UNIVERSIDAD DE LA REPÚBLICA URUGUAY

FACULTAD DE INGENIERÍA

Emulación de Dispositivos Android en un Cyber Range

Informe de Proyecto de Grado presentado por

Nicolás Vidal, Federico Correa

en cumplimiento parcial de los requerimientos para la graduación de la carrera de Ingeniería en Computación de Facultad de Ingeniería de la Universidad de la República

Supervisor

Juan Diego Campo, Rodrigo Martínez

Montevideo, Diciembre 2025

Agradecimientos

En primer lugar, queremos expresar nuestro más sincero agradecimiento a nuestros tutores, Juan Diego Campo y Rodrigo Martínez, por su invaluable guía, apoyo y paciencia durante todo el desarrollo de este proyecto. Su experiencia, conocimiento y gran disposición fueron fundamentales para orientar nuestras decisiones técnicas y mantener el rumbo del trabajo hacia un resultado satisfactorio.

Agradecemos a todos los profesores que nos formaron durante nuestra carrera, cuyas enseñanzas fueron fundamentales para nuestro crecimiento académico y profesional.

Finalmente, agradecemos a nuestras familias y seres queridos por su apoyo incondicional, comprensión y aliento durante todo este proceso. Su paciencia y apoyo fueron esenciales para superar los momentos desafiantes y celebrar los logros alcanzados.

Resumen

El Grupo de Seguridad Informática del Instituto de Computación ha desarrollado Tectonic, una plataforma tipo cyber range que permite el despliegue automatizado de laboratorios de ciberseguridad mediante Infraestructura como Código. Un cyber range es un entorno virtual seguro e interactivo diseñado para la capacitación y el entrenamiento en ciberseguridad, que replica redes, sistemas y aplicaciones en un espacio aislado. En la actualidad, Tectonic solamente soporta el despliegue de sistemas Windows o Linux.

Este proyecto consiste en dotar al cyber range de la capacidad de emular dispositivos móviles Android en los escenarios, permitiendo que una nueva gama de problemáticas de seguridad y mecanismos de protección que afectan a este tipo de sistemas puedan ser estudiadas por los participantes.

Para abordar este desafío, se realizó una investigación del estado del arte en seguridad móvil, estudiando los marcos de referencia más relevantes de la industria y sintetizando sus aportes. Como resultado, se desarrolló una taxonomía de ataques móviles propia que facilita la construcción de escenarios reproducibles y pedagógicamente valiosos. La solución implementada se basa en emulación de Android mediante virtualización anidada y se automatiza mediante Infraestructura como Código, garantizando reproducibilidad y escalabilidad dentro del ecosistema de Tectonic.

Para validar la solución, se construyó y desplegó exitosamente un laboratorio de Ejecución Remota de Código que implementa múltiples categorías de la taxonomía, demostrando la viabilidad técnica de la solución y la utilidad de la taxonomía propuesta. Los resultados incluyen una taxonomía de ataques móviles adaptada al diseño de escenarios prácticos, una solución modular y escalable para la emulación de dispositivos Android, y una implementación completamente automatizada. El proyecto entregó una solución que establece las bases para futuros desarrollos en seguridad móvil dentro de Tectonic.

Palabras clave: Android, Cyber Range, Seguridad Móvil, Emulación, Infraestructura como Código

Índice general

Capítulo 1

Introducción

1.1. Motivación

Hoy en día, casi todo lo que hacemos depende de sistemas digitales. Los usamos para comunicarnos, manejar nuestro dinero, en los servicios de salud, y para que funcionen cosas tan cotidianas como el transporte público, supermercados o aeropuertos. Si bien esto nos trae muchos beneficios, también nos hace más vulnerables a nuevos riesgos. Por eso, la ciberseguridad ya no es solo un tema técnico, sino algo fundamental para proteger el funcionamiento de nuestra sociedad y nuestro día a día.

A nivel global, faltan muchos expertos en ciberseguridad. Según un estudio de $(ISC)^2$, una de las organizaciones más importantes del sector, en 2024 hacían falta casi 4.8 millones de profesionales para cubrir los puestos necesarios (ISC2, 2024). Esto se debe a que la demanda de expertos crece mucho más rápido que la cantidad de personas formadas.

Además, no alcanza con solo estudiar la teoría. La ciberseguridad es un trabajo muy práctico que requiere saber cómo reaccionar rápido durante un ataque, cómo investigar qué pasó en un sistema y cómo usar herramientas para defenderlo. Estas son habilidades que se aprenden en la práctica, no solo leyendo, y la mejor manera de dominarlas es practicándolas una y otra vez en un ambiente seguro.

Pero esta práctica no se puede hacer en sistemas productivos. Es muy arriesgado y puede ser dudoso éticamente. Un pequeño error podría hacer que un servicio importante, como una página web o una aplicación, deje de funcionar para todos sus usuarios. Peor aún, podría causar que se filtre información privada, lo que traería problemas legales y grandes pérdidas de dinero. Por eso, es fundamental tener ambientes seguros y realistas para poder practicar sin causar daños.

Aquí es donde los cyber ranges son clave, ya que ofrecen un entorno controlado donde se pueden realizar ejercicios de ciberseguridad de forma segura. Estas plataformas simulan infraestructuras completas como redes, servidores y

aplicaciones, que se comportan de manera muy similar a los sistemas reales, pero de forma completamente aislada de la infraestructura de producción.

