Herramienta para el Análisis Automático de Parches

Antonio Castro Lechtaler[*,1,2], Julio César Liporace[†,1], Marcelo Cipriano[*,1], Edith García[†,1], Ariel Maiorano[†,1], Eduardo Malvacio[†,1], Néstor Tapia[†,1]

[*]{acastro,marcelocipriano}@iese.edu.ar
[†]{edithxgarcia,jcliporace,maiorano,edumalvacio,tapianestor87}@gmail.com

[1]Grupo de Investigación en Criptografía y Seguridad Informática (GICSI) – Instituto Universitario del Ejército, Argentina
[2]FCE - Universidad de Buenos Aires, Argentina

Enero de 2015

Resumen

El objetivo principal de este artículo es el de presentar el proyecto de una nueva herramienta para el análisis automático del código fuente de parches. Se pretende que podría servir de ayuda a personas o a equipos de personas en los roles de analistas, revisores o auditores, que deban realizar estos análisis de forma repetida y continua. Se procura lograr lo anterior generando un entorno que facilite el trabajo colaborativo entre los usuarios analistas, y automatizando algunas de las tareas que realizarían en evaluaciones y auditorias de esta índole. Se debe aclarar que el alcance de esta automatización, en la versión inicial de la herramienta que se está presentando, es parcial y limitado. Sin embargo se considera que podría resultar de interés el desarrollo de un proyecto de software libre, de código abierto, para ejecutar de manera periódica y automática, al menos los primeros pasos de un análisis de parches o de diferencias entre versiones de código fuente.
Además de la presentación del proyecto, el siguiente objetivo del artículo es el de intentar justificar la utilidad de una herramienta de este tipo. Se presenta un resumen de ciertos problemas de seguridad en proyectos de código abierto u *opensource*. Se expone que estos problemas fueron o pudieron haber sido descubiertos a partir de la revisión de parches o de diferencias entre versiones en software de código abierto, publicados con anterioridad a la divulgación de la vulnerabilidad.

1 Introducción

Se presenta como proyecto pero contando con una primera versión desarrollada, una herramienta para el análisis automático del código fuente de parches. Se planea distribuirla como son comúnmente distribuidos los proyectos de código abierto, y estará disponible al momento en que se haya publicado o expuesto este artículo que la introduce.

Desconocemos la existencia de otras herramientas de acceso libre que analicen específicamente los parches o diferencias entre versiones en código fuente en busca de vulnerabilidades a la manera en que este proyecto lo propone. Sí fueron publicados trabajos académicos que abordaron aspectos relativos a la ingeniería del software, como ser: el recupero de la trazabilidad de requerimientos entre código fuente y defectos o *bugs* corregidos vía el análisis de parches [5] , o la propuesta para la utilización de parches como forma de reportar defectos. El primero de estos trabajos presentó una herramienta para procesar la información registrada en sistemas Bugzilla y relacionarla contra identificadores en el código mediante CVS. En lo relativo al análisis de código fuente que implementa funcionalidad criptográfica, caso de estudios publicados para la plataforma Android indican por ejemplo que, de 269 vulnerabilidades reportadas desde enero de 2011 hasta mayo de 2014, sólo el 17% correspondió a defectos en librerías criprográficas, y el 83% restante a usos incorrectos de estas librerías por aplicaciones [14]. También en [8] se estudiaron aplicaciones Android; 10327 de un total de 11748 aplicaciones (88%) cometieron al menos un error en su implementación. Este último trabajo presentó una herramienta para la detección automática de estos problemas, pero no se ha distribuido libremente [15]. Existen además herramientas de uso general para la ayuda en el manejo de código fuente. Éstas, por lo general, no sólo administran revisiones de parches, sino que incluyen funcionalidad para el manejo de todo el ciclo de desarrollo. Entre las principales alternativas disponibles de código abierto se encuentran por ejemplo [12] y [23].

En relación a la seguridad de la información, la herramienta propuesta podría categorizarse como un analizador estático de código fuente. Sin embargo se diferencia en que pretende, además de automatizar la búsqueda de nuevas vulnerabilidades, permitir seleccionar y configurar reglas particulares para alertar a los usuarios analistas. Ellos podrán manejar esta información, además de aquella relativa a las alertas generadas, operando en el sistema mediante su interfaz Web. Las verificaciones de estas reglas, revisadas automáticamente sobre parches o sobre nuevas diferencias entre ramas o *branches* de un mismo proyecto de software, serán las que produzcan la activación de las alertas en caso de corresponder. Estas reglas incluirían, por ejemplo: el cotejo de patrones específicos; las modificaciones cualesquiera en porciones de

código crítico, o código que, se reconozca como una implementación de criptografía; modificaciones sobre archivos específicos, modificaciones realizadas por programadores específicos, etc. Estas revisiones serán hechas sobre las diferencias o parches que se recuperarán desde repositorios Git [10], locales o remotos. Para el análisis de código estático de vulnerabilidades, en esta versión inicial, la herramienta se valdría de Flawfinder [32]. Respecto a otras herramientas y proyectos del estilo, el apartado siguiente hace referencia a listados de alternativas libres y comerciales [4, 19, 22].

