UNIVERSIDAD DE LA REPÚBLICA
FACULTAD DE INGENIERÍA

Integración y automatización de seguridad en procesos de desarrollo de metodologías ágiles

Autor: Guillermo Gabarrín

Trabajo de Tesis para
la obtención del Título de
Magíster en Seguridad Informática

Centro de Posgrados y Actualización Profesional en Informática, Instituto de Computación, Facultad de Ingeniería, Universidad de la República

Octubre 2019
Montevideo, Uruguay

Tutor: MSc. Felipe Zipitría
Director Académico: Dr. Ing. Gustavo Betarte

Índice general

Índice de cuadros

Índice de figuras

Listados de código

Capítulo 1

Introducción

1.1. Motivación

Durante décadas, múltiples autores han estudiado prácticas y herramientas que permiten integrar seguridad en procesos de desarrollo de software de metodologías tradicionales. Sin embargo, en los últimos años el uso de dichas metodologías ha disminuido, dando lugar a un incremento en la adopción del uso de metodologías ágiles[LSA11]. Las diferencias que estas metodologías mantienen entre sí, han provocado que sea de interés estudiar nuevas técnicas que permitan integrar también seguridad en procesos de desarrollo ágiles. Estos procesos se caracterizan por un desarrollo iterativo e incremental, abierto a cambios en los requerimientos, y promueven obtener un feedback constante por parte de los usuarios finales. A su vez, priorizan la simplicidad y buscan desarrollar únicamente lo indispensable, siendo la entrega de software funcional de manera frecuente uno de sus principales objetivos, incluso en plazos que pueden ser tan cortos como un par de semanas.

Múltiples prácticas de seguridad han surgido alineadas a estas características. En cuanto a la identificación y documentación de requerimientos, aparecen técnicas específicas de seguridad, similares a las user stories, como lo son *abuser* y *security stories*. A su vez, surgen estrategias propias para la gestión de riesgos y vulnerabilidades, así como herramientas y procedimientos que permiten hacer revisiones de código y tests enfocados en seguridad. Sin embargo, los continuos cambios y mejoras en las prácticas ágiles, principalmente en lo que respecta a la automatización de tareas, llevan a que deban surgir constantemente nuevas técnicas de seguridad que se adapten a dichas prácticas.

Los equipos de seguridad deben integrarse a prácticas características de las metodologías ágiles, e intentar beneficiarse al máximo de sus ventajas. La primera de ellas refiere a que los desarrolladores sean capaces de integrar los cambios que realizan sobre el software de una manera sencilla y automatizada, de forma que permita detectar problemas de manera prematura. En cuanto a los despliegues continuos, los mismos permiten, además de integrar el código, desplegar el software en producción también de manera automatizada. De esta forma se reducen riesgos, y se aumenta la confiabilidad y repetibilidad de estos proceso. A su vez, procesos de seguridad pueden ser integrados como parte de estas prácticas, de forma de incluir pruebas de seguridad que se verifiquen continuamente cada

vez un cambio en el código es realizado. Esto permite detectar bugs y fallas en el software de manera automatizada, constante, y prematura, reduciendo el riesgo de liberar vulnerabilidades en ambientes productivos

Por otra parte, en la actualidad estás prácticas comúnmente son acompañadas de un cambio cultural denominado DevOps, que promueve la comunicación y colaboración entre Product Manager, desarrolladores de software, y los encargados de operaciones[Deb10]. El mismo busca automatizar y monitorear los procesos de codificación, building, testing, configuración, empaquetado, entre otros, mediante un único lenguaje en común: el código fuente. Sumado a ello, recientemente han surgido iniciativas que buscan integrar también la seguridad como parte de dicho cambio cultural, dando lugar al concepto de DevSecOps[Mac12]. De esta forma, se busca que la concientización respecto a la seguridad sea incluida como parte de la cultura organizacional y que todos los equipos se sientan responsables de la misma. A su vez, este movimiento busca automatizar tareas centrales de seguridad, incluyendo la implementación de controles, la incorporación de logs, el monitoreo de eventos, la gestión de configuración y parches, el análisis de vulnerabilidades, entre otros.

Finalmente, debido a los constantes cambios que las organizaciones presentan, resulta interesante poder evaluar el nivel de madurez en el cual las organizaciones implementan estas prácticas. Esto ha dado surgimiento recientemente a múltiples modelos de madurez, tanto sobre DevOps, como sobre DevSecOps. Los mismos permiten encontrar oportunidades de mejora y diseñar un roadmap de próximos pasos a seguir.

La motivación de esta tesis es estudiar las técnicas, prácticas y herramientas que permiten integrar y automatizar la seguridad en los procesos de desarrollo ágiles. Particularmente, resulta de interés conocer aquellas que se encuentren alineadas a las prácticas ágiles, cuya implementación resulte transparente para los equipos de desarrollo, y que puedan maximizar los beneficios ofrecidos por las prácticas propias de estas metodologías.

Se evaluará cómo es posible automatizar la integración de seguridad en procesos de desarrollo ágiles con el objetivo de entregar valor al cliente de manera rápida.

Esta tesis busca responder las siguientes preguntas:

- ¿Qué dificultades tienen las organizaciones para integrar seguridad en sus procesos?
- ¿Qué prácticas permiten sobrellevar dichas dificultades?
- ¿Qué controles pueden ser integrados?
- ¿Cómo pueden ser automatizados dichos controles?
- ¿Qué desafíos nuevos surgen al implementar dichos controles y prácticas?
- ¿Existen modelos que permitan evaluar la madurez de los procesos de la organización?
- ¿Qué resultados y métricas pueden ser tomados?

A su vez, se propone la realización de un caso de estudio práctico en el cual se aplicarán los conocimientos obtenidos. Para ello se desarrollarán y adaptarán herramientas con el objetivo de alcanzar niveles de seguridad básicos en un proyecto de desarrollo de software real.

1.2. Contribuciones

La principal contribución de esta tesis refiere al caso de estudio. El mismo describe un escenario que busca representar un proyecto de desarrollo de software real, y detalla las metodologías y tecnologías utilizadas. Presenta el diseño y la implementación de un *application security pipeline*, que busca cubrir prácticas de seguridad correspondientes a un nivel de entendimiento básico. Para ello se toma como referencia el modelo de madurez *DevSecOps Maturity Model* (DSOMM) descrito en la subsección 4.3.2. Dichas prácticas incluyen la definición de procesos de build y deploy, ejecución de tests unitarios y tests de seguridad, tests de componentes con vulnerabilidades inseguras, entre otras. A su vez, presenta métricas acerca de la ejecución del pipeline y un análisis de los resultados obtenidos.

Adicionalmente, esta tesis ha contribuido realizando una investigación del estado del arte respecto a técnicas y prácticas que permiten la integración de la seguridad en los procesos de desarrollo de software ágiles, así como acerca de su automatización. A su vez, presenta y compara brevemente modelos de madurez que permiten identificar posibles áreas a mejorar en este contexto y definir *roadmaps* para su implementación.

Esta tesis esta compuesta de la siguiente forma. El Capítulo 2 describe las metodologías de desarrollo de software, presenta metodologías tradicionales y ágiles comúnmente utilizadas, y detalla ventajas y desventajas del uso de las mismas.

En el Capítulo 3 se presentan actividades y prácticas que permiten integrar y automatizar seguridad en procesos de desarrollo ágiles. A su vez, plantea qué herramientas pueden utilizarse con este fin, y cómo las mismas pueden ser organizadas.

El Capítulo 4 introduce modelos de madurez de DevOps, así como modelos de madurez aplicados al concepto de DevSecOps.

Un caso de estudio es presentado en el Capítulo 5. El mismo consiste en integrar prácticas de seguridad y automatizar la ejecución de herramientas de seguridad en un proyecto de software real de código abierto.

En el Capítulo 6 se presentan trabajos relacionados y diferentes enfoques a esta temática.

Por último, el Capítulo 7 concluye con un resumen del estudio realizado y direcciones para trabajo a futuro.

Capítulo 2

Metodologías de desarrollo de software

Una metodología de desarrollo de software es un marco utilizado para estructurar, planificar y controlar el desarrollo de un sistema[MMS+08]. Su aplicación permite mejorar el diseño, así como la gestión del producto y del proyecto. A su vez, permite alcanzar objetivos como la producción de un producto de software de buena calidad, la entrega del producto a tiempo al cliente, y el cumplimiento del proyecto con un presupuesto determinado. Para ello las metodologías dividen el trabajo en distintas fases, que juntas conforman el ciclo de vida de desarrollo. Dicho ciclo comienza cuando el producto es una idea, y termina una vez que el producto no se encuentre disponible para su uso. Típicamente incluye fases de requerimientos, diseño, implementación, testing y mantenimiento[KB14].

Actualmente existen distintas metodologías de desarrollo disponibles que han ido evolucionando a lo largo de los años, desde genéricas a diseñadas específicamente para resolver determinados tipos de proyectos particulares. Sin embargo, no existe una única metodología que nos asegure que se alcanzarán los objetivos deseados para cualquier tipo de proyecto de software, por lo que es necesario elegir la que mejor se adapte a las necesidades de cada proyecto. Para ello una de las principales decisiones que se debe tomar es la de aplicar una metodología tradicional o una ágil, pudiendo resultar cada una de ellas la más adecuada en situaciones distintas.

2.1. Metodologías de desarrollo tradicionales

Las metodologías tradicionales de desarrollo de software se caracterizan principalmente por enfocarse fuertemente en el análisis de requerimientos y en la planificación. Para ello invierten tiempo y esfuerzo desde el comienzo del proyecto en la identificación, definición y documentación de todos los requerimientos. Idealmente, éstos últimos deben permanecer estables durante todo el ciclo de vida del proyecto. Esto es debido a que en base a su documentación se realiza la planificación de las siguientes fases del proyecto. Esto tiene como resultado que éstas metodologías puedan resultar muy rigurosas para sus practicantes, y que sea difícil hacer una correcta gestión ante un posible cambio de requerimientos[Awa05].

A su vez, éstas metodologías se basan en la ejecución de series secuenciales de distintas etapas. Las mismas incluyen la definición de requerimientos, la construcción de soluciones, el testing y el ´despliegue. A continuación se detallan algunas de las metodologías tradicionales de desarrollo comúnmente utilizadas como lo son *Waterfall*, *Spiral* y *Prototyping*. Las mismas representan paradigmas, y son utilizadas como base por otras metodologías. No se detallarán, quedando fuera del alcance de este trabajo, metodologías y frameworks como *Microsoft Solutions Framework* (MSF) de Microsoft, *Team Software Process* (TSP) del *Software Engineering Institute* (SEI) de la Universidad de Carnegie Mellon y *Unified Process* (UP).

2.1.1. Waterfall

La metodología Waterfall, o desarrollo en cascada, fue definida por Winston W. Royce en 1987 y es reconocida como la primera metodología de desarrollo de software[DES14]. La misma hace un fuerte énfasis desde que comienza el proyecto sobre la planificación, programación de tiempos, presupuestos e implementación del sistema. Esto permite que el avance del desarrollo del sistema sea medible yque se apliquen estrictos controles que aseguren que la documentación sea adecuada, así como que las revisiones del diseño aseguren la calidad y confiabilidad del software desarrollado. Sin embargo, estas características resultan en que Waterfall sea considerada una metodología inflexible, lenta y costosa. Un ejemplo de ello es que inconsistencias y cambios en los requerimientos pueden afectar drásticamente la planificación y presupuestos originales. A su vez, esta metodología presenta otras dificultades, como el hecho de que problemas en el software podrían ser descubiertos de manera tardía durate las etapas de testing, y que la excesiva documentación, que debe ser actualizada de manera continua, consume demasiado tiempo y esfuerzo[MMS+08].

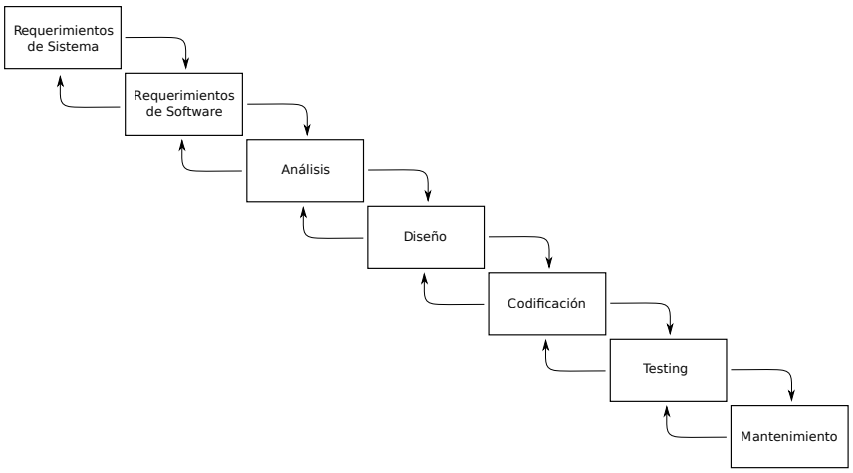

Figura 2.1: Modelo Waterfall

Por otra parte, Waterfall se compone de 7 etapas que deben ser completadas en orden, y cuya ejecución debería tener como resultado software estable. Dichas etapas son: requerimientos de sistemas, requerimientos de software, análisis, diseño, codificación, testing y operaciones (mantenimiento). Su orden se basa en

el siguiente concepto: a medida que cada etapa avanza y el diseño es más detallado, hay una interacción entre los pasos previos y posteriores, pero raramente hay interacción con etapas más lejanas en la secuencia, como se ve en la Figura 2.1. De todas formas, es posible que durante un determinado proyecto sea necesario aplicar excepciones a dicho orden. Un ejemplo de ello puede ser un caso en el cual durante la etapa de testing se identifiquen problemas que afecten el diseño del programa y los requerimientos de software, lo que provocaría que se deba volver a etapas de diseño o de requerimientos, respectivamente[Roy87].

Debido a las mencionadas características, esta metodología es adecuada para proyectos grandes, costosos y complejos que tienen objetivos y soluciones claros. A su vez, los requerimientos deben ser estables y no deberán cambiar durante el ciclo de vida de desarrollo del sistema. Su aplicación también puede resultar adecuada cuando el *Project Manager*[Gad59] asignado al proyecto no tiene una vasta experiencia, o si la composición de los miembros del equipo fuese inestable. Sin embargo, esta metodología no es adecuada para el desarrollo de sistemas cuyos requerimientos no están claros o podrían cambiar, o si se tratase de un sistema de tiempo real u orientado a eventos[MMS+08].

2.1.2. Spiral

La metodología de desarrollo Spiral, o de desarrollo en espiral, fue publicada por Barry Boehm en 1986[Boe86]. La misma se caracteriza por iterar repetidamente sobre un conjunto de etapas con el objetivo de minimizar los riesgos del proyecto. Esto se logra mediante la descomposición del proyecto en segmentos más pequeños, que permiten aplicar cambios con mayor facilidad y realizar una evaluación continua de riesgos a lo largo de todo el ciclo de vida de desarrollo.

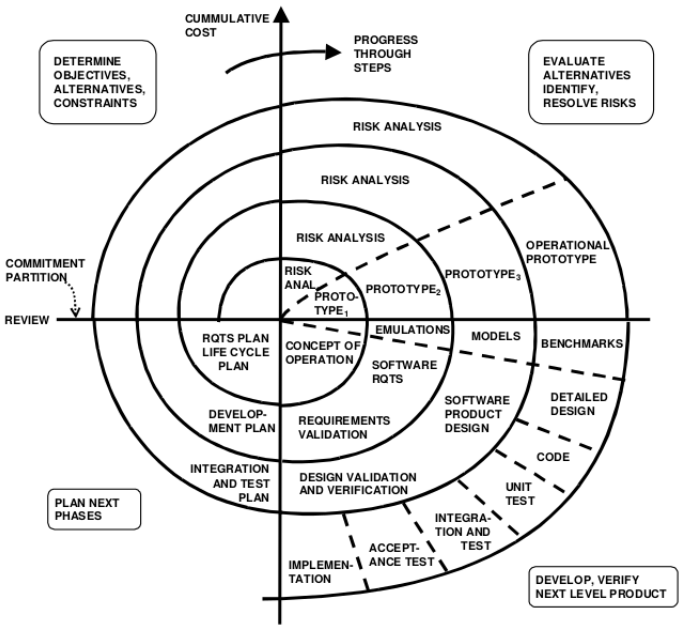

Figura 2.2: Modelo Spiral[Boe86]

Como se ve en la Figura 2.2, el espiral recorre 4 cuadrantes (etapas) básicos. El primer cuadrante consiste en determinar los objetivos, alternativas y restricciones de la iteración. Por su parte, el segundo refiere a evaluar alternativas, e identificar riesgos, así como reducirlos. El tercero incluye el desarrollo y la verificación de los entregables propios de la iteración. Por último, el cuarto consiste en realizar una revisión del proyecto y planificar la siguiente iteración[MG10].

A su vez, este modelo ofrece un alto nivel de customización para cada proyecto, y permite la integración de otras metodologías que se consideren más adecuadas para las etapas de desarrollo en cada iteración del espiral. Debido a que dicha integración puede resultar compleja y podría limitar la reutilización, es conveniente contar con un Project Manager con vasta experiencia y con fuertes conocimientos técnicos para llevarla adelante. Sin embargo, si esta complejidad adicional no es gestionada correctamente puede provocar que el proyecto derive en el uso de Waterfall.

La aplicación de Spiral puede ser apropiada en proyectos en los cuales minimizar riesgos sea una prioridad, así como en aquellos proyectos que requieran una fuerte aprobación de requerimientos y revisión de documentación. También puede resultar conveniente en proyectos medianos y grandes, y ha resultado ser efectiva en la implementación de proyectos internos de una organización debido a que es más fácil identificar riesgos propios[DES14]. Sin embargo, su uso no es recomendado si se desea disminuir el consumo de recursos, o en proyectos en los cuales disminuir riesgos no es prioritario[MMS+08].

2.1.3. Prototyping

La metodología de desarrollo conocida como Prototyping tiene como principal objetivo identificar de manera precisa los requerimientos del sistema a desarrollar. Para ello establece que se deben construir uno o más prototipos de demostración de la aplicación con el objetivo de involucrar a los interesados en el producto y obtener un feedback prematuro de los mismos. De esta forma, se busca mejorar la experiencia de usuario, así como identificar de manera temprana funcionalidades redundantes y requerimientos que hayan sido omitidos.

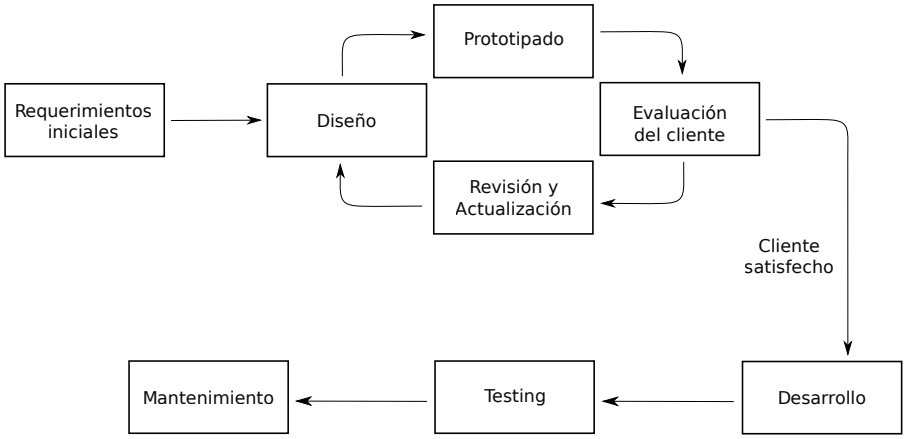

Figura 2.3: Modelo de Prototyping

Como se ve en la Figura 2.3, al comenzar el proyecto se realiza un análisis

inicial de requerimientos. Posteriormente se ejecutan varias iteraciones de diseño, prototipado y evaluación por parte del cliente, así como iteraciones de revisión y actualización a partir del feedback recibido. Este ciclo de iteraciones continúa hasta que el cliente se encuentre satisfecho con las funcionalidades que tendría el software. A partir de ese momento, se comienza con el desarrollo de la solución final, y luego se procede a realizar etapas de testing y mantenimiento.

Esta metodología puede resultar adecuada en proyectos grandes en los que es difícil definir los requerimientos, o proyectos de innovación en los cuales aún no se han desarrollado proyectos similares previamente[DES14]. Sin embargo, los prototipos son descartables y no se espera que sigan siendo desarrollados, por lo que el uso de los mismos tiene un costo asociado. Debido a estas razones, su desarrollo debe ser rápido para evitar un aumento en los costos del proyecto. Por otro lado, la aplicación de Spiral no aporta un valor adicional en proyectos cuyos requerimientos están claros, o en los cuales los riesgos asociados a la definición de requerimientos son bajos.

2.1.4. Resumen

Como se describió anteriormente, las metodologías tradicionales son adecuadas cuando se conocen los requerimientos de software en las etapas iniciales del proyecto, y los mismos son estables. Sin embargo, existe hoy en día una tendencia a nivel global de que los proyectos de software no cumplan con estas características. Por esta razón es necesario evaluar otro tipo de metodologías que puedan manejar de una manera más eficiente estos escenarios.

2.2. Metodologías ágiles de desarrollo

Las metodologías ágiles surgen en la década de los '90 como respuesta a los fracasos en proyectos de software gestionados mediante metodologías tradicionales, siendo su principal objetivo superar estas dificultades, así como permitir una exitosa gestión de proyectos cuyos requerimientos cambian constantemente. Fueron formalizadas en el año 2001, cuando 17 desarrolladores de software que las representaban se reúnen para analizar los conceptos en común entre sus respectivos acercamientos al desarrollo de software[All18]. Dicha reunión tuvo como resultado la fundación de la *Agile Alliance* (una organización sin fines de lucro dedicada a promover los conceptos del desarrollo ágil), así como la creación de un documento llamado *Agile Manifesto*[All01]. En dicho documento se definen las metodologías ágiles como todas aquellas metodologías de desarrollo que cumplen con los conjuntos de valores y principios transcritos a continuación.

Valores:

- Individuos e interacciones sobre procesos y herramientas.
- Software que funciona sobre documentación completa y comprensible.
- Colaboración con el cliente sobre negociación de contratos.
- Responder al cambio sobre seguir un plan.

Principios:

- Nuestra mayor prioridad es satisfacer al cliente mediante la entrega temprana y continua de software con valor.
- Aceptamos que los requisitos cambien, incluso en etapas tardías del desarrollo. Los procesos Ágiles aprovechan el cambio para proporcionar ventaja competitiva al cliente.
- Entregamos software funcional frecuentemente, entre dos semanas y dos meses, con preferencia al periodo de tiempo más corto posible.
- Los responsables de negocio y los desarrolladores trabajamos juntos de forma cotidiana durante todo el proyecto.
- Los proyectos se desarrollan en torno a individuos motivados. Hay que darles el entorno y el apoyo que necesitan, y confiarles la ejecución del trabajo.
- El método más eficiente y efectivo de comunicar información al equipo de desarrollo y entre sus miembros es la conversación cara a cara.
- El software funcionando es la medida principal de progreso.
- Los procesos Ágiles promueven el desarrollo sostenible. Los promotores, desarrolladores y usuarios debemos ser capaces de mantener un ritmo constante de forma indefinida.
- La atención continua a la excelencia técnica y al buen diseño mejora la Agilidad.

- La simplicidad, o el arte de maximizar la cantidad de trabajo no realizado, es esencial.
- Las mejores arquitecturas, requisitos y diseños emergen de equipos auto-organizados.
- A intervalos regulares el equipo reflexiona sobre cómo ser más efectivo para a continuación ajustar y perfeccionar su comportamiento en consecuencia.

A continuación se describen las metodologías *Scrum*, *Lean Software Development*, *Kanban* y *Extreme Programming* (XP), las cuales siguen los valores y principios mencionados.

2.2.1. Scrum

Scrum es un framework flexible y liviano para el desarrollo de software y gestión de proyectos que surge en el año 1993 con Ken Schwaber y Jeff Sutherland[Sch95].

Plantea un desarrollo iterativo e incremental en iteraciones llamadas *sprints*, que normalmente tienen una duración de entre 2 y 4 semanas. Para ello se compone de artefactos, personas y roles, y ceremonias. Dichos componentes son descriptos como se detalla a continuación[SS17].