Dentro de un cyber range, los participantes utilizan herramientas para llevar a cabo una gran variedad de ejercicios prácticos. Por ejemplo, en un ejercicio de defensa (Blue Team), pueden aprender a monitorear una red, analizar alertas y responder a un ciberataque en tiempo real. En un escenario ofensivo (Red Team), practican cómo identificar y explotar vulnerabilidades para entender cómo piensa un atacante. También se pueden realizar investigaciones forenses para reconstruir un ataque en un sistema ya comprometido.

Sin embargo, la mayoría de estas plataformas se han enfocado en computadoras y servidores, dejando de lado a los dispositivos móviles.

Este vacío en la formación es importante porque la seguridad de los celulares tiene desafíos únicos, que no existen en los sistemas tradicionales. A diferencia de un servidor, un celular se mueve y se conecta a redes de todo tipo, tanto seguras como inseguras (por ejemplo, un Wi-Fi público), lo que crea nuevas oportunidades de ataque. Además, es común que la gente use su teléfono personal para trabajar. Cuando esto pasa, las empresas pierden el control sobre qué aplicaciones se instalan o qué configuraciones de seguridad se usan, mezclando los riesgos personales con los de la empresa.

En particular en el ecosistema móvil de Android —el sistema operativo más usado en el mundo, presente en más del 70 % de los dispositivos (StatCounter Global Stats, 2024)— todo gira en torno a las tiendas de aplicaciones, donde pueden aparecer aplicaciones maliciosas, con librerías de terceros vulnerables o que piden más permisos de los necesarios, poniendo en riesgo a los usuarios. Estas diferencias hacen que analizar y proteger un dispositivo móvil requiera habilidades específicas que no se cubren en la formación tradicional.

1.2. Planteo del problema y contexto

Existe un vacío en la formación práctica de ciberseguridad para dispositivos móviles. El desafío, por lo tanto, es cómo crear entornos de entrenamiento para este ecosistema. Abordar este problema requiere un proceso estructurado.

Primero, es necesario analizar las amenazas y características del ecosistema móvil. Luego, definir los escenarios de práctica relevantes, y a continuación, establecer los requerimientos técnicos para implementarlos. Finalmente, desarrollar la tecnología necesaria que cumpla con dichos requerimientos.

Para llevar a cabo este último paso, en lugar de construir una plataforma desde cero, el proyecto se enfoca en extender las capacidades de *Tectonic*, el cyber range desarrollado por el Grupo de Seguridad Informática (GSI) de la Facultad de Ingeniería. Al agregarle soporte para Android, no solo se habilita la creación de laboratorios enfocados exclusivamente en seguridad móvil, sino que también se abre la puerta a escenarios híbridos donde los participantes podrán simular ataques que se mueven entre infraestructuras tradicionales (como servidores y computadoras) y dispositivos móviles.

1.3. Objetivos

Objetivo general

Frente a la necesidad de incluir el ecosistema móvil en la formación de ciberseguridad, el objetivo principal de este proyecto es ampliar las capacidades de la plataforma Tectonic, dotándola de la funcionalidad necesaria para simular dispositivos móviles y permitiendo así la creación de laboratorios de práctica más completos que combinen sistemas convencionales con el ecosistema móvil.

Objetivos específicos

1. Realizar una investigación del estado del arte en seguridad móvil, sintetizando distintos marcos de referencia para la clasificación de amenazas y tácticas de ataque, que permita fundamentar el diseño de escenarios.

2. Diseñar una solución que permita la creación y gestión automatizada de dispositivos móviles emulados, compatible con el ecosistema de Tectonic.

3. Implementar un prototipo funcional de la solución diseñada.

4. Validar la solución mediante la construcción de un caso de estudio práctico, materializado en un escenario de laboratorio que implemente ataques específicos de los estudiados.

Resultados Obtenidos

Los resultados de este proyecto abarcan tanto contribuciones conceptuales como técnicas, culminando en una solución funcional y validada. El aporte principal no es solo la extensión de Tectonic para emular dispositivos Android, sino la creación de un marco de trabajo completo para la enseñanza de la seguridad móvil dentro de un cyber range.

- Una investigación del estado del arte en seguridad móvil, que culminó en una taxonomía de ataques propia, basada en marcos de referencia de la industria (OWASP, MITRE ATT&CK, NIST MTC) y adaptada al diseño de escenarios prácticos en cyber ranges.

- Una solución modular y escalable, basada en el emulador oficial de Android, KVM y Ansible, completamente automatizada que transforma una máquina virtual en un entorno de emulación Android funcional, permitiendo la ejecución eficiente de dispositivos Android.

- La validación de la plataforma a través de la creación y despliegue de un laboratorio, demostrando la utilidad de la solución para la enseñanza de técnicas avanzadas de seguridad móvil.

1.4. Organización del informe

El resto del documento se organiza de la siguiente manera:

Capítulo 2: Antecedentes y Marco Teórico — Presenta los fundamentos conceptuales del proyecto.

Capítulo 3: Análisis y Diseño de la Solución — Expone el proceso de investigación de alternativas, los requisitos y la arquitectura propuesta, justificando las decisiones técnicas que guían la implementación.

Capítulo 4: Implementación — Detalla la implementación de la solución.

Capítulo 5: Caso de Estudio: Laboratorio de Ejecución Remota de Código (RCE) — Presenta el caso de estudio utilizado para validar la solución, su automatización extremo a extremo, el fundamento de la vulnerabilidad explotada y el flujo de ataque en un entorno controlado.

Capítulo 6: Conclusiones y Trabajo Futuro

Capítulo 2

Antecedentes y Marco Teórico

Este capítulo establece el marco teórico y los antecedentes necesarios para contextualizar el proyecto. Se presentan los conceptos clave sobre los cyber ranges y la arquitectura de Tectonic, junto con los principios de Infraestructura como Código utilizados para orquestar escenarios. Luego se introduce el ecosistema Android con su modelo de seguridad, opciones de emulación y la herramienta ADB. Finalmente, se describe el panorama de seguridad móvil.

