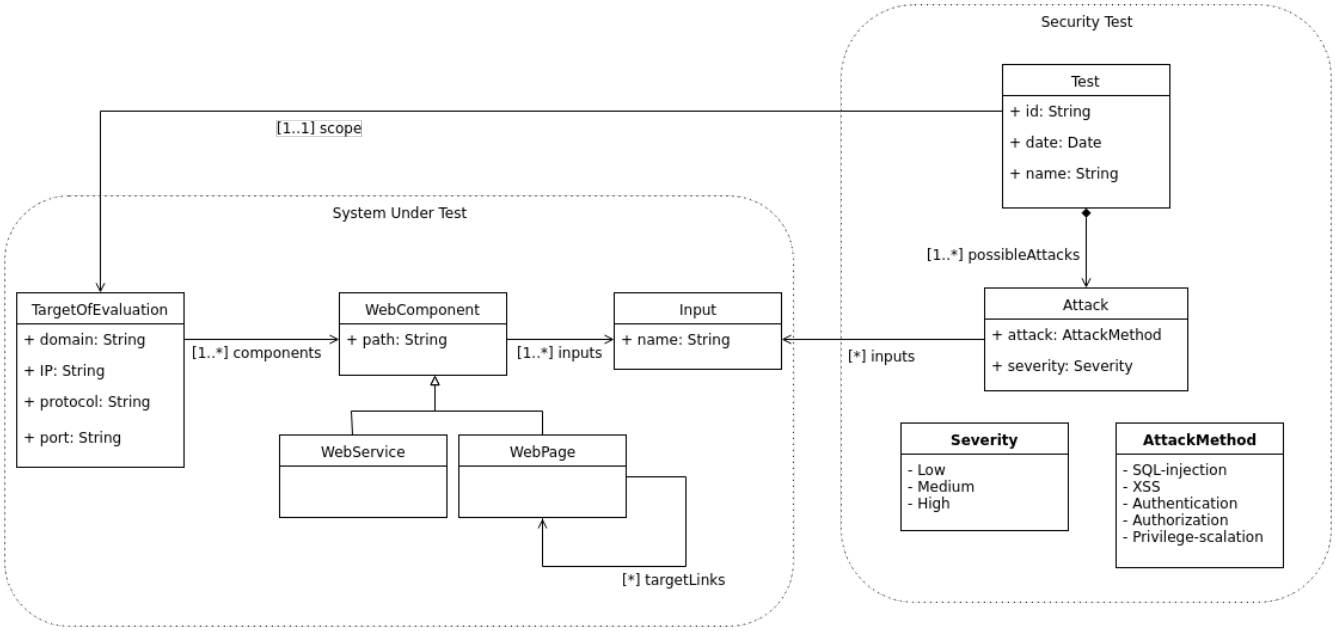


Figura 15: Metamodelo para Pruebas de Seguridad.

5.2.2. Gramática

Las relaciones expresadas en el metamodelo funcionan como restricciones que permiten diferenciar cuándo un modelo no está acorde al matamodelo. Una gramática ofrece las mismas posibilidades para modelos textuales. De hecho nuestra gramática representa

exactamente los mismos conceptos y relaciones que el metamodelo. Expresados en el lenguaje de Xtext[42], un fragmento de la gramática se muestra en la Tabla 4.

```
grammar edu.udea.vulnfinder.gram.SecLanguage with org.eclipse.xtext.common.Terminals

  import "http://udea/vulnfinder/securityTest"
  import "http://www.eclipse.org/emf/2002/Ecore" as ecore

  SecurityTest returns Test:
        {Test}
        '('
        'securityTest' id=EString
              ('(' 'toes' ( scope=TargetOfEvaluation) ')' )?
              ('(' 'attacks' (possibleAttacks+=Attack)+ ')' )?
        ')';
  TargetOfEvaluation returns TargetOfEvaluation:
        {TargetOfEvaluation}
        '('
              ('toe' domain=EString ( components+=WebComponent)* )
        ')';
  Attack returns Attack:
        {Attack}
        '('
              ('attack' name=EAttackMethod ('{' ':level' severity=ESeverity '}')?   )
        ')';
  EString returns ecore::EString:
        STRING | ID;

  WebComponent returns WebComponent:
        {WebComponent}
        '('
              ('page' path=EString ('{' ':target' '[' EString* ']'  '}')? ( inputs+=Input)* )
        ')';
  Input returns Input:
        {Input}
        '('
              ('field' name=EString attacks+=[Attack|ID]*)
        ')';
  enum ESeverity returns ESeverity:
        Low = 'Low' | Medium = 'Medium' | High = 'High'
  ;
```

Tabla 4: Gramática aceptada por Vulnfinder

Un ejemplo del lenguaje reconocido por la gramática se muestra en la Figura 16.

42. *La descripción completa del lenguaje para definición de gramáticas se encuentra en: https://eclipse.org/Xtext/documentation/301_grammarlanguage.html*

```
(securityTest "uni-frankfurt"
    (toes
        (toe "http://iri03.iri.uni-frankfurt.de"
            (page "/")
            (page "/test/index.php?page=login.php"
                    (field "username" ["SQLInjection" "XSS"])
                    (field "password" ["SQLInjection" "XSS" "Authorization"])
                    (field "login-php-submit-button" ["SQLInjection" "XSS"])
                    (field "page" []))
            (page "/test")))
    (attacks
        (attack Authorization { :level High } )
        (attack XSS { :level High } )
        (attack SQLInjection { :level High } )))
```

Figura 16: Modelo textual de una prueba de seguridad.

5.2.3. Prototipo Vulnfinder

Una de las debilidades encontradas en los métodos actuales de MBT aplicados para encontrar vulnerabilidades es que no cuentan con una herramienta de libre acceso que se integre con analizadores de vulnerabilidades y fuzzers ya aceptados por la industria.

Vulnfinder es el nombre del prototipo que se implementó para probar las ideas expuestas en este documento y ofrece un aporte en los siguientes aspectos:

- Incluye un metamodelo propio para aceptar modelos visuales que representen pruebas de seguridad.
- Vulnfinder está diseñado para ser extensible, preparado para conectarse con nuevos analizadores de vulnerabilidades y fuzzers disponibles en el mercado, dado que la transformación se hace de tal manera que permite agregar fácilmente nuevas herramientas de pruebas de seguridad (lenguaje intermedio, luego a lenguaje específico).
- Los casos de prueba utilizados son los que la industria de la seguridad usa en el día a día para encontrar vulnerabilidades Web siguiendo las mejores prácticas internacionales como OWASP.
- Puede llegar a tener cierta adopción por la industria debido a que el modelado de alto nivel facilita el diseño de pruebas y da tranquilidad al analista de ser una

herramienta más exhaustiva que las pruebas manuales habituales[43]. Permitir automatizar pruebas no es algo que sustituya completamente las pruebas manuales, pero sí libera al analista de casos básicos repetitivos para que éste se concentre en los casos críticos no estándar que generalmente son pocos en comparación a toda la superficie de ataque de la aplicación.

- Vulnfinder puede probar aplicaciones Web dinámicas, interpretando el comportamiento de la aplicación en función de la combinación de entradas utilizadas.

Los pasos seguidos para su implementación fueron:

1. Dotar a Vulnfinder de la capacidad de utilizar diferentes herramientas (analizadores de vulnerabilidades y fuzzers) e interpretar los resultados para presentarlos al analista. Por ello se desarrolló teniendo en cuenta la adición de futuros analizadores de vulnerabilidades y fuzzers, no solo los incluidos en el primer prototipo.
2. Seleccionar dos escaneadores de vulnerabilidades de código abierto, inicialmente *ZAP*[44] y *SQLmap*[45]. También se utiliza el fuzzer incluido por ZAP. El criterio de selección fue:
   a) Flexibilidad ofrecida por el API[46] del escaneador.
   b) Calificación obtenida en comparativos de escaneadores Web como SecToolMarket[47] y WASSEC[48].
   c) Inclusión por defecto en la distribución de seguridad Kali Linux[49].
3. Desarrollar capacidad de generación semiautomática de un modelo inicial a partir de una aplicación Web en ejecución.
   - Escaneadores de vulnerabilidades Web contienen ya una funcionalidad de “spider” que se encarga de recorrer automáticamente una aplicación Web para

43. *FLUID S.A (http://fluid.la) y Weepa (http://weepa.co) son empresas que tienen experiencia utilizando analizadores de vulnerabilidades Web y que han decidido probar Vulnfinder porque ven que pueden incorporar las ventajas del MBT sin sacrificar la confianza que tienen en sus analizadores.*
44. *https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project*
45. *http://sqlmap.org/*
46. *Interfaz de programación*
47. *http://www.sectoolmarket.com/*
48. *Web Application Security Scanner Evaluation Criteria*
49. *Es la distribución Unix de seguridad con mejor ranking según http://distrowatch.com/*

determinar la mayor cantidad de páginas y campos existentes en ella. Por tanto en lo que se debe trabajar inicialmente es en controlar dicha funcionalidad por medio del API que expone el analizador de vulnerabilidades.

- Debe escribirse un algoritmo que traduzca los insumos entregados por el spider en modelos de pruebas de seguridad.

4. Desarrollar funcionalidad que permita al usuario complementar el diagrama y modelo textual.
   - Selección de los tipos de ataques que desea realizar.
   - Selección de los campos que desea atacar.
   - Ingreso manual de páginas y campos adicionales.
5. Dotar a VulnFinder de capacidad de ejecución semiautomática de múltiples escaneadores de vulnerabilidades a partir de un solo modelo elaborado por el analista.

5.2.4. Arquitectura propuesta

La arquitectura general de Vulnfinder se muestra en la Figura 17. En secciones posteriores se explicarán cada uno de los módulos.

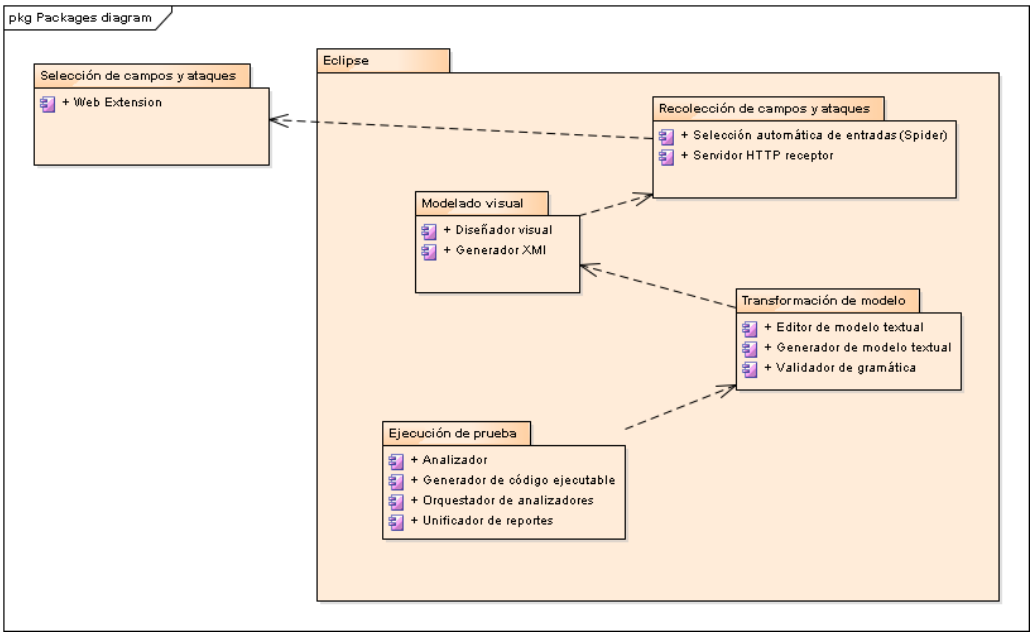


Figura 17: Dependencia entre los componentes de la arquitectura

Puede verse cómo la mayoría de los componentes dependen de la plataforma Eclipse. Es una decisión producto de la revisión del estado del arte en técnicas visuales de modelado. Eclipse una de las alternativas de software libre más completa para modelado visual basado en metamodelos, permitiendo la creación de un entorno de desarrollo integrado para el usuario final. Ha sido utilizado con éxito en recientes plataformas de modelado de seguridad (Amina Souag, 2015b). La principal implicación negativa de la elección de Eclipse es el alto consumo de recursos que se requieren para su instalación y uso. La siguiente alternativa, que aunque no es tan conocida en el medio, permite modelado gráfico con niveles similares de robustez es IDEA MPS (Meta Programming System)[50]. Al ser un entorno de desarrollo integrado también basado en Java, consume recursos similares a los de Eclipse. No se encontró ninguna publicación en la que se utilizara IDEA MPS para modelado de seguridad de software.

5.2.5. Herramientas para definición de lenguajes en Eclipse

50. *https://www.jetbrains.com/mps/*

La definición teórica de un lenguaje de alto nivel para descripción de pruebas de seguridad es en sí mismo un producto de esta investigación. Pero parte de la validación del lenguaje tiene que ver con poder utilizarlo en una herramienta que en verdad encuentre vulnerabilidades en aplicaciones Web. Desde hace al menos 3 décadas ha existido la necesidad de que un usuario sin conocimientos de programación pueda construir diagramas conformes a un metamodelo de entrada y transformar el modelo de alto nivel en código específico (por ejemplo instrucciones de ejecución para una herramienta particular o un programa entendible por un compilador de un lenguaje de programación). Los componentes de software que soportan esas tareas se denominan modelador visual (que permite la construcción de diagramas) y herramientas de transformación de modelo a texto.

Las características que diferencian entre sí a los modeladores visuales son el nivel de detalle que permitan expresar en términos de DSL y la interoperabilidad con otras herramientas no solo de modelado sino también de transformación de modelos y generación de código. La plataforma Eclipse ofrece una de las mejores alternativas respecto a esas características y muchas herramientas de modelado de software y definición de DSL están basadas en ella (Refsdal, 2012). En (Blouin, Moha, Baudry, Sahraoui, & Jézéquel, 2015) se exploran diversas herramientas para la creación de metamodelos y seleccionan el Eclipse Modeling Framework (EMF)[51] para basar el desarrollo de su propia herramienta. ModelSoft[52], un proyecto de investigación que comenzó en 2014 en la Universidad Humboldt de Berlín también seleccionó a EMF como el elemento central de la herramienta producida.