Este framework define al menos 3 artefactos:

- *Product backlog*: es una lista de requerimientos ordenada en base al valor que agregan al negocio, donde cada requerimiento cuenta con una descripción, una prioridad y una estimación. Dicha lista, sumada a los requerimientos, puede incluir mejoras o arreglos, por lo que la misma es dinámica y se espera que cambie a lo largo del proyecto.
- *Sprint backlog*: refiere a un subconjunto de los requerimientos del product backlog. Como se ve en la Figura 2.4[1], son los requerimientos que el equipo de trabajo se compromete a completar en cada uno de los mencionados sprints.
- *Burndown chart*: es un gráfico que permite visualizar de un manera rápida el trabajo restante en un sprint, o en el proyecto en su totalidad.

En cuanto a las personas y roles se definen los siguientes conceptos:

- *Product Owner* (PO): es quien se encarga de priorizar los requerimientos del backlog a partir del valor que cada requerimiento agrega al negocio. Dicho rol solo puede ser ocupado por una única persona en el proyecto.
- *Scrum Master*: es quien actúa de facilitador y debe asegurarse de que todos los involucrados en el proyecto comprendan y apliquen Scrum correctamente. A su vez, es quien ayuda a aquellos que están fuera del equipo de Scrum a comprender qué interacciones con el equipo son útiles y cuáles no. También se encarga de mejorar dichas interacciones con el objetivo de maximizar el valor creado por el equipo.
- Equipo de trabajo: compuesto por entre 5 y 9 personas, es el encargado de que los requerimientos estén listos.

[1] Figura tomada de https://en.wikipedia.org/wiki/File:Scrum_process.svg

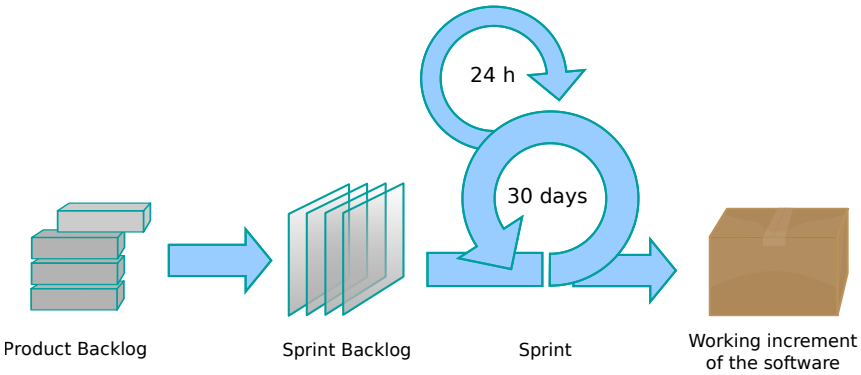

Figura 2.4: Proceso de Scrum

Por otra parte, Scrum se compone de 4 ceremonias:

- *Sprint planning*: es una fase de planificación de Scrum en la cual todo el equipo discute y analiza en qué requerimientos del product backlog se trabajará durante el próximo sprint. Para ello el equipo se basa en el estado actual en que se encuentra el product backlog, así como en lo realizado en el último sprint. A su vez, el equipo de desarrollo se comprometerá a alcanzar el estado de "Listo" para todos aquellos requerimientos y tareas propuestos para dicho sprint. Esta ceremonia puede llevar un tiempo aproximado de 4 horas para sprints de 2 semanas de duración.

- *Daily Scrum* o *daily meeting*: consiste en una reunión diaria en la que participa todo el equipo de Scrum, en la cual cada miembro del equipo de desarrollo describe a los demás compañeros las tareas que ha podido completar desde la última reunión, qué tareas planea realizar ese día, y los problemas que ha encontrado y superado al trabajar en dichas tareas. En cuanto al Scrum Master, el mismo debe oficiar de moderador asegurándose de estipular los turnos en los que cada desarrollador describe los puntos al resto, así como de evitar distracciones y discusiones técnicas por parte de los desarrolladores. Esta ceremonia es informal, y normalmente tiene una duración máxima de 15 minutos. La misma suele realizarse a la misma hora del día, y en el mismo lugar.

- *Sprint reviews*: es una ceremonia que se realiza al finalizar cada sprint. Durante la misma el equipo de Scrum analiza los requerimientos que fueron completados durante el sprint, así como los requerimientos que no pudieron completarse. A su vez, los requerimientos completados son mostrados a los interesados para obtener feedback del trabajo realizado y decidir en qué se deberá trabajar en el próximo sprint. Esta ceremonia normalmente tiene una duración aproximada de 2 horas para sprints de 2 semanas.

- *Sprint retrospective*: consiste en una reunión facilitada por el Scrum Master en la cual cada integrante del equipo de Scrum menciona a los demás aspectos positivos del sprint completado, así como aspectos a mejorar en futuros sprints. La duración aproximada de esta ceremonia es de 1 hora y media para sprints de 2 semanas de largo.

2.2.2. Lean Software Development

El concepto Lean refiere a una práctica aplicada a la producción cuyo principal objetivo es utilizar cada recurso disponible para generar valor al consumidor final. Esta práctica se inspira en el sistema integral de producción Toyota Production System[2] de la empresa automotriz japonesa Toyota, donde los principales objetivos son evitar la sobrecarga (*muri* en japonés), evitar la inconsistencia (*mura* en japonés) y eliminar los desechos (*muda* en japonés). Esto permite mejorar la calidad de los productos reduciendo tiempos y costos.

Lean Software Development es un framework introducido por Mary y Tommy Poppendieck en el año 2003 que aplica las prácticas de Lean para el desarrollo de software, brindando un conjunto de herramientas que permiten identificar y eliminar recursos innecesarios a los que denomina desechos[PP03].

Este framework puede ser resumido en 7 principios, que se transcriben a continuación:

- Eliminar desechos
- Amplificar aprendizaje
- Tomar decisiones en el momento más tardío posible
- Entregar de la manera más rápida posible
- Capacitar al equipo
- Construir integridad intrínseca
- Ver todo el conjunto

De acuerdo a Mary Poppendieck, la aplicación de estos principios permiten entregar mejor software, de una manera más rápida, y a un menor costo de manera simultánea.

2.2.3. Kanban

Kanban, cuyo significado en japonés es tablero de señales, es un sistema de control de producción. Fue creado por Taiichi Ohno, y al igual que Lean, surge en Toyota Production System.

Este sistema también es conocido como "método de supermercados" por su origen. Esto es debido a que en 1940 Toyota estudió los sistemas de reposición aplicados en supermercados con el objetivo de mejorar sus propios sistemas de producción, e identificó que los supermercados utilizaban tarjetas de control en cada uno de sus productos. Dichas tarjetas incluían información propia del producto, así como códigos y ubicación de su almacenamiento. Esto permitía que los supermercados tuvieran en stock los productos que los clientes querían, en la cantidad necesaria, y disponibles para ser vendidos en cualquier momento. Toyota comenzó a aplicar estos conceptos sobre sus procesos de producción a partir de 1953, lo que les permitió determinar qué producir, cuándo y cuánto, así como detectar áreas afectadas por problemas e implementar mejoras.

A comienzos de la década del 2000, David Anderson, quien trabajaba para Microsoft aplicó los conceptos de Kanban al desarrollo de software, siendo su

[2] https://global.toyota/en/company/vision-and-philosophy/production-system/

principal objetivo simplificar y aislar la gestión para que el equipo de desarrollo pudieran dar valor al cliente a través del código.

Kanban aplica una estrategia de *pull* en la que en vez de tratar de pronosticar la demanda (*push*), se responde a los cambios de demanda de una manera rápida, mediante la propagación de dichos cambios a través de tarjetas conocidas como *Kanban cards*. Esto aplicado a software se traduce en que cada miembro del equipo de desarrollo comienza a trabajar en un nuevo ítem de trabajo una vez que termine su ítem anterior ("pull") y no se le asignan varios ítems de trabajo simultáneamente cada vez que surgen nuevos ítems ("push"). Para ello se maneja un *Kanban board*, como el que se muestra en la Figura 2.5 [3], que permite conocer el flujo de trabajo y la participación del equipo, y facilitar su gestión. El tablero muestra como cada ítem de trabajo va pasando de una columna a otra de izquierda a derecha, donde cada columna representa el estado en el que se encuentra cada ítem de trabajo.

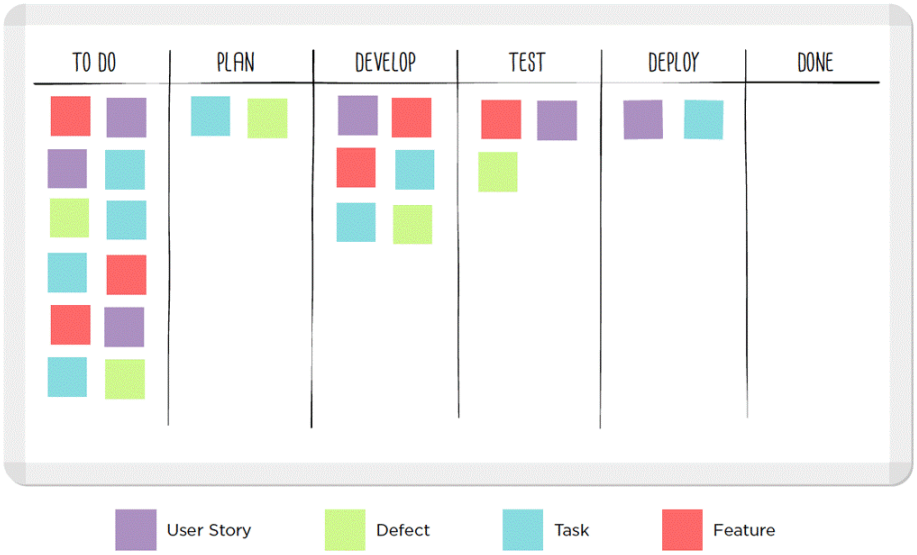

Figura 2.5: Tablero Kanban

2.2.4. Extreme Programming (XP)

Extreme Programming, también conocida como XP, es una metodología de desarrollo ágil que surge en la década de 1990. La misma fue creada por Kent Beck mientras trabajaba en un proyecto de la empresa Chrysler Corporation, y posteriormente descripta también por Beck en 1999 en su libro "Extreme Programming Explained: Embrace Change"[Bec99].

Extreme Programming se basa en los siguientes 5 aspectos clave[Wel03]:

- Comunicación: programadores que siguen esta metodología continuamente se comunican entre sí y con los clientes.
- Simplicidad: programadores mantienen el diseño simple y limpio.

[3]Figura tomada de http://www.astrowerks.com/wonderful-kanban-board-columns/

- Retroalimentación (feedback): programadores obtienen retroalimentación al probar el software desde el primer día. A su vez, se entrega el software al cliente tan pronto como sea posible, y se implementan los cambios sugeridos.
- Respeto: cada pequeño éxito profundiza el respeto por las contribuciones únicas que cada programador aporta al equipo.
- Coraje: programadores son capaces de responder valientemente a cambios en requerimientos y tecnologías.

La idea detrás de Extreme Programming es aplicar estos valores, considerados como las mejores prácticas, al extremo. Beck entiende que de esta forma los desarrolladores son más productivos, desarrollan software de muy buena calidad y responden rápidamente al cambio, al mismo tiempo que disminuyen la probabilidad de tomar decisiones que resulten negativas para el proyecto.

Para alcanzar estos objetivos XP se basa en diversas prácticas como Planning Game, Pair Programming y Test Drive Development que se detallan a continuación.

- *Planning Game*: Es una práctica en la cual se determina en qué requerimientos se va a trabajar. Para ello se definen dos roles: personas de negocio y personas técnicas. Las personas de negocio serán aquellas que definan la prioridad sobre los requerimientos, respondiendo a preguntas como: ¿si Ud. pudiera únicamente contar con un requerimiento sobre el cual los desarrolladores trabajarán, cuál sería?. En cuanto a las personas técnicas, las mismas deben dar un soporte técnico a las decisiones de negocio, tomando decisiones acerca, por ejemplo, de cuánto tiempo puede tomar desarrollar un determinado requerimiento en el software, o qué consecuencias técnicas puede traer elegir una tecnología en lugar de otra.
- *PairProgramming*: Es una práctica que define que todo el código desarrollado siempre será escrito por 2 personas al mismo tiempo. Dichas personas deben trabajar sobre una misma computadora, con un único teclado y únicamente un mouse. Esto brinda diversas ventajas, entre ellas, que el código no es conocido por una única persona, desarrolladores pueden complementarse y sugerir mejoras sobre el código del otro, y que ambas personas buscan como resolver un mismo problema. A su vez, Pair Programming lleva a que se escriba código más fácil de ser probado, más fácil de mantener y siguiendo mejores prácticas de código, entre otros beneficios.
- *Test Driven Development* (TDD): es una de las prácticas más importantes de XP. Cambia el paradigma tradicional a la hora de escribir código. En lugar de escribir código y luego escribir casos de pruebas, TDD define que primero deben escribirse los casos de prueba, y luego de ello se escribe el código propiamente dicho.

Como se ve en la Figura 2.6, TDD se basa en 4 etapas: *red*, *green*, *refactor*, *repeat*, por sus nombres en inglés, los cuales se traducen a rojo, verde, refactorizar y repetir, respectivamente.

La primera etapa, red, consiste en escribir casos de prueba para una funcionalidad que se va a desarrollar. Una vez escritos, dichos casos de prueba serán

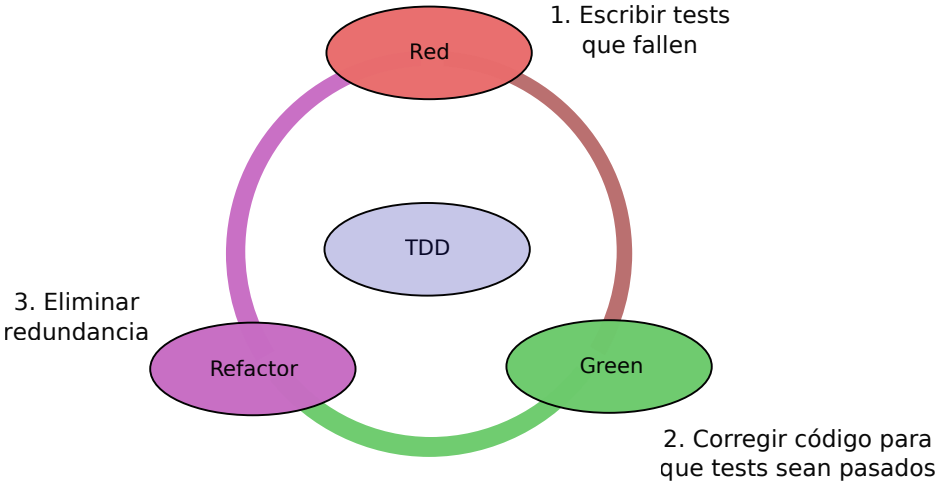

Figura 2.6: Test Driven Development (TDD)

ejecutados, y deberán fallar debido a que aún no se encuentra escrito el código que se desea probar. Por esta razón es que esta etapa recibe el nombre red, ya que las pruebas que fallan son comúnmente identificadas con el color rojo. En la etapa siguiente, green, se escribe el código correspondiente a las pruebas escritas durante la etapa anterior. Al ejecutar los casos de prueba escritos anteriormente, los mismos deberán ser pasados, lo cual se indica con el color verde. De no ser así, el desarrollador sabrá que hay un problema en el código, y que deberá hacer modificaciones hasta que la ejecución de pruebas sea verde. Una vez que se llega a esa situación, se comienza la etapa de refactor, o refactorización, que consiste en hacer cambios sobre el código para mejorar su legibilidad, mantenibilidad y reducción de complejidad. Por último, la etapa de repeat, refiere a repetir los pasos anteriores para el desarrollo del resto de las funcionalidades a implementarse en el sistema. Esto permite que todo el código escrito se encuentre probado, y con casos de prueba que puedan ser ejecutados cada vez que se realiza un cambio.

2.2.5. Resumen

Un equipo de desarrollo de software que valore el desarrollo iterativo e incremental, y una rápida entrega de software al cliente para realizar mejoras en base a su feedback, puede considerar adecuado seguir alguna de las metodologías mencionadas. Si bien las mismas se centran en los valores ágiles, cada una de ellas tiene sus propias herramientas y ventajas. Sin embargo, dichas metodologías son flexibles y cada equipo puede seguir las prácticas que encuentre más apropiadas y que les brinden los mejores resultados. A su vez, los equipos de desarrollo pueden combinar prácticas de distintas metodologías ágiles según sus necesidades. Por ejemplo, un equipo puede trabajar en sprints como define Scrum, utilizar un tablero Kanban como define la metodología que lleva su nombre, y evitar sobrecargas y eliminar desechos como plantea Lean.

Capítulo 3

Integrando seguridad en los procesos de desarrollo ágil

En el presente capítulo se describen actividades y prácticas que permiten integrar seguridad en un proceso de desarrollo ágil. El mismo se encuentra dividido en 3 secciones.

La sección 3.1 plantea la importancia de capacitar a los desarrolladores para que se involucren en la seguridad del software y se sientan responsables de la misma. A su vez, presenta técnicas como las *abuser* y *security stories*. Las mismas permiten identificar potenciales riesgos y vulnerabilidades a la hora de definir los requerimientos del software. También se detallan estrategias para la gestión de riesgos y vulnerabilidades, desde la detección de problemas a los cuales el sistema está expuesto, a su tratamiento. Se presentan además prácticas como los *security sprints* y *hack days* que permiten reducir dichos riesgos, conocidos también como deuda técnica de seguridad. Finalmente, se describen estrategias para detectar problemas en el código fuente de la aplicación, y distintos tipos de pruebas de seguridad que pueden ser ejecutadas sobre el software, siendo éstas últimas conocidas como *Agile Security Testing*.

La sección 3.2 plantea prácticas y herramientas que permiten automatizar la seguridad para construir un sistema de manera rápida e incremental, sin retrasar los tiempos de entrega al cliente. Para ello, se detallan cambios culturales conocidos como DevOps y DevSecOps que buscan integrar los equipos de trabajo en las organizaciones. También se presentan prácticas como $Infrastructure - as - Code$, e integración y despliegue continuos para la gestión de infraestructura y puesta en producción del sistema de una manera confiable y repetible. Por último, se describen nuevos desafíos de seguridad que deben ser considerados por las organizaciones que implementan estos conceptos.

Finalmente, la sección 3.3 presenta herramientas de seguridad que pueden ser integradas en el ciclo de vida de desarrollo del software. Esta sección incluye herramientas de análisis de código, gestión de dependencias, hardening, análisis de vulnerabilidades y monitoreo, entre otras.

3.1. Seguridad en metodologías de desarrollo ágiles

Múltiples autores han planteado distintas estrategias y prácticas para integrar la seguridad en procesos de desarrollo tradicionales. Un ejemplo de ello son los *Software Security Touchpoints* (puntos clave de seguridad del software, en español) propuestos por McGraw[McG06]. Los mismos refieren a 7 actividades que permiten integrar seguridad durante todo el ciclo de vida del desarrollo de software. Como se ve en la Figura 3.1, éstas actividades son realizadas en etapas particulares del desarrollo presentes en las metodologías tradicionales. Actividades como la definición de casos de abuso y requerimientos de seguridad, así como análisis de riesgos, son realizadas en las primeras etapas del ciclo de vida. Por otro lado, actividades como los tests de seguridad basados en riesgos son ejecutadas en las etapas de testing, y las revisiones de código durante la codificación. Finalmente, actividades como análisis de riesgos, tests de penetración y operaciones de seguridad normalmente son ejecutadas en las etapas finales. Sin embargo, los procesos de desarrollo ágiles, caracterizados por desarrollar software de manera iterativa e incremental, no cuentan con dichas etapas y realizan una gestión de tiempos totalmente distinta. Esto provoca que estas actividades y prácticas no puedan ser aplicadas de forma similar en procesos ágiles. Es por ello que para integrar seguridad en procesos ágiles es fundamental implementar actividades de seguridad alineadas a los valores ágiles, y acordes a la forma de trabajo de estos equipos, para aprovechar el buen funcionamiento y las ventajas de las prácticas ágiles ya aplicadas por los desarrolladores de la organización.

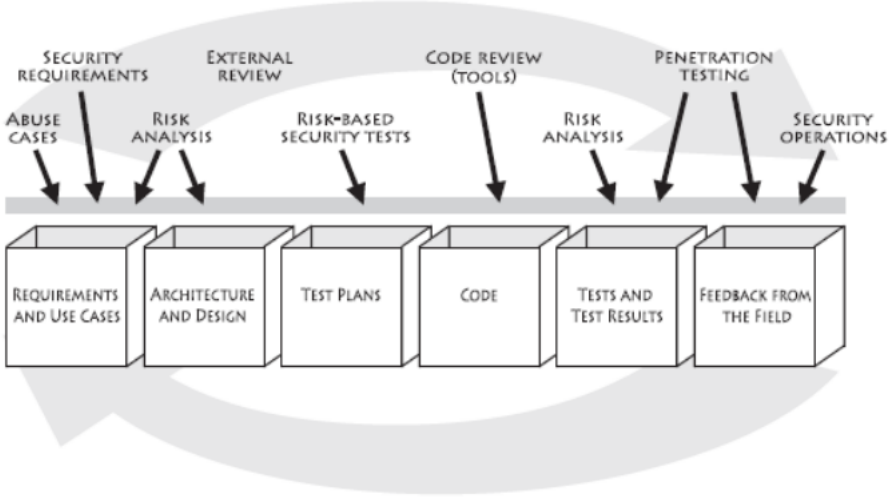

Figura 3.1: Security Touchpoints

A su vez, dichas actividades deben ser capaces de integrar seguridad de manera incremental, y no afectar la rápida entrega de valor a los clientes. Debido a estas razones es imprescindible que puedan ser automatizadas, de manera de poder ejecutar controles automáticamente ante cualquier cambio en el código, y obtener un feedback rápido acerca de qué problemas de seguridad deben ser

resueltos.

3.1.1. Responsabilidades del equipo

Uno de los mayores desafíos en cuanto a la integración de seguridad en el desarrollo de software es lograr el compromiso por parte de todo el equipo de desarrollo sobre la seguridad del software a realizarse. Para ello es imprescindible que el equipo de desarrollo comprenda la importancia y el valor de la seguridad en el software, y que cada miembro de dicho equipo sea responsable de la seguridad de las funcionalidades que desarrolla.

A su vez, es recomendable que las organizaciones cuenten con al menos un ingeniero o especialista de seguridad asociado al proyecto de desarrollo, quien podrá participar en múltiples proyectos en simultáneo, y no necesariamente debe trabajar en exclusividad en un único proyecto a la vez. Dicho ingeniero será responsable de concientizar y capacitar respecto a posibles problemas y consideraciones de seguridad del software a todos los involucrados en el mismo. Esto incluye tanto a desarrolladores, como al *Product Owner*, Scrum Master, encargados de QA (*Quality Assurance*), e incluso a los *stakeholders*, si así lo desearan estos últimos.

Ingenieros de seguridad y desarrolladores deberán trabajar en conjunto y aprender unos de otros. Por esta razón es recomendable que ambos equipos participen juntos de actividades como reuniones de planificación, stand-up meetings y retrospectivas, entre otras[BBSSB17].

Asimismo, el ingeniero de seguridad debe lograr que los desarrolladores consideren que una user story alcanza el estado de "Lista", de acuerdo al concepto de *Definition of Done* (DoD)[1] manejado por el equipo de desarrollo. Dicho estado podrá ser alcanzado únicamente cuando la misma cumpla, tanto con sus criterios de aceptación funcionales, como con sus requerimientos de seguridad asociados.

Por último, el ingeniero de seguridad también deberá actuar como facilitador de seguridad para el equipo de desarrollo, brindando guías y herramientas que permitan que los desarrolladores contemplen la seguridad de la manera más fluida y transparente posible, y no la consideren un impedimento.