Ya sea a partir de la aplicación de controles de calidad -auditorias y revisiones-, o más específicamente, a partir de búsquedas de patrones o códigos determinados y conocidos como causantes típicos de vulnerabilidades, la revisión manual de parches publicados es una actividad común en el ámbito de la seguridad de la información. La auditoria, no sólo del código fuente completo, sino, especialmente, de los cambios que sufrió ese código -lo que se entenderá de manera general como parches o *patches*-, es material de análisis que sería de utilidad para detección de potenciales nuevos problemas de seguridad, como se sugiere el apartado relativo a las vulnerabilidades.

Aunque por supuesto no comparables con las que podría realizar una persona, estas revisiones sistematizadas podrían ser aplicadas repetidamente a una gran cantidad de código fuente o de proyectos de software. Su aporte principal consistiría en organizar y agilizar los primeros pasos de un proceso manual, para finalmente alertar sólo sobre los resultados que deban revisarse manualmente.

Es menester notar que la automatización de este tipo de revisiones tiene sus propios problemas y limitaciones. Un estudio que analizó 210 muestras indicó que más del 40% no fueron reportadas en una evaluación realizada con cinco herramientas de análisis de código fuente C/C++, mientras que sólo el 7% de esas muestras fueron correctamente reportadas por todas las herramientas [30]. Citado desde esta publicación, el mismo estudio, pero ahora utilizando seis herramientas y analizando código fuente en lenguaje Java, obtuvo resultados similares.
Si bien desde el proyecto libre OWASP [22] también se describen por ejemplo los problemas o debilidades de herramientas de análisis automáticas con respecto a la dificultad para la de detección de problemas complejos como pueden ser los de autenticación, de control de acceso y de uso inseguro de criptografía; también se destacan sus fortalezas, a saber: la posibilidad de realizar los análisis repetida y rápidamente sobre una cantidad de código fuente no auditable en los mismos términos por una persona o equipo de personas, la capacidad de detectar problemas típicos (*buffer overflows, SQL Injection, etc.*) con alta probabilidad, y la especificidad de los reportes, generalmente detallando nombre archivo y línea de código fuente.

2 Análisis de código fuente

2.1 Costo de la calidad del software

El análisis y revisión de código fuente es una buena práctica realizada continuamente en busca de mejorar la calidad del software. Vale aclarar que no siendo la cantidad de *bugs* o defectos en un proyecto de software el único indicador de su nivel calidad, sí resulta una métrica valiosa para el control y la potencial mejora de los procesos de desarrollo.
Adviértase también que estrictamente deberíamos decir que el objetivo a lograr con estos análisis y revisiones es reducir el costo de calidad, encontrando y corrigiendo defectos de manera temprana y a un costo menor [25].

En este sentido, se ha demostrado empíricamente [9] que la calidad del software depende de los mecanismos de control de calidad empleados como parte integral de los procesos. Específicamente además, se mostró que la tasa de revisión personal (medida en cantidad de líneas de código fuente por unidad de tiempo) afecta a la efectividad en la detección (y corrección) de defectos. Los datos de este estudio demostraron también por ejemplo que la calidad de las revisiones declina cuando esta tasa excede el máximo recomendado de 200 líneas de código por hora. Los autores citan además trabajos anteriores donde se considera óptima una tasa de 125 líneas por hora [3], y hacen referencia a un comentario en uno de los textos de referencia: *"it is almost one of the laws of nature about inspections, i.e., the faster an inspection, the fewer defects removed"* [25], es decir, cuanto más rápida la inspección, menor la cantidad de defectos corregidos.

2.1.1 No todos los defectos son vulnerabilidades

Como subconjunto de los defectos generales que podrían encontrarse en un proyecto de software, este trabajo se enfocará en aquellos relacionados a problemas de seguridad de la información. Hablamos de *bugs* o defectos que, de explotarse, implicarían algún tipo o grado de compromiso en la seguridad del sistema.

Estos defectos son comúnmente llamados agujeros de seguridad o vulnerabilidades, y suelen ser prioridad en los procesos de "emparche" o *patching* de proyectos de software. Son típicamente categorizados por su criticidad, de acuerdo al grado de exposición o gravedad de la vulnerabilidad.