Una forma de especificar lenguajes de modelado visuales es basándose en metamodelos ya existentes, como por ejemplo el de UML. Eclipse soporta ese enfoque por medio de una herramienta llamada Papyrus, que permite crear diagramas incluso con perfiles UML. Evidentemente el metamodelo de UML es demasiado general como para ser últil en muchos DSL. Por ello debe complementarse con restricciones usando el lenguaje OCL (Object Constraint Language).

51. *http://www.eclipse.org/modeling/emf/*
52. https://www.informatik.hu-berlin.de/de/forschung/gebiete/sam/Lehre/modellbasierte-softwareentwicklung-modsoft/vorlesung/DSLs

Cuando se requiere mayor flexibilidad en cuanto a la selección de los elementos de modelado, es necesario utilizar DSL no basados en perfiles UML. Sirius, Graphiti, Xtext, TEF y GMT son meta-herramientas (herramientas que ayudan a construir nuevas herramientas) de notación (Concrete Syntax Development como las llama Eclipse) que pueden ayudar a construir los editores y los analizadores sintácticos. Meta-herramientas de estructura como EMF, meta-lenguajes como OCL, comprobadores SMT (satisfiability modulo theories) y formatos para los modelos de intercambio como XMI sirven de repositorio y comprobadores de cumpliemiento en las restricciones definidas. Generadores de código o transformación de modelo a texto como Acceleo, JET y xTend o transformadores de modelo a modelo como Operational QVT y ATL, son meta-herramientas para la categoría de la semántica.

La Figura 18 está adaptada de (Fischer et al., 2014) y es un resumen de las relaciones mentre meta-herramientas, herramientas y código generado.

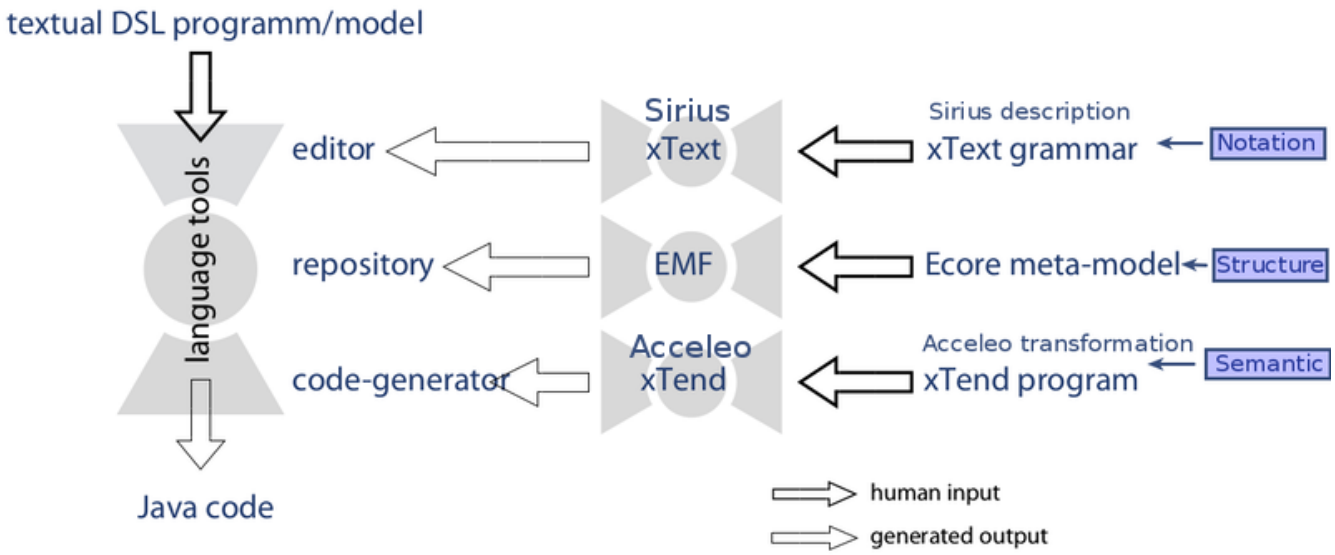


Figura 18: Meta-lenguajes y herramientas para notación, estructura y semántica.

Según el análisis de comunidades MDE realizado conjuntamente entre la Universidad de Nueva York y la Universidad de Manchester (Williams et al., 2014), Sirius[53] es el proyecto

53. *http://www.eclipse.org/sirius/doc/user/general/Modeling%20Project.html*

con mayor actividad de sus usuarios comparado a las otras meta-herramientas de notación visual como Graphiti y GMF, aunque fue solo desde 2013 que empezó a ofrecerse con una licencia de código abierto (el proyecto fue iniciado en 2007). El estudio también concluye que entre todas las categorías de meta-herramientas mencionadas, Xtext[54] y EMF tienen las comunidadades de usuarios más activas. Una de las razones que diferencian a Sirius y Xtext de otras alternativas es que hay empresas que impulsan su desarrollo, Obeo e Itemis respectivamente. El alto nivel de actividad de EMF puede explicarse por ser un componente central de bajo nivel utilizado por muchas de las demás meta-herramientas, incluyendo a Sirius y Xtext. Por ejemplo EMF es quien define el formato Ecore para intercambio de metamodelos.

Una correcta definición del metamodelo es esencial para las etapas de modelado y transformación. El estándar más popular para especificar metamodelos es Meta-Object Facility (MOF)[55] de la organización mundial de modelado OMG[56]. EMF cumple con la parte de la especificación llamada EMOF (Essential MOF). La Figura 19 presenta la arquitectura, la cual tiene tres niveles de abstracción, desde un nivel M0 representando la realidad hasta un nivel M3 usado para delimitar cómo los metamodelos deben ser definidos (es por ello que MOF se describe como un meta-metamodelo). El nivel M2 es el verdero metamodelo y describe las características comunes de cualquier modelo derivado. Esos metamodelos son normalmente representados usando diagramas de clases UML. Finalmente M1 es el nivel donde los modelos son creados por los usuarios que entienden el lenguaje de dominio definido.

Pese a existir una especificación para definir metamodelos, no estamos enterados de la existencia de un metamodelo que siga el estándar EMOF (menos aún MOF) que esté siendo utilizado en herramientas de código abierto para detección de vulnerabilidades.

54. *https://eclipse.org/Xtext/*
55. *Estandar ampliamente adoptado para la espeficación de metamodelos, estandarizado por OMG, la misma organización encargada de la especificación de UML, el lenguaje más utilizado para modelado de software*
56. *http://www.omg.org/mof/*

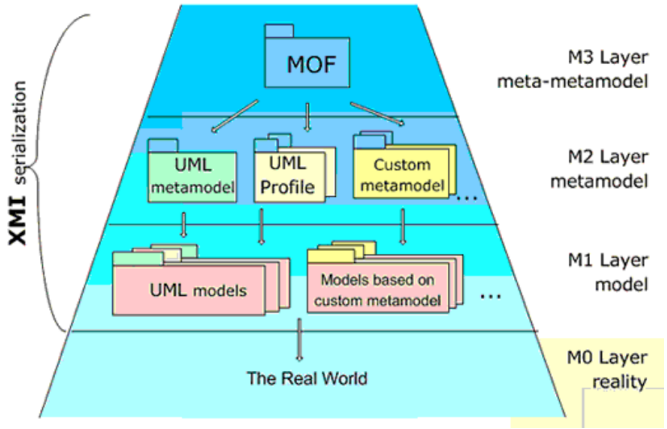


Figura 19: Arquitectura de Meta Object Facility (MOF)

5.2.6. Implementación del prototipo en la plataforma Eclipse

El alcance del análisis elaborado abarca cada uno de los complementos recomendados por el proyecto Eclipse, líder en ofrecer herramientas de modelado de código abierto. Eclipse es un entorno de desarrollo integrado (IDE) que se distribuye en diferentes *bundles*, o conjunto de complementos empaquetados juntos en una sola distribución. Los *bundles* enfocados en modelado y creación de metamodelos y lenguajes visuales son "Eclipse for DSL Developers", con más de 11.000 descargas y "Eclipse Modeling Tools" con más de 10.000. Todas las herramientas a continuación tienen licencia EPL, licencia de código abierto.

Las herramientas analizadas se dividen en dos grupos. La primera categoría es de herramientas que realizan transformación de modelos. Las enumeradas en la Tabla 5 no fueron consideradas debido a que no hacen transformación de modelo a texto.

Logo	**Herramienta**	**Estado de desarrollo**	**Tipo de transformación**

	ATL	Release	Modelo a Modelo
	Operational QVT	Release	Modelo to Modelo

Tabla 5: Herramientas de eclipse que realizan transformación de Modelo a Modelo.

Las herramientas correspondientes a construcción de modelos visuales se presentan en la Tabla 6 y las de transformación de modelos textuales en la Tabla 7.

Logo	**Herramienta**	**Estado de desarrollo**	**Tipo de DSL**	**Tipo de metamodelo**
	Sirius	Release	Visual	Ecore
	Extended Editing Framework	Release	Textual	Ecore
	Graphical Modeling Framework	Release	Textual	Ecore

Tabla 6: Herramientas que están listas para producción y que a la vez permiten desarrollar modelos visuales.

Logo	Herramienta	Estado de desarrollo	Tipo de transformación
	Acceleo	Release	Model to Text
	XPand	Release	Model to Text

Tabla 7: Herramientas que están listas para producción y que a la vez son capaces de realizar transformaciones de modelo a texto.

La herramienta para construcción de editores textuales con autocompletado y resaltado de sintaxis se presenta en la Tabla 8.

Logo	Herramienta	Estado de desarrollo	Tipo de transformación	Formato
	Xtext	Release	Textual	Xtend

Tabla 8: Herramientas que están listas para producción y que a la vez son capaces de construir editores textuales.

Varias de las herramientas ofrecidas por Eclipse no soportaban el estándar de transformación de modelos MOF o no tenían un nivel de madurez suficiente (se encontraban en estado de “incubation” en vez “release” según como lo clasifica el proyecto Eclipse) y fueron excluidas de la evaluación. La Tabla 9 presenta ese listado.

Logo	Herramienta	Estado de desarrollo	Tipo de transformación	Formato
	Eclipse Generation Factories	Incubation	Modelo a Texto	-
	Graphiti	Incubation	Visual	Ecore
	Papyrus	Release	Visual	Metamodelo UML

	MoDisco	Incubation	(sirve para construir DSL a partir de ingeniería inversa sobre código fuente)	-
	OCL Tools	Release	(se integra con Xtext editors para completar documentos OCL)	Metamodelo UML

Tabla 9: Herramientas descartadas por no representar metamodelos MOF o por no están en estado de producción.

EMF está compuesto por un grafo de objetos, mecanismos de serialización y deserialización de modelos y un editor en forma de árbol para manipular los modelos. Sobre EMF se han construido otras herramientas de modelado visual más avanzadas:

- GEF / Draw2D: Permiten la creación de editores gráficos. Para ser utilizado requiere que el usuario tenga conocimientos en Java, XML y la arquitectura de plugins de Eclipse.
- GMF / Graphiti: Oculta complejidades de GEF, pero sigue siendo necesario mucho conocimiento de Java y de los plugins de Eclipse.
- Sirius: Basado en GMF, provee rápido desarrollo de lenguajes de modelado[57] sin necesidad de configurar directamente las tecnologías subyacentes.

Sirius es capaz de representar cualquier modelo compatible con EMF, entre ellos modelos UML, SysML o TOGAF. Pero su característica principal es la capacidad de crear editores gráficos para DSL que cumplen con un metamodelo, es decir que permite la definición de lenguajes de modelado visuales facilitando muchas tareas habituales requeridas por MDE. La herramienta de modelado construida con Sirius puede desplegarse como una distribución de Eclipse independiente o como plugin. Sirius garantiza que un modelo sea consistente con su metamodelo. De hecho desde la misma paleta de componentes

57. *Ese lenguaje de modelado es un DSL, que en el contexto de MDE se denomina Domain-Specific Modeling (DSM). Sin embargo, dentro de la literatura de MBT el término más usado es simplemente lenguaje de modelado.*

visuales restringe operaciones que no tienen sentido de acuerdo al metamodelo. Los componentes principales de Sirius se describen a continuación:

- Elementos de construcción: Incluyen a GEF, Draw2D, contenedores (elemento que tiene sub-objetos) y relaciones entre objetos.
- *Representation*: Ofrece una perspectiva de modelado para la construcción gráfica de elementos del dominio. Por defecto Sirius ofrece algunos dialectos, o clases de representaciones, y por medio de programación pueden agregarse nuevos. Los dialectos soportados van más allá de lo permitido por UML, incluyendo no solo diagramas sino también tablas y árboles. Los archivos con extensión *.aird* contienen los datos necesarios para mostrar las representaciones pero sin semántica (la cual es almacenada en los propios modelos).
- *Viewpoint*: Presenta una forma específica de ver un modelo y está definido por un conjunto de *representations*. Puede crearse más de un *Viewpoint* para un mismo modelo, de modo que diferentes usuarios vean solo los fragmentos relevantes para cada uno. Se almacenan en un *Viewpoint Specification Model (VSM)*, archivos con extensión *.odesign*.
- *Mapping*: Determina la correspondencia entre las partes del modelo con los elementos visuales mostrados al usuario.
- *Style*: Define la apariencia visual de los elementos gráficos.
- *Section*: Usado para crear nuevos componentes de modelado que permitan por ejemplo modificar atributos o crear instancias y relaciones.

El insumo inicial para construir una herramienta de modelado con Sirius es el metamodelo (archivo con extensión *.ecore*), allí se especifican las restricciones que deberá cumplir cualquier modelo definido con la herramienta. Los modelos creados son serializados (almacenados en un archivo) por Sirius en el formato estándar XMI que permite el intercambio con herramientas creadas por terceros. Los modelos podrán tener cualquier extensión de archivo definida por el diseñador de la herramienta de modelado.

XMI es un formato diseñado para ser utilizado por herramientas, no para interacción textual directa por parte de los usuarios. XMI se basa en XML, formato conocido por ser

poco compacto (requerir muchos símbolos para expresar ideas simples) y muchas personas según su preferencia de lenguaje de programación optan por otros formatos como anotaciones en Java, EDN en Clojure o JSON en Javascript. Puede ser conveniente entonces transformar el modelo en XMI a un nuevo lenguaje más compacto y fácil de manipular directamente a nivel textual. Una forma de definir lenguajes es utilizando gramáticas, mecanismo soportado por varias herramientas incluso dentro de Eclipse como es el caso de Xtext.