2.1. ¿Qué es un cyber range?

Un *cyber range* es un entorno virtual seguro e interactivo diseñado para la capacitación y el entrenamiento en ciberseguridad, así como para la evaluación de tecnologías e infraestructuras (IBM, 2024). Estas plataformas replican redes, sistemas, aplicaciones y datos en un espacio aislado para permitir a individuos y equipos adquirir y mejorar sus habilidades prácticas a través de la experimentación con escenarios realistas. El propósito fundamental de un *cyber range* es funcionar como un laboratorio de entrenamiento donde se practican ataques a infraestructuras simuladas, la defensa contra ataques simulados, la emulación de adversarios, el análisis forense, y la validación de posturas de seguridad (Threat Defence, s.f.).

2.1.1. Tipos de Ejercicios en un cyber range

Los cyber ranges se estructuran para simular distintos escenarios de ciberseguridad, permitiendo la recreación de roles ofensivos y defensivos, así como competiciones de habilidades. Estos son los tipos de ejercicios más comunes que se pueden representar:

- **Equipo Rojo (Ofensiva)**: Los participantes actúan como atacantes, enfocándose en la penetración y la explotación de vulnerabilidades.

    - Ejemplo: Modificar el código de una aplicación o interceptar comunicaciones de red para robar datos.

- **Equipo Azul (Defensiva)**: Los participantes asumen el rol del defensor, centrándose en la detección, la contención y la respuesta a incidentes (RI). Esta función incluye el **fortalecimiento de sistemas** (aplicar configuraciones de seguridad para reducir su superficie de ataque).

    - Ejemplo: Analizar la actividad del dispositivo para identificar intrusiones o investigar qué datos se expusieron.

- **Equipo Púrpura (Colaboración)**: Combina los objetivos y las lecciones aprendidas del Equipo Rojo y el Equipo Azul en un ciclo de retroalimentación continua para mejorar la postura de seguridad.

    - Ejemplo: Analizar un ejercicio anterior donde el Equipo Rojo logró explotar una vulnerabilidad y, junto al Equipo Azul, ajustar las reglas de detección o las configuraciones del sistema para evitar que el mismo ataque vuelva a tener éxito.

- **Captura la Bandera (CTF)**: Son competiciones de tiempo limitado centradas en la resolución de desafíos técnicos específicos para obtener banderas que demuestran la finalización del objetivo.

    - Ejemplo: Extraer claves o *tokens* de autenticación de una aplicación vulnerable o descifrar datos almacenados de forma insegura.

A diferencia de un laboratorio de seguridad estático, un *cyber range* ofrece un ambiente dinámico y adaptable capaz de simular amenazas del mundo real. Esta capacidad de emulación y simulación permite a los usuarios familiarizarse con herramientas ofensivas y defensivas sin el riesgo de comprometer la infraestructura real de una organización. La relevancia de los *cyber ranges* radica en su capacidad para ejecutar diversos escenarios de ataque y defensa, lo cual permite desarrollar un pensamiento adversario, adoptando la perspectiva del atacante para anticipar sus movimientos y fortalecer las defensas para responder de manera efectiva a los desafíos de seguridad. A la vez que, en el ámbito académico, facilita la enseñanza práctica y el desarrollo de competencias mediante ejercicios reproducibles y medibles.

2.1.2. Público Objetivo

La utilidad de un *cyber range* abarca diversos segmentos de la audiencia, cada uno con objetivos de capacitación específicos, como lo establece la guía del NIST (National Institute of Standards and Technology, 2023). Estos entornos son utilizados por:

- **Educadores:** Para implementar cursos y programas de estudio que requieren una práctica activa para complementar la teoría.

- **Individuos:** Quienes buscan en estas plataformas un medio para la formación continua y el desarrollo de habilidades especializadas en roles como analistas de seguridad, forenses digitales y operadores.

- **Organizaciones:** Para validar las capacidades de sus equipos, evaluar a candidatos durante los procesos de contratación o probar la postura de seguridad de nuevos productos y sistemas antes de su despliegue (IBM, 2024).

2.1.3. Componentes

El funcionamiento de un *cyber range* se basa en una arquitectura de múltiples capas que trabajan de forma coordinada para crear y administrar entornos de entrenamiento seguros y realistas (National Institute of Standards and Technology, 2023), como se ilustra en la Figura 2.1.

En la base se encuentra la **infraestructura subyacente**, compuesta por los recursos físicos o en la nube (por ejemplo, AWS o Azure) donde se alojan los escenarios. Sobre ella opera la **capa de virtualización**, que permite ejecutar múltiples entornos de práctica de manera aislada, utilizando hipervisores (como KVM o Hyper-V) o contenedores (como Docker).

La **capa de orquestación** coordina la creación, configuración y despliegue de los escenarios. Se apoya en herramientas de Infraestructura como Código (IaC) para automatizar la provisión de recursos y garantizar entornos reproducibles. Finalmente, la **infraestructura objetivo** corresponde al entorno simulado con el que interactúan los participantes, formado por redes, servidores y aplicaciones diseñadas para cada ejercicio.

Todas estas capas se integran bajo un **sistema de gestión del aprendizaje (RLMS)**, que actúa como interfaz central de la plataforma. Desde allí, los instructores pueden definir escenarios, administrar usuarios y evaluar el progreso de los estudiantes durante las actividades.