2.2 Sistematización del análisis

De lo visto en el apartado referido a la calidad del software, respecto a la eficacia que tendrá una revisión en relación a la cantidad de tiempo que el analista dedicó, se desprende que la automatización del análisis de código haría más eficientes los procesos de control de calidad y desarrollo. Las herramientas de aseguramiento de calidad son actualmente un recurso fundamental para la mejora de las aplicaciones de software, según una publicación del NIST, donde ya se proponen especificaciones mínimas de funcionalidad para lo que sería un software analizador de código fuente en busca de vulnerabilidades [2]. En relación a esto último, se considera en esa publicación que, por ejemplo, la herramienta debería identificar debilidades de seguridad, reportar éstas indicando el tipo de debilidad de cada una y donde se encuentra ubicada, y por último, no reportar demasiados (*too many* en el original) falsos positivos.

Valerse únicamente de la implementación de políticas, estándares y buenas prácticas de desarrollo de software para el aseguramiento de la calidad sería inadecuado según un estudio del SEI/CERT [30]. Se indica que puede que no sean aplicados correcta y consistentemente, y que serían las auditorías manuales de código fuente las que pueden ser complementadas, ayudadas, por herramientas de análisis sistematizado y automático.

2.2.1 Herramientas actualmente disponibles

Diferentes entidades mantienen listados desde los cuales se comprueba que existe una buen variedad de herramientas para analizar código fuente de manera general. Entre ellas, la página del NIST, *Source Code Security Analyzers* [19], o "Analizadores de seguridad de código fuente"; de parte de la división CERT, de su *Secure Coding* o "Programación segura", ver *Secure Coding Tools* [4] o "Herramientas de programación segura"; por último, una referencia obligada es el proyecto OWASP, que también mantiene un listado de *Source Code Analysis Tools* [22], o "Herramientas de análisis de código fuente".

Si bien se planea incorporar otras herramientas generales de análisis de código fuente a la herramienta presentada en el futuro, actualmente se utiliza Flawfinder [32] para el análisis de código en lenguajes C/C++.

3 Vulnerabilidades post parches

3.1 Errores de clasificación de vulnerabilidades

Este apartado trata acerca del problema de la clasificación, o mejor dicho, de la no-clasificación de *bugs* o defectos de software como vulnerabilidades. Se refiere a continuación a los trabajos de un autor que estudio esta problemática y publicó resultados empíricos de sus experimentos con proyectos de software libre de código abierto.

Trabajos de Jason Wright Conocido por su trabajo en el *framework* criptográfico del sistema operativo OpenBSD [13], su reciente tesis [34], que revisa y extiende otros trabajos anteriores [35, 36, 33], deja en claro la importancia de la revisión de parches como mecanismo para el descubrimiento de vulnerabilidades.

Dividida en tres secciones principales, su última publicación analiza en primer lugar el impacto que tuvieron los cambios de los períodos de gracia otorgados por parte de las organizaciones de divulgación de vulnerabilidades a los fabricantes de software. En segundo lugar, propone dos nuevas métricas para vulnerabilidades de software. Por último, en la tercera sección, introduce el concepto de *hidden impact bugs*, o defectos de impacto oculto. La definición se encuentra presentada también en [35] y en [33], y aunque en este último trabajo aparezca como *hidden impact vulnerabilities*, citando a otro trabajo [1], se trata por supuesto del mismo concepto. Esto es, vulnerabilidades que fueron en primera instancia reportadas como *bugs* o defectos, y que su impacto, en relación a los aspectos de seguridad, tiempo después de ese reporte inicial, cuando ya se ha publicado un parche que, hasta ese entonces, se entendía que solucionaba sólo un defecto del software en cuestión. En relación a esto último, se debe tener en cuenta una métrica utilizada por el autor a la hora de exponer sus resultados: El retraso de impacto, o *impact delay*, al que se definió como el tiempo transcurrido desde la publicación del defecto -en la forma de parche-, hasta el momento en que se asignó un CVE al *bug* por haber sido entonces identificado como una vulnerabilidad.

El autor, junto a otros, publicaron los resultados, en un segundo período de análisis, de una revisión de defectos de impacto oculto en el código fuente (parches) del *Kernel* de Linux y en el motor de base de datos MySQL en el año 2012 [33]. Luego, en el año 2013, realizaron y publicaron [35] un análisis extendido sobre MySQL exclusivamente.

Los experimentos consistieron en revisar detalladamente un subconjunto de los defectos o *bugs* reportados en para cada uno de los proyectos de soft-

ware de código abierto analizados. El código fuente relativo o asociado a los defectos de este subconjunto fue minuciosamente analizado para determinar la proporción que no había sido clasificada como vulnerabilidad cuando debían haberlo sido. Los autores finalmente utilizan estos resultados para hacer una extrapolación y así poder estimar un porcentaje sobre el total de los defectos reportados.

Los resultados de estos trabajos, como se ha comentado más arriba, han sido revisados y extendidos al incorporarlas el autor a su tesis de maestría, publicada provisoria e informalmente. Desde este último trabajo se resume lo que sigue:

- En relación a los resultados que obtuvieron con el experimento del *Kernel* de Linux, en el segundo período de análisis que correspondió a Enero de 2009 hasta Abril de 2011, los resultados podrían resumirse de la siguiente manera: de un total de 185 defectos de impacto oculto, 73 de ellos (el 39%) tuvieron un retraso de impacto de al menos 2 semanas; 55 defectos (30%) un retraso de al menos 4 semanas; y 29 (16%) de al menos 8 semanas.