Para convertir el modelo XMI en el nuevo lenguaje definido debe realizarse un proceso de transformación de modelo a texto o M2T con herramientas como Acceleo[58]. De hecho Sirius utiliza *Acceleo Query Language (AQL)* para referirse a los nodos del modelo en las llamadas "expresiones candidatas de semántica". Esa compatiblidad entre Sirius y Acceleo se debe a que ambos son creados por la misma empresa (Obeo). Acceleo usa un lenguaje de transformación basado en Java y además es compatible con el estándar MOFM2T de OMG que dio mucha fuerza a los enfoques basados en plantillas en vez de a los "visitor based" (Ogunyomi, Rose, & Kolovos, 2014).

Las plantillas son una forma de definir las reglas de transformación de modelos que ofrece simplicidad cuando gran parte del código resultante es estático. Consisten en definir la estructura general de la salida junto con las secciones que serán reemplazadas. Cuando en cambio el código de salida es altamente variable o se realizan transformaciones de código a código puede ser más apropiado el enfoque "visitor based", que consiste en recorrer el modelo fuente generando texto para un subconjunto de los elementos encontrados. También existen aproximaciones híbridas que combinan ambos enfoques. Acceleo trabaja por medio de plantillas (archivo con extensión *.mtl*), las cuales especifican qué secciones permanecen estáticas y cuáles serán transformadas. Por medio de expresiones se indica cómo cambiar cada sección a ser transformada en una salida. Las expresiones interpretan el metamodelo de entrada para extraer la información correcta del modelo para transformarlo en cualquier clase de código, incluso en instrucciones ejecutables por otros programas. Xtend[59] es otra herramienta que usa

58. *https://eclipse.org/acceleo/*
59. *http://www.eclipse.org/xtend/*

plantillas para realizar transformaciones, similar a Acceleo, y fue construida utilizando Xtext.

Xtext también es una meta-herramienta que permite definir lenguajes. La diferencia con Sirius está en su enfoque a lenguajes textuales y la necesidad de especificarlos por medio de gramáticas. Existen dos escenarios diferentes para crear el nuevo lenguaje, según se cuente con un metamodelo o no antes de empezar a definir la gramática. Cuando no se tiene el metamodelo, una persona puede crear la gramática con mayor libertad y posteriormente Xtext generará automáticamente el metamodelo que no restringe de ninguna manera la definición del lenguaje, sino que busca representar de modo diferente modelos compatibles con el lenguaje. Cuando en cambio se tiene previamente un metamodelo, puede usarse para generar automáticamente una gramática inicial, que no siempre es perfecta y generalmente debe ser modificada.

En cualquiera de los casos Xtext verifica que la gramática esté de acuerdo con el metamodelo (en este contexto el metamodelo funciona como un agente que define estructuras y restricciones lógicas). Una vez se tiene la gramática correcta, Xtext creará automáticamente un editor con autocompletado y resaltado de sintaxis capaz de validar que los modelos ingresados por un usuario cumplen o no con las restricciones definidas por la gramática y el metamodelo.

5.3. PROCESO DESDE DISEÑO HASTA EJECUCIÓN

5.3.1. Proceso

El proceso desde que se concibe la prueba de seguridad hasta que se encuentran las posibles vulnerabilidad puede resumirse en cuatro actividades principales mostradas en la Figura 20:

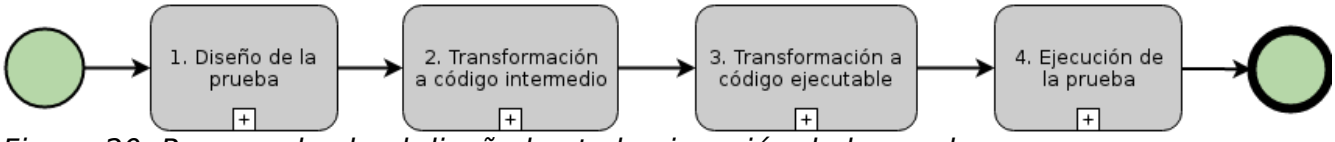


Figura 20: Proceso desde el diseño hasta la ejecución de la prueba.

En “*Diseño de la prueba*” Vulnfinder presenta al analista una versión modificada del TOE en la cual se pueden seleccionar las entradas a ser probadas y los ataques respectivos con un nivel de severidad que determina cuantos y qué tan avanzados serán los casos de prueba utilizados para detectar la presencia de vulnerabilidades. Mientras mayor sea el nivel de severidad más tiempo tomará después la ejecución de la prueba y mayor probabilidad existirá de causar impactos no deseados en sistemas críticos generalmente relacionados a una denegación de servicio. La forma clásica de mitigar esas consecuencias negativas es haciendo pruebas en ambientes de prueba en vez de la versión de producción. Otra medida es con monitoreo detallado, pruebas funcionales automatizadas y respaldos frecuentes, de modo que ante cualquier incidente, pueda retornarse el sistema a la última versión funcional correcta.

Permitir un mayor nivel de severidad aumenta la probabilidad de detección, así que es una decisión que el analista deberá tomar según el contexto en el que se realiza la prueba. La entradas seleccionadas por el analista se utilizan luego como insumo para que un componente llamado “spider” encuentre entradas adicionales en el sitio que pueden ser difíciles de identificar a simple vista (por ejemplo campos ocultos de HTML). Todas estas entradas son presentadas utilizando un modelo de alto nivel utilizando un DSL gráfico, similar a UML, pero más flexible en el sentido que permite por ejemplo representar tablas de datos que simplifican la presentación de información.

En la actividad “*Transformación a código intermedio*” se traduce todo el modelo gráfico a un DSL textual. Un analista experimentado y conocedor de este lenguaje textual puede incluso diseñar la prueba directamente sin pasar por de la creación del modelo visual. Un ejemplo de modelo textual de una prueba para el sitio www.udea.edu.co fue mostrado en la Figura 16. En la prueba se intentan, entre otros ataques, un XSS (con nivel de severidad High) y SQL Injection (con nivel de severidad Low) en el campo *user* de la página *login.php*.

En la etapa de “*Transformación a código ejecutable*” Vulnfinder transforma los ataques descritos en el modelo de la prueba a instrucciones específicas para cada uno de los analizadores de vulnerabilidades soportados. Por ejemplo para la herramienta *SQLmap*

produciría el comando:

sqlmap --url 'http://www.udea.edu.co/login.php&user=val' --data 'user=val' –banner –batch --crawl=5

En la actividad final denominada "*Ejecución de la prueba*" se utilizan interfaces de consola o de servicios Web de cada uno de los analizadores de vulnerabilidades y se realiza un consolidado en un solo informe de las vulnerabilidades reportadas por todos ellos.

La Figura 21 describe en más detalle la relación entre las diferentes actividades.

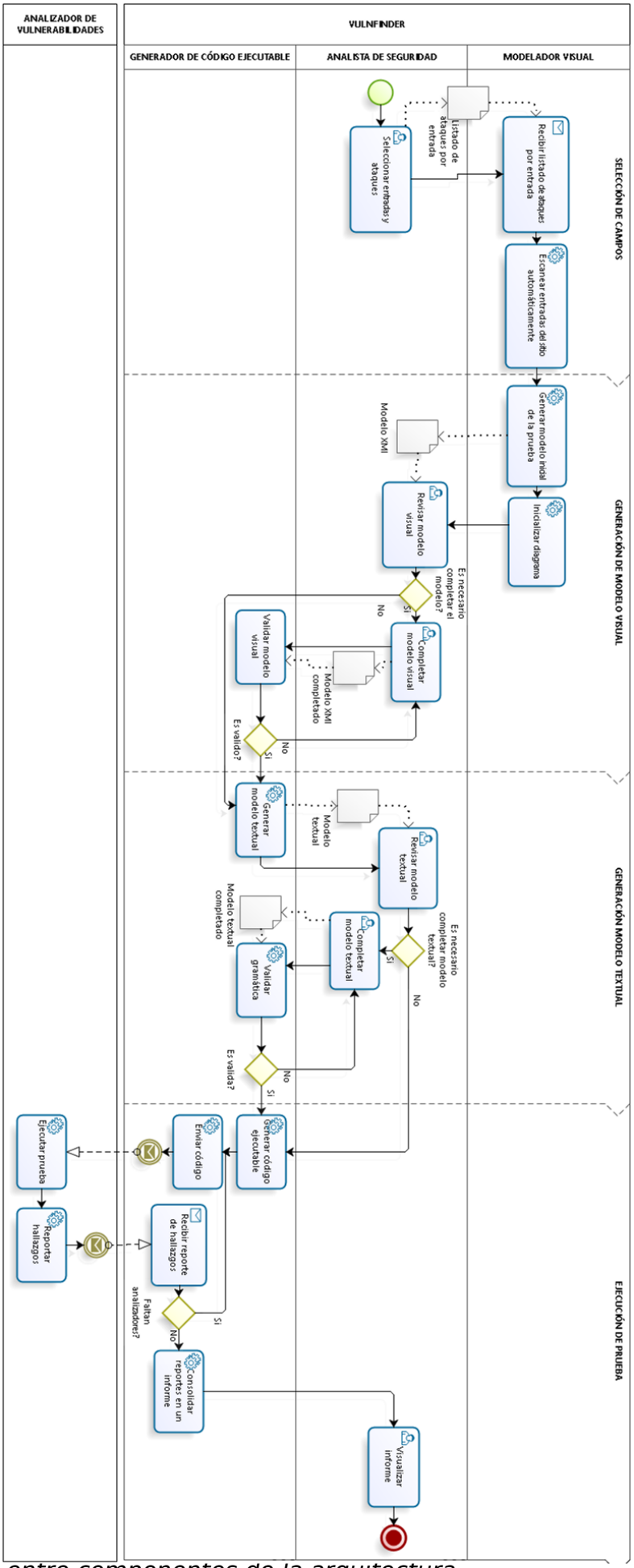


Figura 21: Interacción entre componentes de la arquitectura.

Los diferentes componentes de Vulnfinder pueden agruparse en dos partes: Modelado y Transformación. Estas serán explicados en las dos siguientes secciones.

5.3.2. Componentes de la sección de modelado

Esta sección corresponde a la etapa indicada en la Figura 22.

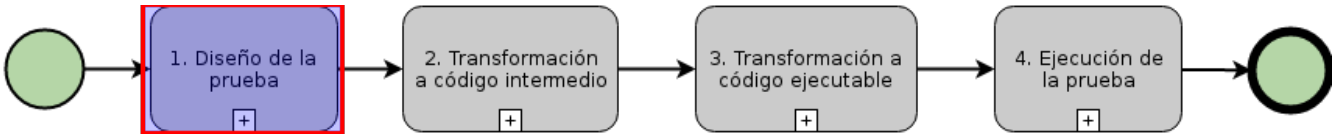


Figura 22: Primera etapa del proceso.

Los componentes de la sección de modelado se listan a continuación y se ilustran en la Figura 23.

- *Web Extension:* Es un componente que se ejecuta en el cliente (navegador Web), encargado de transformar las respuestas del TOE para presentar campos visuales al usuario para que este indique cuáles de las entradas desea probar.
- *Recolector de datos*: Define y valida el formato para intercambio de información entre el *Web Extension* y los módulos del lado servidor. Entre los campos que define se encuentran el dominio de la URL, en nombre de la acción del formulario y el campo *referer* de cada petición. Este componente interpreta la selección del usuario y la procesa para ser utilizada luego como insumo en el componente *Selección automática de entrada.*
- *Selección automática de entrada (Spider)*: A partir de la selección de entradas del usuario se coordina al *Spider* para encontrar campos no identificados por el usuario.
- *Generador XMI*: Procesa el modelo de entrada que representa la prueba para generar un modelo de salida.
- *Diseñador visual*: Con las entradas seleccionadas, se construye un modelo visual de navegación entre páginas y se presenta la posibilidad de seleccionar los ataques para cada entrada. Internamente incluye una instancia de *Sirius*

(diseñador visual): Valida la coherencia del modelo contra el metamodelo base de pruebas de seguridad.

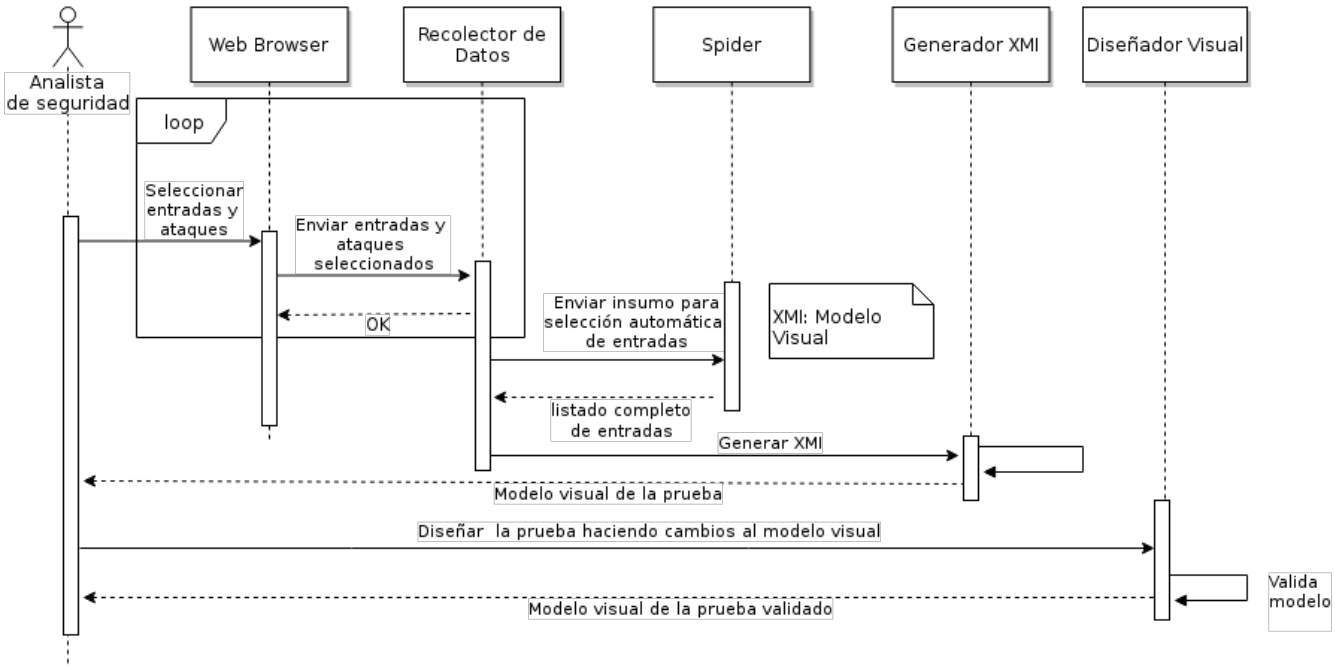


Figura 23: Interacción del analista con los componentes de la sección de modelado.