2.1.4. Roles

Los siguientes roles son utilizados para definir las funcionalidades de los usuarios de un *cyber range*, estableciendo la estructura de la plataforma y el manejo de los ejercicios.

- **Instructor:** Define los objetivos de entrenamiento, especifica la topología de red y personaliza las configuraciones de los escenarios. Es responsable de asegurar que se cumplan los requisitos didácticos y técnicos del escenario.

- **Estudiante:** Es el usuario final que accede al ambiente virtualizado. Interactúa con las máquinas virtuales del escenario para ejecutar tareas de ciberseguridad, aplicando habilidades prácticas de ataque o defensa.

- **Administrador de Infraestructura:** Se encarga del mantenimiento del *cyber range* y la gestión de los recursos subyacentes. Su función es garan-

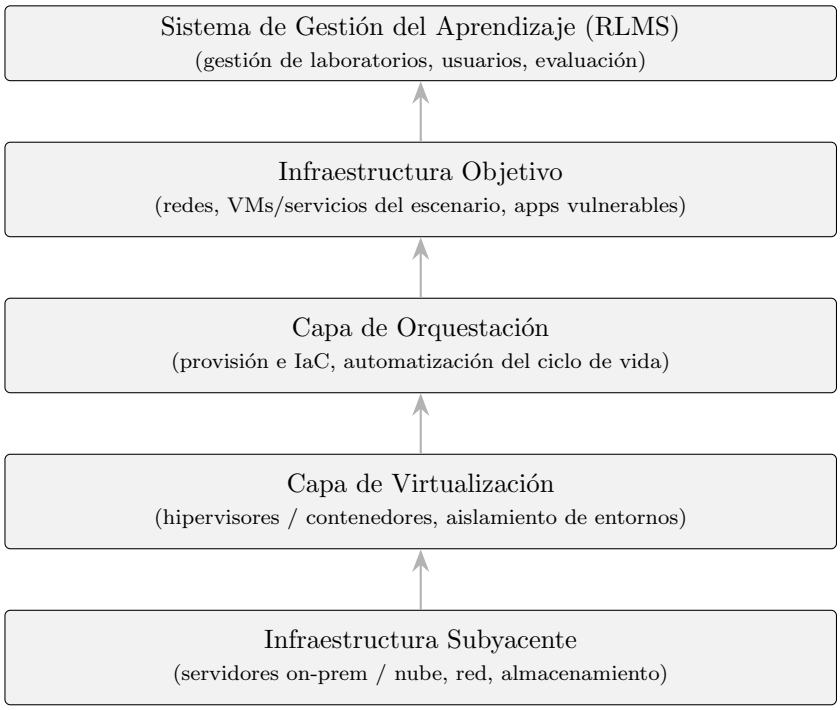

Figura 2.1: Arquitectura por capas de un *cyber range* (adaptado de NIST, *The Cyber Range: A Guide*, 2023).

tizar la disponibilidad, integridad y escalabilidad de la infraestructura que aloja los escenarios.

2.1.5. Tipos de cyber ranges

Los *cyber ranges* se clasifican según su nivel de fidelidad y la tecnología subyacente que utilizan para la simulación y emulación (National Institute of Standards and Technology, 2023). Las principales categorías incluyen:

- **Simulación:** Estos entornos replican el comportamiento de los componentes de red de forma sintética, utilizando modelos matemáticos y lógicos en lugar de hardware o software real. Son ideales para escenarios a gran escala o de despliegue rápido, ya que ofrecen la ventaja de una rápida reconfiguración y la capacidad de utilizar equipo genérico de servidor y almacenamiento. Sin embargo, su principal limitación radica en la fidelidad, ya que pueden no reproducir con precisión las latencias, el jitter y otras características inherentes a una red física real, lo que podría afectar el realismo de ciertos ejercicios.
- **Emulación:** A diferencia de la simulación, la emulación opera sobre una

infraestructura de red dedicada o un subconjunto de ella. Este enfoque permite un alto grado de fidelidad al replicar los sistemas, flujos de tráfico y el comportamiento de los protocolos de manera más auténtica que la simulación. Los entornos de emulación son valiosos para entrenamientos que requieren interacciones precisas y el uso de herramientas de ataque y defensa en tiempo real.

- **Híbrido:** Estos entornos combinan elementos de la simulación y la emulación, aprovechando las ventajas de ambos modelos para crear plataformas de entrenamiento flexibles y adaptables. Un entorno híbrido puede utilizar la emulación para los sistemas críticos que requieren alta fidelidad (ej., servidores o firewalls), mientras que otros componentes de la red menos cruciales se representan mediante simulación. Esto permite optimizar los recursos sin sacrificar el realismo en los puntos clave del ejercicio.

Una clasificación adicional se basa en el modelo de alojamiento (Threat Defence, s.f.), lo que es fundamental para la escalabilidad y el acceso:

- ***On-Premise*:** Se instalan en la infraestructura local de la organización. Ofrecen un control total y alta seguridad, pero su escalabilidad es limitada por los recursos físicos disponibles.
- **Cloud:** Se alojan en plataformas en la nube (como AWS, Azure). Proporcionan alta escalabilidad y flexibilidad, permitiendo un acceso geográfico amplio y la provisión de recursos bajo demanda.
- **Híbrido:** Combinan el control de un entorno local con la flexibilidad y escalabilidad de la nube.

Otra distinción crucial es entre:

- **Entornos Virtuales:** Se ejecutan completamente en software y son más flexibles para la simulación de escenarios teóricos.
- **Entornos *Live-Fire* (Fuego Real):** Utilizan hardware real en un ambiente controlado, lo que permite un entrenamiento con herramientas y tácticas de ataque del mundo real de alta fidelidad, con el objetivo de evaluar la respuesta a amenazas en tiempo real.