3.1.2. Requerimientos

La seguridad de los sistemas comúnmente es contemplada por primera vez al momento de escribir código o de desplegar los sistemas en producción. Sin embargo, la misma debe ser considerada incluso desde que se evalúan y diseñan los requerimientos. Para ello han surgido múltiples técnicas con distintos enfoques que permiten expresar requerimientos relacionados a la seguridad en procesos de desarrollo ágiles. Algunas de ellas permiten escribir requerimientos que definen mecanismos para prevenir determinados tipos de ataques. Otras expresan requerimientos basados en problemas que podría querer explotar un atacante en el sistema. A su vez, otras plantean la incorporación de roles especiales, como el del *Security Master*, quien es definido como el encargado de gestionar y evaluar que los aspectos de seguridad sean contemplados en todos los requerimientos del sistema.

[1] https://www.agilealliance.org/glossary/definition-of-done/

Abuser Stories

Johan Peeters ([Pee05]) propone una técnica llamada *abuser stories*. La misma consiste en escribir requerimientos de seguridad describiendo cómo un atacante podría abusar del sistema que se desarrolla. Como se ve en la Figura 3.2, se expresan de una manera corta e informal, similar a las historias de usuario, en un lenguaje con el cual se encuentran familiarizados, tanto los desarrolladores, como el cliente.

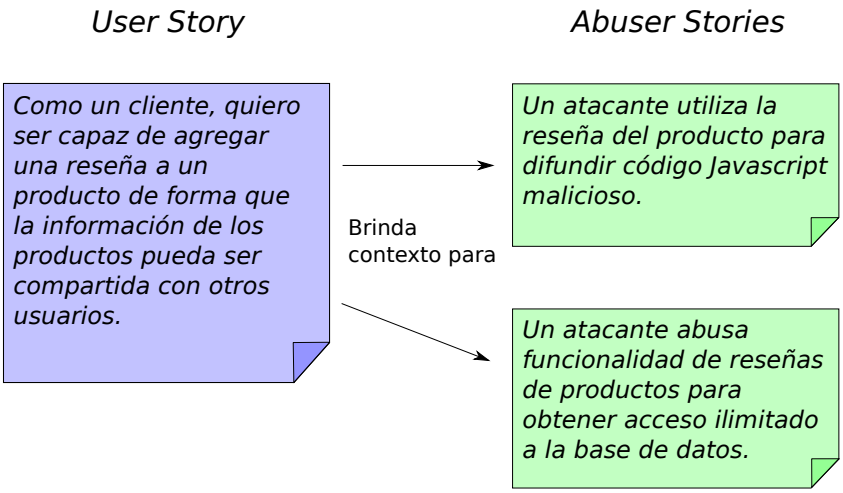

Figura 3.2: Ejemplos de abuser stories[Bos17]

Gustav Boström et. al. ([BWB+06]) describe un conjunto de pasos para definirlas. En primer lugar, propone identificar los activos críticos del sistema a ser desarrollado, y lograr un buen entendimiento de los mismos. Al mismo tiempo, plantea que un ingeniero de seguridad, posiblemente en conjunto con desarrolladores y miembros del cliente, definan escenarios de amenazas que puedan presentar riesgos altos a los activos críticos identificados. Boström destaca el rol de dicho ingeniero como esencial. Esto es debido a que cuenta conocimientos acerca de vulnerabilidades de seguridad, amenazas y riesgos, y es muy importante a la hora de definir los escenarios de amenaza.

Las abuser stories, al igual que las user stories, también pueden ser priorizadas y puntuadas en base a la amenaza que las mismas presentan frente a los activos de la organización. A su vez, los puntos de historia asignados a dichas stories podrán cambiar durante el ciclo de vida de desarrollo ante escenarios como un posible aumento en el impacto o en la probabilidad del riesgo que describen. Al momento de planificar en qué stories se trabajará en cada iteración, el ingeniero en seguridad deberá elegir en conjunto con el cliente qué abuser stories serán incluidas en dichas iteraciones.

Security Stories

A su vez, Gustav Boström ([BWB+06]) también propone el uso de user stories para expresar requerimientos funcionales de seguridad y las denomina security stories. Dichas stories permiten describir contramedidas frente a las amenazas identificadas previamente al momento de definir abuser stories.

Como se ve en la Figura 3.3, las security stories se expresan de similar manera que las user stories convencionales. Se escriben también en un lenguaje claro y comprendido con facilidad por desarrolladores y clientes. Al igual que las user stories, también permiten ser validadas mediante tests unitarios, tests de sistema y tests de integración.

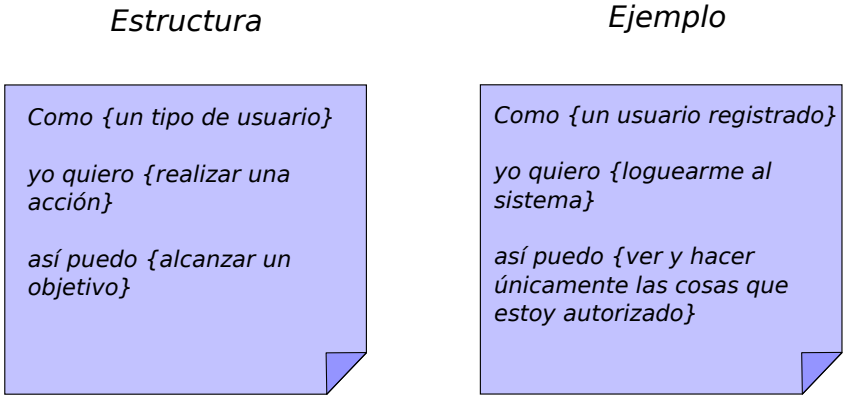

Figura 3.3: Security stories

La definición de éstas stories requiere un alto conocimiento de seguridad, principalmente en cuanto a arquitecturas seguras para mitigar los riesgos identificados previamente. Por esta razón, es muy importante contar con un ingeniero de seguridad que cuente con sólidos conocimientos al respecto a la hora de escribirlas. A su vez, es valioso que participen de esta actividad, tanto desarrolladores por su conocimiento técnico del producto, como el cliente por sus conocimientos de negocio. Sin embargo, es posible que el cliente no tenga interés de participar en dicha actividad debido a que la misma tiene un alto componente técnico.

Existen diversos recursos disponibles en Internet que pueden ser útiles a la hora de definir requerimientos de seguridad. El *Application Security Verification Standard* (ASVS)[OWA16] es un recurso *open source* de la organización *Open Web Application Security Project* (OWASP)[2]. El mismo define un conjunto de controles de seguridad a nivel técnico que deben ser aplicados para proteger las aplicaciones frente a diversas amenazas. El uso de este recurso puede resultar conveniente, principalmente para definir sus criterios de aceptación. El mismo incluye requerimientos relacionados a distintos aspectos de la seguridad de los sistemas, como lo son la verificación de autenticación, gestión de sesiones, control de acceso, manejo de entradas maliciosas, entre otros. Otra fuente recomendada para definir security stories es el documento llamado *Practical Security Stories and Security Tasks for Agile Development Environments*[ATO+12], definido por la *Software Assurance Forum for Excellence in Code* (SAFECode)[3]. Dicho documento plantea estrategias para incluir seguridad en la planificación e implementación de requerimientos en procesos de desarrollo ágiles.

[2]https://www.owasp.org/index.php/Main_Page
[3]https://safecode.org/

Security Backlog

Azham et al. [AGI11] plantea que al dar comienzo a un proyecto ágil a ser desarrollado utilizando la metodología Scrum, tanto el cliente como los desarrolladores, no cuentan con suficientes conocimientos acerca de los potenciales riesgos de seguridad del producto a desarrollarse. Entiende además que la creación de múltiples requerimientos de seguridad y *epics* (conjunto de requerimientos relacionados entre sí) en el backlog tradicional puede afectar la agilidad de Scrum. Por esta razón, los autores proponen la incorporación de un backlog adicional, al que denominan *Security Backlog*, que será utilizado para gestionar únicamente riesgos de seguridad. Dicho backlog no será gestionado por el Product Owner, sino que será gestionado por un *Security Master*.

El Security Master será quien esté a cargo de la seguridad en el proyecto de desarrollo, y deberá contar con extensos conocimientos de seguridad. Sus tareas pueden incluir actividades como la identificación de riesgos de seguridad y su documentación en el Security Backlog. A su vez, como se ve en la Figura 3.4, podrá seleccionar requerimientos de éste backlog y asignarlos a los desarrolladores para que se trabaje sobre los mismos durante un sprint específico. Esto es realizado con el objetivo de reducir el riesgo asociado al proyecto. Además podrá ejecutar tests de seguridad sobre dichos requerimientos así como sobre cualquier conjunto de funcionalidades que entienda conveniente.

Por otra parte, otra tarea del Security Master puede estar referida a realizar capacitaciones de seguridad, tanto a los desarrolladores, como a los stakeholders (en caso de que los mismos estén interesados). De esta forma es posible que todos los involucrados en el proyecto cuenten al menos con conocimientos básicos de seguridad.

3.1.3. Gestión de Vulnerabilidades y Riesgos

Una vez identificados los requerimientos de seguridad es necesario hacer una correcta gestión de los mismos. Por esta razón, es imprescindible contar con una adecuada gestión de vulnerabilidades y riesgos. La misma debe proveer una visión centralizada e integral de los riesgos de seguridad a los que se expone la organización, y dicha información debe estar disponible para todo el personal involucrado en el desarrollo, tanto a nivel técnico como gerencial.

Vulnerabilidades en el software son encontradas a diario, pudiendo exponer a las organizaciones a diversos riesgos. Debido a esto ya no es suficiente realizar análisis de vulnerabilidades periódicos. Las organizaciones deben incentivar a los equipos de desarrollo a identificar vulnerabilidades en el software de manera continua. A su vez, deben proveer metodologías y técnicas a los desarrolladores para que tengan un entendimiento básico de riesgos del proyecto, así como de riesgos técnicos.

Los equipos de desarrollo, en conjunto con especialistas en seguridad, deberán capacitarse acerca de los riesgos de seguridad más comunes y conocer técnicas para un correcto manejo de los mismos. A su vez, deberán ejecutar escaneos de vulnerabilidades continuos, tanto de software como de infraestructura, con el objetivo identificar vulnerabilidades que afecten los sistemas. Los resultados de los escaneos realizados podrán integrarse al backlog de desarrollo. De esta forma, se puede realizar una trazabilidad de las vulnerabilidades identificadas. Éstos últimos podrán ser analizados y discutidos durante las reuniones retrospectivas

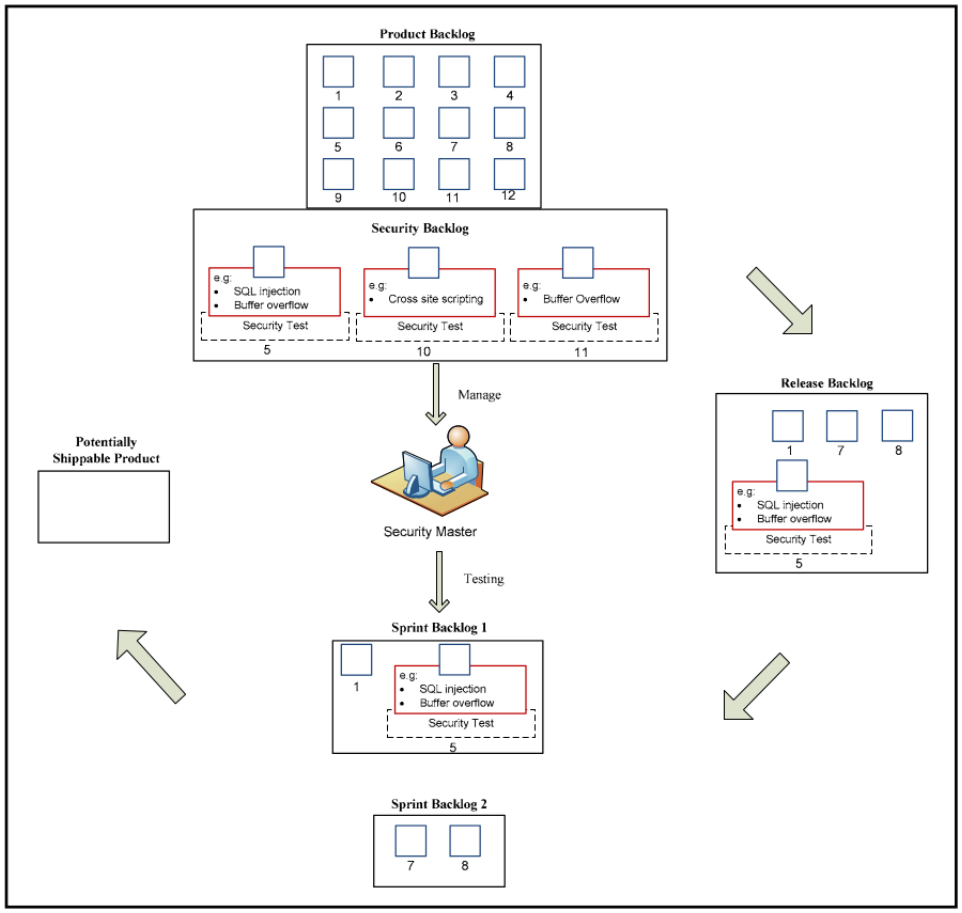

Figura 3.4: Security Master en proceso de Scrum[AGI11]

de cada iteración de desarrollo para identificar posibles aspectos a mejorar.

En cuanto al tratamiento de las vulnerabilidades identificadas, el primer paso a ejecutar es lograr un entendimiento del riesgo al cual exponen a la organización. Posteriormente se deben evaluar qué acciones se desean ejecutar al respecto. De tratarse de riesgos críticos las vulnerabilidades podrán ser corregidas de manera inmediata. De no ser así, los riesgos pueden ser agregados al backlog del proyecto, y se trabajará en ellos en futuras iteraciones de desarrollo. También es posible que los riesgos sean descartados por no ser suficientemente importantes.

Por otra parte, también resulta conveniente realizar trazabilidad de vulnerabilidades en componentes de terceros utilizados. Esto incluye, tanto dependencias utilizadas en el código fuente como el software de base. Para ello es posible realizar un inventario del software y de los productos utilizados, y realizar suscripciones a boletines de seguridad de los mismos. Esto permite estar informado de nuevas vulnerabilidades que sean detectadas, y actuar en consecuencia. Este proceso debe ir acompañado de una correcta gestión de parches. La misma debe permitir evaluar el correcto funcionamiento de actualizaciones de seguridad, así como el despliegue de las mismas en ambientes productivos de manera automatizada.

A su vez, es posible definir procedimientos y equipos de respuesta frente a posibles incidentes de seguridad. Para ello es conveniente plantear escenarios de potenciales amenazas y planificar cuál será la respuesta a ejecutar. De esta forma, si algún problema sucede los equipos de respuesta sabrán qué acciones deben realizar y cómo deben comunicarse. Dichas acciones pueden incluir procedimientos como cambiar configuraciones en firewalls, tomar imágenes forenses de equipos comprometidos y restaurar sistemas que fueron dañados, entre otras. Además es recomendable hacer simulacros periódicos de problemas de seguridad para entrenar al personal, así como para medir tiempos de respuesta e identificar puntos a mejorar. Dichos simulacros son conocidos como *game days* o *war games* (días de juego o juegos de guerra, en español respectivamente).

3.1.4. Security Sprints

La consideración de seguridad como parte del concepto de "Listo" de una user story no siempre se respeta de manera absoluta, y pueden haber excepciones a la misma. Asthana et. al. utiliza el término *security debt*[ATO+12] (deuda de seguridad) para describir todas las tareas incompletas y pendientes que tienen relevancia para la seguridad del sistema. Dichas tareas surgen de posponer, saltear, depriorizar o ignorar tareas asociadas a seguridad, y potencialmente exponen a la aplicación a distintas vulnerabilidades.

Los security sprints buscan resolver esta problemática, y consisten en sprints en los que el equipo de desarrollo se dedica únicamente a trabajar en user stories relacionadas a seguridad. De esta forma se reduce la deuda de seguridad del sistema, y se remedian vulnerabilidades del mismo. Estas stories pueden describir requerimientos referidos a problemas como validación de entradas, manejo de logs, autenticación y autorización. A su vez, pueden describir requerimientos relacionados a riesgos técnicos como $Cross - Site\ Scripting$ (XSS), *SQL injections*, entre otros[JM].

Es posible que el equipo de desarrollo tenga dificultades para justificar este tipo de sprints, tanto al cliente interesado en el proyecto como a la gerencia de la organización. Esto es debido a que los security sprints son una actividad costosa, y durante los mismos se pausa totalmente el desarrollo de nuevas funcionalidades o mejoras en la arquitectura del sistema. Una alternativa más sencilla y económica a los security sprints son los *hack days*, o días de hacking en español. Los mismos refieren a jornadas puntuales en las cuales los equipos de desarrollo dejan de lado sus tareas habituales, y se dedican enteramente a aprender nuevas cosas, trabajar en nuevas ideas y resolver problemas juntos. Pueden consistir en capacitar al equipo de desarrollo acerca de cómo identificar y mitigar un tipo de vulnerabilidad en el software. De esta forma todo el equipo, trabajando en grupos, podrá dedicar un día entero a resolver este tipo de problema.

3.1.5. Security Code Review

Una revisión de seguridad de código, o *security code review* en inglés, es una actividad que consiste en analizar el código fuente del software con el objetivo de encontrar errores y problemas de seguridad. Dicha actividad puede ser realizada, tanto en procesos de desarrollo tradicionales como ágiles. Permite identificar problemas como errores comunes de programación, utilización incorrecta de

herramientas de seguridad provistas por frameworks, uso inseguro de primitivas de criptografía, desarrollo incompleto de un requerimiento, entre otros.

En las metodologías ágiles suele ser conveniente realizar revisión de código cada vez que se realiza un cambio sobre el mismo. Éstas revisiones podrán ser llevadas a cabo por el propio desarrollador que realizó el cambio, así como por otro compañero. Las mismas proveen varios beneficios al código. Dichos beneficios incluyen el cumplimiento de normativas y legislaciones, realización de controles ante cambios no autorizados por parte de otro desarrollador, aprendizaje de estrategias aplicadas por compañeros para resolver determinados problemas, entre otras. A su vez, mejora la calidad de código al verificar que se cumpla con guías y buenas prácticas de codificación, ya sean propias de cada lenguaje como de la organización.

Existen diversos tipos de revisiones que pueden realizarse sobre el código con el objetivo de detectar problemas seguridad, como lo son *peer review*, auditorías de código y revisiones de código automatizadas, entre otros.

Peer review

Las revisiones de código pueden realizarse de varias formas, siendo habituales en las metodologías ágiles las revisiones entre pares, o *peer review* en inglés. Dichas revisiones consisten en revisiones informales entre los propios compañeros de desarrollo. Son realizadas previamente a que los cambios sobre el código fuente sean impactados sobre el repositorio central de código.

Estas revisiones pueden realizarse tanto de manera asíncrona como síncrona. En la primera de ellas el desarrollador del código solicita mediante un software la revisión por parte de un compañero, y una vez realizada recibe una notificación con el resultado de la revisión. En cuanto a las revisiones síncronas, las mismas consisten en hacer la revisión durante el propio desarrollo de software. En metodologías como Extreme Programming (XP) donde se aplica Pair Programming (PP), dicha revisión se da de manera natural, ya que 2 o más desarrolladores trabajan al mismo tiempo sobre el mismo código.

Auditorías de código

Las auditorías de código son procesos formales de revisión de código. Las mismas son realizadas por especialistas de seguridad en conjunto con un grupo de desarrolladores, siendo todos los participantes de la auditoría externos al equipo de desarrollo del software a analizar.

En estas auditorías el código fuente es revisado con el objetivo de identificar principalmente problemas de seguridad. Para ello se definen las funcionalidades a ser revisadas, y se envía el código a cada uno de los participantes para que lo analicen de manera individual. Posteriormente, todos los participantes se reúnen uno o más días, de ser necesario, a revisar en conjunto cada línea del código a ser auditado.

Estos procesos pueden ser muy costosos, ya que involucran a muchas personas revisando el mismo código en simultáneo. A su vez, estas revisiones suelen ser muy extensas, y la realización de las mismas durante varias horas al día lleva a que se reduzca la concentración por parte de los participantes. Esto puede provocar que haya problemas en el código que no sean detectados.

Revisión de código automatizada

A su vez, es posible realizar revisiones de código automatizadas integrando distintas herramientas. Las mismas permiten revisar el código en su totalidad, lo que podría resultar muy costoso o imposible de revisar manualmente. Además, si bien las mismas son parametrizables, sus resultados son repetibles y uniformes, es decir, los mismos son objetivos y no varían de acuerdo a qué desarrollador las ejecuta.

Existen herramientas de análisis estático de código, también conocidas como SAST por su sigla en inglés (*Static Application Security Testing*), que permiten realizar revisiones sobre código fuente y código compilado, sin necesidad de ejecutar el código. Dichas herramientas suelen integrarse a los ambientes de desarrollo de cada desarrollador, de forma de escanear el código automáticamente mientras el mismo es escrito. Sin embargo, tienen ciertas limitaciones, como el hecho de que brindan un alto número de falsos positivos. Esto refiere a que las herramientas reportan un problema de seguridad cuando en realidad el sistema no es vulnerable. A su vez, presentan dificultades para automatizar detección de determinados problemas particulares[OWA17].

Por otro lado existen herramientas que realizan revisiones dinámicas analizando la ejecución de archivos binarios. Dichas herramientas son también conocidas como DAST por su sigla en inglés (*Dynamic Application Security Testing*). Las mismas deben generar distintas entradas para evaluar el comportamiento de la ejecución del sistema y determinar la existencia de vulnerabilidades.

Consideraciones adicionales

Existen estrategias y prácticas que los desarrolladores pueden seguir con el objetivo de mejorar la calidad del código, y que a la vez facilitan la ejecución de revisiones sobre el mismo. Una de éstas prácticas es realizar cambios pequeños y concisos en el código fuente. De esta manera la revisión es más sencilla y corta. Esto permite que quien revise el código sea capaz de entender los cambios realizados de mejor manera, aumentando por tanto la probabilidad de identificar problemas. A su vez, es recomendable capacitar a desarrolladores respecto a buenas prácticas de desarrollo y codificación segura. Esto permitirá que los desarrolladores escriban mejor código, que será más fácil de entender por los demás. Además brindará herramientas a los desarrolladores para evitar introducir problemas en el código, así como para detectarlos.

Otra práctica recomendada refiere a revisar especialmente el código de operaciones sensibles, con el objetivo de identificar problemas de seguridad que podrían tener un alto impacto en caso de ser explotados. También es recomendable prestar especial atención a operaciones criptográficas. Las mismas suelen ser complejas y es común que los desarrolladores cometan errores en las mismas, pudiendo exponer información sensible a problemas de integridad o confidencialidad.

Además resulta conveniente realizar *hashes* sobre secciones de código que no se modifican comúnmente o que refieren a operaciones sensibles. Dichas secciones normalmente se encuentran revisadas e incluyen diversas pruebas para asegurar que su funcionamiento es el adecuado. La inclusión de los mencionados hashes sobre dichas secciones permite identificar potenciales problemas sobre las mismas que podrían ser introducidos por los desarrolladores. También permiten enviar una alerta a el/los encargados de la revisión de código para que revisen

particularmente la sección afectada. Por otra parte, el uso de esta técnica permite la detección de cambios sobre funciones sensibles que sean introducidos de manera deliberada por desarrolladores con el objetivo de exponer al software a un problema de seguridad.

Las revisiones de código deben aplicarse de manera obligatoria sobre el código escrito por cualquiera de los desarrolladores, independientemente de su experiencia. Los desarrolladores con menor experiencia posiblemente tienen un bajo entendimiento del funcionamiento del sistema. Escriben comúnmente código sobre funcionalidades sencillas, y es más probablemente que introduzcan errores en su código. Por esta razón dicho código es revisado por desarrolladores más experientes. Éstos últimos son los encargados de desarrollar las funcionalidades más complejas debido a sus conocimientos y mayor entendimiento del producto. Sin embargo, la complejidad de esas funcionalidades puede llevar a que el desarrollador cometa algún tipo de error o introduzca alguna vulnerabilidad de manera involuntaria. Debido a estas razones es recomendable que también el código de desarrolladores con experiencia sea revisado. Esto también permite evitar que desarrolladores que cuenten con cualquier tipo de experiencia introduzcan problemas de seguridad en el software de manera intencional.