5.3.3. Componentes de la sección de transformación

Esta sección corresponde a la etapa indicada en la Figura 24.

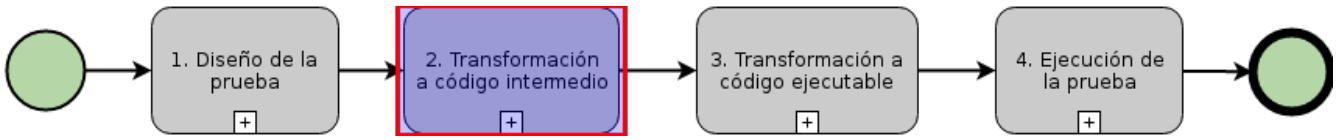


Figura 24: Segunda etapa del proceso.

Los componentes de la sección de modelado se listan a continuación y se ilustran en la Figura 25.

- *Editor y validador de gramática:* Herramienta que permite ingresar el modelo textual de la prueba, diseñado a partir de gramáticas con Xtext.

- *Transformación a código intermedio (generador modelo a texto)*: A partir del modelo de salida, construye el texto que representa las instrucciones que dirigen el funcionamiento de diferentes analizadores de vulnerabilidades y fuzzers.
- *Orquestador de analizadores*: Prioriza y comparte hallazgos entre diferentes fuzzers, analizadores de vulnerabilidades y spider. Materializa el lenguaje intermedio producido por en el *Generador XMI* para llevarlo a casos específicos.
- *Unificador de reportes*: Combina los resultados de cada herramienta para proveer un solo reporte con todas las vulnerabilidades encontradas.

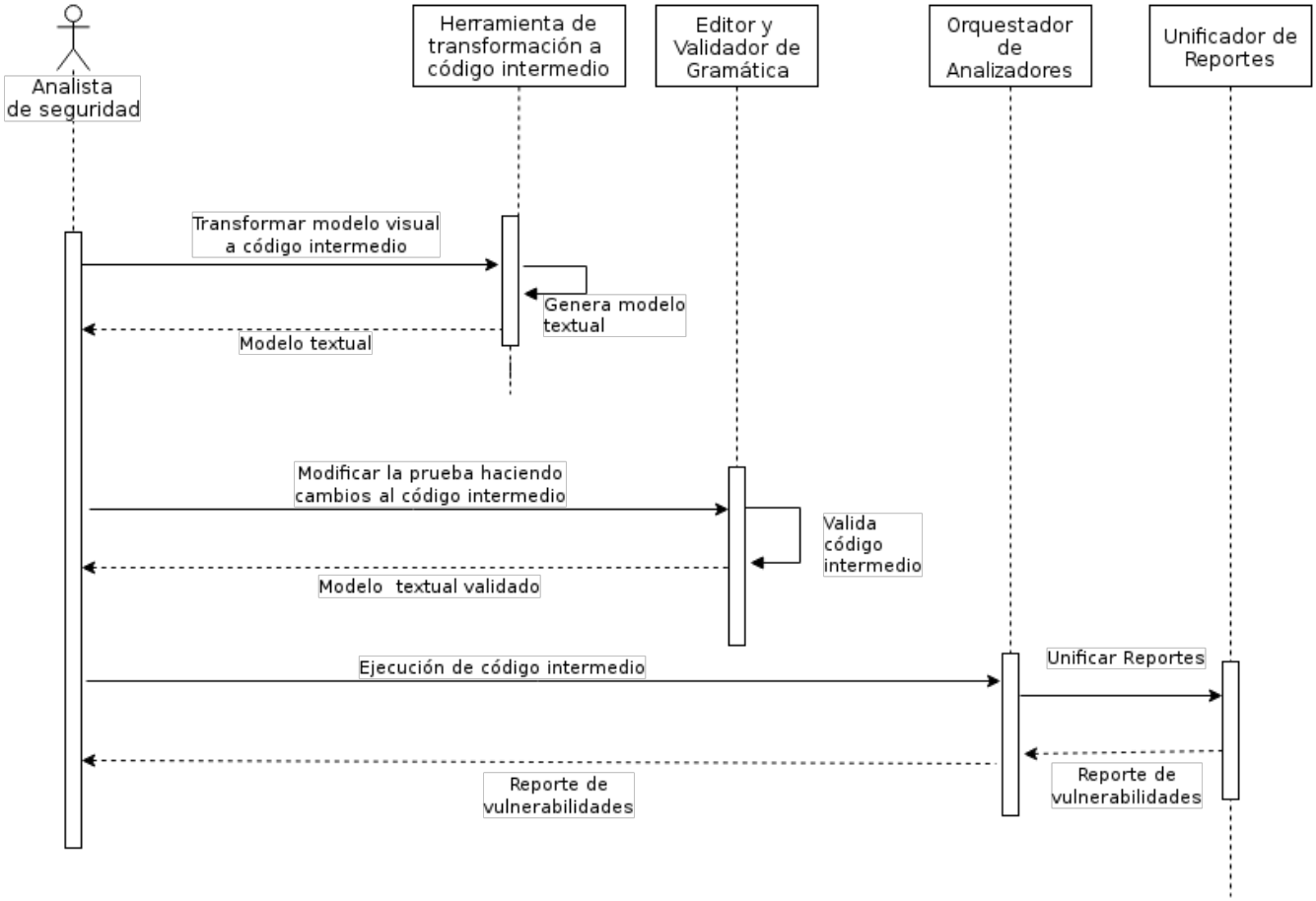


Figura 25: Interacción del analista con los componentes de la sección de transformación.

5.3.4. Estrategia general de la transformación de modelo a código ejecutable

Esta sección corresponde a la etapa indicada en la Figura 26.

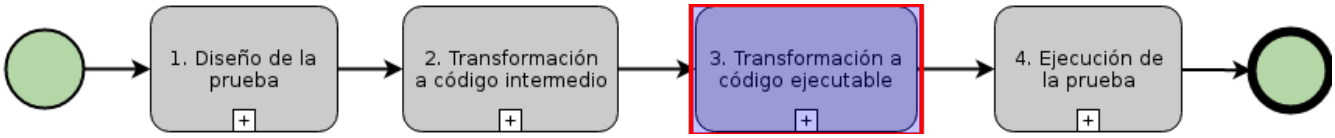


Figura 26: Tercera etapa del proceso.

En (Reina Quintero, 2011) se presentan los elementos que conforman el proceso de transformación de modelos, ver Figura 27.

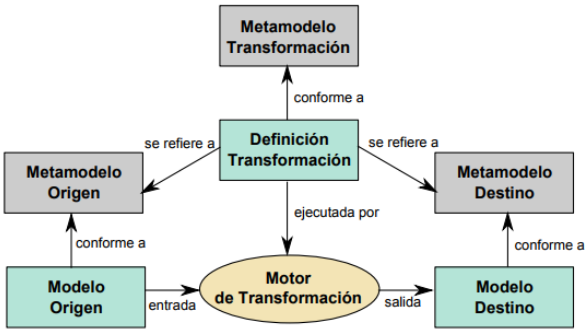


Figura 27: Elementos de transformaciones de modelos.

En el caso de Vulnfinder se requiere soportar dos transformaciones, una de *modelo a texto* y la otra de *texto a código ejecutable para herramienta específica.* La primera utiliza un metamodelo general como origen mientras que la segunda una gramática. Los metamodelos de las transformaciones son directamente provistos por las herramientas de Eclipse (Acceleo y Xtext). La transformación de modelo a texto sigue el estándar MOFM2T[60]. Un diagrama de los elementos de las transformaciones en el contexto de Vulnfinder se muestran en la Figura 28.

60. http://www.omg.org/spec/MOFM2T/1.0/

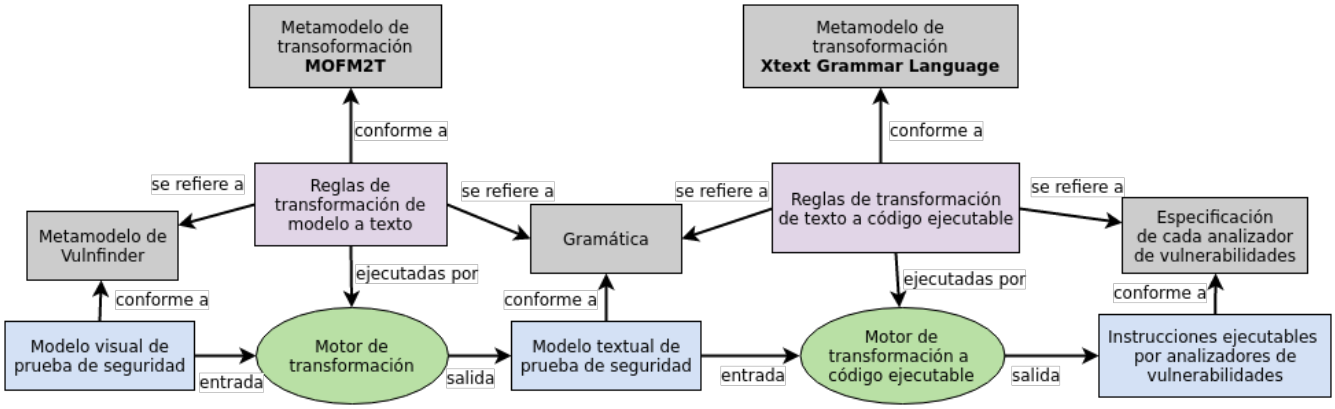


Figura 28: Elementos de las transformaciones de modelos de Vulnfinder.

Para pasar desde un modelo de alto nivel hacia un código ejecutable para cada una de las herramientas, se decidió dividir el proceso en dos etapas:

- *Transformación de modelo a texto:* En esta fase se convierte el modelo de alto nivel y visual especificado por el usuario, en un modelo textual que representa la misma información pero en un formato que es más fácilmente validable por una gramática formal. La representación textual es un punto de partida más cercano al resultado final esperado por cada herramienta, lo que facilita que en etapas posteriores la lógica específica de cada analizador de vulnerabilidades pueda estar encapsulada en módulos individuales.
- *Transformación de texto a código ejecutable para herramienta específica*: Cada herramienta usa su propio lenguaje, es por ello que se requiere una transformación especial por cada una. Por cada analizador de vulnerabilidades o fuzzer se diseña un adaptador que traduce el modelo textual en el código ejecutable.

La Figura 29 ilustra un ejemplo de una transformación desde un modelo con un ataque *SQL Injection* para el campo *user*.

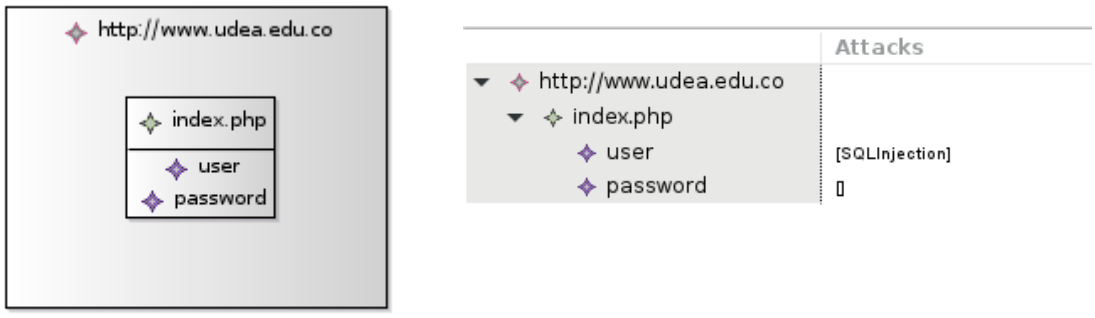


Figura 29: Modelo de prueba de seguridad.

Al realizarse la transformación se obtiene el modelo textual mostrado en la Figura 30. Puede observarse cómo están presentes diferentes elementos del metamodelo como por ejemplo:

- TOE: http://www.udea.edu.co
- Page: index.php
- Field: user
- Attack: SQLInjection

```
(securityTest Example
    (toes
        (toe "http://www.udea.edu.co"
            (page "index.php"
                    (field "user" [ SQLInjection ])
                    (field "password" [])
            )
        )
    )
    (attacks
        (attack SQLInjection { :level Medium } )
    )
)
```

Figura 30: Modelo textual.

Acceleo es el programa encargado de la transformación. La sintaxis principal que usa se resume en las siguientes sentencias:

- ***template****:* Para especificar una nueva plantilla.
- ***comment***: Comentario a ser ignorado durante la transformación.
- ***file***: Archivo destino con el resultado de la transformación.
- ***for***: Instrucción de iteración.
- ***if***: Instrucción de condicional.
- ***atributo.campo***: El punto (.) es la operación de acceso a uno de los miembros de un objeto definido en el metamodelo.
- ***[sentencia] cuerpo [/sentencia]***: Indica que la sentencia será ejecutada aplicando al *cuerpo* especificado.

Para una explicación exhaustiva de toda la sintaxis, consulte la documentación oficial en Eclipse[61]. A continuación se explican las porciones más importantes de las reglas de transformación para Acceleo.

*[**module generate**('www-udea-edu-co.scan')]*
Especifica archivo html destino con los resultados del escaneo.

```
pages = {
[for (dominio : DominioWeb | anEscaneo.dominios)]
        [for (pagina: Pagina | dominio.paginas)]
                "[dominio.nombre/]/[pagina.ruta/]" : {
                        "params" : {
                                [for (entrada : Entrada | pagina.entradas)]
                                [let atts : Ataque = entrada.ataques]
                                "[entrada.nombre/]" : { "value": "[entrada.valor/]", "tests": ['['/][atts.SQL =
'Si'/], [atts.fuzzer = 'Si'/], [atts.XSS = 'Si'/], [atts.authenticationAndSession = 'Si'/], [atts.auth = 'Si'/],
[atts.generalInjection = 'Si'/][']'/]},
                                [/let]
                                [/for]
                        },
                        "cookies" : {
                                [for (sess : COOKIE | pagina.cookies)]"[sess .nombre/]" : { "value": "[sess
.valor/]"[comment], "tests": ['['/]1, 2, 3[']'/][/comment]}[/for]
```

61. *https://wiki.eclipse.org/Acceleo/*

```
                    }
                }[if dominio <> anEscaneo.dominios->last() or pagina <> dominio.paginas->last()],[/if]
        [/for]
[/for]
}
```

Para cada página dentro del dominio elegido por el usuario, se enumeran todas las entradas.