2.2. Tectonic

Tectonic es un *cyber range* académico desarrollado por el GSI (FIng–UdelaR) para crear y gestionar escenarios de ciberseguridad mediante Infraestructura como Código, automatizando su ciclo de vida (Guerrero, Betarte, y Campo, 2024). Como se describe en el repositorio oficial, la plataforma está diseñada para proporcionar "escenarios de ciberseguridad realistas para educación y entrenamiento a través del despliegue de redes, sistemas y aplicaciones" (GSI-Fing-Udelar, 2024).

2.2.1. Arquitectura por Capas

Como se puede ver en la Figura 2.2, la arquitectura de Tectonic (Grupo de Seguridad Informática, FIng-Udelar, 2024) se organiza en cinco capas funcionales, cada una cumpliendo un rol específico en la operación de la plataforma:

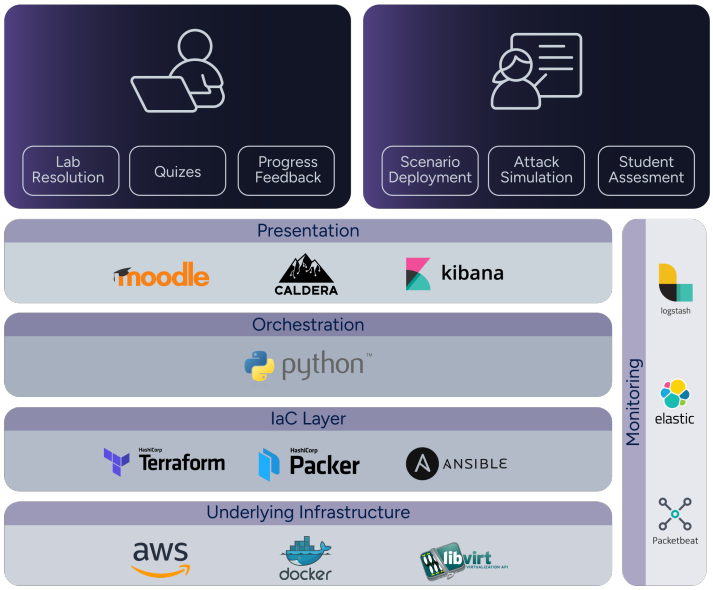

Figura 2.2: Arquitectura por capas de Tectonic.

1. **Infraestructura Subyacente:** Constituye la infraestructura física o en la nube donde se despliegan los sistemas y redes que forman la base de un escenario particular. Actualmente soporta despliegues en AWS y *on-premise* usando Libvirt y Docker, con más plataformas planificadas.

2. **Herramientas de IaC:** Para lograr el despliegue automatizado de la infraestructura, se utilizan herramientas como Packer (Hashicorp, 2013) (creación de imágenes), Terraform (Hashicorp, 2014) (creación de recursos) y Ansible (Michael DeHaan, 2012) (configuración de sistemas). La plataforma utiliza extensivamente los *Playbooks de Ansible* (Ansible, 2023) para la configuración específica de los escenarios.

3. **Orquestación:** Un componente en Python orquesta estas herramientas y gestiona el ciclo de vida completo de los escenarios, incluyendo su despliegue, eliminación, encendido, apagado y listado de información.

4. **Especificación de Escenarios:** Los escenarios se describen mediante una especificación declarativa que permite a los usuarios definir aspectos

como las máquinas a desplegar, las redes utilizadas para conectarlas y las configuraciones a aplicar, entre otros.

5. **Capa de Presentación y Monitoreo:** Incluye el sistema de gestión de aprendizaje (gestión de laboratorios), el acceso remoto a los entornos y la pila Elastic como SIEM para monitoreo y evaluación.

2.2.2. Ciclo de Vida de Escenarios

El proceso de gestión de escenarios en Tectonic sigue un flujo bien definido:

- **Creación de Imágenes Base:** El sistema toma la especificación del escenario y utiliza la herramienta Packer, que permite crear imágenes de máquinas idénticas a partir de una única plantilla fuente, para generar o seleccionar las imágenes base de los sistemas operativos. Estas imágenes son discos virtuales que contienen un sistema operativo base, software preinstalado y configuración inicial. Estas imágenes base son el punto de partida para crear las máquinas virtuales del escenario, lo que permite una creación rápida y eficiente de múltiples clones de máquinas al compartir el mismo disco y almacenar solo las diferencias individuales de cada instancia.

- **Clonación y Despliegue:** Tectonic emplea Terraform para gestionar la creación de las instancias de máquinas virtuales y desplegar la topología de red definida en el escenario. Terraform es una herramienta que provee la capacidad de gestionar infraestructura creando, modificando y eliminando recursos virtuales a partir de una especificación declarativa del escenario, definida internamente en Tectonic.

- **Configuración Post-clonación:** Ansible ejecuta los playbooks definidos por el instructor para aplicar las configuraciones específicas del escenario. Estas configuraciones abarcan la instalación de herramientas, la inicialización de servicios y la personalización de la configuración inicial de la máquina. Esta es la única capa de Infraestructura como Código cuyo contenido es diseñado por el instructor que configura los escenarios.

- **Gestión del Ciclo de Vida:** La Interfaz de Línea de Comandos (CLI) de Tectonic es el principal punto de acceso que permite al Instructor automatizar y controlar el ciclo de vida completo de los escenarios. Estas operaciones incluyen la capacidad de crear y desplegar réplicas individualizadas y aisladas del escenario para cada participante, así como encender, apagar y destruir estas instancias, asegurando el control y la escalabilidad de los recursos para múltiples grupos de estudiantes.