- En el caso de MySQL, el total de defectos de impacto oculto fue de 29, y de ellos, 19 (65%) tuvo retraso de impacto de al menos 2 semanas, también 19 para al menos 4 semanas de retraso; y por último, 16 defectos (55%) tuvieron retraso de impacto de al menos 8 semanas.

3.2 Problemas ya corregidos en la versión *devel* de NTP

Una implmentación de referencia del protocolo para el sincronizar los relojes de equipos en red, conocido como NTP, de sus siglas en inglés *Netowork Time Protocol*, se encuentra por defecto distribuida como parte de sistemas operativos Linux, entre otros. En diciembre de 2014, una serie de problemas de seguridad, algunos de los cuales fueron categorizados como vulnerabilidades críticas, fueron descubiertos. La información relacionada estos problemas fue hecha pública junto con los parches que corregirían los defectos; sin embargo, en la base de datos de defectos o *bugs* del equipo de desarrollo, en los mensajes relacionados a algunos de de estos defectos -donde también participaron quienes descubrieron los problemas-, se puso de manifiesto que algunos de estos defectos ya habían sido corregidos en la versión de desarrollo o *devel* de la implementación, años atrás. Mas información puede consultarse en su portal de seguridad [16]; aquí, a continuación, se describen brevemente algunos detalles, para notar que una comparación entre versiones podría haber puesto en evidencia correcciones aún no implementadas en la versión estable o *release*, no *devel*, de la implementación.

Aunque de acuerdo a registros históricos en su *Bugzilla* esta información estuvo públicamente accesible recién el día 20/12/2014, en los mensajes correspondientes al *bug* 2665 [17] en relación a una llave criptográfica débil por defecto, se comenta que la vulnerabilidad ya había sido "corregida" en la versión de desarrollo anterior del proyecto, ntp-dev (4.2.7), y más adelante se especifica "(4.2.7p11) 2010/01/28".

Seguidamente, otro ejemplo: en el *bug* 2666 [18], relativo a al generador de números aleatorios -no criptográficamente seguro e inicializado con una semilla débil-, aunque luego volvió a emparcharse antes del *release* de la versión 4.2.8, se comentó que en la versión de desarrollo n 4.2.7p230 (del primero de noviembre de 2011) el problema ya había sido corregido, refiriendo una mejora en el manejo de la semilla. En este caso particularmente, la corrección primera, del año 2011, hecha en la versión de desarrollo, es criticada y no resulta ser lo que finalmente se corrige en la versión estable. Sin embargo, valdría citar de cualquier forma este ejemplo para el propósito de mostrar cómo algunos problemas son corregidos y publicados -a través del código fuente- por los desarrolladores antes de que el problema de seguridad o vulnerabilidad asociada se haga también pública.

3.3 Más de un parche para vulnerabilidades *shellshock*

Descubiertas también recientemente, en el *shell* de sistemas UNIX bash, estas vulnerabilidades representan otro ejemplo de problemas de seguridad críticos que no son completa o correctamente corregidos con los primeros parches publicados. La importancia y criticidad de la vulnerabilidad se destacó por ejemplo en el medio electrónico *The Register* [28] al titular imperativamente que debía instalarse un parche en lo inmediato porque el *bug* dejaría a los sistemas operativos Linux y OS X completamente abiertos. La noticia, al igual que el aviso público de Red Hat [26] y los paquetes actualizados, son del día 24 de septiembre de 2014. Sin embargo, luego y durante el lapso de unos pocos días, otros problemas relacionados fueron descubiertos, para los cuales Red Hat no proveyó paquetes actualizados sino hasta el día 26 de septiembre de 2014 [27].

3.4 Descubrimiento simultáneo de vulnerabilidad *Heartbleed*

Otra vulnerabilidad importante, crítica, del año 2014 fue la conocida como Heartbleed. Se trataba de un *bug* explotable en la ampliamente implementada librería de código abierto OpenSSL. Medio millón de sitios afectados indicó Bruce Schneier, calificando el problema como "catastrófico" [29]. Aunque desde esta misma fuente se hace referencia a un artículo que habla de posible evidencia de la explotación de la vulnerabilidad en el año

2013, lo que va destacarse en este apartado es lo relativo al supuesto descubrimiento simultáneo del problema. Según un representante de Red Hat Security [6], la coincidencia de dos hallazgos del mismo problema, al mismo tiempo, incrementa el riesgo de mantener por mayor tiempo esta vulnerabilidad sin publicar los parches que para su corrección. Ese mismo día entonces, a consecuencia de los anterior, se supone, OpenSSL libera paquetes actualizados. Para ese entonces, la vulnerabilidad era ya pública y conocida, sin embargo, por el inesperado apuro al que se vieron sujetos las personas encargadas de hacer la publicación, o divulgación responsable, del problema, no pudieron coordinar como hubiera sido óptimo la disponibilidad de parches y paquetes de actualización para todas las distribuciones de Linux. Se cita este ejemplo únicamente para comentar que no es poco común que diferentes distribuciones de Linux publiquen parches de código fuente y/o paquetes de actualización de forma no coordinada.