5.3.5. Ejecución de la prueba

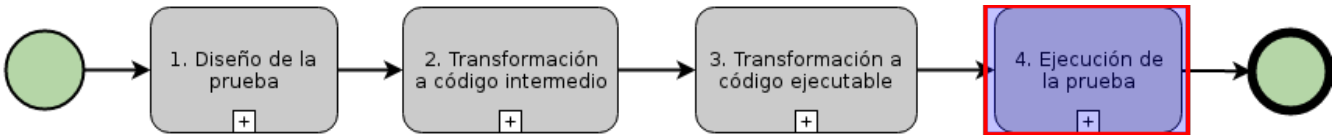


Figura 31: Cuarta etapa del proceso.

Esta sección corresponde a la etapa indicada en la Figura 31.

Se quiere ahora lograr la transformación de texto a comandos a ser ejecutados por la herramienta SQLMap (de modo similar se haría con otras herramientas como ZAP):

```
#Método POST
./sqlmap -u 'http://www.udea.edu.co:80/index.php'
--data 'user=aaaa&password=aaaa' -p user -b

#Método GET
./sqlmap -u 'http://www.udea.edu.co:80/index.php?username=aaaa&password=aaaa' -p user --dbms mysql -b
```

Para lograrlo se prefirió utilizar un lenguaje de propósito general llamado Clojure por su flexibilidad en comparación a Acceleo.

```
(defn concat-fields
 "Join a group of fields with &"
 [[_ page & fields] filter-fn]
 (str (correct-page page)
    (if (= 1 (count (s/split page #"\?"))) ;no ? found
     "?"
     "&")
    (s/join "&"
        (->> fields
           (filter filter-fn)
           (mapv
            (fn [[_ field attacks]]
             (str field "=a"))
            )))))
(defn concat-fields-sqlmap
```

```
"Concat only fields specified with the SQLInjection attack"
[page-with-fields]
(concat-fields
 page-with-fields
 (fn [[_ field-name attacks]]
  (let [attack-set (set attacks)]
   (or (attack-set "*")
     (attack-set "SQLInjection"))))))
(defn fields->sqlmap
"Construct the comand of a specific field for SQLmap"
[dir {:keys [url-with-params post-params]} severity]
(s/join " "
    ["sqlmap"
     "--batch"
     "--smart"
     (str "--risk=" (risk severity))
     (str "--level=" (level severity))
     "--time-sec=2"
     "--delay=1"
     "--timeout=5"
     "--retries=1"
     "--keep-alive"
     "--random-agent"
     "--threads=10"
     "--answers='extending=N,others=Y'"
     (format "--url '%s'" (validate-url url-with-params))
     (format "--data '%s'" (validate-url post-params))
     (format "--output-dir '%s'" (validate-path dir))]))
```

Hasta este punto concluye la fase de transformación. En la sección siguiente se explica cómo se unifican los resultados obtenidos luego de la ejecución de los comandos específicos para cada analizador de vulnerabilidades.

5.3.6. Reportes

Con la información anterior se construye un listado de vulnerabilidades. Cada vulnerabilidad tiene el siguiente formato:

Riesgo (risk)	Número entero entre 1 y 3 inclusive. 1 es bajo riesgo, 2 es medio y 3 es alto.
Nombre (name)	Nombre de la vulnerabilidad.
URL	Es la URL completa en la que el parámetro vulnerable se envía.
Entrada (field)	Nombre del parámetro donde se encontró la vulnerabilidad.
Ataque (attack)	Valor ingresado como entrada que llevó a la detección de

	la vulnerabilidad.
Descripción (description)	Descripción detallada de la vulnerabilidad.
Solución (solution)	Mejor práctica sugerida para resolver la vulnerabilidad.

A continuación unas secciones de pseudocódigo que ilustran la lógica principal.

```
vulns = [ '[' /][ ']' /]
def addVuln(name, risk, url, param, attackStr, description, solution):
vulns.append(
{'arisk': risk, 'name': name, 'url': url, 'field': param, 'attack': attackStr, 'description': description,
'solution': solution})
return
```

Se agregan las vulnerabilidades encontradas en cada entrada. Por cada vulnerabilidad se especifican los campos: riesgo, nombre, URL, campo afectado, ataque, descripción del ataque y mitigación:

```
def isRelevant(param, uri, tipoVuln):
        param = para.strip().split()[ '[' /]0[ ']' /];
        for k, v in pages.iteritems():
                if uri == k:
        for parIter, tests in v[ '[' /]"params"[ ']' /].iteritems():
                if parIter == param and tipoVuln in tests[ '[' /]"tests"[ ']' /]:
                        return True
        return False
```

Aplica los tipos de ataque solo a las entradas que tienen sentido. Por ejemplo si la entrada no se muestra en el contenido de la respuesta, no será vulnerable ante ataques XSS.

5.3.7. Resumen de la arquitectura empleada

Las relaciones entre los componentes de Vulnfinder se presentan en la Figura 32.

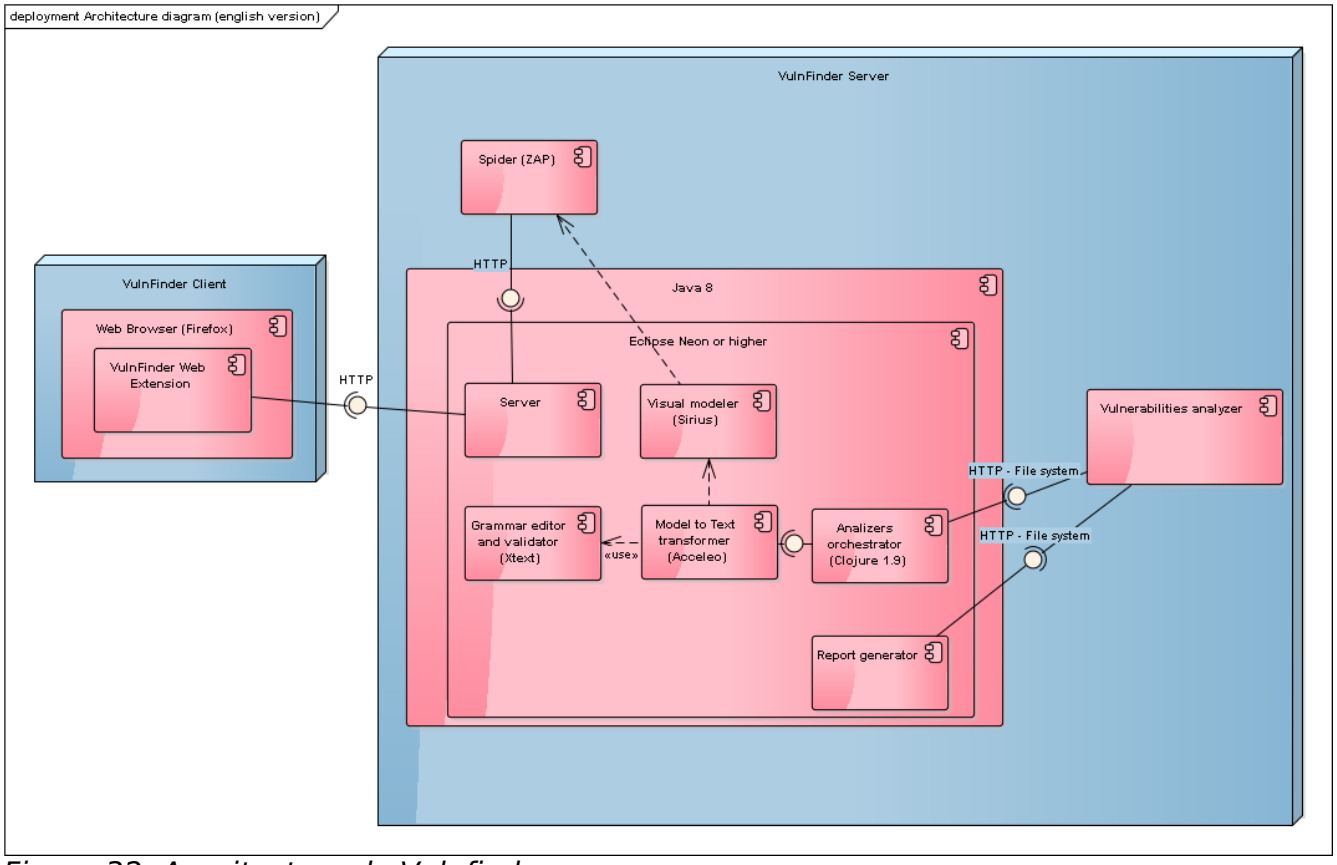


Figura 32: Arquitectura de Vulnfinder

CAPÍTULO 6. VALIDACIÓN

Este capítulo reporta la ejecución de Vulnfinder contra diversos sitios y compara sus resultados con herramientas del estado del arte. También reporta vulnerabilidades encontradas en sitios en producción elaborados por terceros.

6.1. ESTRATEGIA DE VALIDACIÓN

Los seres humanos no necesariamente procesan la misma información siempre de la misma forma. La seguridad de la información tiene como uno de sus principales retos la administración de los aspectos humanos de quienes ejecutan procesos previamente definidos. Por ello, durante las pruebas de seguridad, es deseable automatizar ciertos pasos por medio de herramientas como los analizadores de vulnerabilidades. En la distribución de linux llamada Kali se encuentran algunos de los más utilizados por la industria de la seguridad, incluidos ZAP y SQLmap. En adelante ambos analizadores serán llamados las *herramientas alternativas*, para marcar la diferencia con Vulnfinder (el prototipo desarrollado en la investigación) que internamente también las emplea. La idea es que haya una correlación entre los mecanismos de detección comparados, para que verificar que el empleo de modelos no disminuya la detección en las herramientas originales.

Las herramientas alternativas requieren una parametrización detallada individual, que las hace muy dependientes de las habilidades diferenciadas entre analistas. Personas con experiencia similar, usando esas mismas herramientas para atacar un determinado TOE (objetivo de la prueba), no necesariamente encontrarán las mismas vulnerabilidades. Vulnfinder permite especificar las pruebas por medio de modelos, evitando que el analista configure por sí mismo cada analizador.

Especificar mediante modelos (visuales o textuales) las pruebas de seguridad para ser aplicadas de modo automatizado, hace que el conjunto básico de ataques ejecutados sean más estándar y repetibles. Esas ventajas son bien conocidas en el área de MBT y

cada vez aplican más a otros campos como lo es la infraestructura definida por código, donde se usan lenguajes como Ansible para describir la infraestructura en vez de configurar manualmente cada servidor. Aplicar esos ataques contra un TOE permite obtener cierto número de vulnerabilidades, cada una con su propia *criticidad* (severity). En conjunto, estas vulnerabilidades representan una medida sobre el estado de seguridad del TOE, que denominaremos *nivel de riesgo*.

6.1.1. Repetibilidad y nivel de detección

Una prueba es más repetible cuando puede especificarse una sola vez y ejecutarse múltiples veces, encontrando de nuevo el mismo nivel de riesgo. Esta es una propiedad inherente a la funcionalidad ofrecida por Vulnfinder, dado que en la repetición de la prueba no interviene ninguna decisión humana, puede considerarse que es un proceso determinístico. Es solo afectado por condiciones externas como por ejemplo que el TOE no esté disponible. Por tanto, no hay una necesidad evidente de validar este aspecto mediante un experimento. La posibilidad de existencia de errores de implementación no se descarta, pero ellos afectarían principalmente la efectividad de la prueba en vez de la repetibilidad.

Diferentes pruebas de seguridad no necesariamente encuentran las mismas vulnerabilidades en un TOE determinado. Resultados de interés son la cantidad y criticidad de las vulnerabilidades detectadas. Factores que influyen en la variación de los resultados incluyen a la persona que la ejecuta, tiempo asignado para la prueba y herramientas utilizadas. Este último aspecto es el que Vulnfinder busca ayudar a estandarizar. Utilizar más herramientas en la misma unidad tiempo podría favorecer un mejor nivel de detección.

La estandarización de las pruebas provista por el MBT ayuda a especificar políticas que incluyan factores específicos del contexto de la prueba, como lo es el nombre del propietario del TOE. Un ejemplo es que toda prueba de aplicación realizada para el cliente UdeA tiene que probar siempre al menos:

- *udea' OR 1<2 -- -*

- " *script>alert(udea')</script>*
- " *onerror=alert(udea').*
- *">alert(333)</script><!--*

Esto ayuda a complementar las pruebas normales que hace cada analista con ciertos casos mínimos estandarizados que hayan mostrado alta efectividad en el pasado.

6.1.2. Métrica para resultados de una prueba de seguridad

Cuando se realiza una prueba de seguridad, el principal objetivo es diagnosticar el nivel de riesgo al que se expone la organización que utiliza el TOE (sistema objetivo de la prueba). Ese nivel de riesgo de negocio es difícil de estimar sin conocimiento específico de funcionamiento interno de la organización y puede ser subjetivo porque implica variables difíciles de medir como por ejemplo la percepción de imagen ante los clientes.

En la práctica, los informes de pruebas de seguridad se concentran en el riesgo técnico que es más fácil de medir y que de hecho es uno de los insumos para que luego un experto de la organización pueda calcular el riesgo de negocio. Para este experimento se utilizará una escala que ignora el riesgo de negocio y tiene como alcance solo el aspecto técnico. El TOP 10 de OWASP, mostrado en la Tabla 10, será utilizado como los tipos de vulnerabilidades posibles a ser detectadas.