2.3. Principios de Automatización: Infraestructura como Código y Ansible

Un pilar fundamental en la arquitectura de Tectonic y, por extensión, en la solución propuesta, es el paradigma de **Infraestructura como Código (IaC)**.

2.3.1. Infraestructura como Código (IaC)

La Infraestructura como Código es la práctica de gestionar y aprovisionar la infraestructura de TI (redes, máquinas virtuales, balanceadores de carga, etc.) a través de archivos de definición legibles por máquina, en lugar de utilizar configuración manual o herramientas interactivas. Este enfoque trata a la infraestructura de la misma manera que el software, permitiendo aplicar prácticas de desarrollo como el control de versiones, la revisión de código y la integración continua.

Para un *cyber range* como Tectonic, los beneficios de IaC son críticos:

- **Reproducibilidad:** Garantiza que un escenario definido en código se pueda desplegar de forma idéntica una y otra vez, eliminando la variabilidad que introduce la configuración manual. Esto es esencial para la validez de ejercicios y evaluaciones.

- **Escalabilidad:** Permite el despliegue de múltiples copias aisladas (réplicas) de un mismo escenario de forma rápida y automática. Esta funcionalidad es importante, ya que asegura que varios equipos o estudiantes puedan trabajar simultáneamente sin riesgo de interferencia, facilitando el trabajo concurrente. El aislamiento entre estas réplicas previene que las acciones ofensivas o defensivas de un usuario afecten la integridad del entorno de otro.

- **Versionado y Colaboración:** Almacenar la infraestructura como código (por ejemplo, en un repositorio Git) permite tener un historial de cambios, colaborar en el diseño de escenarios y revertir a versiones anteriores si es necesario. Esta facilidad para versionar y compartir el código es un objetivo importante, ya que simplifica la generación de un catálogo de escenarios al que los instructores pueden contribuir para sus entrenamientos.

2.3.2. Ansible como Motor de Configuración

Ansible es una herramienta de automatización de código abierto que se utiliza para tareas como la gestión de configuración, el despliegue de aplicaciones y la orquestación de flujos de trabajo. Es el motor de Infraestructura como Código (IaC) utilizado por Tectonic para la configuración de las máquinas virtuales una vez que han sido desplegadas. El instructor define gran parte de la especificación del escenario utilizando archivos de configuración de Ansible, llamados playbooks.

- **Playbooks:** Estos son archivos de definición escritos en formato YAML que describen el estado final deseado de un sistema. En el contexto de Tectonic, permiten declarar la configuración final requerida (como la instalación de software o la modificación de archivos), a diferencia de la ejecución de comandos paso a paso.

Sus características clave, relevantes para este proyecto, son:

- **Naturaleza Declarativa:** En lugar de especificar los pasos para llegar a un estado, se describe el estado final deseado en un *playbook*. Ansible se encarga de determinar los pasos necesarios para alcanzar ese estado.

- **Idempotencia:** Una operación es idempotente si aplicarla múltiples veces produce el mismo resultado que aplicarla una sola vez. Los *playbooks* de Ansible están diseñados para ser idempotentes: si un paquete ya está instalado o un archivo ya tiene el contenido correcto, Ansible no realizará ninguna acción. Esto hace que las ejecuciones repetidas sean seguras y predecibles.

- **Arquitectura sin Agentes:** Ansible se comunica con las máquinas gestionadas a través de un protocolo que permite la ejecución segura de comandos remotos a través de una red cifrada llamado SSH (Secure Shell), sin necesidad de instalar ningún software de agente especial en ellos, lo que simplifica la gestión y tiene menos interferencia en los escenarios.

2.4. Android: Fundamentos y Herramientas

Para poder diseñar e implementar escenarios de seguridad móvil, es fundamental comprender el sistema operativo Android, las principales soluciones disponibles para emularlo y una herramienta esencial que permite interactuar y analizar estos dispositivos emulados.

2.4.1. El Sistema Operativo Android

Android es un sistema operativo de código abierto basado en Linux. Fue desarrollado por Android Inc., una compañía adquirida por Google en 2005, y actualmente es gestionado por el Android Open Source Project (AOSP). Desde su lanzamiento en 2008, se ha convertido en el sistema operativo más utilizado en el mundo, presente en una amplia variedad de dispositivos.

La seguridad de Android se apoya en un modelo de aislamiento conocido como sandbox. En este modelo, cada aplicación se ejecuta en un entorno separado y seguro, sin acceso directo a los datos o procesos de las demás. Este aislamiento se implementa a nivel del núcleo del sistema (Linux kernel), que asigna a cada aplicación una identidad de usuario única, llamada UID (User Identifier). Gracias a este mecanismo, una aplicación solo puede acceder a sus propios recursos y necesita permisos explícitos para comunicarse con otras o con el sistema. Este

enfoque reduce la posibilidad de que una aplicación maliciosa comprometa el resto del dispositivo o los datos personales del usuario.

2.4.2. Soluciones de Emulación

La emulación de Android permite recrear mediante software el comportamiento de un teléfono real. Esto incluye su sistema operativo y sus componentes principales como la memoria, la pantalla o la red, pero sin necesidad de disponer del dispositivo físico. Gracias a esto, es posible ejecutar aplicaciones, realizar pruebas o recrear escenarios de laboratorio de forma segura y controlada.