3.1.6. Agile Security Testing

Martin Fowler ([Fow12]) define el concepto de *Test Pyramid*, o pirámide de tests en español, como un enfoque para clasificar distintos tipos de tests automatizados. Dicho concepto había sido utilizado por primera vez por Mike Cohn ([Coh09]) para referirse a un modelo eficiente y efectivo para automatizar tests en proyectos de metodologías ágiles. Como se ve en la Figura 3.5 la pirámide se compone de tests de 3 tipos: tests de interfaz gráfica, tests de servicios, y tests unitarios.

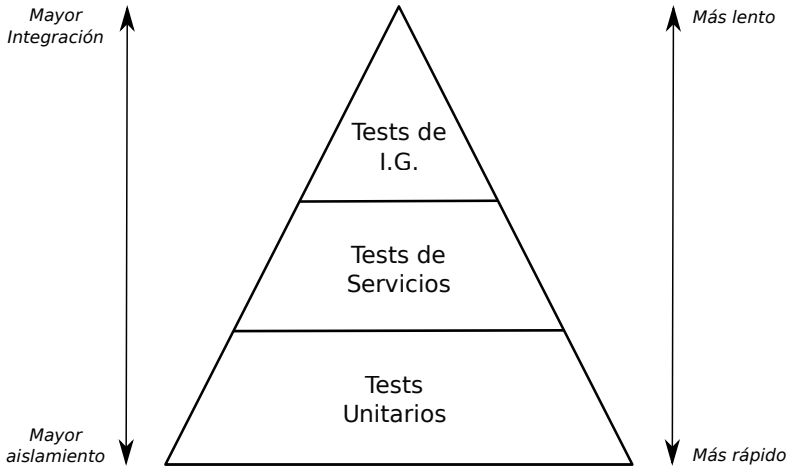

Figura 3.5: Pirámide de Agile Testing

- Los tests de interfaz gráfica ocupan la punta de la pirámide. El autor entiende que si bien los mismos son fáciles de realizar, e incluso pueden ser ejecutados por personas sin conocimientos de programación, son lentos

y difíciles de automatizar en scripts. Por otra parte, su funcionamiento implica que un usuario grabe una interacción con la interfaz del sistema para su posterior ejecución automatizada. Esto provoca que sean muy frágiles, debido a que un cambio en el software puede romper los tests con facilidad. Si esto ocurre, los tests tienen que ser grabados nuevamente, y por tanto, se incurre en incrementos en costos y tiempo. Por dichas razones, la pirámide muestra que es recomendable realizar otros tipos de tests menos frágiles, y que a la vez permiten un mayor grado de automatización.

- Los tests de servicios componen la capa central de la pirámide. Permiten abstraerse de la complejidad de las interfaces gráficas, permitiendo evaluar la lógica de negocio sin involucrar la interfaz de usuario. Estos tests prueban la interacción entre servicios, así como interacciones entre APIs.

- La capa inferior de la pirámide refiere a tests unitarios. Dichos tests son un método en el cual partes individuales de código fuente son evaluadas para verificar si su funcionamiento es el indicado. Tienen un costo asociado bajo y pueden ser totalmente automatizados. A su vez, proveen un feedback rápido ante cambios en el sistema, y permiten identificar rápidamente las secciones de código con problemas, por lo que simplifican la detección y corrección de errores.

Como se ve en la Figura 3.5, las flechas que acompañan a la pirámide indican que los tests unitarios son los más rápidos y baratos, mientras que los tests de interfaz son más lentos y costosos.

El concepto de Agile Security Testing busca aplicar los mismos conceptos de la pirámide de tests con pruebas específicas sobre la seguridad del sistema a desarrollar. A nivel de tests de interfaz gráfica pueden utilizarse herramientas como scanners de vulnerabilidades o herramientas de navegación automatizadas como Selenium[4], como propone Kongsli ([Kon07]). Respecto a los tests de servicio, pueden utilizarse herramientas de fuzzing a nivel de API, encargadas de proveer entradas inválidas, inesperadas, o randomizadas a los programas, así como herramientas para evaluar configuración de servicios mediante la especificación del estado deseado de la misma. En cuanto a los tests unitarios, es posible escribir tests sobre funciones de código con entradas que estén por fuera del camino esperado del software, y evaluar que entradas podría querer usar un atacante para afectar el correcto funcionamiento del software.

3.2. Automatización de seguridad

La automatización de la seguridad es imprescindible para construir un sistema de manera rápida e incremental, sin retrasar los tiempos de entrega al cliente. Para ello se requiere un cambio cultural en las organizaciones en el cual los equipos de desarrollo, operaciones y seguridad trabajen de forma conjunta sobre los despliegues y la seguridad del sistema. Esto permite realizar despliegues automatizados, así como ejecutar pruebas y controles de seguridad automáticamente ante cualquier cambio en el código. De esta forma, se obtiene un feedback rápido acerca de componentes del sistema que pueden presentar riesgos de seguridad, y se puede actuar en consecuencia.

[4]https://docs.seleniumhq.org/

A continuación se detallan cambios culturales conocidos como DevOps y DevSecOps que buscan integrar los equipos de trabajo en las organizaciones. También se presentan prácticas como *Infrastructure − as − Code*, e integración y despliegue continuos para la gestión de infraestructura y puesta en producción del sistema. Por último, se describen nuevos desafíos de seguridad que deben ser considerados en las organizaciones que implementan estos conceptos.

3.2.1. DevOps

Uno de los principales objetivos de las metodologías ágiles es el de ofrecer valor al cliente de la manera más rápida posible. Para alcanzarlo las organizaciones deben contar con herramientas que permitan desarrollar nuevas funcionalidades, así como desplegarlas en producción rápidamente. Sin embargo, estas tareas tradicionalmente se realizaban por equipos independientes. Por un lado los desarrolladores se encargaban del código, y por otro, el área de operaciones se encargaba de realizar los despliegues. Esto comúnmente llevaba a problemas de relacionamiento entre dichos equipos, en los que se responsabilizaban unos a otros ante problemas a la hora de desplegar el software y entregar valor a los clientes. A este problema también debía sumarse el miedo al cambio, que llevaba a retrasar y a evitar cambios en producción por miedo a que los sistemas dejaran de funcionar. A su vez, existían despliegues en los cuales los propios equipos no tenían confianza en que fueran a ser exitosos, y situaciones en las que los desarrolladores aseguraban que el software funciona en sus propios equipos de desarrollo pero no funcionaba correctamente a la hora de desplegarlo en producción, entre otras. Por otra parte, las organizaciones sufrían de un problema conocido como "Siloficación", que describe a organizaciones con tendencias a dividir a los miembros de un equipo en distintos "silos" según su especialización, como lo son desarrolladores, testers, administradores de sistemas, entre otros[Deb10].

DevOps es un movimiento que surge en el año 2009 como respuesta a estos problemas. Su objetivo es incentivar la colaboración entre todos los involucrados en el despliegue del software para entregar valor al cliente más rápido y de una manera más confiable. Para ello define que la responsabilidad de que el software sea desplegado correctamente sea compartida entre desarrolladores y personal de operaciones. Como se ve en la Figura 3.6, esto lleva a un cambio cultural en la organización, en la que equipos de desarrollo y de operaciones deberán trabajar en conjunto, para alcanzar los objetivos.

Rouan Wilsenach ([Wil16]) plantea que DevOps busca que los desarrolladores no pierdan el interés en operaciones y mantenibilidad de los sistemas que desarrollan, y que se involucren y sean responsables de mantener sus sistemas a lo largo de todo el ciclo de vida de los mismos. Expresa que de esa forma los desarrolladores entienden las preocupaciones y problemas a los que se enfrenta el personal de operaciones, y pueden identificar posibles soluciones para simplificar, tanto el despliegue como el mantenimiento de los sistemas. Esto permite que los desarrolladores puedan brindar el enfoque de las metodologías ágiles a la administración de sistemas y operaciones. De esta manera, la integración de DevOps como parte de la cultura de la organización permite poner código en producción de manera más rápida. Para ello, será imprescindible que todos los equipos se involucren en la calidad del proceso de desarrollo, y que den y reciban feedback con el objetivo de continuar mejorando el proceso. También deberán involucrarse en la automatización, de forma de acelerar tareas repetitivas y

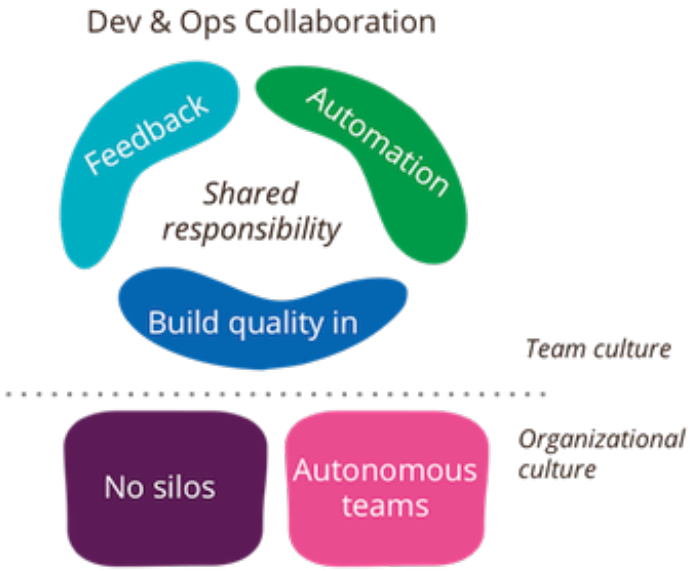

Figura 3.6: Colaboración en DevOps[Wil16]

disminuir el error humano.

3.2.2. Infrastructure-as-Code

Infrastructure-as-Code (IaC), o infraestructura como código en español, es definido como un proceso que permite gestionar la infraestructura de los sistemas de la organización mediante el uso de herramientas utilizadas para el desarrollo de software[ABDN+17].

Previamente cuando una organización quería hacer cambios en su infraestructura, debía comprar servidores físicos, e instalar y configurar tanto sistemas operativos como software de manera manual. Estos procesos podían llegar a demorar días, e incluso semanas, creando una brecha entre el momento en que la organización se decidía a hacer los cambios y el momento en que efectivamente los cambios eran realizados. En la actualidad, la computación sobre la nube (o *cloud computing* en inglés) permite desplegar infraestructuras en segundos o minutos. IaC explota la ventajas de esta tecnología y permite que las organizaciones puedan abstraerse de la capa física de sus sistemas, brindando las mejores prácticas del software a la gestión y manejo de la infraestructura. Para ello IaC integra herramientas como software de versionado de código permitiendo manejar un repositorio centralizado que almacena todos los códigos utilizados para gestionar la infraestructura de la organización. Dichos repositorios pueden incluir información acerca de sistemas operativos utilizados, configuración de red y firewall, software a instalarse, archivos de configuración para el software, configuración de seguridad, usuarios, manejo de dependencias, entre otros.

Existen además diversas herramientas como ansible[5], saltstack[6] y puppet[7]

[5]https://www.ansible.com/overview/how-ansible-works
[6]https://www.saltstack.com/
[7]https://puppet.com/

que permiten definir rutinas de configuración de infraestructura con facilidad. En el listado 3.1[8] se ve un código de ejemplo escrito usando ansible mediante el cual a través de simples pasos es posible configurar un servidor de base de datos MySQL[9].

```
1  ---
2  - hosts: server
3    sudo: yes
4    sudo_user: root
5
6    tasks:
7    - name: install mysql-server
8      apt: name=mysql-server state=present update_cache=yes
9
10   - name: install ansible dependencies
11     apt: name=python-mysqldb state=present
12
13   - name: Ensure mysql is running
14     service: name=mysql state=started
15
16   - name: Create user with the password and all
       previleges
17     mysql_user: login_user=root login_password="" name={{
         mysql_user }} password={{ mysql_password }} priv
         =*.*:ALL host=% state=present
18
19   - name: Delete test database
20     mysql_db: name=test state=absent
21
22   - name: Create ansible_example database
23     mysql_db: name=ansible_example state=present
24
25   - name: Copy mysql back up dump to the remote_user
26     copy: src=dump.sql.bz2 dest=/tmp
27
28   - name: Restore the dump into ansible_example database
29     mysql_db: name=ansible_example state=import target=/
         tmp/dump.sql.bz2
```

Listado de código 3.1: Ejemplo de IaC utilizando la herramienta ansible

Para ello define el servidor sobre el cual se aplicarán los cambios (línea 2), y el usuario con el que el script será ejecutado (líneas 3 y 4). Posteriormente plantea múltiples tareas (*tasks*) que se componen de un nombre (*name*), y de una acción a ejecutar. La línea 7, por ejemplo, define una tarea para la instalación del servidor web, mientras que la línea 8 define la acción asociada,

[8]Ejemplo tomado de https://www.thoughtworks.com/insights/blog/infrastructure-code-reason-smile

[9]https://dev.mysql.com/doc/mysql-getting-started/en/

siendo en este caso la ejecución del comando apt para instalar el paquete mysql-server en el servidor. Las siguientes tareas muestran como es posible configurar un usuario y contraseña (líneas 16 y 17), crear una base de datos llamada ansible_example (línea 22), así como tomar un respaldo, y recuperarlo (líneas 25 a 29).Por otra parte, la infraestructura deja de ser gestionada únicamente por el personal de operaciones, e involucra a desarrolladores, de acuerdo al movimiento cultural que brinda DevOps a la organización. De esta forma, tanto el personal de operaciones como los desarrolladores serán responsables de que la infraestructura de la organización funcione eficientemente, y compartirán una misma base de conocimiento con el objetivo de brindar valor al cliente de manera rápida. A su vez, la implementación de IaC en la organización brinda diversas ventajas en cuanto a seguridad, trazabilidad, gestión, e incluso a nivel de cumplimiento de normativas. Disponer de un repositorio centralizado con la configuración de toda la infraestructura permite generar homogeneidad en cuanto a los sistemas instalados. Esto es debido a que todos los sistemas son instalados y configurados a partir del mismo código, por lo que dichos sistemas siempre serán iguales, y el proceso de instalación de los mismos es repetible. Previo a IaC esta situación resultaba más compleja debido a que las organizaciones contaban con documentación acerca de como configurar sus sistemas, pero la ejecución de dicha documentación podía depender de quién fuese el encargado de ejecutarla. Esto resultaba comúnmente en distintos resultados ante la ejecución de la misma documentación por parte de dos personas distintas. A su vez, IaC reduce el riesgo de un posible error humano, y brinda trazabilidad y funcionalidades de logging que permiten conocer si cada instrucción fue ejecutada correctamente, o si ocurrió algún problema.

IaC permite que los repositorios sean accedido por todos los involucrados, y que cualquiera pueda proponer cambios sobre los mismos. Como se ve en la Figura 3.7, para ello solo basta con sugerir un cambio sobre los repositorios usando las herramientas tradicionales de gestión de código, y el mismo puede ser revisado y auditado por el resto de los involucrados. Esto genera transparencia, y trazabilidad acerca de los cambios propuestos por cada integrante de la organización. Además, cada cambio sugerido podrá ser testeado, previo a volverse definitivo, y aplicarse sobre la infraestructura ya desplegada o nueva infraestructura.

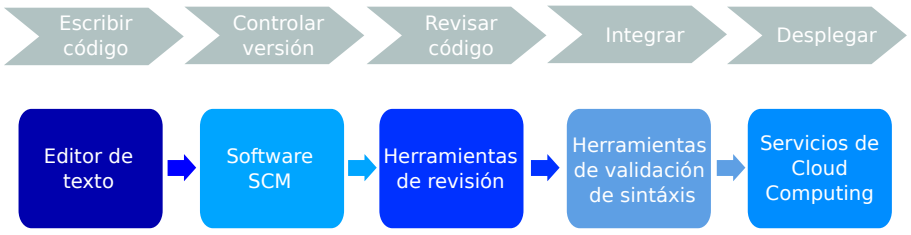

Figura 3.7: Infrastructure-as-Code

Asimismo, puede resultar muy útil para la detección de comportamientos y cambios anómalos que pueda sufrir la infraestructura, permitiendo detectar incidentes de seguridad rápidamente. Esto es debido a que también provee herramientas que permiten identificar si la actual configuración de los sistemas desplegados es consistente con la configuración inicial que fue aplicada sobre los

mismos.

3.2.3. Continuous Integration

Continuous integration (CI), o integración continua en español, es una práctica en la cual los miembros de un equipo de desarrollo integran su trabajo múltiples veces al día. Esto permite detectar errores rápidamente y localizarlos en el código con más facilidad[Tho]. A su vez, de esta forma se evitan problemas que pueden surgir posteriormente a la hora de integrar distintos sistemas que se desarrollan de manera independiente.

Desde la perspectiva de un desarrollador la integración continua puede verse de la siguiente manera:

Cada vez que un desarrollador comienza a trabajar en una nueva funcionalidad del producto, debe realizar una copia del código funcional almacenado en el repositorio centralizado, y almacenarla en su equipo local. Posteriormente aplica los cambios que entienda necesarios únicamente sobre su copia local del código, sin afectar el repositorio centralizado. A su vez, debe incluir tests unitarios que verifiquen que los cambios introducidos se comportan de la manera esperada. Una vez finalizada la nueva funcionalidad con sus tests funcionando, el desarrollador debe actualizar su copia local con los nuevos cambios que haya habido en el repositorio central que pudieron haber realizado otros desarrolladores desde que realizó la copia original. Si alguno de los nuevos cambios realizados rompe la funcionalidad en desarrollo, deberán solucionarse los problemas hasta que todos los tests (tanto los nuevos como los anteriores) efectivamente sean aprobados nuevamente. Únicamente a partir del momento en que todos los tests son ejecutados correctamente el desarrollador podrá impactar su trabajo en el repositorio central de código, en un proceso conocido como commit. Una vez que esto ocurre, el sistema debe ser construido desde cero. Este proceso es conocido como *build* e incluye tareas como compilación de código fuente, integración de librerías de terceros, creación de binarios, entre otras. A su vez, se ejecutan todos los tests nuevamente, en una etapa conocida como testing. Si los mismos fallan los cambios deben ser descartados. De esta forma, se asegura que el código en el repositorio centralizado siempre se encuentra funcional y evita problemas de integración posteriores. Como se ve en la Figura 3.8, Continuous Integration consiste en que los desarrolladores apliquen estos pasos de manera cíclica múltiples veces al día, permitiendo brindar valor de manera iterativa e incremental.

Para que esta forma de trabajo funcione correctamente es fundamental que todos los desarrolladores respeten las siguientes prácticas[FF06]:

- Mantener un único repositorio de código
- Automatizar el build
- Hacer que el build sea auto-testeable
- Por cada commit debe hacerse un build en una máquina de integración
- Mantener el build rápido
- Realizar tests en un clon del ambiente de producción
- Todos pueden ver lo que esta ocurriendo

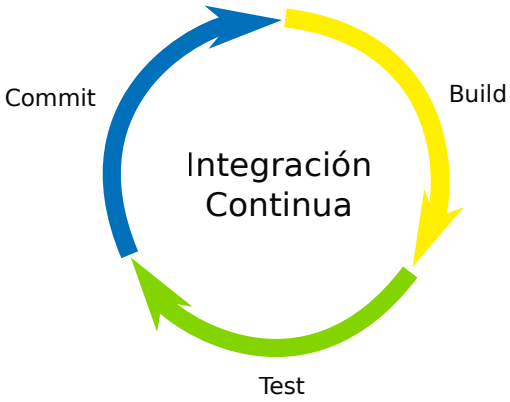

Figura 3.8: Proceso de integración continua

- Automatizar los despliegues

Su aplicación brinda numerosas ventajas en una organización. Permite reducir tiempos y problemas a la hora de integrar cambios, aumentando así la velocidad de entrega al cliente. A su vez, permite verificar continuamente que el nuevo código es estable y que no tenga errores. Esto es posible debido a la insistencia que se realiza en cuanto a escribir tests sobre cualquier cambio. Si bien esto no es obligatorio, como en otras prácticas como Test Driven Development (TDD), es conveniente para contar con un buen cubrimiento de tests (*test coverage*)[10] sobre el código. Esto brinda un feedback valioso a los desarrolladores, por lo que es recomendado mantener la ejecución de los procesos de build de la manera más rápida posible. A su vez, debido a que cualquier cambio en el código que resulte en un fallo en cualquier etapa del ciclo implica que dichos cambios no podrán ser entregados al cliente, los mismos deberán ser arreglados de manera inmediata.

Continuous Integration permite además la ejecución frecuente del código fuera del sistema de los desarrolladores. Esto permite asegurar que el código puede ejecutar correctamente en un ambiente con características idénticas, o muy similares al ambiente productivo en el cual se desplegarán finalmente.

Por otra parte, el repositorio centralizado de código, alineado a la cultura de DevOps y al proceso de Infrastructure-as-Code, debe incluir información como scripts de instalación de sistemas, archivos de configuración, listas de dependencias, entre otros. De esta forma, cualquier desarrollador que realice una copia del proyecto en su equipo puede ejecutar las acciones de build y testing en su propia computadora. Asimismo, el repositorio incluso puede incluir archivos de configuración de los entornos integrados de desarrollo (*Integrated Development Environments* (IDE), en inglés). De esta forma es posible que todas las computadoras utilizadas para desarrollar tengan una configuración similar.

Herramientas comúnmente utilizadas para integración continua pueden incluir Jenkins[11], Circle CI[12], Travis CI[13] y GitLab[14], entre otras.

[10] https://www.thoughtworks.com/insights/blog/are-test-coverage-metrics-overrated
[11] https://jenkins.io/doc/
[12] https://circleci.com/product/
[13] https://travis-ci.com/
[14] https://about.gitlab.com/product/continuous-integration/

3.2.4. Continuous Deployment

Una vez que una organización adopta un cambio cultural hacia DevOps e Infrastructure-as-Code, y es capaz de integrar código todos los días mediante integración continua, el siguiente paso en cuanto a automatización refiere a desplegar la aplicación a producción de manera totalmente automatizada. *Continuous Delivery* y *Continuous Deployment* son dos prácticas de DevOps que permiten alcanzar este objetivo, y permiten que las organizaciones sean capaces de brindar valor a sus clientes de manera más rápida.

Continuous Delivery, comúnmente abreviada CDE, refiere a mantener siempre el sistema en un estado en el cual pueda ser liberado a producción. Jez Humble ([Hum06]) plantea que está definida por los siguientes 5 principios:

- Construir calidad dentro del producto
- Trabajar en lotes pequeños
- Las computadoras ejecutan tareas repetitivas, las personas resuelven problemas
- Buscar la mejora continua de manera implacable
- Todos son responsables

Su implementación brinda diversas ventajas a las organizaciones. Debido a que los cambios realizados son más pequeños, es más fácil verificar que los mismos funcionan correctamente, y por tanto, que sean liberados de manera constante. Esto a su vez disminuye los riesgos asociados a cada puesta en producción, y los tiempos necesarios para las mismas. Asimismo, los procesos automatizados reducen el riesgo de error humano, brindando repetibilidad y trazabilidad de todos los pasos ejecutados[Hum06].

Por otra parte, Continuous Deployment, comúnmente abreviada CD, refiere a liberar a producción efectivamente cada uno de los cambios realizados de manera automatizada todos los días. Éstos cambios pueden consistir en la inclusión de nuevas funcionalidades, la corrección de defectos (*bugs*) en el código, o incluso cambios en la infraestructura de los sistemas.

Como se ve en la Figura 3.9 [15]la principal diferencia entre ambas prácticas refiere a que en la primera de ellas, la decisión del pasaje a producción debe ser realizada manualmente por una o varias personas, pudiendo definir en qué momento o qué funcionalidades en particular se liberan, mientras que en Continuous Deployment todos los cambios son liberados de manera automática sin ningún tipo de intervención humana.

Para alcanzar estos objetivos, estos procesos son implementados en forma de *pipelines*. Los mismos se descomponen en distintas fases, o *stages* en inglés, que pueden incluir procedimientos como: testing, build, orquestación de ambientes y máquinas virtuales, instalación de sistemas operativos, configuración de sistemas, instalación de software de base, ejecución de tests de aceptación, entre otras. Debido a que la ejecución de las mismas puede llevar un tiempo considerable, es posible paralelizar algunas de ellas. De esta forma, los tiempos de ejecución disminuyen y se puede brindar un feedback rápido a los equipos involucrados.