A1 – Inyección
A2 – Pérdida de Autenticación y Gestión de Sesiones
A3 – Secuencia de Comandos en Sitios Cruzados (XSS)
A4 – Referencia Directa Insegura a Objetos
A5 – Configuración de Seguridad Incorrecta
A6 – Exposición de Datos Sensibles
A7 – Ausencia de Control de Acceso a las Funciones
A8 – Falsificación de Peticiones en Sitios Cruzados (CSRF)
A9 – Uso de Componentes con Vulnerabilidades Conocidas
A10 – Redirecciones y reenvíos no validadoS

Tabla 10: TOP 10 de OWASP

El riesgo técnico está conformado por la probabilidad e impacto de un evento en el que un agente de amenaza se aproveche de un tipo de vulnerabilidad determinado. Por ejemplo, si el tipo de vulnerabilidad encontrada es A1-Inyección, el resultado puede ser la alteración completa de la información almacenada en el sistema. Si el tipo de vulnerabilidad es A3-XSS, puede ocurrir la suplantación de identidad de algunos usuarios. Es aceptado por la industria que A1-Inyección, en términos técnicos, representa un mayor riesgo que otros tipos de vulnerabilidades Web (de hecho está catalogado como el TOP 1 de los riesgos según OWASP). Sin embargo, cada vulnerabilidad puede ser más crítica que otra (aún incluso perteneciendo al mismo tipo) según el escenario de negocio particular y por eso es que existen escalas de calificación como CVSS[62] y VRSS que tienen en cuenta diferentes aspectos. Por ejemplo, consideran la complejidad de llevar a cabo la explotación de la vulnerabilidad. VRSS será utilizado durante el experimento porque facilita el análisis estadístico de los resultados obtenidos. Esto se debe a su alta correspondecia con una distribución normal en un análisis de 34.093 vulnerabilidades listadas por CVE[63] publicadas entre 1999 y 2008 (Qixu Liu, 2011).

Establecer diferentes pesos a los tipos de vulnerabilidades en función de su criticidad (concepto *severity* en el metamodelo), permite catalogar como un mejor diagnóstico a aquel que no solo encuentra un alto número de vulnerabilidades sino que además encuentra las más críticas. VRSS tiene en cuenta ese tipo de ponderación, asignando valores que van desde 0 hasta 10 para cada vulnerabilidad, siendo 10 el nivel más crítico.

VRSS define la criticidad de cada tipo de vulnerabilidad como la suma del puntaje de *impacto* y de *explotabilidad*. El impacto depende de la medida en que es afectada cada una de las componentes *confidencialidad (C)*, *integridad (I)* y *disponibilidad (A)*. Los posibles valores para ellas son *completo (C)*, *parcial (P)* o *ninguno (N)*.

El puntaje de explotabilidad se calcula en función de tres componentes. El *vector de acceso (AV)* es el lugar desde dónde se debe realizar la explotación, de menor a mayor explotabilidad según si es el equipo *local (L)*, *red adyacente (A)* o *red externa (N)*. La

62. *Common Vulnerability Scoring System - https://www.first.org/cvss/*
63. *Common Vulnerabilities and Exposures - https://cve.mitre.org/*

complejidad de acceso (AC) depende de las circunstancias por fuera del control del atacante para que la explotación pueda llevarse a cabo (por ejemplo condiciones de carrera o necesidad de interacción de un usuario), con valores de *bajo (L)*, *medio (M)* y *alto (H)*. La *autenticación (Au)* recibe una calificación dependiendo del tipo de credenciales requeridas por el agente de amenaza: *ninguna (N)*, *un solo tipo de autenticación (S)* o *múltiples factores de autenticación (M)*.

La Figura 33, tomada del documento de referencia de VRSS ya citado, resume los posibles valores para las métricas de impacto y explotabilidad.

Possible impact metrics cases	Qualitative level
[C:C/I:C/A:C]	High
[C:P/I:C/A:C], [C:C/I:P/A:C], [C:C/I:C/A:P]	High
[C:N/I:C/A:C], [C:C/I:N/A:C], [C:C/I:C/A:N]	High
[C:C/I:P/A:P], [C:P/I:C/A:P], [C:P/I:P/A:C]	High
[C:C/I:P/A:N], [C:C/I:N/A:P], [C:P/I:C/A:N], [C:P/I:N/A:C], [C:N/I:C/A:P], [C:N/I:P/A:C]	Medium
[C:C/I:N/A:N], [C:N/I:C/A:N], [C:N/I:N/A:C]	Medium
[C:P/I:P/A:P]	Medium
[C:N/I:P/A:P], [C:P/I:N/A:P], [C:P/I:P/A:N]	Medium
[C:P/I:N/A:N], [C:N/I:P/A:N], [C:N/I:N/A:P]	Low
[C:N/I:N/A:N]	Low

Exploitability metric	Metric value
Access vector (AV)	Local (L)/Adjacent network (A)/Network (N)
Access complexity (AC)	High (H)/Medium (M)/Low (L)
Authentication (Au)	None (N)/Single (S)/Multiple (M)

Figura 33: Impacto y explotabilidad en VRSS

Cada vulnerabilidad puede tener diferente puntuación VRSS aún siendo del mismo tipo. Por ejemplo pueden existir dos calificaciones diferentes para A3-XSS, según el atacante

puede explotar la vulnerabilidad desde internet o si requiere estar en la red adyacente donde está desplegado el TOE. Nótese que en ese caso la diferencia de la calificación no depende del TOE como tal, sino de la ubicación de este. Teniendo en cuenta que para el experimento es de interés la naturaleza de la vulnerabilidad en sistemas Web y no particularidades del despliegue del TOE, para el modelo presentado en este experimento se asume equivalencia entre el puntaje de cada instancia de un mismo tipo de vulnerabilidad.

Se buscó asignar un único valor a cada tipo de vulnerabilidad a partir de datos históricos reportados por la empresa FLUID S.A a sus clientes entre diciembre de 2015 y septiembre de 2016. Se hizo una corrección sobre el impacto de la vulnerabilidad A2[64], debido a que presentaba divergencia amplia respecto al tipo de impacto recomendado por OWASP. Los datos de FLUID S.A indican un promedio de calificación para A2 de C:P/I:N/A:N, que significa P en confidencialidad, N en integridad y N en disponibilidad, lo que según VRSS indica un valor para el impacto de 1 (bajo). Sin embargo, OWASP le asigna un nivel de impacto severo, justificando que un fallo de ese tipo pueden comprometer algunas o incluso todas las cuentas atacadas, con lo que el atacante puede hacer todo lo que la víctima pudiera hacer en el sistema (las cuentas con privilegios son con frecuencia las más atacadas).

En análisis conjunto con FLUID S.A, se llegó a la conclusión de que durante un proyecto típico el tiempo de la prueba es muy restringido para alcanzar a realizar ataques de diccionario o fuerza bruta contra las todas las cuentas del sistema, por lo que generalmente se identifican contraseñas débiles para alguno de los usuarios existentes pero no necesariamente la del usuario de mayor privilegios. Aunque el hallazgo permita extrapolar conclusiones sobre una debilidad general en las políticas de contraseñas, FLUID S.A reporta lo realmente encontrado y procede luego a explicar detalladamente la situación para que haya una valoración de negocio con el cliente dueño de la aplicación. La Tabla 11 muestra la criticidad máxima (valor de referencia de cuál fue la vulnerabilidad más grave encontrada) y la criticidad promedio por cada tipo de vulnerabilidad, calculada a partir de las métricas parciales “explotabilidad” e “impacto”.

64. *https://www.owasp.org/index.php/Top_10_2013-A2-Broken_Authentication_and_Session_Management*

Tipo de vulnerabilidad	AV	AC	Au	C	I	A	Explotab.	Impacto	Criticidad promedio	Criticidad máxima
A1	0.65	0.61	0.56	P	P	C	0.44	6.00	6.44	10.00
A2	0.65	0.71	0.70	C	P	P	0.65	6.00	6.65	10.00
A3	0.65	0.61	0.56	C	P	N	0.44	5.00	5.44	9.00
A4	1.00	0.71	0.56	P	P	N	0.80	2.00	2.80	8.50
A5	0.65	0.71	0.70	P	P	P	0.65	3.00	3.65	6.80
A6	0.65	0.71	0.70	P	N	P	0.65	2.00	2.65	8.50
A7	0.65	0.71	0.70	N	P	P	0.65	2.00	2.65	10.00
A8	1.00	0.71	0.56	P	P	N	0.80	2.00	2.80	7.90
A9	0.65	0.71	0.70	P	P	P	0.65	3.00	3.65	10.00
A10	1.00	0.56	0.61	P	P	P	0.68	3.00	3.68	7.50

Tabla 11: Puntaje por tipo de vulnerabilidad

Muchas recomendaciones de seguridad hablan de disminuir la superficie de ataque, que es la parte de la aplicación con la que puede interactuar un atacante. Se considera parte de la superficie de ataque todos los posibles campos o entradas de datos a la aplicación Web, incluyendo encabezados del protocolo HTTP, mecanismos de persistencia en el navegador, etiquetas INPUT y SELECT de HTML y parámetros enviados mediante los métodos PUT, GET, POST y DELETE. Buscando ofrecer un alto nivel de abstracción, el lenguaje de modelado de Vulnfinder solo expone campos de formularios y parámetros en la URL, que son un tipo de entradas muy asociadas a lógica de negocio de la aplicación. Las demás entradas generalmente están asociadas a detalles de implementación y por ende se asume que siempre deben ser probadas cuando el tipo de análisis se configura como alto (High) en Vulnfinder.

Con mayor exposición de superficie de ataque, existe mayor probabilidad de ocurrencia de vulnerabilidades. Es decir que hablar tan solo del número de vulnerabilidades encontradas en una aplicación tiene más sentido si se expresa haciendo mención al número de entradas (campos) probados. Por ejemplo, es normal que sea más difícil

encontrar un fallo entre 20 campos que entre 2500. Otra ventaja de considerar el número de entradas en la unidad de medición, es que permite a futuro hacer cierto tipo de comparaciones entre aplicaciones.

La métrica usada para asignar una *puntuación* a una prueba de seguridad será llamada *nivel de riesgo,* donde una puntuación de 0 significa que no existe ninguna vulnerabilidad, mientras que 10 representa que cada campo de la aplicación tiene el tipo de vulnerabilidad más alta posible. Para calcular la puntuación, se suma la criticidad de cada vulnerabilidad encontrada y se divide por el número total de entradas de la aplicación. Por ejemplo en una aplicación de 5 campos en la que existe una instancia de A1-Inyección y dos de A3-XSS, la puntuación de la prueba sería 3.46[65]. Ese valor puede interpretarse como que, en promedio, cada entrada de la aplicación tiene un tipo de vulnerabilidad de criticidad 3.46. Ese estilo de asignar puntuación se utilizará para medir el resultado de cada participante durante el experimento.

6.1.3. Efectividad

Para que una prueba sea *efectiva*, el nivel de riesgo detectado debe estar acorde con el nivel de riesgo real que posee el TOE. Emplear modelos para dirigir pruebas, no debe modificar la efectividad en la detección respecto a utilizar las herramientas de modo independiente.

El aspecto efectividad contempla la parte de la hipótesis que menciona que es posible especificar pruebas mediante modelos (visuales o textuales) para ser aplicadas de modo automatizado. Se contemplan dos características:

- *Configuración según mejores prácticas*: Cada una de las herramientas alternativas requieren configuraciones en un cierto número de parámetros. La mayoría de los parámetros están configurados según las guías de OWASP (puede verificarse observando el código fuente del módulo *orchestrator* de Vulnfinder[66]). La sintaxis y

65. *La criticidad promedio de A1-Inyección es 6.44 y la de A3-XSS es 5.44, de ese modo el cálculo a realizar es: (6.44*1 + 5.44*2) / 5*
66. *https://gitlab.com/ryepesg/vulnfinder/tree/master/orchestrator/src*

llamadas al API utilizadas para dirigir las herramientas se encuentran en las siguientes guías:
https://www.owasp.org/index.php/Automated_Audit_using_SQLMap
https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project
Esto explica que un modelo de prueba creado utilizando Vulnfinder sí se transforma correctamente en la sintaxis aceptada por los analizadores de vulnerabilidades Web y el fuzzer, permitiendo encontrar vulnerabilidades en aplicaciones reales como se mostrará más adelante.

- *Número de Vulnerabilidades detectadas:* Es la cantidad de vulnerabilidades en un tipo de vulnerabilidad determinado. No es necesario emplear la métrica *Nivel de riesgo*[67]*,* explicada en la sección siguiente "6.1.2. Métrica para resultados de una prueba de seguridad", porque las aplicaciones tienen el mismo número de entradas y se realizan comparaciones en cada tipo de vulnerabilidad de modo independiente.

Durante la revisión de la literatura, no se encontró un TOE estándar que sea utilizado sistemáticamente para comparar herramientas de detección. En (Faustin, Junyi, Damien, & Lancine, 2015) llegaron a esta misma conclusión y ellos por su cuenta decidieron desarrollar un sistema vulnerable de pruebas (el cuál no fue liberado como código abierto). Otros autores han utilizado sistemas vulnerables disponibles libremente como Mutillidae y WebGoat (Maag, 2013) (van der Loo, 2011). Estas aplicaciones cuentan con vulnerabilidades previamente conocidas y una prueba hacia ellas permite estimar el nivel de detección de la herramienta probada.

La Tabla 12 muestra una comparación por separado para Mutillidae y WebGoat. Se encontró al menos las mismas vulnerabilidades que la herramienta con mayor detección entre los estudios citados (columna *Máximo exponente del estado del arte*, en cada celda se indica entre paréntesis la herramienta que reporto el mayor número) y además se confirmó que la ejecución de las herramientas alternativas por separado encuentran las mismas vulnerabilidades que al utilizarse Vulnfinder. En ambas ocasiones se revisó que

67. *Criticidad promedio encontrado por entrada, medido en escala VRSS (Vulnerability Rating and Scoring System). Esta métrica ofrece un estimado del nivel de detección de una prueba contra un TOE, permitiendo comparar cuál de ellas encontró mejores resultados*

los casos de prueba se ejecutan exitosamente cubriendo la mayor parte de la superficie de ataque y usando los tipos de ataques Web incluidos en el TOP 10 OWASP.

	Número de Vulnerabilidades detectadas en el TOE Mutillidae usando el TOP 10 OWASP	
Tipo de vulnerabilidad	**Máximo exponente del estado del arte**	**Vulnfinder usando SQLmap y ZAP**
A1: Injection	9 (Arachni)	12
A2: Broken Access Control	5 (W3af)	8
A3: Cross-Site Scripting (XSS)	12 (Wapiti)	13
A8 - Cross-Site Request Forgery (CSRF)	3 (Arachni)	3

Tabla 12: Vulnerabilidades encontradas en Mutillidae

Adicionalmente, se confirma que el lenguaje propuesto sirve para describir modelos que especifican pruebas de seguridad, aplicables en entornos no diseñados a propósito para ser vulnerables. Es es decir, que el usuario especifica solo las entradas y los tipos de ataque deseados, en vez de los casos de prueba concretos a ser enviados al TOE. La validación en este sentido consistió en encontrar vulnerabilidades de criticidad alta (puntaje de impacto VRSS entre 6 y 9) en una aplicación realizada por terceros.