3.5 Vulnerabilidad en OpenBSD luego de parche mal categorizado

En el año 2007 fue publicada una vulnerabilidad descubierta en la implementación de IPv6 del *kernel* del sistema operativo OpenBSD [21]. Un paquete ICMP especialmente construido, al ser recibido por una función interna que no calcula correctamente el tamaño o longitud de un buffer, generaría un desbordamiento o *buffer overflow* explotable remotamente. Se ejemplifica con este caso el hecho de que, según el autor, la vulnerabilidad fue descubierta al analizar, e intentar reproducir, el problema corregido por un parche no categorizado como vulnerabilidad, sino como un *Reliability fix*, o parche de fiabilidad [20].

4 Proyecto de herramienta de análisis de parches

4.1 Herramienta para el Análisis Automático de Parches, o AAP, versión 0.1b

Si bien entre las herramientas de análisis automático de software se incluyen a aquellas que realizan análisis estático, análisis dinámico, herramientas de compilación y varias técnicas de testeo [30], esta herramienta, especializada en parches y diferencias entre versiones, implementa lo que sería equivalente a un análisis estático; valiéndose de reglas por patrones definidas por el usuario de la interfaz Web. Es el proceso desatendido que realiza estas revisiones, ejecutando también otras herramientas externas de análisis de código fuente generales, para luego alertar y reportar según corresponda, de acuerdo a cómo se haya configurado en cada caso o para cada usuario analista. De esta manera, éste podrá revisar ordenadamente la información filtrada de acuerdo a sus criterios, a la vez que participar vía notas o comen-

tarios en intercambios con otros usuarios analistas.

4.1.1 Idea

El proyecto se basa principalmente en dos consideraciones:

- La primera corresponde a la idea detrás de implementaciones actuales de herramientas para el análisis automático de malware, como por ejemplo *Cuckoo Sandbox* [11]. En la presentación referenciada se hace mención a dos problemas que sufren quienes realizarían este tipo de análisis manualmente: el volumen de malware *in the wild*, creciente día a día, y el tiempo que le demandaría a un analista una revisión acabada. Esto hace que ya no sea posible, por una persona o equipo de personas, analizar manualmente más que un subconjunto relativamente muy pequeño de todas las muestras capturadas día a día. De manera análoga, el análisis de código fuente, aunque de un ámbito distinto, también debe resolver los mismos problemas: el de volumen de proyectos, código y parches en constante crecimiento, y lo trabajoso de su análisis. Es en este sentido que el análisis de código, automatizado hasta dónde sea posible, se vuelve un mecanismo valedero para abarcar un mayor alcance. Diferentes organizaciones relacionadas directa o indirectamente con la seguridad de la información realiza análisis del código fuente de los sistemas que utilizan o que auditan. Una parte de este análisis corresponde a verificar que modificaciones no introduzcan nuevos problemas o vulnerabilidades, como así también, si se trata de un parche, que el defecto esté correctamente categorizado y que la corrección resuelva efectivamente el problema.

- La segunda consideración se desprende de los trabajos publicados referidos al descubrimiento de nuevas vulnerabilidades -o del alcance de aquellas ya conocidas, a partir de, o a *posteriori*, de la distribución de parches en proyectos de código abierto. Los trabajos citados en el tercer apartado de este artículo ejemplifican concretamente esta consideración. En este sentido cabe destacar el trabajo académico de uno de los autores citados en su experimento con el *kernel* de Linux y el motor de base de datos MySQL [34, 35, 36, 33], los comentarios relativos a los últimos parches de la implementación de referencia del protocolo NTP [17, 18], y el descubrimiento de la vulnerabilidad explotable remotamente en OpenBSD [20].

4.1.2 Tecnologías utilizadas

La herramienta consiste básicamente de una aplicación basada en Web desarrollada en el lenguaje de programación Python [24]. Se utilizó el frame-

work Web Django [7], y se vale de la versatilidad de Git [10] para el manejo de código fuente. Las reglas propias de la herramienta, que son administradas desde la interfaz Web, se aplican luego de que se haya recuperado las últimas modificaciones aún no analizadas desde el repositorio remoto. Como herramienta externa de análisis estático de código fuente, la versión presentada utilizaría, opcionalmente, la herramienta Flawfinder [32].