A diferencia de un dispositivo real, un emulador ofrece ventajas importantes: puede reiniciarse fácilmente, ejecutarse en paralelo con otros, mantenerse aislado del sistema donde se ejecuta y configurarse siempre de la misma manera. Estas características lo hacen ideal para entornos de práctica o automatización. A continuación se presentan distintas soluciones para emular Android, tanto oficiales como de terceros:

- **Android Emulator (SDK oficial de Google):** Emulador oficial de Google (Android Developers, 2024). Permite crear dispositivos virtuales con diferentes versiones del sistema operativo y configuraciones de hardware (por ejemplo, cantidad de memoria o tamaño de pantalla). Se destaca por su fidelidad al sistema original y su integración con otras herramientas de desarrollo y prueba automatizada.
- **Genymotion:** Es una alternativa comercial que ofrece emuladores listos para usar, tanto en computadoras personales como en la nube. Está orientado a la facilidad de uso y al rendimiento gráfico, e incluye la posibilidad de simular sensores como el GPS o la cámara, lo que lo hace útil para demostraciones o prácticas interactivas.
- **Otros**
  - **ReDroid:** Permite ejecutar Android dentro de contenedores (como Docker), lo que facilita desplegar muchas instancias al mismo tiempo en un mismo servidor.
  - **docker-android:** Ofrece imágenes preconfiguradas que pueden ejecutarse directamente y controlarse desde un navegador web, lo que simplifica las pruebas remotas.
  - **Android-x86:** Permite instalar Android en un ordenador o máquina virtual como si fuera un dispositivo normal.

2.4.3. Android Debug Bridge (ADB)

El Android Debug Bridge es una herramienta de línea de comandos que permite comunicarse con un dispositivo o emulador Android. Es utilizada para instalar aplicaciones, realizar pruebas, obtener información del sistema o automatizar tareas.

A través de ADB es posible realizar tareas como:

- Instalar y desinstalar aplicaciones.
- Ejecutar comandos en el dispositivo desde una terminal.
- Transferir archivos entre el sistema anfitrión y el dispositivo.
- Ver los registros del sistema en tiempo real.
- Configurar opciones avanzadas, como el uso de un proxy de red para observar el tráfico.

2.5. El Ecosistema de la Seguridad Móvil

La comprensión del panorama de amenazas y vulnerabilidades del ecosistema móvil es esencial para diseñar escenarios de laboratorio realistas y relevantes.

Esta sección presenta, en primer lugar, los componentes principales del ecosistema móvil y sus vectores de ataque. Luego, se describen los marcos de referencia y estándares de la industria más relevantes para la clasificación de amenazas móviles, incluyendo el OWASP Mobile Top 10, MITRE ATT&CK for Mobile y el NIST Mobile Threat Catalogue.

2.5.1. Panorama de Amenazas y Componentes del Ecosistema

En los últimos años, los incidentes de seguridad en dispositivos móviles han aumentado de manera significativa, impulsados por factores como el crecimiento del número de dispositivos en uso, la mayor cantidad de información sensible que gestionan (financiera, salud, trabajo) y el aumento de aplicaciones maliciosas (Kivva, 2024; StatCounter Global Stats, 2024). Como resultado, estos dispositivos se han convertido en un objetivo prioritario para los atacantes, tanto para robar datos sensibles como para utilizarlos como punto de entrada a otros sistemas. Por ello, es fundamental que los profesionales de ciberseguridad comprendan el ecosistema móvil y sus vulnerabilidades para poder mitigar estos riesgos.

El ecosistema de seguridad móvil es un entorno complejo que incluye el dispositivo físico, el sistema operativo, las aplicaciones, las redes y los servicios de backend (servidores con los que las apps se comunican para autenticación, datos y otras funciones). La forma en que todos estos elementos se conectan crea una gran superficie de ataque que los atacantes pueden aprovechar de varias maneras (Russo, Merla, y Verderame, 2020). El panorama de amenazas se puede entender mejor analizando los componentes principales de este ecosistema:

- **Dispositivo y Sistema Operativo (SO) Móvil:** Las vulnerabilidades en esta capa, a nivel de hardware o del sistema operativo (por ejemplo, Android), pueden comprometer el dispositivo de forma crítica. Un caso

común es el *rooting* no autorizado, es decir, obtener privilegios de administrador que permiten al atacante controlar todas las funciones del teléfono y desactivar sus mecanismos de seguridad. Otro punto crítico es el kernel, el núcleo del sistema operativo que gestiona los recursos del dispositivo (procesador, memoria, comunicaciones). Una falla en el kernel puede dar acceso directo al hardware, permitiendo a un atacante ejecutar código malicioso con los máximos privilegios.

- **Aplicaciones Móviles:** Son el principal vector de ataque en el ecosistema. Las debilidades en las aplicaciones pueden surgir de malas prácticas de desarrollo, incluyendo el uso de librerías obsoletas, validación insuficiente de entradas/salidas o almacenamiento inseguro de datos (National Institute of Standards and Technology, 2018; OWASP Foundation, 2020).

- **Redes de Comunicación:** Conformadas por las redes Wi-Fi, celulares y conexiones Bluetooth. Los ataques a esta capa buscan interceptar, modificar o bloquear la comunicación entre el dispositivo y los servidores, como es el caso de los ataques de *Man-in-the-Middle* (MITM), los cuales consisten en ataques de interceptación y escucha de comunicaciones para extraer y potencialmente modificar información sensible.

- **Tiendas de Aplicaciones y Servicios de *Backend*:** Son un punto de entrada clave para la distribución de malware. Los atacantes pueden inyectar código malicioso en aplicaciones para luego distribuirlas a través de tiendas no oficiales o incluso intentar infiltrarse en tiendas oficiales (Russo y cols., 2020). Los servicios de *backend* con los que las aplicaciones se comunican son también un objetivo para obtener acceso a bases de datos o información sensible.