[15]Figura tomada de https://softcrylic.com/blogs/testing-strategies-continuous-delivery/

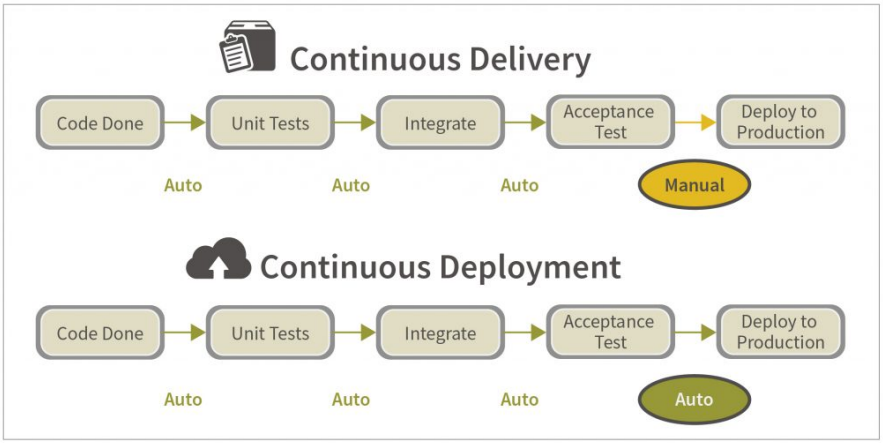

Figura 3.9: Continuous Delivery y Continuous Deployment

3.2.5. DevSecOps

Neil MacDonald ([Mac12]) plantea que en las organizaciones existen problemas de comunicación y de procesos. Los mismos, a su entender, afectan tanto a las áreas de desarrollo y operaciones, como al área de seguridad de la información. A su vez, considera que si bien la práctica de DevOps busca resolver estos problemas, la misma no es suficiente. Entiende que esta práctica representa una visión incompleta de las estructuras de una organización debido a que falta la inclusión de un "silo" referido a la seguridad de la información. Por esta razón, plantea que dichas áreas deben colaborar entre sí, y propone un cambio cultural conocido por el nombre de *DevOpsSec*. De acuerdo a un estudio realizado por Myrbakken et. al., este cambio puede ser referido también con términos como *DevSecOps*, *SecDevOps* o *Rugged DevOps*, entre otros[MCP17]. Nos referiremos a este concepto de aquí en adelante como DevSecOps.

Esta práctica consiste en incluir seguridad en DevOps con el objetivo de adoptar las buenas prácticas y beneficios que éste último ofrece en los procesos de desarrollo. Para ello propone realizar un *shift left* (o corrimiento a la izquierda, en español) de la seguridad[Lie16]. Esto refiere a incorporar prácticas de seguridad desde las etapas más tempranas del ciclo de desarrollo[Sha12]. Dichas prácticas comienzan con la capacitación y concientización de los desarrolladores, así como con el involucramiento de los stakeholders, siendo imprescindible el compromiso de todas las partes respecto a la seguridad del software.

De igual manera que DevOps propone la integración de herramientas para asegurar la correctitud del código durante todo el ciclo de desarrollo, DevSecOps sugiere la integración de herramientas de seguridad como parte de los procesos de integración y despliegue continuos[Muk]. La integración de dichas herramientas componen pipelines conocidos como *application security pipelines*, que pueden ser abreviados como *app sec pipelines*[OWA18]. Dichos pipelines podrán incluir fases como automatización de revisión de código, testing de seguridad, escaneos de seguridad, monitoreo y generación automatizada de reportes, entre otras. A su vez, se destacan actividades de aprendizaje y medición, así como de predicción de potenciales problemas.

3.2.6. Nuevos desafíos

La implementación de las prácticas expresadas anteriormente por parte de una organización permiten integrar y automatizar la seguridad en los sistemas desde el primer momento. Sin embargo, estos conceptos y cambios culturales pueden exponer a las organizaciones a nuevos riesgos. Esto se debe, por ejemplo, al hecho de que todo el código fuente esta alojado en un repositorio centralizado, así como que toda la infraestructura de la organización puede llegar a ser manejada de manera automatizada por código. El valor y la importancia de estas tecnologías provocan que sean un claro objetivo para posibles atacantes. Es por ello que las organizaciones deben evaluar los riesgos asociados a las mismas para protegerlas adecuadamente.

Activos como los repositorios de código pueden ser un importante objetivo, tanto para amenazas externas a la organización, como incluso para amenazas internas (también conocidas como *insider threats* en inglés). Por dicha razón resulta imprescindible implementar estrictos procesos de control de acceso sobre ellos. Esto permite mantener la confidencialidad del código, así como prevenir cambios no autorizados sobre el mismo, protegiendo su integridad. A su vez, se pueden implementar revisiones de código obligatorias entre pares. De esta forma se impide que una única persona pueda impactar cambios con objetivos maliciosos. Es posible además integrar herramientas de análisis de código con el objetivo de detectar la introducción, tanto deliberada como no intencional, de cambios no deseados. Sumado a estas medidas, también resulta conveniente que la organización cuente con procedimientos que brinden trazabilidad total sobre los cambios realizados.

Otro objetivo importante para un atacante refiere a los servidores encargados de llevar a cabo la integración continua de código[BHR+15][DPH18]. Un ataque exitoso sobre dichos procesos permitirían el compromiso y acceso no autorizado a activos críticos de la organización. Esto podría resultar en problemas como la troyanización e inclusión de malware en el software durante los procesos de build y la ejecución de pipelines. A su vez, atacantes podrían tomar control de herramientas de seguridad incluidas en el pipeline y extraer sus resultados, así como iniciar escaneos y pruebas contra otros equipos o servicios de la organización, o incluso de terceros. Esta situación podría agravarse en el caso de que la organización también implemente continuous deployment, ya que podría derivar en el compromiso de servidores y equipos críticos en ambientes productivos.

A su vez, los servidores encargados de ejecutar los procesos de continuous deployment pueden incluir scripts de despliegue con información sensible. Dicha información puede incluir usuarios y contraseñas de servidores, claves privadas de certificados SSL/TLS, claves privadas utilizadas para gestión de equipos mediante protocolo SSH, tokens de APIs, entre otros. Por esta razón es imprescindible realizar una configuración segura de los servidores de integración y despliegues continuos, incluyendo un estricto proceso de *hardening* sobre los mismos. Asimismo es importante mantener estas herramientas actualizadas y evitar problemas como incluir configuraciones por defecto, uso de contraseñas inseguras y uso de software con vulnerabilidades conocidas.

También resulta conveniente proteger herramientas de gestión de vulnerabilidades y herramientas de logs con detalles de ejecución de pipelines. Las mismas contienen información sensible acerca de todos los problemas de seguridad presentes en el software de la organización, y su compromiso podría exponer los

sistemas en producción a riesgos y ataques que aún no han sido resueltos.

Por otra parte, herramientas de mensajería instantánea, al igual que el correo electrónico, son comúnmente utilizadas por equipos de desarrollo para comunicarse entre sí, así como para comunicarse con los stakeholders. Estas herramientas son utilizadas con múltiples objetivos como coordinar actividades, discutir requerimientos, analizar detalles de implementación, intercambiar archivos, compartir snippets de código fuente, entre otras. Su uso puede exponer información sensible a atacantes, por lo que su compromiso puede resultar de interés a atacantes. A dicho interés debe sumarse el hecho de que permiten integrarse, mediante plugins, a otras herramientas de desarrollo, como repositorios centralizados y herramientas de integración continua. De esta forma, cada vez que algún desarrollador hace un push de cambios de código o ejecuta pipelines, canales de comunicación de mensajería instantánea reciben notificaciones al respecto, con sus respectivos resultados. Éstas herramientas incluso permiten ejecutar determinados procesos mediante el envío de comandos particulares en dichos canales de comunicación, característicos de una práctica conocida como ChatOps[16]. Esto presenta nuevos riesgos para la organización, por lo que es importante proteger estas herramientas.

3.3. Diseño de AppSec Pipeline

Esta sección tiene como objetivo presentar prácticas y herramientas que pueden ser consideradas a la hora de implementar un AppSec pipeline (o application security pipeline) y organizar sus tests. Su diseño debe ser realizado a medida y dependerá de múltiples factores relacionados a cada organización incluyendo su tamaño, industria, cultura y madurez, así como las metodologías y tecnologías utilizadas.

El objetivo de un AppSec pipeline es identificar problemas de seguridad de la manera más rápida posible para que puedan ser corregidos a la brevedad. A su vez, debe permitir liberar a producción dichas correcciones de manera rápida y segura. Estos pipelines se componen, tanto de procesos manuales como de procesos automatizados.

Los procesos manuales (o semi-automatizados) comienzan con la comprensión de las aplicaciones a ser testeadas, permitiendo entender sus requerimientos de seguridad. A partir de dicha información se definen las user stories relacionadas a seguridad, que pueden expresarse en forma de security y abuser stories. A su vez, estos procesos incluyen la ejecución de modelados de amenazas y la definición de pruebas manuales, que luego serán utilizadas como insumo a la hora de definir qué herramientas automatizadas se utilizarán. Asimismo se pueden incluir revisiones de código entre pares, e incluso tests de penetración, aunque éstos últimos son ejecutados de forma paralela.

En cuanto a los procesos automatizados, consisten en ejecutar múltiples tests y herramientas de seguridad sobre el sistema cada vez que se realiza un cambio. Para maximizar sus beneficios es importante que su ejecución sea sencilla, y que sus resultados sean consistentes y repetibles. A su vez, los tiempos de ejecución deben ser bajos para poder brindar un feedback temprano. Para ello es importante optimizar la ejecución de cada herramienta, y en consecuencia reducir el tiempo total de ejecución. También resulta conveniente considerar que

[16] https://www.pagerduty.com/blog/what-is-chatops/

la ejecución de un pipeline es fallida si alguna de sus principales herramientas identifica un problema de gravedad, o si se identifica una cantidad particular de problemas. De esta forma se reducen los tiempos de ejecución y se fuerza a los equipos a que solucionen los problemas identificados previamente a la ejecución de pruebas más complejas.

Un application security pipeline puede incluir la ejecución de herramientas (a ser presentadas en la sección 3.4) como firewalls de código, tests unitarios, tests de seguridad, gestión de dependencias, analizadores estáticos de código, builds hardenizados, analizadores dinámicos y scanners de vulnerabilidades, entre otras. Cada una de ellas puede expresarse como una fase independiente en el pipeline, y se podrán ejecutar tanto de manera serial como en paralelo, dependiendo de las necesidades de cada sistema. Una vez ejecutadas, sus resultados deben ser consolidados y normalizados. A su vez, los mismos pueden ser ingresados de manera automatizada en herramientas de *tracking* de bugs para ser procesados posteriormente. Este procesamiento puede incluir procesos de verificación para descartar falsos positivos, así como el análisis de las vulnerabilidades identificadas y su eventual remediación.

Por último, resulta conveniente acompañar la ejecución de procesos manuales, como automatizados, de toma de métricas. Éstas últimas permiten acelerar los procesos e identificar oportunidades de mejora, así como ofrecer un feedback continuo al proceso completo.

3.4. Herramientas

A continuación se hará una introducción a distintos tipos de pruebas y herramientas que pueden ser incluidas en un flujo de desarrollo ágil con el objetivo de integrar y automatizar la seguridad. Como se ve en la Figura 3.10, existen herramientas que se integran en distintas fases del ciclo. Se describen algunas de ellas capaces de detectar problemas en el código fuente del software, tanto con enfoques de caja blanca (white box) como con enfoques de caja negra (black box). También se detallan otras que permiten evaluar la configuración de los sistemas, identificando posibles problemas de seguridad en sistemas operativos, software de base y servicios, entre otros. Por último, se presentan herramientas capaces de verificar que los sistemas y aplicaciones se encuentran funcionando adecuadamente mediante el uso de técnicas de recolección de logs y de monitoreo.

3.4.1. Firewall de código

Permiten definir reglas de seguridad simples en el software de gestión de código (*Source Code Management* (SCM) en inglés)[Roh16]. Soluciones SCM como lo son git[17] o subversion[18] permiten definir *hooks* con acciones personalizadas que se ejecutan ante determinados eventos. Dichos eventos pueden ocurrir tanto en los clientes como en servidores, y refieren a acciones como $pre - commit$ y $post - commit$. De esta forma, es posible procesar los cambios en el código previamente a que sean centralizados y aplicar controles de seguridad básicos.

Estas herramientas pueden detectar problemas como la inclusión de claves privadas, contraseñas en texto plano, *tokens* de APIs u otro tipo de información

[17] https://git-scm.com/

[18] https://subversion.apache.org/features.html

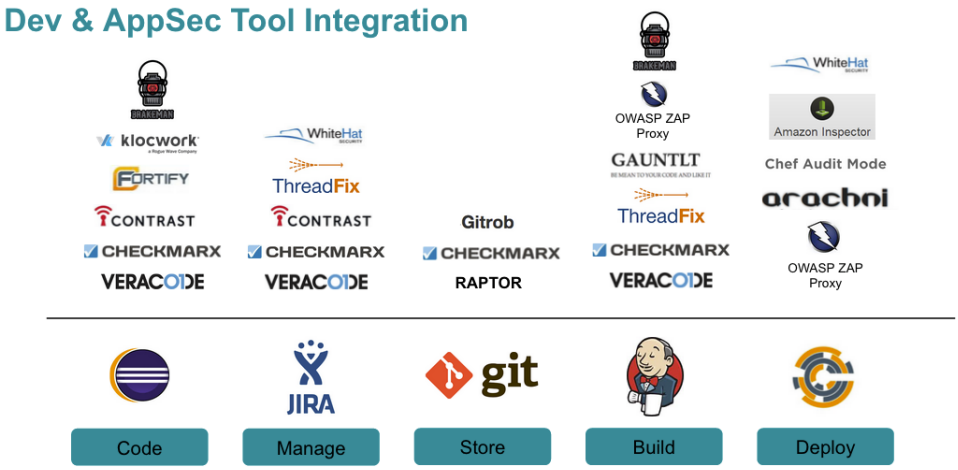

Figura 3.10: Ejemplo de herramientas en AppSec Pipeline

sensible como parte del código fuente. A su vez, permiten detectar potenciales problemas de inyecciones, tanto en sentencias SQL como en ejecución de comandos a nivel de sistema operativo, entre otros.

3.4.2. Análisis de código estático

Como se mencionó anteriormente en la sección 3.1.5 existen herramientas automatizadas que permiten encontrar vulnerabilidades mediante la revisión de código fuente, *byte code* y archivos binarios. Las mismas son conocidas como herramientas de análisis estático de código, o SAST por sus siglas en inglés (*Static Application Security Testing*).

Estas herramientas son flexibles y se integran con facilidad al ciclo de desarrollo, brindando feedback incluso mientras el código está siendo escrito. Debido a que no requieren que el código sea ejecutado para identificar problemas de seguridad, es posible integrarlas a los entornos de desarrollo de cada desarrollador, también conocidos como IDE (*Integrated Development Environment*). De esta forma los errores pueden ser detectados rápidamente, y los equipos de desarrollo pueden trabajar en mitigaciones de manera temprana. Esto provoca mejoras en la integridad del código y lleva a que los desarrolladores escriban código más seguro.

Debido a que son ejecutadas de forma previa a realizar commits de código a los repositorios centralizados, reducen la probabilidad de liberar código vulnerable. Sin embargo, las mismas también pueden integrarse directamente en estos repositorios, de forma de agregar distintas capas de protección. Su ejecución puede ser obligatoria previo a realizar un *merge* con código a las ramas estables de código o a producción.

Su uso permite identificar vulnerabilidades web como inyecciones SQL, ataques de *Cross − Site Scripting* e inyecciones de comandos, entre otras. A su vez, en lenguajes de bajo nivel permiten detectar problemas en manejo de memoria, como *buffer* y *heap overflows*.

3.4.3. Gestión de dependencias

El uso de componentes con vulnerabilidades conocidas es uno de los principales problemas de seguridad que afecta a las aplicaciones web hoy en día, ocupando la posición número 9 en el proyecto OWASP Top Ten 2017[19]. Una de las razones que llevan a este problema es la complejidad que tienen actualmente las aplicaciones, que normalmente hacen uso de múltiples componentes de terceros. Esto dificulta que los equipos de desarrollo estén al tanto de todos los componentes utilizados, así como de sus respectivas versiones, y por tanto, de las vulnerabilidades que estas últimas presentan. Este problema se ve agravado debido a que puede haber componentes con dependencias anidadas, así como componentes que dejan de ser soportados por sus desarrolladores[OWA18].

Para resolver estos problemas es imprescindible mantener un inventario con todos los componentes utilizados con sus respectivas versiones. De esta forma, los equipos de desarrollo pueden suscribirse a boletines de seguridad relacionados a dichos componentes y mantenerse al tanto de las vulnerabilidades publicadas. Esto puede ser complementado con la ejecución de escaneos de seguridad regulares. A su vez, es recomendable remover dependencias no utilizadas, así como funcionalidades y archivos innecesarios, de forma de reducir la superficie de ataque.

Existen herramientas que permiten automatizar estos procesos. Para ello identifican las versiones de componentes utilizados, y buscan las mismas en una base de datos de componentes con problemas conocidos. Dichas base de datos pueden ser consultados en línea, o pueden descargarse para ser utilizadas de forma local. En caso de identificar un problema, notifican a los desarrolladores y sugieren la actualización de los componentes afectados a versiones en las cuales las vulnerabilidades fueron corregidas.

3.4.4. Build hardenizado

El hardening de un equipo refiere al procedimiento de configurar de manera segura el sistema operativo y las aplicaciones de un sistema, con el objetivo de reducir los problemas de seguridad del mismo[SJT08]. Dicha definición es dada por el *National Institute of Standards and Technology* (NIST)[20], o Instituto Nacional de Estándares y Tecnología en español. Estos procedimientos puede incluir distintos tareas como:

- Configurar el equipo para realizar instalación automatizada de actualizaciones
- Remover o deshabilitar servicios, aplicaciones y protocolos de red innecesarios
- Configurar las cuentas de usuario a nivel de sistema operativo
- Forzar una política de contraseñas robusta
- Configurar el firewall
- Instalar software antivirus

[19] https://www.owasp.org/index.php/Top_10-2017_Top_10
[20] https://nist.gov/

Los procesos de hardening pueden ser extensos, y los pasos a ejecutar pueden variar según la experiencia y conocimientos de quien los aplica. Sin embargo, debido a que los mismos pueden ser automatizados en forma de scripts, es posible integrarlos a prácticas como Infrastructure-as-Code. Esto permite escribir rutinas de hardening en plataformas como Chef[21] y Puppet. A su vez, éstas rutinas podrán ser escritas de manera colaborativa por personal de desarrollo, infraestructura y de seguridad como propone DevSecOps, y se podrá tener trazabilidad sobre sus cambios.

3.4.5. Gestión de configuración

Este tipo de herramientas permite verificar que los sistemas se encuentran configurados de manera segura y que cumplen con determinados estándares y normativas. Pueden ser ejecutadas una vez que los sistemas son aprovisionados para verificar que la configuración de los mismos es correcta y que las rutinas de hardening fueron aplicadas de la manera esperada. Para ello ejecutan múltiples tests que realizan consultas al sistema operativo y analizan archivos de configuración. De esta forma, detectan problemas como una configuración indebida de firewalls, configuraciones inseguras en servicios (HTTP, DNS y NTP, entre otros), instalación de paquetes innecesarios en sistemas operativos y posibles vectores de escalación de privilegios, entre otros.

A su vez, pueden ser aplicadas periódicamente sobre los sistemas de producción. De esta forma, es posible identificar cambios no deseados o no autorizados, reduciendo riesgos y detectando prematuramente potenciales incidentes de seguridad.

La integración de múltiples tests puede resultar compleja, debido principalmente a su automatización y al procesamiento posterior de sus resultados. Es por ello que estas herramientas normalmente son utilizadas a través de frameworks de testing de seguridad. Éstos permiten listar múltiples comandos que son ejecutados y definir reglas para determinar si los tests son fallidos o no. Dichas reglas pueden definirse como expresiones regulares buscando palabras particulares en la salida de los comandos o a partir del código de salida de cada herramienta.

3.4.6. Análisis dinámico de software

Las herramientas de análisis estático pueden ser complementadas con herramientas de análisis dinámico. Éstas últimas son conocidas también como herramientas de testing de caja negra, o como DAST, por sus siglas en inglés (*Dynamic Application Security Testing*). Permiten identificar problemas de seguridad mediante la ejecución del sistema a probar. Para ello, deben generar distintas entradas y evaluar el comportamiento del sistema respecto a las mismas.

Debido a requieren que el sistema pueda ser ejecutado, su uso al aplicar integración continua se encuentra limitado a las últimas etapas de testing. Como se ve en la Figura 3.11, requieren que el build sea ejecutado de manera completa previamente[Sol15], a diferencia de las herramientas SAST que pueden ejecutarse durante todas las etapas mostradas. Esto provoca que los problemas de seguridad sean encontrados de forma más tardía, y por tanto, que los desarrolladores vuelvan a familiarizarse con el código desarrollado para trabajar en posibles mitigaciones. A su vez, debido a que no tienen acceso al código fuente, puede resultar

[21] https://docs.chef.io/chef_overview.html

complejo identificar qué porciones de código se ven afectadas por determinadas vulnerabilidades.

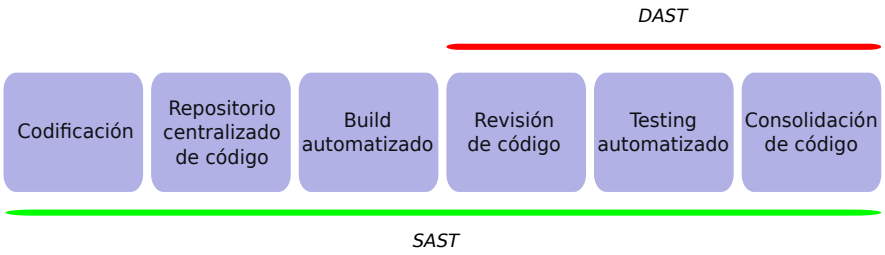

Figura 3.11: Integración de herramientas SAST y DAST

Una desventaja que presentan es que ante cualquier tipo de cambio en el código todo el software debe ser probado de nuevo. Esto provoca incrementos en tiempos y costos de ejecución. Este problema no afecta herramientas de tipo SAST, debido a que las mismas pueden ser configuradas para analizar únicamente el código que fue modificado, y no el código en su totalidad.

3.4.7. Fuzzing

Sumado a los análisis estáticos y dinámicos sobre el software, también es recomendable utilizar herramientas de *fuzzing*. El fuzzing, o *fuzz testing*, es una técnica de testing de caja negra que consiste en inyectar datos malformados al software de manera automatizada[OWA18]. Dichos datos cubren casos de borde, y son enviados como entradas al sistema en forma de archivos, protocolos de red y llamadas a API, entre otros[Oeh05].

Como se ve en la Figura 3.12, éstas herramientas deben ser capaces de reconocer que la aplicación falla ante entradas específicas. Para ello comúnmente integran herramientas de debugging, que monitorean la aplicación y permiten identificar comportamientos no deseados. Una vez que un problema es detectado, se almacena la entrada que lo provocó así como el estado en que quedó la aplicación. Dicha información podrá ser procesada posteriormente para determinar si se trató o no de un problema de seguridad.

3.4.8. Análisis de vulnerabilidades

Los *scanners* de vulnerabilidades son herramientas automatizadas que permiten identificar vulnerabilidades, tanto a nivel de infraestructura como a nivel de aplicación, sin requerir acceso al código fuente. Permiten imitar ataques externos de manera económica, detectando vulnerabilidades y permitiendo evaluar las protecciones implementadas.

Son capaces de identificar configuraciones inseguras, uso de software obsoleto o con vulnerabilidades conocidas, problemas como inyecciones en aplicaciones web, entre otros. A su vez, ofrecen recomendaciones para resolver las vulnerabilidades identificadas, así como reportes específicos para el cumplimiento de determinadas regulaciones y normativas[GBMB10].