4.1.3 Funcionalidad actualmente implementada

A través de su interfaz Web, la herramienta permitiría mantener un registro organizado de proyectos de software, referencias a sus repositorios, ramas o *branches*, sus actualizaciones, y las referencias a los usuarios analistas asignados a cada repositorio. También de esta forma se parametrizan las configuraciones generales para la revisión automática de parches y de diferencias entre versiones de los proyectos de software analizados. Esta información es manejada por los usuarios administradores del sistema.

Proceso automático Mediante el proceso *batch* o desatendido, integrado a la herramienta, se actualizarían los repositorios Git [10] y se realizarían los análisis configurados para cada una de sus ramas o *branches* automáticamente. El resultado de este proceso, en tanto corresponda, incluirá la generación de alertas en el sistema -opcionalmente enviando correos electrónicos de aviso- para los analistas asignados. Se registrará además información de *log* o bitácora detalladamente que posibilitará el seguimiento y control del proceso.

Análisis Se recuperarán todos las diferencias actualizadas o *commits* del repositorio para la evaluación de las reglas configuradas. Así por ejemplo se realizarían pruebas como la búsqueda por patrones especificados -en código fuente y en nombres de directorios y archivos-, la ejecución de herramientas de análisis de código generales para comparar sus resultados post actualización contra resultados previos (la implementación propuesta de momento sólo utilizaría Flawfinder), la revisión de nuevas diferencias entre las ramas o *branches* de un repositorio, y demás chequeos que podrían registrar alertas a ser revisadas luego por los usuarios analistas. Ejemplo de esto último podría ser el aviso de detección de código que implementaría funcionalidades criptográficas.

Interfaz Web - manejo de configuración y alertas La interfaz Web permitiría la consulta de las alertas generadas por el proceso automático,

sumar notas personales y consultar las notas de otros analistas o usuarios del sistema. A su vez, aunque en futuras versiones se planea hacer más flexible este módulo, la interfaz permitiría configurar parte de los parámetros involucrados en los análisis automáticos que se realizan en proceso desatendido.

4.1.4 Software libre de código abierto

El código fuente de la herramienta será publicado en un servicio gratuito para el control de versiones de proyectos de software libre u *opensource*. Será licenciado bajo los términos de la GPL, o *GNU General Public License* de la *Free Software Foundation*. Se planea la continuación del desarrollo para ampliar la funcionalidad actual de la herramienta. También se espera que su publicación pueda lograr que otros desarrolladores se interesen e involucren en el proyecto.

5 Trabajo a futuro

En este trabajo, los esfuerzos de investigación se han concentrado determinadas cuestiones específicas, habiéndose reservado para trabajos futuros una serie de aspectos que se comentan seguidamente.

- Continuar con la mejora en las funcionalidades de la herramienta, aún quedaría por agregarle algunas como pueden ser las de incorporar algoritmos que permitan la lectura e interpretación de los comentarios que los programadores realizan sobre el código fuente, posibilitando de esta manera detectar mediante el uso de análisis sintáctico posibles errores en los parches dentro de un mismo proyecto.

- Sería interesante poder detectar los errores en los parches analizando los comentarios dentro de los foros destinados a la comunicación entre los desarrolladores.

- Realizar pruebas de las funcionalidades de la herramienta para detectar errores de programación y mejorar las reglas y los algoritmos de análisis.

- Investigar, probar e incorporar algoritmos de inteligencia artificial que permitan ayudar al análisis automático de parches permitiendo simplificar el filtrado de mayor cantidad de líneas para una posterior lectura por parte del analista (humano).

- Considerar la integración de una base de datos de reportes de vulnerabilidades o *advisories*, a actualizar automáticamente, de manera de

relacionarlos con sus respectivos parches y permitir estudios comparativos.

- En lo que sería prácticamente otra línea, investigar si aplicaría, al proyecto y a la misma herramienta, el análisis de parches distribuidos como código compilado y no fuente (binario, *bytecode*, ...).
- Investigar factibilidad de implementar alternativas al análisis estático, también de manera automática (por ejemplo, utilizando técnicas de *fuzz testing*).
- Implementar los mecanismos necesarios en la herramienta para para poder realizar análisis estadísticos y minería de datos que puedan aportar nuevo conocimiento.

6 Resumen

Se han descripto brevemente cómo pueden contribuir a la calidad -y por ende a la seguridad- del software las revisiones automatizadas de su código fuente. Se diferenciaron los conceptos de defecto o *bug* y vulnerabilidad. Una serie de ventajas y desventajas de la sistematización de estos análisis fue enumerada, junto con algunas estadísticas de trabajos citados y referencias a herramientas disponibles. A través de ejemplos concretos de vulnerabilidades publicadas, se intentó justificar la necesidad de la revisión de parches y diferencias entre versiones de proyectos de software. A continuación, de acuerdo al objetivo principal del trabajo, se presentó una herramienta para el análisis automático de código fuente de parches. Basada en otras herramientas de código libre u *opensource*, este desarrollo se distribuirá bajo el mismo esquema de licenciamiento. Se han comentado lo planeado para trabajos futuros sobre el desarrollo de la herramienta.