El dominio con la aplicación vulnerable es: https://alice.*.*.de/[68]. La Tabla 13 lista las vulnerabilidades encontradas. Para más información sobre los tipos de vulnerabilidades y sus calificaciones, consulte la sección siguiente (6.1.2).

Tipo de vulnerabilidad	**Descripción**	**Valoración VRSS**
A2 – Pérdida de Autenticación y Gestión de Sesiones	Un atacante puede engañar al sistema de autenticación para que valide las credenciales contra un servidor cualquiera (incluyendo uno que el atacante configure de antemano para responder	6.44 - Alta

68. *Detalles de la vulnerabilidad han sido removidos utilizando asteriscos (*) para proteger la confidencialidad del sistema. Solo los evaluadores de la investigación y los propietarios de software vulnerable tienen acceso a la información completa.*

	afirmativamente).	
A3 – Secuencia de Comandos en Sitios Cruzados (XSS)	Un atacante puede ejecutar código en el navegador de la víctima y lograr suplantación de identidad por medio del robo del identificador de sesión.	5.44 - Media
A4 – Referencia Directa Insegura a Objetos A5 – Configuración de Seguridad Incorrecta	Es posible causar denegación de servicio cambiando el parámetro connect_timeout y apuntando a direcciones de servidor que no contesten o que tarden mucho en responder.	2.80 - Media

Tabla 13: Vulnerabilidades encontradas en sitio de tercero.

El mecanismo utilizado para la detección de vulnerabilidades no fue diseñado para esta aplicación en específico, sino que corresponde a técnicas recomendadas por las mejores prácticas respecto a pruebas de seguridad de proyectos tan generales como OWASP, que no proponen pruebas de caja negra enfocadas en tecnologías específicas, sino que son de utilidad para encontrar vulnerabilidades del mismo tipo en una aplicación Web cualquiera.

CAPÍTULO 7. CONCLUSIONES Y TRABAJO FUTURO

7.1. Conclusiones

Existe un alto grado de investigación previa asociada a pruebas basadas en modelos orientadas a la funcionalidad, pero apenas recientemente se han aplicado esas técnicas en pruebas de seguridad. Este proyecto definió un lenguaje que permite describir pruebas de seguridad de caja negra automatizadas para sistemas Web que sean basadas en modelos. Dentro del conocimiento de los autores, Vulnfinder es la primera herramienta de MBT visual y textual, de libre acceso, que se integra con múltiples analizadores de vulnerabilidades aceptados incluidos en las distribución de seguridad más utilizadas por la industria.

Vulnfinder mostró que las herramientas empleadas habitualmente en la industria pueden ser complementadas de manera efectiva con técnicas de MBT sin sacrificar calidad en la detección. Las pruebas quedan documentadas en un lenguaje soportado por herramientas de modelado y pueden ser ejecutadas de nuevo sin intervención humana que pueda comprometer la repetibilidad. El lenguaje en el que se describen las pruebas es independiente de analizadores de vulnerabilidades particulares, lo cual permite que una sola definición de una prueba emplee diversas herramientas especializadas en detección de vulnerabilidades específicas. Vulnfinder, a diferencia de muchas propuestas de MBT, encuentra un amplio rango de tipos de vulnerabilidades en sistemas Web.

Vulnfinder encontró resultados satisfactorios en detección de vulnerabilidades comparados a otras herramientas del estado del arte y fue aplicado contra un sistema Web en producción, expuesto al público, encontrando vulnerabilidades de nivel de criticidad alto.

Para crear un lenguaje específico de dominio (DSL), como lo es la propuesta presentada en esta investigación que se enfoca en describir pruebas de seguridad, se pueden usar perfiles UML o metamodelos. Aunque en la academia es habitual el uso de UML como lenguaje de modelado, se encontró que en la industria es más que todo utilizado para representaciones de alto nivel y arquitecturas, en vez de especificaciones técnicas detalladas. En particular para las pruebas de seguridad, no es común que los analistas dominen completamente UML, así que no es una ventaja significativa el apegarse solo a perfiles UML. Aunque el soporte actual de herramientas es mayor para UML que para metamodelos, se encontró que la plataforma Eclipse proveía todo lo necesario para construir una herramienta con validación de modelos y gramáticas y todo el soporte interactivo esperado de un entorno integrado de desarrollo.

El lenguaje textual propuesto es un subconjunto del lenguaje de programación Clojure[69] (inventado en 2007), que pertenece a la familia de Lisp (inventado en 1968). Son conocidos por su gran utilidad en áreas como la inteligencia artificial. Lo que diferencia a esta familia de lenguajes es que los programas se escriben directamente usando el árbol de sintaxis abstracta (AST) del lenguaje. Esto permite que manipular cualquier programa desde código sea tan fácil como transformar cualquier otra estructura de datos, es decir, proveen mecanismos excepcionales para metaprogramación (programas que escriben otros programas) y construcción de lenguajes específicos de dominio. Dadas esas características, el lenguaje textual de Vulnfinder se benefició no solo de los editores de texto que soportan Clojure, sino que permitió simplificar mucho la implementación de la conversión de modelos en instrucciones para los analizadores de vulnerabilidades.

El lenguaje de modelado propuesto fue complementado con una estructura de tablas para relacionar ataques con entradas, que se asemeja a las plantillas basadas en hojas de cálculo utilizadas por FLUID S.A para presentar sus informes técnicos de detección de vulnerabilidades, en los que son muy rigurosos especificando la cobertura de la prueba (es decir exactamente qué campos fueron probados y cuales no, junto con la justificación de por qué no aplicaban para el tipo de prueba actual).

69. *http://clojure.org*

7.2. Líneas de trabajo futuro

7.2.1. Aprendizaje de máquinas

Una posible línea futura de investigación se encuentra en el área del aprendizaje de máquinas, la cual provee un vasto conjunto de algoritmos que pueden tomar mejores decisiones luego de ser expuestos a entrenamiento en ambientes controlados. Usando esa técnica, Vulnfinder podría aprender por sí mismo nuevos tipos de ataques o determinar, por sí mismo, los vectores de ataque más efectivos, sin depender de los autores de la herramienta para incorporar esas nuevas capacidades. Así, Vulnfinder podría ser efectivo en áreas del conocimiento cada vez más relevantes, como lo son los sistemas Web en dispositivos móviles o sistemas de control industrial como los sistemas SCADA. A partir de entrenamientos con técnicas de aprendizaje de máquinas, Vulnfinder podría ofrecer soluciones creativas generalmente asociadas a capacidades humanas; ese sería un paso más para intentar cerrar la brecha entre un ingeniero de pruebas y la herramienta de automatización.

7.2.2. Evolución como proyecto de software libre

No se cuenta con activos de propiedad intelectual previos al proyecto. Se obtendrá un nuevo activo de propiedad intelectual al finalizar el prototipo, aunque el software a desarrollar será liberado con licencias de software de código abierto. Las licencias de código abierto (open source), como por ejemplo MIT o BSD y las de software libre como GPL, permiten que los códigos fuentes y binarios puedan ser modificado y redistribuidos libremente sin que se tenga que pagar por una licencia para ello. Ese aspecto hace que otros investigadores puedan luego profundizar más en la investigación sin tener que empezar desde cero y podrá facilitar la adopción de la herramienta en la academia y en la industria. Vulnfinder se diferencia de otras iniciativas de integración de herramientas como Faraday[70] en que ofrece lenguajes para MBT, pero el prototipo integra pocos analizadores de vulnerabilidades. Podría ser más eficiente integrar directamente

70. *https://www.faradaysec.com/*

Vulnfinder con Faraday en vez de con diferentes analizadores por separado, de modo que el esfuerzo se concentre más en mejorar el soporte del modelado como tal.

Todo el código fuente se encuentra disponible en: https://gitlab.com/ryepesg/vulnfinder. Los artefactos que componen al proyecto son:

- **fields and attacks selection**: Plugin para firefox para seleccionar campos y ataques desde el propio navegador.
- **fields and attacks gathering**: Servidor HTTP que recibe la selección campos y ataques desde el navegador.
- **display model**: Componentes para modelado visual.
- **model transformation**: Transformación de modelos visuales y textuales.
- **orchestrator**: Orquestador de analizadores de vulnerabilidades.
- **report generator**: Generador de reportes con las vulnerabilidades encontradas.

A continuación algunas capturas de la herramienta final construida. Puede consultar más información visitando la guía de uso rápida[71] o avanzada[72].

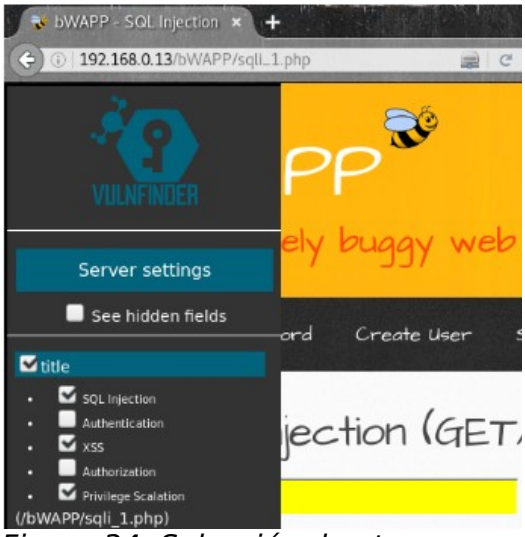


Figura 34: Selección de ataques por campo.

71. *https://gitlab.com/ryepesg/vulnfinder/wikis/guide*
72. *https://gitlab.com/ryepesg/vulnfinder/wikis/advanced-guide*

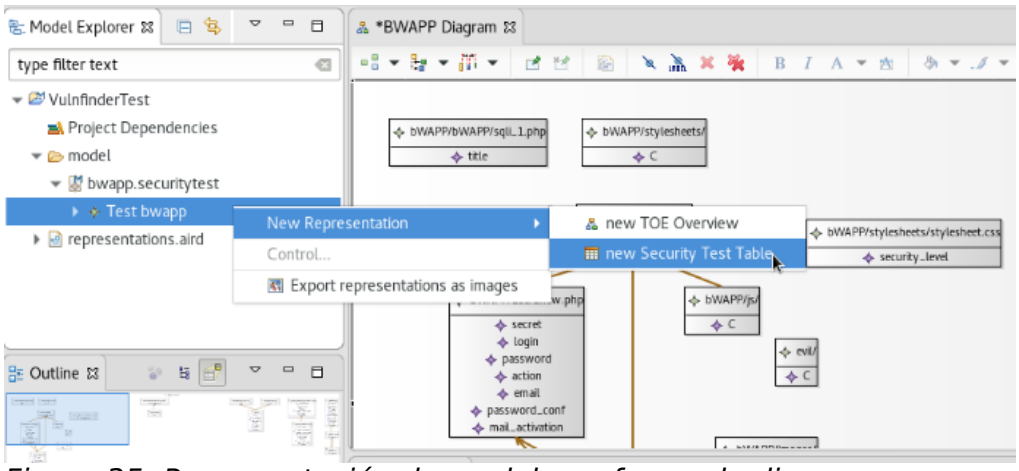


Figura 35: Representación de modelo en forma de diagrama.

```
(attacks
    (attack Authorization { :level High } )
    (attack PrivilegeScalation { :level High } )
    (attack Authentication { :level High } )
    (attack XSS { :level High } )
    (attack SQLInjection { :level H } )
)
```

High

Figura 36: Soporte de autocompletado.

```
9          )
10         (pag "/"
11         )
12     )
```

Figura 37: Señalado de errores de sintaxis.

```
4          (page "/mapp/index.php?page=login.php"
5          Multiple markers at this line                    ["SQLIn"])
6          - Couldn't resolve reference to Attack 'SQLIn'.  ["SQLIn
7          - The word 'username' is not correctly spelled   -submit-
8              )
9              (page "/"
10             )
11         )
12     )
13     (attacks
14
```

Change to 'Authentication'
Change to 'Authorization'
Change to 'PrivilegeScalation'
Change to 'SQLInjection'
Change to 'XSS'

Figura 38: Sugerencias de corrección de errores.

7.2.3. Pruebas de cierre

La detección de vulnerabilidades a nivel profesional se realiza con el ánimo de que sean solucionadas. Una prueba que se realiza por primera vez sobre un TOE específico, se denomina del tipo *búsqueda*. En contraste, las pruebas de *cierre* (o re-test) son realizadas en un momento posterior, para determinar la calidad en la corrección de las vulnerabilidades identificadas durante la prueba de búsqueda.

Las empresas de seguridad, durante las pruebas de cierre, deben decidir entre probar solo los campos detectados previamente o hacer en cambio una nueva prueba desde cero para cubrir los casos en que se hayan abierto nuevas vulnerabilidades en distintos lugares. Evidentemente un modelo creado con Vulnfinder para realizar la prueba de búsqueda podrá aplicarse sin cambios durante la prueba de cierre, pero igualmente los campos adicionales que hayan aparecido desde la primera prueba no serán considerados.

Desde la primera búsqueda hasta el efectivo cierre de las vulnerabilidades más críticas puede ocurrir mucho tiempo, por la coordinación entre todos los actores (quien detecta la vulnerabilidad, propietario de la aplicación y desarrollador). Un posible uso de las pruebas por modelos es combinarlas con estrategias de integración continua, de modo que se ejecuten cada determinado tiempo para mostrar constantemente el progreso del cierre de lo antiguo. Esto no evita la necesidad de realizar una nueva búsqueda, pero puede ofrecer realimentación automatizada a las partes para que tengan más información al tomar decisiones que lleven a reducir el número de iteraciones necesarias para corregir todas las vulnerabilidades importantes.

7.2.4. Adopción en la industria de la seguridad

Como esfuerzo de llevar el MBT a la industria, se está llevando a cabo un ejercicio de utilizar Vulnfinder en proyectos reales ejecutados por analistas de la empresa FLUID S.A. Entre las observaciones notadas hasta el momento se encuentran:

- Como los analistas tienen un perfil técnico muy avanzado, muestran preferencia por los modelos textuales en vez de los visuales.