Los scanners de aplicaciones web suelen incluir funcionalidades como *crawling* (scripts que recorren sitios web de manera sistemática), fuerza bruta de archivos y directorios, y detección de múltiples problemas como inyecciones SQL, Cross-Site

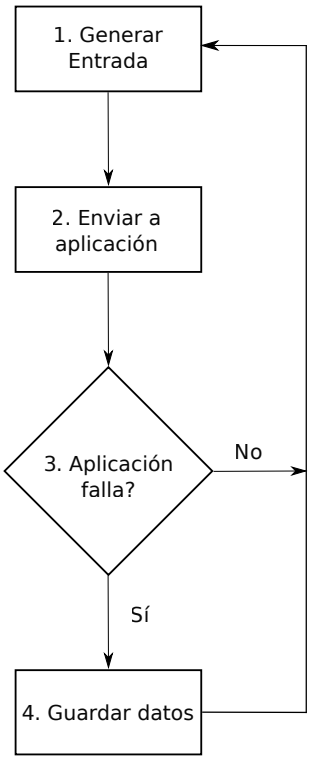

Figura 3.12: Proceso de fuzzing[Oeh05]

Scripting (XSS), manejo de sesiones inseguro, entre otros. A su vez, permiten configurar credenciales de inicio de sesión o *cookies* para escanear funcionalidades protegidas por procesos de login. Debido a no requieren acceso al código fuente de las aplicaciones, normalmente se adaptan a distintos tipos de tecnologías y pueden abstraerse de detalles de implementación. De todas formas, también existen scanners disponibles diseñados para detectar vulnerabilidades sobre tecnologías particulares.

En cuanto a los scanners de infraestructura, los mismos comúnmente se componen de un motor encargado de ejecutar las pruebas y de un conjunto de plugins que especifican cada una de las pruebas a realizarse. Dichos plugins pueden describir problemas como uso de versiones inseguras de software, identificación de recursos por defecto, uso de protocolos inseguros, problemas conocidos en dispositivos de red, uso de puertos inseguros, entre otros. A su vez, permiten basar los escaneos en regulaciones como son PCI DSS (*Payment Card Industry Data Security Standard*)[22], SOX[23] y FISMA[24].

Si bien estas herramientas pueden ejecutarse en su configuración por defecto, las mismas ofrecen múltiples parámetros y configuraciones distintas. Debido a que por defecto pueden realizar escaneos innecesarios, o que no apliquen a las necesidades de los sistemas a ser escaneados, es importante realizar una configuración personalizada con el objetivo de reducir tiempos de ejecución. A

[22]https://www.pcisecuritystandards.org/
[23]http://www.soxlaw.com/
[24]https://www.dhs.gov/fisma

su vez, dado que los proveedores de este tipo de herramientas frecuentemente actualizan dichos sus funcionalidades y plugins, es necesario actualizarlas de manera continua para obtener mejores resultados.

Configuración de SSL/TLS

En la actualidad utilizamos aplicaciones web para transmitir información sensible como usuarios, contraseñas, números de tarjetas de crédito y números de cuentas bancarias. El envío de este tipo de información a través de Internet provoca que sea imprescindible garantizar la seguridad de las comunicaciones. Esto aplica tanto a comunicaciones entre clientes y servidores, como entre posibles equipos y componentes que conforman el sistema a desarrollarse.

Actualmente la opción disponible más utilizada para alcanzar este objetivo es el uso del protocolo SSL/TLS (*Secure Socket Layer/ Transport Layer Security*). El principal beneficio que ofrece este modelo es que impide que la información sensible que es enviada pueda ser accedida o modificada mientras es transmitida entre el cliente (navegador web) y el servidor web. A su vez, su configuración más habitual permite que los clientes verifiquen la autenticidad del servidor. Si bien es posible que los servidores también verifiquen la autenticidad de los clientes mediante el uso de certificados de cliente, esta funcionalidad no es utilizada frecuentemente[OWA18].

Este protocolo se compone de distintos tipos de algoritmos y de distintas implementaciones, permitiendo distintas configuraciones de acuerdo a las necesidades de cada sistema. Debido a que las configuraciones por defecto de servidores SSL/TLS habitualmente no son seguras, y a que continuamente son descubiertos problemas en algoritmos utilizados, es recomendable evaluar de manera continua la configuración de nuestros sistemas. A esto deben sumarse también factores como la vida útil de los certificados.

Existen herramientas disponibles en Internet que nos permiten evaluar qué tan segura es la configuración que presentan nuestros equipos. Permiten detectar problemas como:

- Utilización de versiones del protocolo consideradas inseguras/obsoletas
- Falta de cifrado o uso de algoritmos considerados débiles
- Uso de claves de cifrado de largos considerados inseguros
- Cifrado con modos de operación inseguros
- Certificados autofirmados, no válidos, expirados o próximos a expirar
- Vulnerabilidades conocidas en protocolos/algoritmos (por ejemplo, BREACH, CRIME, POODLE, entre otras)

Es posible configurar estas herramientas para que sean ejecutadas de manera automatizada al momento de configurar los equipos. De esta forma, si se detecta algún tipo de problema de seguridad es posible detener los procedimientos de configuración y despliegues, y ejecutar alertas pertinentes. Esto permite garantizar continuamente que nuestros equipos cuentan con la configuración adecuada.

3.4.9. Logging y Monitoreo

Una vez que las aplicaciones son desplegadas en producción de manera segura y automatizada, también resulta conveniente realizar un monitoreo sobre las mismas. Esto permite asegurar que los sistemas se encuentran funcionando correctamente y que su performance es adecuada. Este proceso puede automatizarse, y es conocido como *Continuous Monitoring*, o monitoreo continuo en español. Consiste en configurar y desplegar herramientas de monitoreo de manera automatizada que aseguren que los sistemas se comportan de la manera esperada. Esto permite, tanto detectar como responder rápida y efectivamente a posibles problemas de seguridad.

Asimismo permiten detectar problemas de performance o en el código, excepciones, ataques de *Denial of Service* (DoS), ataques de fuerza bruta, entre otras. Para ello es necesario contar con herramientas de logging que brinden trazabilidad acerca de eventos y actividad de los sistemas. A su vez, pueden almacenar información que resulte útil ante un potencial incidente de seguridad, o para el cumplimiento de normativas.

En el siguiente capítulo daremos una introducción al concepto de modelos de madurez, y presentaremos ejemplos de dichos modelos aplicados a DevOps, así como a DevSecOps.

Capítulo 4

Modelos de madurez

4.1. Introducción

Los modelos de madurez son instrumentos que nos permiten medir el grado de mejora continua en el que se encuentra una organización. Comúnmente presentan un conjunto de escalas o niveles, que pueden ser medidos y comparables. Esto permite que las organizaciones obtengan una visión objetiva del nivel de madurez en que se encuentran, que identifiquen oportunidades de mejora, y que definan qué nivel u objetivos se desean alcanzar.

En la sección 4.2se presentan modelos de madurez de DevOps. Los mismos definen ciertas áreas o categorías a mejorar, y para las mismas definen múltiples niveles de madurez, organizados en estructuras piramidales o matriciales.

A su vez, en la sección 4.3 se presentan modelos de madurez aplicados a DevSecOps. Estos modelos también definen dominios o dimensiones, con sus respectivos niveles y métricas para medir su nivel de progreso, e identificar oportunidades de mejora.

4.2. Modelos de madurez de DevOps

Existen múltiples modelos de madurez que permiten evaluar el grado en que las prácticas de DevOps son implementadas. Los mismos se basan en aspectos como procedimientos implementados, tecnologías utilizadas y cultura organizacional. A continuación se detallan algunos de estos modelos presentados por distintas empresas.

4.2.1. IBM

Permite evaluar las prácticas implementadas por las organizaciones, definir un *roadmap*, y medir la mejora continua. Define 4 pares de categorías: planear/medir, desarrollar/testear, liberar/desplegar, y monitorear/optimizar[Bah13]. Por otra parte, a cada uno de dichos pares se le asignan 4 niveles de madurez incrementales que comienzan desde practicado, consistente y confiable, hasta escalado. Propone desde prácticas simples como estandarizar deployments, a prácticas complejas como automatizar el aislamiento y resolución de problemas.

4.2.2. Solinea

Presenta un camino para que las organizaciones progresivamente adopten agilidad. Para ello define 4 pilares, que considera comunes a las organizaciones, y sobre los cuales deben aplicarse cambios[Sol17]. Los mismos son: procesos, personas, tecnologías y cultura. A su vez, para cada uno de ellos define 5 niveles, que representan un nivel de transición hacia el siguiente nivel, a excepción del nivel 5 en el que consideran que las organizaciones deben permanecer una vez que el mismo es alcanzado. Acciones en sus niveles iniciales pueden incluir ejemplos como procesos manuales, testing ad hoc y build de ambientes manuales. Mientras tanto, acciones en su nivel 5 (optimizado) pueden incluir reportes predictivos, educación continua y requerimientos de negocio claros.

4.2.3. Capgemini

Este modelo plantea que las organizaciones deben implementar herramientas y procesos de manera coordinada, así como aplicar cambios sobre su cultura para implementar DevOps[Cap15]. Al igual que el modelo anterior, divide a la organización en múltiples dimensiones. Sin embargo, en este caso las dimensiones elegidas son: personas, procesos y herramientas. Cada una de ellas podrá ser clasificada en 5 niveles de madurez (básico, emergente, coordinado, mejorado y "nivel superior"). Su nivel más básico describe una organización dividida en silos tradicionales (desarrollo y operaciones), así como procesos separados y caóticos. En su nivel superior define a las organizaciones como un único equipo, y procesos totalmente automatizados.

4.3. Modelos de madurez de DevSecOps

Analizaremos a continuación los siguientes modelos de madurez de DevSecOps:

- Security DevOps Maturity Model (SDOMM)
- DevSecOps Maturity Model (DSOMM)
- DevSecOps Guide

Dichos modelos buscan evaluar la integración de la seguridad con las prácticas de desarrollo y operaciones de las organizaciones.

4.3.1. Security DevOps Maturity Model (SDOMM)

Christian Schneider propone en el OWASP AppSecEU 2015[1] el Security DevOps Maturity Model (SDOMM) como un road-map para proyectos que buscan implementar Security DevOps[Sch15]. Como se resume en la tabla 4.1 este modelo se basa en 4 ejes distintos, para los cuáles se definen 4 niveles de madurez. Dichos ejes son:

- Profundidad dinámica: ¿qué tan profundos son los escaneos dinámicos ejecutados?

[1] https://2015.appsec.eu/

Modelo	Autor	Áreas	Niveles
SDOMM	Christian Schneider	Ejes: - Profundidad dinámica - Profundidad estática - Intensidad - Consolidación	Niveles 1 a 4
DSOMM	Timo Pagel	Dimensiones: - Build y Deployment - Cultura y Organización - Recopilación de información - Infraestructura - Tests y verificación	Niveles según entendimiento de prácticas de seguridad. - Nivel 1: básico - Nivel 2: normal - Nivel 3: alto - Nivel 4: avanzado
DevSecOps Guide	General Services Administration (GSA)	Dominios de responsabilidad: - Dominio global - Gestión de imágenes - Logging, monitoreo y sistema de alertas - Gestión de parches - Gobernanza de plataforma - Gestión de cambios - Desarrollo de la aplicación, testing y operaciones - Despliegue de la aplicación - Cuentas de usuario, privilegios, credenciales y gestión de secretos - Disponibilidad y gestión de performance - Gestión de redes - Autorización para operar procesos (ATO) - Gestión de ciclo de vida de backups y datos - Gestión de acuerdos y finanzas	- Nivel 1: considerado inviable para DevSecOps - Nivel 2 - Nivel 3

Cuadro 4.1: Tabla comparativa de modelos de madurez de DevSecOps

- Profundidad estática: ¿qué tan profundos son los escaneos estáticos ejecutados?
- Intensidad: ¿qué tan intensos son la mayor parte de los ataques ejecutados?
- Consolidación: ¿qué tan completo es el proceso de manejo de hallazgos de las pruebas ejecutadas?

4.3.2. DevSecOps Maturity Model (DSOMM)

Timo Pagel define el DevSecOps Maturity Model (DSOMM) como un modelo que presenta múltiples medidas de seguridad que pueden aplicarse en organizaciones que implementan DevOps, y cómo las mismas pueden ser priorizadas[Pag]. Como se ve en la tabla 4.1 las dimensiones manejadas por este modelo son:

- Build y Deployment
- Cultura y Organización
- Recopilación de información
- Infraestructura
- Tests y verificación

Cada dimensión se compone de hasta 4 niveles de madurez, los cuales incluyen múltiples tareas a implementar. En su nivel más bajo se espera un nivel de entendimiento básico de prácticas de seguridad, que incluya tareas como tests unitarios de seguridad en componentes importantes, escaneos simples de vulnerabilidades, y tests de vulnerabilidades conocidas sobre componentes utilizados. En cuanto al nivel más alto, se espera un entendimiento avanzado de prácticas de seguridad a escala que incluya, por ejemplo, firma de artefactos, ejecución de modelados de amenazas avanzados, testing de alta intensidad, entre otros.

4.3.3. DevSecOps Guide

La *General Services Administration* (GSA)[2], una agencia independiente de los Estados Unidos que facilita y ayuda en gestión y funcionamiento de agencias federales, define una guía de DevSecOps[Adm]. El objetivo de dicha guía es describir a alto nivel las expectativas, alcance de responsabilidades, modelo de madurez y métricas asociadas a las plataformas de DevSecOps de acuerdo a los requerimientos de la GSA.

Esta guía define múltiples dominios de responsabilidad, los cuales se componen de una descripción, un modelo de madurez, un conjunto de métricas y una lista de artefactos esperados. Los dominios definidos son los siguientes:

- Dominio global
- Gestión de imágenes
- Logging, monitoreo y sistema de alertas
- Gestión de parches

[2]https://www.gsa.gov/

- Gobernanza de plataforma
- Gestión de cambios
- Desarrollo de la aplicación, testing y operaciones
- Despliegue de la aplicación
- Cuentas de usuario, privilegios, credenciales y gestión de secretos
- Disponibilidad y gestión de performance
- Gestión de redes
- Autorización para operar procesos (ATO)
- Gestión de ciclo de vida de backups y datos
- Gestión de acuerdos y finanzas

4.3.4. Otros modelos

La comunidad DevSecOps Days se encuentra desarrollando un modelo de madurez de DevSecOps[Day] que permita medir el progreso de implementación de iniciativas. Dicho modelo tendrá como objetivo definir etapas de madurez de DevSecOps en contextos empresariales.

A partir del análisis de éstos modelos, tomamos las siguientes conclusiones. El modelo SDOMM de Christian Schneider se enfoca principalmente en la integración de herramientas, evaluando la profundidad e intensidad de las pruebas realizadas, así como la consolidación de los resultados obtenidos. En cuanto al modelo DSOMM, consideramos que el mismo tiene un enfoque práctico y define múltiples medidas de forma clara y concisa. Finalmente, entendemos que el modelo propuesto por la GSA tiene un enfoque formal, y hace un fuerte énfasis en los procesos aplicados en múltiples aspectos de las organizaciones. Debido a su complejidad, consideramos que su implementación puede resultar en un importante overhead para las organizaciones.

A continuación presentamos un caso de estudio. El mismo se basa en una organización en un nivel de madurez básico de DevOps, de acuerdo a los modelos presentados. A su vez, se muestra cómo son integradas algunas de las prácticas básicas de DevSecOps propuestas en el modelo DSOMM.

Capítulo 5

Caso de estudio

En este capítulo se presenta un caso de estudio en el cual se implementan prácticas y herramientas mencionadas en el Capítulo 3. Para ello se describe un escenario que busca representar un proyecto de desarrollo de software real, detallando las metodologías y tecnologías utilizadas en el mismo. Posteriormente, se diseña y especifica la implementación de un application security pipeline sobre dicho proyecto. Se describen las fases definidas, así como la interacción entre las mismas, y las herramientas utilizadas. Por último, se toman métricas acerca de la ejecución del pipeline y se analizan los resultados obtenidos.

El caso de estudio cubre además prácticas de seguridad correspondientes a un nivel de entendimiento básico de acuerdo al modelo de madurez DevSecOps Maturity Model (DSOMM) descrito en la subsección 4.3.2. Dichas prácticas incluyen la definición de procesos de build y deploy, ejecución de tests unitarios y tests de seguridad, tests de componentes con vulnerabilidades inseguras, entre otras.

5.1. Descripción del proyecto

La aplicación elegida para ser estudiada es un micro blog llamado Flaskr[1]. La misma ofrece un sistema de login, y permite que sus usuarios agreguen entradas en un blog, como se ve en la Figura 5.1.

Flaskr está disponible para su descarga en Internet, y es utilizada como tutorial para enseñar a desarrolladores a programar utilizando el framework de desarrollo web Flask[2]. Dicho framework, escrito en el lenguaje de programación Python[3], permite desarrollar aplicaciones en ese mismo lenguaje.

5.2. Gestión de código y seguimiento de issues

El código de la aplicación es gestionado utilizando el sistema de control de versionado git, y el mismo es almacenado en un repositorio centralizado utilizando la herramienta GitLab[4].

[1] http://flask.pocoo.org/docs/0.12/tutorial/introduction/
[2] http://flask.pocoo.org/
[3] https://www.python.org/doc/
[4] https://about.gitlab.com/

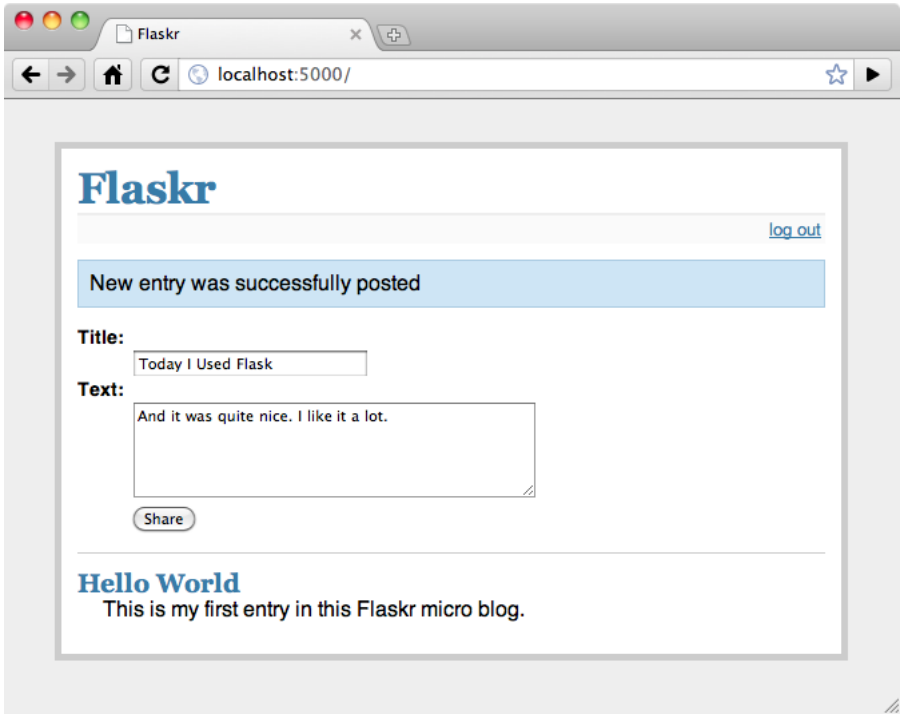

Figura 5.1: Usuario en Flaskr agregando entrada en blog

En cuanto a los requerimientos del proyecto, cada uno de ellos es también almacenado en GitLab en forma de *issues*, y se aplica el flujo conocido como *git feature branch*[Atl] para trabajar sobre los mismos. Dicho flujo de trabajo define que el código utilizado en producción es almacenado en la rama *master*, mientras que cada issue es desarrollado en una rama independiente al resto, conocida como *feature branch*. Una vez que un desarrollador considera que el código de dicha rama alcanza el estado de "Listo", debe realizar un *pull request*, es decir, un pedido para agregar su trabajo en la rama master. Dicho pull request debe ser revisado manualmente por otros desarrolladores, e incluso por personal de seguridad, con el objetivo de identificar potenciales problemas.

5.3. Arquitectura

Los sistemas de producción se encuentran alojados en el proveedor de cloud computing Google Cloud Platform (GCP)[5], haciendo uso de 2 de sus servicios: Google Compute Engine (GCE) y Google Container Registry (GCR).

- GCE es un servicio que provee instancias virtuales de servidores donde la aplicación es ejecutada.
- GCR es un servicio que permite gestionar imágenes de Docker[6]. Docker es una plataforma que permite utilizar contenedores para el despliegue

[5]https://cloud.google.com/
[6]https://www.docker.com/why-docker

de la aplicación. Esta tecnología permite realizar virtualización a nivel de sistema operativo, también conocida como contenedorización.

En cuanto a la persistencia de datos en el sistema, tanto la información de los usuarios como las entradas en el blog, son almacenadas en el sistema de gestión de base de datos relacional SQLite[7].

5.4. Integración y despliegue continuos

El proyecto implementa prácticas de integración y despliegue continuos, haciendo uso de la plataforma GitLab. Esto permite automatizar tareas como la creación del build de la aplicación, la ejecución de tests unitarios y tests de integración, el despliegue de la aplicación en distintos ambientes, entre otras. Para ello, GitLab permite agrupar dichas tareas en múltiples etapas a ser ejecutadas (*stages*, en inglés), conformando un pipeline de integración y despliegue continuos. Las mismas deben ser definidas en un archivo de configuración llamado $.gitlab-ci.yml$ que debe ser almacenado en el directorio raíz del repositorio de código[Git]. Las etapas definidas en este caso de estudio son detalladas en el listado 5.1.

```
stages:
 - build
 - test
 - security-pre
 - deploy
 - security-post
 - cleanup
```

Listado de código 5.1: Lista de etapas definidas en pipeline

A continuación se resume en qué consiste cada una de ellas:

- build: etapa en la que se prepara la aplicación a ser ejecutada.
- test: etapa en la cual se ejecutan los tests unitarios.
- security-pre: etapa que se compone de múltiples pruebas de seguridad que se ejecutan previo al despliegue de la aplicación.
- deploy: etapa encargada de desplegar la aplicación.
- security-post: etapa en la que se realizan pruebas de seguridad sobre la aplicación y el sistema en el que la misma es ejecutada, una vez que la misma fue desplegada.
- cleanup: etapa encargada de ejecutar tareas de limpieza.

[7] https://sqlite.org/index.html

Dichas etapas serán ejecutadas automáticamente cada vez que un desarrollador realiza un cambio en el código (*commit*) y que sube cambios al repositorio (*push*). A su vez, es importante destacar que por defecto cada una de estas etapas es ejecutada de manera serial en el orden en que están definidas, y que las mismas únicamente son ejecutadas si la etapa anterior finaliza exitosamente.

Otro factor a considerar es que la ejecución de tareas no es realizada por la instancia de GitLab, sino que las mismas son delegadas a otra instancia específica para ello conocida como GitLab *runner*[8]. Ambas ejecutan en máquinas virtuales del tipo "n1-standard-2" provistas por el servicio Google Cloud Engine, lo que significa que cada una de ellas cuenta con 2 procesadores virtuales y 7.5 GB de memoria RAM.

5.4.1. Build

Como se mencionaba anteriormente, la etapa de build consiste en preparar el ambiente en que la aplicación es ejecutada. El listado 5.2 muestra una tarea llamada "build" (línea 1) correspondiente a la etapa de mismo nombre (línea 2), ambas definidas en el archivo .gitlab-ci.yml.

```
1 build:
2   stage: build
3   script:
4   - docker build -t gcr.io/crucial-cabinet-\
5     222523/flaskr:$CI_COMMIT_SHA -f Dockerfile .
```

Listado de código 5.2: Etapa de build

La línea 3 da comienzo a los comandos a ser ejecutados como parte de la tarea. Como se ve en dicho comando, la tecnología utilizada es Docker. La misma permite ejecutar aplicaciones basadas en contenedores. Para generar dicho contenedor es necesario definir un archivo llamado Dockerfile[9] que incluye una especificación del ambiente.