Referencias

[1] J. Arnold, T. Abbott, W. Daher, G. Price, N. Elhage, G. Thomas, A. Kaseorg *Security Impact Ratings Considered Harmful*. Proceedings 12th Conference on Hot Topics in Operating Systems. USENIX. Mayo de 2009. [en línea: `http://www.inl.gov/technicalpublications/Documents/5588153.pdf` - accedido el 29/12/2014].

[2] P. Black, M. Kass, M. Koo, M. Fong. *Source Code Security Analysis Tool Functional Specification Version 1.1*. NIST Special Publication 500-268 v1.1. Febrero de 2011. [en línea: `http://samate.nist.gov/docs/source_code_security_analysis_spec_SP500-268_v1.1.pdf` - accedido el 29/12/2014].

[3] F. Buck. *Indicators of Quality Inspections.* IBM Technical Report TR21.802, Systems Comm. Diciembre de 1981.

[4] CERT Division - Secure Coding *Secure Coding Tools.* CERT, Software Engineering Institute (SEI), Carnegie Mellon University. [en línea: `http://www.cert.org/secure-coding/tools/index.cfm` - accedido el 29/12/2014].

[5] C. Corley, L. Etzkorn, N. Kraft, S. Lukins. *Recovering Traceability Links between Source Code and Fixed Bugs via Patch Analysis.* University of Alabama. 2008. [en línea: `http://www.cs.wm.edu/semeru/tefse2011/papers/p31-corley.pdf` - accedido el 29/12/2014].

[6] M. Cox *Heartbleed.* Mark J. Cox Google+. [en línea: `https://plus.google.com/+MarkJCox/posts/TmCbp3BhJma` - accedido el 29/12/2014].

[7] Django Framework. *Django overview.* Django Software Foundation. [en línea: `https://www.djangoproject.com/start/overview/` - accedido el 29/12/2014].

[8] M. Egele, D. Brumley, Y. Fratantonio, C. Kruegel. *An empirical study of cryptographic misuse in android applications.* CCS '13 Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. Pages 73-84. 2013. EE.UU. [en línea: `http://www.cs.ucsb.edu/~chris/research/doc/ccs13_cryptolint.pdf` - accedido el 29/12/2014].

[9] C. Kemerer, M. Paulk. *The Impact of Design and Code Reviews on Software Quality: An Empirical Study Based on PSP Data.* IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 35, NO. XX Abril de 2009. [en línea: `http://www.pitt.edu/~ckemerer/PSP_Data.pdf` - accedido el 29/12/2014].

[10] Git SCM. *About Git.* Git - Software Freedom Cnonservancy. [en línea: `http://git-scm.com/about` - accedido el 2912/2014].

[11] C. Guarnieri *One Flew Over the Cuckoo's Nest.* Hack In The Box 2012. Mayo de 2012. Hollanda. [en línea: `http://sebug.net/paper/Meeting-Documents/hitbsecconf2012ams/D1T1%20-%20Claudio%20Guarnieri%20-%20One%20Flew%20Over%20the%20Cuckoos%20Nest.pdf` - accedido el 29/12/2014].

[12] Gerrit Website *Gerrit Code Review.* Google Inc. [en línea: `https://code.google.com/p/gerrit/` - accedido el 2912/2014].

[13] A. Keromytis, J. Wright, T. de Raadt. *The Design of the OpenBSD Cryptographic Framework.* International Conference on Human System

Interactions (HSI). Junio de 2012. Australia. [en línea: http://www.thought.net/papers/ocf.pdf - accedido el 29/12/2014].

[14] D. Lazar, H. Chen, X. Wang, N. Zeldovich. *Why does cryptographic software fail?: a case study and open problems*. Proceedings of 5th Asia-Pacific Workshop on Systems Article No. 7. 2014. EE.UU. [en línea: http://pdos.csail.mit.edu/papers/cryptobugs:apsys14.pdf - accedido el 29/12/2014].

[15] A. Mujic. *Reimplementation of CryptoLint tool*. Blog for and by my students. Diciembre de 2013. [en línea: http://sgros-students.blogspot.com.ar/2013/12/reimplementation-of-cryptolint-tool.html - accedido el 29/12/2014].

[16] Network Time Protocol project. *NTP Security Notice*. NTP support website. Network Time Foundation. [en línea: http://support.ntp.org/bin/view/Main/WebHome - accedido el 29/12/2014].

[17] Network Time Protocol project. *Bug 2665 - Weak default key*. NTP Bugzilla. Network Time Foundation. [en línea: http://bugs.ntp.org/show_bug.cgi?id=2665 - accedido el 29/12/2014].

[18] Network Time Protocol project. *Bug 2666 - non-cryptographic random number generator with weak seed*. NTP Bugzilla. Network Time Foundation. [en línea: http://bugs.ntp.org/show_bug.cgi?id=2666 - accedido el 29/12/2014].