- La problemática abordada sobre estandarización de casos de prueba es una necesidad real de la industria.
- Que el proyecto sea de código abierto ha motivado el interés de los participantes.
- La posibilidad de utilizar múltiples herramientas sin que eso represente mayores costos (el analista se demora lo mismo especificando el modelo para una o para 10 herramientas), ha permitido plantear la idea de ejecutar proyectos empresariales en los que se cobre por tiempo de parametrización y no de ejecución como se hace hoy en día.

BIBLIOGRAFÍA

Bau, J., Bursztein, E., Gupta, D., & Mitchell, J. (2010). State of the Art: Automated Black-Box Web Application Vulnerability Testing. *2010 IEEE Symposium on Security and Privacy*, 332–345. http://doi.org/10.1109/SP.2010.27

Blouin, A., Moha, N., Baudry, B., Sahraoui, H., & Jézéquel, J.-M. (2015). Assessing the use of slicing-based visualizing techniques on the understanding of large metamodels. *Information and Software Technology*, *62*(1), 124–142. http://doi.org/10.1016/j.infsof.2015.02.007

Bozic, J., Garn, B., Kapsalis, I., Simos, D., Winkler, S., & Wotawa, F. (2015). Attack Pattern-Based Combinatorial Testing with Constraints for Web Security Testing. In *2015 IEEE International Conference on Software Quality, Reliability and Security* (pp. 207–212). IEEE. http://doi.org/10.1109/QRS.2015.38

BSI. (2004). IT Grundschutz Manual.

Calvi, A., & Viganò, L. (2016). An automated approach for testing the security of web applications against chained attacks. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing - SAC '16* (pp. 2095–2102). New York, New York, USA: ACM Press. http://doi.org/10.1145/2851613.2851803

Cenzic. (2014). Application Vulnerability Trends Report : 2014. Retrieved from https://www.info-point-security.com/sites/default/files/cenzic-vulnerability-report-2014.pdf

Chenmeng Sui, Yanzhao Liu, Y. L. (2012). A Software Security Assessment System Based On Analysis of Vulnerabilities. Retrieved May 10, 2016, from http://www.aicit.org/JCIT/ppl/JCIT Vol7 No6_part26.pdf

Dai, Z. R. (2006). Model-Driven Testing with UML 2.0. Retrieved November 10, 2014, from http://www.uml.org.cn/UMLSearch/Dai.pdf

El-far, I. K., & Whittaker, J. A. (2001). Model-based Software Testing, 1–22.

F Massacci, Mylopoulos, J., & Zannone, N. (2007). An ontology for secure socio-technical systems. *Handbook of Ontologies for Business Interactions, IDEA Group*.

Faustin, K., Junyi, L., Damien, H., & Lancine, C. (2015). Effectiveness of Web Application Security Scanners at Detecting Vulnerabilities behind AJAX/JSON. Retrieved from http://www.ijirset.com/upload/2015/june/79_Effectiveness.pdf

Felderer, M., Agreiter, B., Zech, P., & Breu, R. (2011). A Classification for Model-Based Security Testing, (c), 109–114.

Fenz, S., & Ekelhart, A. (2009). Formalizing Information Security Knowledge. *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security, New York, USA*, 183–194.

Fischer, J., Scheidgen, M., & Blunk, A. (2014). Domain Specific Languages. Retrieved March 4, 2016, from https://www.informatik.hu-berlin.de/de/forschung/gebiete/sam/Lehre/modellbasierte-softwareentwicklung-modsoft/vorlesung/DSLs/v9-modsoft-ii.semantics-i.pdf

FOKUS. (2013). ITEA 2 Diamonds Case Studies Overview. Retrieved January 21, 2017, from http://www.itea2-diamonds.org/overview-9396b9e5a2ca3dc7

Ford, R., & Frincke, D. (2010). Ethics in Security Vulnerability Research. *IEEE Security & Privacy*, (April), 1.

Gorshkova, E., & Novikov, B. (2002). Exploiting UML Extensibility in the Design of Web Applications, 3–5.

Guoxiang, Y. (2012). Test Model for Security Vulnerability in Web Controls based on Fuzzing. *Journal of Software Vol. 7, No. 4*, *7*(4).

Hennicker, R. (2001). Systematic Design of Web Applications with UML, 1–20.

Henry, A., Zenteno, T., & X-t, N. I. E. (2008). Metodo de pruebas de sistema basado en modelos navegacionales en un contexto MDWE.

Hewlett Packard. (2012). What will 2020 look like? *Enterprise 20/20*.

Homeland Security ICS Cyber Emergency Response Team. (2012). ICS-CERT Year in Review. Retrieved from http://ics-cert.us-cert.gov/sites/default/files/Year_in_Review_FY2012_Final_0.pdf

Informatik, F., Fakult, M., Humboldt-universit, I. I., Dipl, H., Wei, S., Pr, B., … Peleska, J. (2010). Test Models and Coverage Criteria for Automatic Model-Based Test Generation with UML State Machines DISSERTATION.

ISO/IEC. (2005). ISO/IEC 27001:2005, Information technology - Security techniques - Information security management systems - Requirements.

ITEA. (2014). Security-testing regime for interconnected software-based systems and networks. Retrieved from https://itea3.org/innovation-report/security-testing-regime-for-interconnected-software-based-systems-and-networks.html

Jagadish. S., Lawrence, C, S. R. . (2008). cmUML - A UML based Framework for Formal Specification of Concurrent. *Reactive Systems, Journal of Object Technology (JOT),*

Vol. 7, No. 8, Novmeber- December 2008, Pp 188-207. http://www.jot.fm/issues/issue_200.

Jan, J. (2004). Model-based Security Engineering. *Software & Systems Engineering*, (Dep. of Informatics, TU Munich, Germany), 1. Retrieved from http://www4.in.tum.de/?juerjens

Jorgensen, P. (1995). *Software Testing: A Craftsman's Approach*. *CRC Press*. CRC Press.

Jürjens, J. (2008). Model-based Security Testing Using UMLsec. *Electronic Notes in Theoretical Computer Science*, *220*(1), 93–104. http://doi.org/10.1016/j.entcs.2008.11.008

Kerr, P., Rollins, J., & Theohary, C. (2010). The Stuxnet Computer Worm: Harbinger of an Emerging Warfare Capability. *Congressional Research Service*, 5. Retrieved from http://pubs.mantisintel.com/R41524.pdf

Kramer, A. (Software engineer), & Legeard, B. (2016). *Model-based testing essentials : guide to the ISTQB certified model-based tester foundation level*.

Landoll, D. J., & Landoll, D. (2005). The Security Risk Assessment Handbook: A Complete Guide for Performing Security Risk Assessments, 39.

Lebeau, F., Legeard, B., Peureux, F., & Vernotte, A. (2013). Model-Based Vulnerability Testing for Web Applications (MBVT). *Smartesting R&D Center*.

Maag, S. (2013). D5.WP2-Final Security Testing Techniques. *DIAMONDS Consortium*.

Magazine, T., & Testers, P. (2012). Model-Based Testing, (March).

Mallouli, W. (2011). D2.WP2-Concepts for Model-Based Security Testing. *DIAMONDS Consortium*, 1–134.

Matulevicius, R., & Dumas, M. (2010). A Comparison of SecureUML and UMLsec for Role-based Access Control.

Mayer, N. (2012). Model-based Management of Information System Security Risk. ".

Microsoft. (2016). Model-Based Testing. Retrieved March 3, 2016, from https://msdn.microsoft.com/en-us/library/ee620469.aspx

Model-Based Testing for Embedded Systems. (2011). CRC Press. Retrieved from http://books.google.com/books?id=fzgzNW_alD0C&pgis=1

Mohamed Mussa, Samir Ouchani, Waseem Al Sammane, A. H.-L. (2009). A Survey of Model-Driven Testing Techniques. *Proceedings - International Conference on Quality Software*. Retrieved from http://users.encs.concordia.ca/~abdelw/sba/papers/QSIC09-Model-DrivenTestingTechniques.pdf

Mouratidis, H., Giorgini, P., Schumacher, M., & Manson, G. (2003). Security patterns for agent systems. *In Proceedings of the Eight European Conference on Pattern Languages of Programs (EuroPLoP), Irsee, Germany*.

Na Wang, L. L. (2011). Evaluation on Web Application Security Scanner.

Need, T. C., Both, W., Today, B., Complements, C. T., Threats, G., & For, N. (2013). Information Security Consulting Services, Q1 2013. *The Forrester Wave*.

Neto, A. C. D., & Vieira, M. (2007). A Survey on Model-based Testing Approaches : A Systematic Review.

NIST. (1995). An Introduction to Computer Security : *The NIST Handbook Technical Report, NIST (National Institute of Standards and Technology). Special Publication 800-12*, *October*.

Ogunyomi, B., Rose, L., & Kolovos, D. (2014). On the Use of Signatures for Source Incremental Model-totext Transformation. *Springer International Publishing, MODELS 2014, LNCS 8787*, 84–98.

Paydar, S. (2011). An Agent-Based Framework for Automated Testing of Web-Based Systems. *Journal of Software Engineering and Applications*, *4*(2), 86–94. http://doi.org/10.4236/jsea.2011.42010

PCI Security Standards Council. (2015). PCI DSS Quick Reference Guide: Understanding the Payment Card Industry Data Security Standard version 3.1. Retrieved from https://www.pcisecuritystandards.org/documents/PCIDSS_QRGv3_1.pdf

Peleska, J. (2013). Industrial-Strength Model-Based Testing - State of the Art and Current Challenges. *Electronic Proceedings in Theoretical Computer Science*, *111*(Mbt), 3–28. http://doi.org/10.4204/EPTCS.111.1

Pierce, B. (2016). A Deep Specification for Dropbox. Retrieved January 21, 2017, from https://www.youtube.com/watch?v=Y2jQe8DFzUM

Pressman, R. S. (2010). *Ingeniería del Software. Un enfoque práctico*. (M.- Hill, Ed.) (7th ed.).

PriceWaterhouseCoopers. (2010). Revolution or evolution? *Technology Strategy Board*, 9.

Qixu Liu, Y. Z. (2011). VRSS: A new system for rating and scoring vulnerabilities. *Computer Communications, vol.34, pp.264–273*.

Refsdal, I. (2012, June 4). Comparison of GMF and Graphiti based on experiences from the development of the PREDIQT tool. Retrieved from https://www.duo.uio.no/handle/10852/9000

Reina Quintero, A. (2011). Separación Avanzada de Conceptos en Entornos Web. *ETS Ingeniería Informática. Universidad de Sevilla*. Retrieved from http://www.lsi.us.es/~reinaqu/doc/PhdThesis-reinaqu.pdf

Richier, J.-L. (2011). D1.WP2-Security Testing Techniques. *DIAMONDS Consortium*.

Schieferdecker, I., Grossmann, J., & Schneider, M. (2012). Model-Based Security Testing. *Electronic Proceedings in Theoretical Computer Science*, *80*(Mbt), 1–12. http://doi.org/10.4204/EPTCS.80.1

Schieferdecker, I., Großmann, J., & Schneider, M. (2012). Model-Based Fuzzing for Security Testing. *ITEA 2*, (April 2012), 24.

Schieferdecker, I., & Rennoch, A. (2013). Model Based Security Testing Selected Considerations, (March 2011).

Schwinger, W., & Koch, N. (2006). Modeling Web Applications, 39–64.

Souag, A. (2015a). *AMAN-DA. Une approche basée sur la réutilisation de la connaissance pour l'ingénierie des exigences de sécurité*.

Souag, A. (2015b). Amina Souag AMAN-DA : Une approche basée sur la réutilisation de la connaissance pour l ' ingénierie des exigences de sécurité.

Souag, A., Salinesi, C., & Comyn-Wattiau, I. (2012). Ontologies for Security Requirements: A Literature Survey and Classification. *Advanced Information Systems Engineering Workshops*, *112*(Éd. Berlin, Heidelberg: Springer Berlin Heidelberg), 61- 69.

Souag, A., Salinesi, C., Mazo, R., & Comyn-Wattiau, I. (2015). A Security Ontology for Security Requirements Elicitation. *International Symposium on Engineering Secure Software and Systems (ESSoS)*. http://doi.org/10.1007/978-3-642-36563-8

Stepien, B., & Peyton, L. (2012). Using TTCN-3 as a modeling language for web penetration testing. *2012 IEEE International Conference on Industrial Technology*, 674–681. http://doi.org/10.1109/ICIT.2012.6210016

Sujithra Sriganesh. (2005). A model based approach for white box testing.

Suto, L. (2010a). Analyzing the Accuracy and Time Costs of Web Application Security Scanners.

Suto, L. (2010b). Analyzing the Effectiveness and Coverage of Web Application Security Scanners. Retrieved from http://ha.ckers.org/files/Accuracy_and_Time_Costs_of_Web_App_Scanners.pdf

Symantec. (2015). Internet Security Threat Report, Volume 20.

Symantec. (2016). 2016 Norton Cybercrime report.

The Open Web Application Security Project. (2013). Threat Risk Modeling. Retrieved from https://www.owasp.org/index.php/Threat_Risk_Modeling

Undercoffer, J., Joshi, A., & Pinkston, J. (2003). Modeling Computer Attacks: An Ontology for Intrusion Detection. *6th International Symposium on Recent Advances in Intrusion Detection, 113-35. Springer.*

van der Loo, F. (2011). Comparison of penetration testing tools for web applications, 50.

Vorobiev, A., & Han, J. (2006). Security attack ontology for web services. *Second International Conference on Semantics, Knowledge and Grid, 2006. SKG'06., 42-42. IEEE.* Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.464.2973&rep=rep1&type=pdf

Williams, J., Matragkas, N., Kolovos, D., Korkontzelos, Y., Annaniadou, S., & Paige, R. (2014). Software Analytics for MDE Communities. *Proceedings of the 1st Workshop on Open Source Software for Model Driven Engineering Co-Located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems, OSS4MDE@MoDELS, Valencia, Spain.*, 53–63.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). *Experimentation in Software Engineering*. Springer Science & Business Media. Retrieved from https://books.google.com.co/books/about/Experimentation_in_Software_Engineering.html?id=QPVsM1_U8nkC&pgis=1

Wotawa, F. (2012). D3.WP2-Initial Model-Based Security Testing Methods. *DIAMONDS Consortium*, 2–5.

Zuluaga, M. J. B. (2012). Propuesta de gestión de riesgos para SCADA en sistemas eléctricos, *3*(2), 12–21.