El listado 5.3 muestra el archivo Dockerfile utilizado. El mismo toma como base una imagen del sistema operativo Debian (línea 1), y define una dirección de mail mediante una etiqueta (LABEL) de un encargado de la imagen a generar. Posteriormente define variables de entorno necesarias para la ejecución del servidor web (líneas 3 a 6), y crea un usuario con permisos limitados que ejecutará la aplicación (línea 7). Luego instala paquetes necesarios a nivel de sistema operativo mediante el comando apt, y copia el código fuente de la aplicación a la imagen. Finalmente, se instalan las dependencias del proyecto y se indica a Docker cuál será el script de arranque de la imagen (mediante el comando CMD en la línea 24), así como se define que el puerto 5000 deberá poder ser accesible (línea 25).

A su vez, durante esta etapa se implementan principios de seguridad[HL01]. Se reduce la superficie de ataque al instalar únicamente el software necesario

[8] https://docas.gitlab.com/runner/
[9] https://docs.docker.com/engine/reference/builder/

```
1  FROM debian
2  LABEL maintainer="g.gabarrin@gmail.com"
3  ENV FLASK_APP=flaskr
4  ENV FLASK_ENV=production
5  ENV LC_ALL=C.UTF-8
6  ENV LANG=C.UTF-8
7  RUN addgroup --gid 10001 app \
8      && adduser --gid 10001 -u 10001 --disabled-password \
9      --gecos "" --home /app --shell /sbin/nologin app
10 RUN apt update && apt install --no-install-recommends -y \
11     python3-pip \
12     python3-setuptools \
13     python3-dev \
14     python3-pytest \
15     curl \
16     net-tools
17     && rm -rf /var/lib/apt/lists/*
18 RUN mkdir -p /app/flaskr
19 COPY . /app/flaskr
20 WORKDIR "/app/flaskr"
21 RUN pip3 install -e . \
22     && pip3 install -r requirements.txt
23 USER app
24 CMD ./run_flask.sh
25 EXPOSE 5000
```

Listado de código 5.3: Dockerfile utilizado

para la ejecución del sistema. También se implementa el principio de uso de menores privilegios, debido a que se ejecuta la aplicación con una cuenta de usuario limitada ("app"), que no provee una shell al momento de login y que no tiene contraseña.

5.4.2. Test

La tarea de test, correspondiente a la etapa de mismo nombre, consiste en la ejecución de tests unitarios definidos en el proyecto. Para ello se hace uso de la tecnología pytest[10], una plataforma que permite escribir tests unitarios en aplicaciones desarrolladas en Python.

Como se ve en el listado 5.4, el script de esta tarea consiste en ejecutar el comando "pytest-3" en el contenedor Docker creado anteriormente en la etapa de build.

El listado 5.5 muestra un caso de test de ejemplo en el que un usuario logueado en la aplicación es capaz que crear una nueva entrada de blog en el sistema.

[10] https://pytest.org/

```
test:
  stage: test
  script:
  - docker run -p 4444:5000 \
    gcr.io/crucial-cabinet-222523/flaskr:\
    $CI_COMMIT_SHA pytest-3
```

Listado de código 5.4: Etapa de testing

```
def test_create(client, auth, app):
    auth.login()
    assert client.get('/create').status_code == 200
    client.post('/create',
        data={'title': 'created', 'body': ''}
    )

    with app.app_context():
        db = get_db()
        count = db.execute(
            'SELECT COUNT(id) FROM post'
        ).fetchone()[0]
        assert count == 2
```

Listado de código 5.5: Ejemplo de test unitario

Esta etapa de testing también incluye tests unitarios orientados a seguridad, que se explican posteriormente en el listado 5.8.

5.4.3. Deploy

Esta etapa es la encargada de desplegar el contenedor de Docker en un ambiente idéntico a producción. Como se ve en el listado 5.6, esta tarea se encarga de subir el contenedor al repositorio de imágenes de Google Cloud Registry (GCR). Posteriormente se crea un servidor virtual a través del servicio Google Compute Engine (GCE), y se le indica a dicho servidor que ejecute el contenedor subido. Al finalizar dichas acciones, se obtiene la dirección IP en que el servidor se encuentra ejecutando, y mediante el comando *curl* se comprueba que la aplicación Flaskr es accesible.

5.4.4. Cleanup

Durante esta etapa se realizan tareas de limpieza. Como se ve en el listado 5.7, en la misma se ejecutan comandos para eliminar la instancia de Google Compute Engine en que la aplicación es ejecutada.

```
deploy:
  stage: deploy
  script:
  - echo $GOOGLE_APPLICATION_CREDENTIALS | \
    docker login -u _json_key --password-stdin \
    https://gcr.io
  - docker push gcr.io/crucial-cabinet-222523/\
    flaskr:$CI_COMMIT_SHA
  - gcloud compute instances create-with-\
    container flaskr-vm-$CI_COMMIT_SHA \
    --container-image gcr.io/crucial-cabinet-\
    222523/flaskr:$CI_COMMIT_SHA \
    --project crucial-cabinet-222523 \
    --machine-type=g1-small --zone=us-east1-b
  - IP_ADDRESS=$(gcloud --format="value("\
    "networkInterfaces[0].accessConfigs[0]"\
    ".natIP)" compute instances list --filter= \
    "name=('flaskr-vm-$CI_COMMIT_SHA')")
  - echo "External IP:" $IP_ADDRESS
  - echo "Removing previous IP"
  - sed 's/.*www.myflaskrinstance.com.*//' \
    /etc/hosts
  - echo "Adding new IP"
  - echo "$IP_ADDRESS  www.myflaskrinstance.com" \
    > /etc/hosts
  - sleep 60
  - curl -k -v --silent \
    https://www.myflaskrinstance.com:5000 \
    2>&1 | grep "Flaskr"
```

Listado de código 5.6: Etapa de deploy

5.5. Diseño e implementación de AppSec Pipeline

A continuación se presentarán las herramientas utilizadas como parte de un application security pipeline sugerido. A su vez, se define la siguiente regla: de encontrarse cualquier tipo de falla en cualquiera de las fases, se considera que la ejecución del pipeline en su totalidad fue fallida. Queda por fuera de esta caso de estudio la normalización y consolidación de resultados, así como la implementación de herramientas de logging y monitoreo.

En cuanto a la selección de herramientas, existen múltiples factores a evaluar a la hora de elegir qué herramientas se van a integrar a un application security pipeline. Dichos factores pueden incluir:

- Facilidad de instalación
- Facilidad de configuración y customización

```
cleanup:
  stage: cleanup
  script:
  - gcloud compute instances delete flaskr-vm-
    $CI_COMMIT_SHA \
    --zone=us-east1-b --quiet
  when: always
```

Listado de código 5.7: Etapa de limpieza

- Tiempos de ejecución
- Capacidad de ejecutar herramientas desde línea de comandos o en modo *headless* (sin interfaz gráfica).
- Formato de salida de resultados y reportes
- Mantenimiento de herramientas
- Requerimientos de hardware necesario

Por otra parte, es conveniente que las herramientas utilizadas provean API que permitan automatizar su uso, y que puedan ser ejecutadas como servicios o en modo headless. Todas las herramientas utilizadas en este caso de estudio son *open source*.

5.5.1. Tests unitarios de seguridad

A su vez, se incorporan tests unitarios de seguridad como parte del pipeline. Los mismos permiten probar que funcionalidades sensibles para la seguridad del sistema se comporten de manera esperada. Dichas funcionalidades pueden incluir procesos de login, validación de entradas, control de acceso sobre determinados recursos, entre otras. En el listado 5.8 se ve un test de ejemplo sobre la funcionalidad de login, en el que se valida que no es posible ingresar al sistema con un usuario o contraseña inválidos.

```
@pytest.mark.parametrize(
    ('username', 'password', 'message'), (
    ('a', 'test', b'Incorrect username.'),
    ('test', 'a', b'Incorrect password.'),
))
def test_login_validate_input(auth, username,
      password, message):
    response = auth.login(username, password)
    assert message in response.data
```

Listado de código 5.8: Ejemplo de test unitario de seguridad

5.5.2. Gestión de dependencias

Actualmente existen múltiples proyectos que permiten descargar y gestionar paquetes desarrollados en Python. Entre ellos se encuentran pip[11], un herramienta de línea de comandos para instalar paquetes, y setuptools[12], que permite realizar builds, descargar, instalar y gestionar paquetes de manera automatizada. En este caso de estudio se optó por utilizar pip, herramienta sugerida por Python en su documentación oficial.

Pip permite definir un archivo de requerimientos que contiene una lista de todas las dependencias del software. Dicha lista, a su vez, puede incluir la versión específica necesaria de cada dependencia. La aplicación Flaskr define el archivo *requirements.txt*, como se ve en el listado 5.9, donde se incluye el paquete flask (un microframework de desarrollo web), junto a paquetes como Click, itsdangerous, Jinja2 (utilizado para el manejo de templates), MarkupSafe y Werkzeug. A su vez, se incluyen pytest y coverage (ambas librerías utilizadas para la ejecución de tests unitarios), y pyOpenssl (utilizada para desplegar la aplicación utilizando SSL/TLS).

```
asn1crypto==0.24.0
atomicwrites==1.3.0
attrs==18.2.0
cffi==1.12.0
Click==7.0
coverage==4.5.2
cryptography==2.5
flask==1.0.2
itsdangerous==1.1.0
Jinja2==2.10
MarkupSafe==1.1.0
more-itertools==6.0.0
pathlib2==2.3.3
pluggy==0.8.1
py==1.7.0
pytest==4.2.0
pycparser==2.19
pyOpenSSL==19.0.0
pytest==4.2.0
six==1.12.0
Werkzeug==0.14.1
```

Listado de código 5.9: Archivo de requerimientos requirements.txt

Por otra parte, se elige safety[13] como herramienta para la detección de dependencias con vulnerabilidades conocidas. Para ello safety mantiene una base de datos de vulnerabilidades que es actualizada por su proveedor mensualmente, y

[11] https://pypi.org/
[12] https://setuptools.readthedocs.io/en/latest/
[13] https://pyup.io/safety/

se ejecuta directamente sobre el archivo *requirements.txt* definido anteriormente. En caso de que algún problema sea identificado, el test se considera fallido, y por tanto, el pipeline en su totalidad también.

```
dependency-safety:
  stage: security-pre
  script:
  - safety check -r requirements.txt
```

Listado de código 5.10: Etapa de análisis de dependencias

5.5.3. Ejecución de herramientas SAST

A su vez, se integra la herramienta bandit[14]. La misma permite identificar problemas de seguridad introducidos en código Python. Para ello procesa los archivos de código fuente, construye un árbol AST a partir de los mismos, y ejecuta múltiples plugins sobre los nodos de dicho árbol. Esto permite identificar problemas como potenciales inyecciones de comandos, inyecciones a nivel de *templates* (conocidos como *Server − Side Template Injection*, o SSTI), inyecciones SQL, uso de funciones de hashing consideradas débiles, entre otros. A su vez, permite exportar los resultados del escaneo en múltiples formatos, e incluso formatos personalizados, que luego pueden ser parseados por otras herramientas.

Como se ve en el listado 5.11 en este caso bandit es ejecutado sobre el código fuente de Flaskr, y se le indica a la herramienta que ignore el directorio que almacena los tests unitarios. También se indica que los resultados sean desplegados en formato JSON.

```
sast-bandit:
  stage: security-pre
  script:
  - bandit -r . -x tests -f json -s B108
```

Listado de código 5.11: Etapa de análisis SAST

5.5.4. Gestión de configuración

Gauntlt[15] es un framework que permite definir tests de seguridad. Para ello, se escriben tests denominados *attacks* (ataques, en español) en los que se especifican comandos a ejecutarse, y se definen reglas que permiten considerar el test como fallido o pasado. A continuación se detallan múltiples herramientas que integramos con Gauntlt.

[14] https://bandit.readthedocs.io/en/latest/
[15] http://gauntlt.org/

Nmap[16] es una herramienta open source que permite realizar descubrimientos de redes y auditorías de seguridad. Permite identificar sistemas, así como puertos abiertos y servicios expuestos en redes. En el listado 5.12 se ve como es posible integrar nmap al definir un ataque en Gauntlt. Para ello se describen escenarios en los cuales se ejecutan comandos y se definen reglas sobre su salida para determinar si los tests son pasados o no. En este caso, el primer escenario comprueba que el puerto 5000 en el que ejecuta la aplicación se encuentre abierto, y el segundo comprueba que los puertos 80 y 443 se encuentren cerrados.

```
@slow

Feature: nmap attacks
  Background:
    Given "nmap" is installed
    And the following profile:
      | name        | value                      |
      | host        | www.myflaskrinstance.com   |
      | port        | 5000                       |

  Scenario: Verify server is open in port 5000
    When I launch an "nmap" attack with:
      """
      nmap -Pn -p <port> <host>
      """

    Then the output should contain:
      """
      5000/tcp open
      """

  Scenario: Verify that there are no unexpected
     ports open
    When I launch an "nmap" attack with:
      """
      nmap -Pn -F <host>
      """

    Then the output should not contain:
      """
      80/tcp
      443/tcp
      """
```

Listado de código 5.12: Ejemplo de Gauntlt attack

En el listado 5.13 se ve como es definida esta etapa utilizando el comando

[16] https://nmap.org/

gauntlt e indicando el ataque a ser ejecutado.

```
networking-nmap:
  stage: security-post
  script:
  - gauntlt security/nmap.attack
```

Listado de código 5.13: Etapa de escaneo de puertos con Nmap

También se integra la herramienta sslyze[17], un scanner que permite evaluar que la configuración de un servidor SSL/TLS al conectarse al mismo. El listado 5.14 permite ver como es definida esta tarea. El ataque en el archivo 'security/sslyze.attack' comprueba que el servidor acepte únicamente conexiones TLS, y que los protocolos SSLv2 y SSLv3 se encuentren deshabilitados.

```
networking-sslyze:
  stage: security-post
  script:
  - gauntlt security/sslyze.attack
```

Listado de código 5.14: Etapa de análisis de configuración de SSL/TLS

Nikto[18] es un scanner open source de vulnerabilidades en servidores web. Permite identificar problemas como componentes desactualizados, archivos sensibles, encabezados inusuales, potenciales métodos HTTP inseguros, entre otros. En el listado 5.15 se ve la tarea "vulnerability-scanner-nikto" en la que se ejecuta un ataque de guantlt. Dicho ataque fue configurado para comprobar que nikto no identifique componentes desactualizados y que los métodos HTTP DELETE y PUT se encuentren deshabilitados.

```
vulnerability-scanner-nikto:
  stage: security-post
  script:
  - gauntlt security/nikto.attack
```

Listado de código 5.15: Etapa de análisis con scanner Nikto

[17] https://github.com/nbla-c0d3/sslyze
[18] https://cirt.net/Nikto2

5.5.5. Ejecución de herramientas DAST

Zed Attack Proxy (ZAP)[19] es una herramienta gratuita y open source que permite identificar vulnerabilidades en aplicaciones web. Es mantenida por OWASP y cuenta con la contribución activa de voluntarios internacionales.

Ofrece múltiples modos de uso, siendo flexible y extensible. Uno de ellos, es conocido como "Man-in-the-middle proxy" en el cual la herramienta es configurada de forma tal que intercepta e inspecciona todos los mensajes intercambiados en el navegador del usuario y la aplicación web a ser evaluada. Sin embargo, dicho modo es adecuado para pruebas de tipo manual. Por otro lado, es posible configurar ZAP para que de manera autónoma realice ataques automatizados sobre aplicaciones. En este modo procede a navegar el sitio con una funcionalidad conocida como *spider*, y a escanear de manera pasiva cada página que identifica. Luego hace uso de su scanner activo con el objetivo de detectar vulnerabilidades sobre las páginas identificadas previamente.

Los resultados de las pruebas realizadas son mostrados en forma de alertas. Las mismas ofrecen información detallada de los problemas identificados, incluyendo un listado de los recursos vulnerables, una explicación de cada problema y posibles soluciones para remediarlos[OWA17]. También incluyen una prioridad, que puede tomar los valores de informacional, baja, media y alta. ZAP permite exportar estos resultados en formatos XML y JSON, que podrán ser parseados posteriormente por otras herramientas.

Para integrar esta herramienta en el pipeline se desarrolló un script en Python que consume una API provista por ZAP. Dicho script le indica a ZAP qué aplicación será escaneada, inicia el escaneo conocido como spider mencionado anteriormente, y por último dispara un escaneo activo. Una vez que el mismo finaliza se extraen los resultados, y a partir de los mismos se determina si la tarea debe ser considerada pasada o fallida. En el listado 5.16 se ve como es definida la tarea encargada de ejecutar este script. El ataque de gauntlt considerará que el test es fallido si se identifican vulnerabilidades de prioridad alta.

```
vulnerability-scanner-zap:
  stage: security-post
  script:
  - gauntlt security/zap.attack
```

Listado de código 5.16: Etapa de análisis DAST

5.6. Métricas y resultados

Para evaluar el desempeño del application security pipeline se ejecutó el pipeline en 100 ocasiones y se comprobó el tiempo de ejecución de cada una de sus etapas. En la tabla 5.1 se pueden ver el tiempo más rápido y más lento obtenido en cada tarea, así como el tiempo de la ejecución completa más rápida

[19] https://www.owasp.org/index.php/ZAP

Tarea	Etapa	Mejor tiempo	Peor tiempo
build	build	00:15,40	00:19,80
test	test	00:03,20	00:03,80
dependency-safety	security-pre	00:00,70	00:02,50
sast-bandit	security-pre	00:00,70	00:01,30
deploy	deploy	01:20,50	01:57,60
networking-nmap	security-post	00:02,50	00:06,50
networking-sslyze	security-post	00:01,10	00:01,50
vulnerability-scanner-nikto	security-post	01:35,60	01:41,70
vulnerability-scanner-zap	security-post	00:46,30	01:02,20
cleanup	cleanup	00:43,70	01:35,80
Tiempo total	-	05:07,10	06:13,50

Cuadro 5.1: Tiempos más rápidos y más lentos de ejecución de tareas del pipeline

y más lenta. Todos los tiempos mencionados a continuación son expresados en minutos, excepto que se aclare lo contrario.

En la figura 5.2 se ve un gráfico con los 100 tiempos totales de ejecución expresados en segundos.

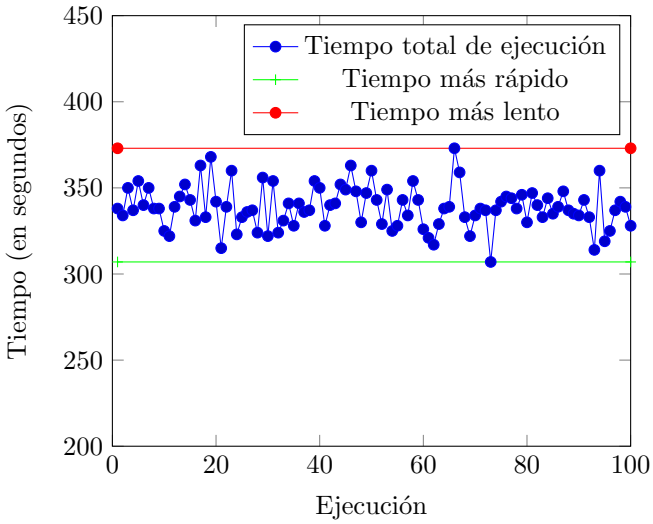

Figura 5.2: Gráfica de tiempos de ejecución de pipeline

Analizando estos resultados se observó que sobre un total de 100 ejecuciones, el tiempo total promedio de ejecución fue de 05:39,22. También se observó que el pipeline más rápido tuvo una duración de 05:07,10, mientras que el más lento tomó 06:13,50.

Asimismo, se calculó el tiempo de ejecución promedio de cada una de las tareas de manera individual, con el objetivo de entender qué tareas fueron ejecutadas más rápida y lentamente. Los resultados obtenidos se pueden ver en la tabla 5.2. A partir de los mismos se detectó que el análisis de vulnerabilidades en dependencias y de análisis estático de código fueron las más rápidas, ambas promediando 00:00,8.

En cuanto a las tareas más lentas, las mismas fueron el análisis de vulnerabilidades utilizando la herramienta nikto, con una duración de 01:04,2 y el deploy, que finalizó en 01:33,7. Sin embargo, entendemos que en tareas como la ejecución de tests, el análisis estático de código (SAST), y el análisis dinámico (mediante el uso de la herramienta ZAP) el tiempo de ejecución podría aumentar si se agregaran nuevas funcionalidades a la aplicación, y por tanto, hubiese un aumento en las líneas de código. Específicamente consideramos que la tarea que se vería más afectada sería el análisis dinámico.

A su vez, es relevante considerar que tanto la aplicación como los instancias de GitLab y GitLab Runner fueron desplegadas en Google Cloud Platform. Esto permite reducir tiempos de operaciones de red, como el push de imágenes de Docker al servicio Google Container Registry (GCR), y todas las tareas de la etapa security-post (ejecución de herramientas nmap, sslyze, nikto y ZAP).

Tarea	**Etapa**	**Tiempo promedio**
build	build	00:03,4
test	test	00:16,3
dependency-safety	security-pre	00:00,8
sast-bandit	security-pre	00:00,8
deploy	deploy	01:33,7
networking-nmap	security-post	00:03,6
networking-sslyze	security-post	00:01,2
vulnerability-scanner-nikto	security-post	01:40,2
vulnerability-scanner-zap	security-post	00:54,1
cleanup	cleanup	01:05,1
Tiempo total	-	**05:39,2**

Cuadro 5.2: Tiempos promedio de ejecución de tareas del pipeline

Capítulo 6

Trabajo relacionado

Múltiples trabajos han estudiado como integrar actividades de seguridad en metodologías ágiles, incluyendo los requerimientos ([Pee05; BWB+06; AGI11]) y la gestión de riesgos ([VS10]. A su vez, varios autores han presentado estudios sobre testing aplicado seguridad ([TBM+05; Kon07]) y diversas herramientas que pueden ser utilizadas.

La investigación de Ahmed Alnatheer ([Aln14]) busca identificar los principales problemas relacionados a seguridad que afectan a los procesos de desarrollo ágiles. A su vez, estudia cómo evaluar la cohesión y completitud de las soluciones predominantes a dichos problemas. Entre dichas soluciones destaca el desarrollo de software teniendo la seguridad en consideración y el uso de controles de seguridad. El autor también hace énfasis en la inclusión de un ingeniero de seguridad o de desarrolladores experimentados en los equipos de desarrollo, y la modificación de procesos de documentación y análisis de riesgos.

Andreas Broström ([Bro15]) propone cómo y dónde la seguridad puede ser integrada en procesos de desarrollo ágiles de aplicaciones web. Para ello se enfoca en los procesos de integración continua, y centra su estudio en herramientas de análisis estático de código (SAST) y de análisis dinámico (DAST). Durante su investigación configura ambientes con aplicaciones deliberadamente vulnerables y ejecuta las herramientas sobre los mismos, y posteriormente compara los resultados obtenidos por cada una de ellos.

Asimismo, Kuusela et al. ([Kuu+17]) analiza herramientas y técnicas de seguridad, y evalúa la factibilidad de su incorporación en procesos de integración continua. Su investigación consiste en la ejecución de herramientas similares a las planteadas por Broström, sumadas a otras de gestión de configuración, verificación de seguridad y análisis de dependencias. Para ello despliega distintas aplicaciones web basadas en diferentes frameworks y lenguajes (Ruby on Rails[1], Java[2] y Scala[3]), determina qué herramientas son más indicadas para analizar cada aplicación, y estudia los resultados obtenidos de su ejecución. Gupta et. al. ([GBMB10] y Daud et. al. ([DBH14]) también realizaron sus trabajos comparando herramientas con un enfoque similar a Kuusela et. al.

Los trabajos de Dave Shackleford[Sha12] y Francois Raynaud [Ray17] plantean los beneficios de impulsar procesos de DevSecOps en las organizaciones,

[1] https://rubyonrails.org/
[2] https://go.java/index.html
[3] https://scala-lang.org/

promoviendo el trabajo colaborativo entre desarrolladores, y personal de operaciones y seguridad. Ambos presentan múltiples áreas y prácticas que pueden implementarse en este contexto.

Düllmann et. al. [DPH18] y Bass et. al. ([BHR+15]) plantean la importancia de proteger los pipelines de integración y despliegues continuos. Ambos autores coinciden en que dichos pipelines son críticos para las organizaciones, y presentan estrategias para detectar, diagnosticar y resolver problemas de seguridad sobre ellos. Particularmente, Bass et. al. describe cómo se diseñó un pipeline involucrando las tecnologías Chef, Jenkins, Docker, GitHub y AWS, y detalla el proceso que se siguió para hardenizarlo.