[19] NIST *Source Code Security Analyzers*. SAMATE - NIST. [en línea: http://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html - accedido el 29/12/2014].

[20] A. Ortega *OpenBSD Remote Exploit*. Core Security. Julio de 2007. [en línea: https://www.blackhat.com/presentations/bh-usa-07/Ortega/Whitepaper/bh-usa-07-ortega-WP.pdf - accedido el 29/12/2014].

[21] A. Ortega, G. Richarte. *OpenBSD Remote Exploit*. Core Security. Abril de 2007. [en línea: https://www.blackhat.com/presentations/bh-usa-07/Ortega/Whitepaper/bh-usa-07-ortega-WP.pdf - accedido el 29/12/2014].

[22] OWASP Wiki. *Source Code Analysis Tools*. The Open Web Application Security Project (OWASP). Últ. mod. 29/10/2014. [en línea: https://www.owasp.org/index.php/Source_Code_Analysis_Tools - accedido el 29/12/2014].

[23] Phabricator Website. *Phabricator, an open source, software engineering platform.*. Phacility, Inc. [en línea: `http://phabricator.org/` - accedido el 29/12/2014].

[24] Python Website. *About Python.* Python Software Foundation. [en línea: `https://www.python.org/about/` - accedido el 29/12/2014].

[25] R. Radice. *High Quality Low Cost Software Inspections.* Paradoxicon Publishing. 2002.

[26] Red Hat. Seguridad. Base de datos de CVE. *CVE-2014-6271.* Red Hat Customer portal. 24 de Septiembre de 2014. [en línea: `https://access.redhat.com/security/cve/CVE-2014-6271` - accedido el 29/12/2014].

[27] Red Hat. Seguridad. Base de datos de CVE. *CVE-2014-7169.* Red Hat Customer portal. 24 de Septiembre de 2014. [en línea: `https://access.redhat.com/security/cve/CVE-2014-7169` - accedido el 29/12/2014].

[28] The Register. J. Leyden. *Patch Bash NOW: 'Shellshock' bug blasts OS X, Linux systems wide open.* The Register online tech publication. 24 de Septiembre de 2014. [en línea: `http://www.theregister.co.uk/2014/09/24/bash_shell_vuln/` - accedido el 29/12/2014].

[29] B. Schneier. *Heartbleed.* Schneier on Security, Blog. Abril de 2014. [en línea: `https://www.schneier.com/blog/archives/2014/04/heartbleed.html` - accedido el 29/12/2014].

[30] R. Seacord, W. Dormann, J. McCurley, P. Miller, R. Stoddard, D. Svoboda, J. Welch *Source Code Analysis Laboratory (SCALe).* CERT, Software Engineering Institute (SEI), Carnegie Mellon University. Abril de 2012. [en línea: `https://resources.sei.cmu.edu/asset_files/TechnicalNote/2012_004_001_15440.pdf` - accedido el 29/12/2014].

[31] W. Weimer. *Patches as Better Bug Reports.* University of Virginia. 2006. [en línea: `https://www.cs.virginia.edu/~weimer/p/p181-weimer.pdf` - accedido el 29/12/2014].

[32] D. Wheeler. *Flawfinder.* David A. Wheeler's Personal Home Page - Flawfinder Home Page. [en línea: `http://www.dwheeler.com/flawfinder/` - accedido el 29/12/2014].

[33] D. Wijayasekara, M. Manic, J. Wright, M. McQueen. *Mining Bug Databases for Unidentified Software Vulnerabilities.* Proceedings International Conference on Human System Interactions (HSI). Junio de 2012, Perth, Australia. [en línea: `http://www.inl.gov/technicalpublications/Documents/5588153.pdf` - accedido el 29/12/2014].

[34] J. Wright. *Software Vulnerabilities: Lifespans, Metrics, And Case Study*. Master of Science Thesis. University of Idaho. Mayo de 2014. [en línea: `http://www.thought.net/papers/thesis.pdf` - accedido el 29/12/2014].

[35] J. Wright, J. Larsen, M. McQueen. *Estimating Software Vulnerabilities: A Case Study Based on the Misclassification of Bugs in MySQL Server*. Proceedings International Conference of Availability, Reliability, and Security (ARES). Septiembre de 2013. pp. 72-81. Regensburg, Alemania. [en línea: `http://www.inl.gov/technicalpublications/Documents/5842499.pdf` - accedido el 29/12/2014].

[36] J. Wright, M. McQueen, L. Wellman. *Analyses of Two End-User Software Vulnerability Exposure Metrics (Extended Version)*. Information Security Technical Report, 17(4), Elsevier. Abril de 2013. pp. 44-55. [en línea: `http://www.thought.net/papers/INL-JOU-12-27465-preprint.pdf` - accedido el 29/12/2014].