Capítulo 7

Conclusiones

Durante la realización de esta tesis hemos estudiado las principales características de los procesos de desarrollo de metodologías tradicionales. Presentamos los problemas que los afectan, y las razones que impulsaron el surgimiento de las metodologías ágiles de desarrollo. A su vez, observamos que las organizaciones tienen la necesidad de desplegar cambios en sus sistemas de manera continua mediante procesos confiables y repetibles, y que las metodologías ágiles resultaban adecuadas para lidiar con esas necesidades.

Sin embargo, vimos que tanto en procesos gestionados con metodologías tradicionales como ágiles, resulta imprescindible contemplar la seguridad en todas las etapas del ciclo de vida de desarrollo, así como en todas las herramientas y procesos involucrados en el mismo. Entendemos que esto presenta múltiples desafíos, tanto a nivel organizacional como técnico. Los primeros refieren a alcanzar el compromiso de la organización respecto a considerar aspectos de seguridad en sus acciones en el día a día, y que cada persona se sienta responsable de la misma. En cuanto a los desafíos técnicos, incluyen la integración de nuevas prácticas, herramientas y roles, el uso de diferentes tecnologías y la necesidad de incluir componentes de terceros, entre otros. Estos factores derivan en un aumento en la complejidad, tanto de los procesos como de los sistemas.

Hemos estudiado múltiples herramientas y prácticas de seguridad que permiten sobrellevar estas dificultades, desde herramientas para el diseño y gestión de requerimientos, prácticas enfocadas a la seguridad del código fuente, estrategias de testing, y gestión de riesgos y vulnerabilidades, entre otras. Sin embargo, consideramos que no hay una única solución para resolver estos problemas. Factores como la heterogeneidad de tecnologías utilizadas, las necesidades y objetivos propios de cada organización, y la cultura organizacional llevan a que sea necesario desarrollar soluciones diseñadas a medida en cada organización. Por ello todas las partes involucradas deben ser responsables de la seguridad del sistema, y la misma debe ser contemplada desde la concepción del proyecto. Esto permite integrar la seguridad en múltiples capas, y de manera iterativa e incremental.

Por otra parte, hemos visto que surgen en las organizaciones cambios culturales como DevOps, que buscan integrar equipos de desarrollo y operaciones, para resolver estas dificultades de una manera colaborativa. Para ello se toma el código fuente como lenguaje en común entre dichos equipos, y se comparten conocimientos y responsabilidades sobre las tareas. Sumado a ello, vimos el

surgimiento de prácticas como integración y despliegue continuos para responder a necesidades de despliegue de cambios en producción de manera continua. Para ello se diseña un pipeline que consiste en la ejecución automatizada de distintas tareas, desde el build de la aplicación, la ejecución de tests, hasta la orquestación de la infraestructura y la puesta en producción. Estas prácticas pasaron a ser imprescindibles en múltiples organizaciones, que también acompañaron dicho cambio con prácticas que promueven la gestión de infraestructura como código, y que muchas veces adoptan los beneficios de la integración de tecnologías de cloud computing. Sin embargo, vimos que la implementación de estas prácticas y de DevOps se tornan insuficientes para resolver problemas de seguridad, y surge la necesidad de integrar la seguridad como parte de ese cambio cultural. Esto dio surgimiento al concepto de DevSecOps, que propone la coordinación y colaboración de todas las partes: desarrolladores, operaciones y seguridad.

A su vez, consideramos que si bien estos cambios ofrecen múltiples ventajas, por si mismos no son sufcientes para resolver los problemas de seguridad de acuerdo a las necesidades de las organizaciones en la actualidad. Para que ello sea posible, vimos que surgen implementaciones de pipelines, conocidos como pipelines de seguridad, que permiten evaluar de manera automatizada la seguridad del sistema ante cualquier cambio en su código. Sin embargo, no existe un estándar a la hora de diseñarlos y las soluciones propuestas al respecto se encuentran fragmentadas. Nuestra investigación nos permitió concluir que existen múltiples variables que deben ser evaluadas, principalmente acerca de qué pruebas realizar, con qué herramientas, con qué configuración, entre otras. Por un lado, vimos que existen fases y tareas que resultan convenientes en determinados escenarios, pero podrían no aportar valor en otros. Por otro, notamos que existen múltiples herramientas a utilizar, y los productos varían desde soluciones open source o de código cerrado, como a soluciones gratuitas o comerciales. A pesar de ello, no pudimos identificar tipos de herramientas o productos específicos que fueran utilizados mayoritariamente en las implementaciones evaluadas. Entendemos a su vez, que un aspecto importante a considerar es el de mantener los tiempos de ejecución del pipeline a un mínimo, por lo que resulta necesario hacer un balance para evaluar qué etapas se consideran imprescindibles, y cuáles podrían ser opcionales, o incluso descartadas.

Asimismo, pudimos concluir que, si bien los procesos de integración y despliegues continuos y los pipelines de seguridad aportan un gran valor respecto a la seguridad del sistema, éstos también aumentan la superficie de ataque y exponen a las organizaciones a nuevos desafíos. El nivel de automatización que estos procesos ofrecen lleva a que los mismos tengan acceso a múltiples activos críticos de la organización. Esto deriva en que una configuración insegura de los sistemas de CI/CD pueda exponer a la organizacón a riesgos críticos. La gestión de información sensible como credenciales y tokens de sesión almacenados en texto plano en código fuente, de las credenciales de administración remota de servidores de producción, o de la información sensible en clientes de mensajería instantánea y chatbots, son solo algunos de los factores a tener en cuenta al implementar estos procesos. Por esta razón, entendemos que también es imprescindible contemplar la seguridad de estas herramientas a la hora de implementarlas.

Por otra parte, hemos visto que existen múltiples modelos de madurez que nos permiten evaluar el grado en que una organización implementa prácticas de DevOps y DevSecOps, e incluso que nuevos modelos ya están siendo diseñados.

Consideramos que los mismos resultan útiles a la hora de entender en qué nivel se encuentra la organización a evaluar, así como para definir métricas y procesos que permitan definir objetivos y alcanzar niveles superiores.

Respecto al caso de estudio, entendemos que una de las principales métricas a tener en cuenta refiere al tiempo de ejecución del pipeline. Concluimos que en tiempos razonables (menores a 10 minutos) es posible ejecutar pruebas básicas de seguridad en un pipeline de integración y despliegue continuos. A su vez, consideramos que otras posibles métricas pueden incluir la cantidad de problemas identificados, así como cantidad de falsos positivos. Por otra parte, puede resultar beneficioso conocer en qué aspectos del sistema se introducen problemas de seguridad con mayor frecuencia para capacitar al equipo respecto a los mismos, o conocer qué tipos de herramientas son capaces de dar resultados más valiosos.

7.1. Trabajo a futuro

Durante este estudio surgieron múltiples lineamientos de trabajo a futuro. En lo que refiere al análisis inicial de metodologías ágiles se evaluaron algunas de ellas como Scrum, Lean, Kanban y Extreme Programming. Entendemos que puede agregar valor conocer otras metodologías y prácticas ágiles más allá de las discutidas. A su vez, se evaluaron técnicas que permiten incorporar seguridad en las mismas. Sin embargo, no se detallaron procesos manuales como la evaluación de cumplimiento de normativas y una exhaustiva gestión de riesgos. Tampoco fueron considerados procesos como la ejecución de tests de penetración sobre los sistemas, ni la incorporación de un programa de recompensas por bugs, conocidos en inglés como *bug bounty programs*, en los que se involucra a terceros en la identificación de riesgos y vulnerabilidades, a cambio de reconocimiento y compensaciones monetarias.

En lo que respecta al caso de estudio consideramos que hubo múltiples puntos que pudieron haber sido evaluados con mayor profundidad. Con el objetivo de simplificar la solución planteada no se implementó la práctica de infraestructura como código. Entendemos que pudo agregar valor la integración de herramientas como Terraform[1] para la orquestación de infraestructura, así como Chef y Puppet para la gestión de configuración de los sistemas, debido a la popularidad de las mismas. Por otra parte, consideramos que pudo ser conveniente realizar una evaluación más exhaustiva de los principales proveedores de servicios de cloud computing en el mercado. El proveedor Google Cloud Platform (GCP) fue elegido frente a otros como Amazon Web Services (AWS)[2] o Microsoft Azure[3] únicamente por razones de costos, y debido a que no requería la solicitud de permisos especiales para la ejecución de herramientas de seguridad desde y hacia los servidores contratados. Consideramos que en un proyecto real también debe ser relevante evaluar otros factores como el listado de los servicios ofrecidos por ellos, la integración de los mismos, facilidad de uso, y compatibilidad con el resto de las herramientas elegidas.

En cuanto al software de integración continua utilizado durante el caso de estudio, la decisión de utilizar GitLab fue tomada únicamente debido a que

[1] https://www.terraform.io/
[2] https://aws.amazon.com/
[3] https://azure.microsoft.com/

contábamos con experiencia previa profesional con este producto. Sin embargo, existen otras plataformas reconocidas que pueden ser consideradas, como lo son Jenkins, Circle CI y Travis CI, que podrían adaptarse con mayor facilidad a la realidad de la organización en que se implanten. A su vez, consideramos relevante la arquitectura en la cual estos productos son instalados. En este caso, tanto las instancias de GitLab y GitLab Runner, como las instancias de la aplicación fueron ejecutadas en el proveedor GCP. A pesar de ello, entendemos que podría evaluarse la ejecución de todos estos componentes en infraestructura propia de la organización, o incluso en un modelo mixto. Este último podría consistir en la ejecución de GitLab y GitLab Runner en infraestructura propia, y la ejecución de la aplicación en la nube. Consideramos que los tiempos de ejecución del pipeline de seguridad dependerán directamente de este factor, afectando principalmente los tiempos de ejecución de tareas que requieran operaciones de red si se eligiera un modelo mixto.

Entendemos que también puede resultar beneficiosa la incorporación de nuevas tareas al pipeline de seguridad en múltiples áreas, buscando alcanzar también niveles de madurez superiores. Herramientas como firewalls de código, herramientas de fuzzing, soluciones de logging y monitoreo, entre otras, podrían brindar un valor agregado. Por otra parte, la mayoría de los productos utilizados a la hora de diseñar dicho pipeline en el caso de estudio fueron open source y de uso libre, por lo que se pudo haber evaluado la integración de productos comerciales.

Finalmente, consideramos que en organizaciones maduras en las que se realizan múltiples pruebas de seguridad sobre los pipelines, puede resultar imprescindible contar con un sistema que permita integrar y normalizar los resultados de las herramientas ejecutadas. Entendemos que estos sistemas deben ofrecer funcionalidades como consolidación y normalización de resultados, manejo de falsos positivos, así como creación y gestión de tickets para su posterior tratamiento.

Bibliografía

[JM] E. K. Jim Manico, *Securing the SDLC*, Disponible en: https://www.owasp.org/images/7/76/Jim_Manico_(Hamburg)_-_Securiing_the_SDLC.pdf.

[VS10] A. Vähä-Sipilä, "Product security risk management in agile product management", *Stockholm, Sweden*, 2010.

[Roy87] W. W. Royce, "Managing the Development of Large Software Systems: Concepts and Techniques", en *Proceedings of the 9th International Conference on Software Engineering*, ép. ICSE '87, IEEE Computer Society Press, 1987, págs. 328-338, ISBN: 0-89791-216-0. dirección: http://dl.acm.org/citation.cfm?id=41765.41801.

[DBH14] N. I. Daud, K. A. A. Bakar y M. S. M. Hasan, "A case study on web application vulnerability scanning tools", en *2014 Science and Information Conference*, IEEE, 2014, págs. 595-600.

[Gad59] P. O. Gaddis, *The project manager*. Harvard University Boston, 1959.

[DES14] M. L. DESPA, "Comparative study on software development methodologies", *Database Systems Journal*, vol. 5, n.º 3, 2014.

[MMS+08] C. for Medicare & Medicaid Services y col., "Selecting a development approach", *Centers for Medicare & Medicaid Services*, págs. 1-10, 2008.

[Atl] Atlassian, *Git Feature Branch Workflow | Atlassian Git Tutorial*, Disponible en: https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow.

[Git] GitLab, *Getting started with GitLab CI/CD | GitLab*, Disponible en: https://docs.gitlab.com/ee/ci/quick_start/README.html.

[MG10] N. M. A. Munassar y A Govardhan, "A comparison between five models of software engineering", *IJCSI International Journal of Computer Science Issues*, vol. 7, n.º 5, págs. 94-101, 2010.

[OWA18] OWASP, *Top 10-2017 A9-Using Components with Known Vulnerabilities*, Disponible en: https://www.owasp.org/index.php/Top_10-2017_A9-Using_Components_with_Known_Vulnerabilities, 2018.

[Awa05] M. Awad, "A comparison between agile and traditional software development methodologies", *University of Western Australia*, 2005.

[Pag] T. Pagel, *DSOMM - Information*, Disponible en: https://dsomm.timo-pagel.de/information.php.

[DPH18] T. F. Düllmann, C. Paule y A. van Hoorn, "Exploiting devops practices for dependable and secure continuous delivery pipelines", en *2018 IEEE/ACM 4th International Workshop on Rapid Continuous Software Engineering (RCoSE)*, IEEE, 2018, págs. 27-30.

[OWA17] OWASP, *OWASP ZAP 2.7 - Getting Started Guide*, Disponible en: https://github.com/zaproxy/zaproxy/releases/download/2.7.0/ZAPGettingStartedGuide-2.7.pdf, nov. de 2017.

[OWA18] ——, *OWASP AppSec Pipeline*, Disponible en: https://www.owasp.org/index.php/OWASP_AppSec_Pipeline, mar. de 2018.

[Sol17] Solinea, *The Solinea DevOps Maturity Model*, Disponible en: https://solinea.com/blog/solinea-devops-maturity-model, 2017.

[OWA17] OWASP, *Static Code Analysis*, Disponible en: https://www.owasp.org/index.php/Static_Code_Analysis/, 2017.

[Day] D. Days, *The DevSecOps Maturity Model*, Disponible en: https://www.devsecopsdays.com/resources/the-devsecops-maturity-model.

[OWA18] OWASP, *Fuzzing - OWASP*, Disponible en: https://www.owasp.org/index.php/Fuzzing, 2018.

[Boe86] B. Boehm, "A spiral model of software development and enhancement", *ACM SIGSOFT Software engineering notes*, vol. 11, n.º 4, págs. 14-24, 1986.

[Fow12] M Fowler, *Test Pyramid*, Disponible en: https://martinfowler.com/bliki/TestPyramid.html, 2012.

[Lie16] S. Lietz, *<- Shifting security to the Lef - devsecops*, Disponible en: http://www.devsecops.org/blog/2016/5/20/-security, 2016.

[Hum06] J. Humble, *What is Continuous Delivery?*, Disponible en: https://www.continuousdelivery.com/, 2006.

[OWA18] OWASP, *Transport Layer Protection Cheat Sheet*, Disponible en: https://www.owasp.org/index.php/Transport_Layer_Protection_Cheat_Sheet, 2018.

[Adm] G. S. Administration, *DevSecOps - Tech at GSA*, Disponible en: https://tech.gsa.gov/guides/dev_sec_ops_guide/.

[Cap15] Capgemini, *DevOps - The Future of Application Lifecycle Automation - 2nd Edition*, Disponible en: https://www.capgemini.com/wp-content/uploads/2017/07/devops_pov_2015-12-18_final.pdf, 2015.

[Coh09] M. Cohn, *Succeeding with Agile: Software Development Using Scrum*, ép. Addison-Wesley Signature Series (Cohn). Pearson Education, 2009, ISBN: 9780321660565.

[McG06] G. McGraw, *Software Security: Building Security in*, ép. Addison-Wesley professional computing series. Addison-Wesley, 2006, ISBN: 9780321356703.

[OWA16] OWASP, *Application Security Verification Standard 3.0.1*, Disponible en: `https://www.owasp.org/images/3/33/OWASP_Application_Security_Verification_Standard_3.0.1.pdf`, jul. de 2016.

[Tho] ThoughtWorks, *Continuous Integration | ThoughtWorks*, Disponible en: `https://www.thoughtworks.com/continuous-integration`.

[Wel03] D. Wells, *Extreme Programming: A gentle introduction (2006)*, 2003.

[Muk] J. Mukherjee, *DevSecOps: Injecting Security into CD Pipelines*, Disponible en: `https://www.atlassian.com/continuous-delivery/devsecops`.

[Sha12] D. Shackleford, *A DevOpsSec Playbook*, Disponible en: `https://www.sans.org/reading-room/whitepapers/analyst/devsecops-playbook-36792`, 2012.

[Sch95] K. Schwaber, "Scrum Development Process. OOPSLA'95 Workshop on Business Object Design and Implementation", *Austin, USA*, 1995.

[Wil16] R Wilsenach, *DevOps Culture*, Disponible en: `http://martinfowler.com/bliki/DevOpsCulture.html`, 2016.

[MCP17] H. Myrbakken y R. Colomo-Palacios, "DevSecOps: A Multivocal Literature Review", en *Software Process Improvement and Capability Determination*, A. Mas, A. Mesquida, R. V. O'Connor, T. Rout y A. Dorling, eds., Cham: Springer International Publishing, 2017, págs. 17-29, ISBN: 9783319673837.

[GBMB10] D. Gupta, J. Bau, J. Mitchell y E. Bursztein, "State of the Art: Automated Black-Box Web Application Vulnerability Testing", en *2010 IEEE Symposium on Security and Privacy(SP)*, vol. 00, mayo de 2010, págs. 332-345. DOI: 10.1109/SP.2010.27.

[Sol15] S. Solomon, *SAST vs DAST - Why SAST*, Disponible en: `https://www.checkmarx.com/2015/04/29/sast-vs-dast-why-sast-3/`, 2015.

[Deb10] P. Debois, *What is This Devops Thing, Anyway?*, Disponible en: `http://www.jedi.be/blog/2010/02/12/what-is-this-devops-thing-anyway/`, 2010.

[Bah13] P. Bahrs, *Adopting the IBM DevOps approach for continuous software delivery - IBM Developer*, Disponible en: `https://developer.ibm.com/articles/d-adoption-paths/`, 2013.

[Bec99] K. Beck, "Embracing change with extreme programming", *Computer*, vol. 32, n.º 10, págs. 70-77, 1999.

[Pee05] J. Peeters, "Agile security requirements engineering", en *Symposium on Requirements Engineering for Information Security*, 2005.

[BBSSB17] L. Bell, M. Brunton-Spall, R. Smith y J. Bird, *Agile Application Security: Enabling Security in a Continuous Delivery Pipeline*. O'Reilly Media, 2017, ISBN: 9781491938812.

[Aln14] A. Alnatheer, "The investigation of security issues in agile methodologies", Tesis doct., University of Southampton, 2014.

[All18] ——, *Agile Practices Timeline*, Disponible en: `https://www.agilealliance.org/agile101/practices-timeline/`, 2018.

[All01] ——, *Agile manifesto*, Disponible en: `http://www.agilemanifesto.org`, 2001.

[Bos17] G. Bostrom, *Crisp's Blog ■ Security Test-Driven Development - Spreading the STDD-virus*, Disponible en: `https://blog.crisp.se/2017/03/20/gustavbostrom/security-test-driven-development-spreading-the-std-virus`, 2017.

[Mac12] N. Macdonald, *DevOps Needs to Become DevOpsSec*, Disponible en: `https://blogs.gartner.com/neil_macdonald/2012/01/17/devops-needs-to-become-devopssec/`, 2012.

[KB14] G. Kumar y P. K. Bhatia, "Comparative analysis of software engineering models from traditional to modern methodologies", en *Advanced Computing & Communication Technologies (ACCT), 2014 Fourth International Conference on*, IEEE, 2014, págs. 189-196, ISBN: 9781479949106.

[BHR+15] L. Bass, R. Holz, P. Rimba, A. B. Tran y L. Zhu, "Securing a deployment pipeline", en *Proceedings of the Third International Workshop on Release Engineering*, IEEE Press, mayo de 2015, págs. 4-7. DOI: `10.1109/RELENG.2015.11`.

[Kon07] V. Kongsli, "Security Testing with Selenium", en *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, ép. OOPSLA '07, ACM, 2007, págs. 862-863, ISBN: 978-1-59593-865-7. DOI: `10.1145/1297846.1297927`. dirección: `http://doi.acm.org/10.1145/1297846.1297927`.

[TBM+05] A Tappenden, P Beatty, J Miller, A Geras y M. Smith, "Agile security testing of Web-based systems via HTTPUnit", vol. 0, ago. de 2005, págs. 29-38, ISBN: 0769524877. DOI: `10.1109/ADC.2005.11`.

[Oeh05] P. Oehlert, "Violating Assumptions with Fuzzing", *IEEE Security and Privacy*, vol. 3, n.º 2, págs. 58-62, mar. de 2005, ISSN: 1540-7993. DOI: `10.1109/MSP.2005.55`. dirección: `http://dx.doi.org/10.1109/MSP.2005.55`.

[ABDN+17] M. Artac, T. Borovssak, E. Di Nitto, M. Guerriero y D. A. Tamburri, "DevOps: introducing infrastructure-as-code", en *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*, IEEE, 2017, págs. 497-498. DOI: `10.1109/ICSE-C.2017.162`.

[Roh16] M. Rohr, *Application Security Testing*, Disponible en: `https://www.owasp.org/images/e/e9/GOD16-agiletesting.pptx`, 2016.

[Kuu+17] J. Kuusela y col., "Security testing in continuous integration processes", 2017.

[Ray17] F. Raynaud, *DevSecOps Whitepaper - The business benefits and best practices of DevSecOps implementation*, Disponible en: `https://www.devseccon.com/wp-content/uploads/2017/07/DevSecOps-whitepaper.pdf`, 2017.

[Sch15] C. Schneider, *Security DevOps - staying secure in agile projects*, Disponible en: `https://www.christian-schneider.net/slides/OWASP-AppSecEU-2015_SecDevOps.pdf`, 2015.

[Bro15] A. Broström, "Integrating Automated Security Testing in the Agile Development Process", *KTH Royal Institute of Technology, Stockholm, Sweden*, 2015.

[BWB+06] G. Boström, J. Wäyrynen, M. Bodén, K. Beznosov y P. Kruchten, "Extending XP Practices to Support Security Requirements Engineering", en *Proceedings of the 2006 International Workshop on Software Engineering for Secure Systems*, ép. SESS '06, ACM, 2006, págs. 11-18, ISBN: 1-59593-411-1. DOI: `10.1145/1137627.1137631`. dirección: `http://doi.acm.org/10.1145/1137627.1137631`.

[ATO+12] V. Asthana, I. Tarandach, N. ODonoghue, B. Sullivan y M. Saario, "Practical security stories and security tasks for agile development environments", *Online, July*, 2012.

[SS17] K. Schwaber y J. Sutherland, *The Scrum Guide (2017)*, Disponible en: `https://www.scrumguides.org/docs/scrumguide/v2017/2017-Scrum-Guide-US.pdf`, 2017.

[AGI11] Z. Azham, I. Ghani y N. Ithnin, "Security backlog in Scrum security practices", en *2011 Malaysian Conference in Software Engineering*, dic. de 2011, págs. 414-417. DOI: `10.1109/MySEC.2011.6140708`.

[HL01] M. Howard y D. Leblanc, *Writing Secure Code*. Redmond, WA, USA: Microsoft Press, 2001, ISBN: 9780735617223.

[PP03] M. Poppendieck y T. Poppendieck, *Lean Software Development: An Agile Toolkit: An Agile Toolkit*. Addison-Wesley, 2003, ISBN: 0321150783.

[FF06] M. Fowler y M. Foemmel, *Continuous integration*, Disponible en: `http://www.thoughtworks.com/ContinuousIntegration.pdf`, 2006.

[SJT08] K. Scarfone, W. Jansen y M. Tracy, "Guide to general server security", *NIST Special Publication*, vol. 800, pág. 123, 2008.

[LSA11] M. Laanti, O. Salo y P. Abrahamsson, "Agile methods rapidly replacing traditional methods at Nokia: A survey of opinions on agile transformation", *Information and Software Technology*, vol. 53, n.º 3, págs. 276-290, 2011.