En este contexto, un ataque puede aprovechar una o varias vulnerabilidades en diferentes capas del ecosistema. Por ejemplo, un atacante podría comprometer un dispositivo mediante una aplicación maliciosa, usar la conexión de red del dispositivo para acceder a una red corporativa interna y, finalmente, explotar una vulnerabilidad en un servidor de *backend* para robar datos (Russo y cols., 2020).

2.5.2. Frameworks y Estándares de la Industria

Para entender y defenderse de las amenazas móviles, el sector de la ciberseguridad utiliza marcos de trabajo y catálogos de amenazas creados por organizaciones de referencia que proporcionan una base sólida para la comprensión y categorización de las amenazas (Russo y cols., 2020; National Institute of Standards and Technology, 2018). Los marcos más relevantes para este proyecto incluyen el OWASP Mobile Top 10, MITRE ATT&CK for Mobile y el NIST Mobile Threat Catalogue.

OWASP Mobile Top 10

El **Open Web Application Security Project** (OWASP) es una fundación sin fines de lucro dedicada a mejorar la seguridad del software a nivel global (OWASP Foundation, 2020). Su proyecto **OWASP Mobile Top 10** identifica y clasifica las diez vulnerabilidades de seguridad más críticas en aplicaciones móviles, basándose en datos de riesgos, fallas y vulnerabilidades del mundo real (OWASP Foundation, s.f.). Este documento es un punto de referencia esencial para desarrolladores, analistas de seguridad y profesionales, ya que proporciona una guía clara para el desarrollo de aplicaciones más seguras.

El marco se organiza en categorías de fallas que no solo describen el problema, sino que también ofrecen orientación sobre cómo prevenirlo. A continuación, se detallan las categorías más recientes del OWASP Mobile Top 10:

- **M1: Uso Incorrecto de Credenciales (Improper Credential Usage):** Cubre fallas en la gestión de credenciales, como incluir credenciales directamente en el código, su almacenamiento o transmisión en forma insegura, y mecanismos de autenticación débiles que facilitan el acceso no autorizado. Estas debilidades pueden derivar en fuga de información, suplantación de identidad y abuso de funcionalidades sensibles.

- **M2: Seguridad Inadecuada de la Cadena de Suministro (Inadequate Supply Chain Security):** Se relaciona con la integración de componentes de terceros (librerías, SDKs) que contienen vulnerabilidades conocidas. Un atacante puede explotar estas debilidades o inyectar código malicioso durante el proceso de desarrollo.

- **M3: Autenticación/Autorización Insegura (Insecure Authentication/Authorization):** Son fallas en los controles de acceso que permiten a un atacante eludir la verificación de identidad o acceder a datos y funcionalidades que deberían ser inaccesibles para su perfil.

- **M4: Validación Insuficiente de Entrada/Salida (Insufficient Input/Output Validation):** Representa la debilidad o ausencia en la sanitización de datos que ingresan a la aplicación. Esta falla es la causa principal de ataques de Inyección y problemas de acceso al sistema de archivos.

- **M5: Comunicación Insegura (Insecure Communication):** Describe vulnerabilidades durante la transmisión de datos en la red. Se manifiestan por la omisión del cifrado o por el uso de algoritmos criptográficos débiles u obsoletos, facilitando ataques de Man-in-the-Middle (MITM).

- **M6: Controles de Privacidad Inadecuados (Inadequate Privacy Controls):** Abarca fallas relacionadas con la recolección y manejo de Información Personal Identificable (PII). Resulta en la exposición inadvertida de datos sensibles, como en *logs* del sistema, o la ausencia de mecanismos de consentimiento apropiados.

- **M7: Protección Binaria Insuficiente (Insufficient Binary Protections):** Se refiere a la falta de mecanismos de seguridad que resguarden el código de la aplicación. Esto simplifica la ingeniería inversa para descubrir secretos internos y la manipulación del binario para alterar su funcionamiento legítimo.

- **M8: Configuración de Seguridad Errónea (Security Misconfiguration):** Ocurre cuando la implementación de la seguridad no es óptima, ya sea en la aplicación o el servidor. Esto incluye controles de acceso excesivamente permisivos o el despliegue de servicios con configuraciones por defecto inseguras.

- **M9: Almacenamiento Inseguro de Datos (Insecure Data Storage):** Este riesgo aparece cuando una aplicación guarda información sensible en el dispositivo sin cifrado o con mecanismos de protección débiles. En ese caso, un atacante que logre acceder al sistema de archivos —ya sea mediante otra aplicación maliciosa o a través de un acceso físico al dispositivo— puede leer esos datos y utilizarlos indebidamente.

- **M10: Criptografía Insuficiente (Insufficient Cryptography):** Involucra las vulnerabilidades que provienen del uso de algoritmos de cifrado obsoletos o implementaciones criptográficas defectuosas. Esto permite a un atacante descifrar o falsificar datos sensibles de manera más eficiente que si se utilizaran estándares robustos.

MITRE ATT&CK for Mobile

Desarrollado por la corporación MITRE, el marco **ATT&CK** (Adversarial Tactics, Techniques, and Common Knowledge) es una base de conocimiento global de tácticas y técnicas de adversarios basada en observaciones del mundo real (MITRE, 2020). A diferencia de OWASP, que se centra en vulnerabilidades, ATT&CK se enfoca en las acciones que los atacantes llevan a cabo.

El sub-marco **ATT&CK for Mobile** documenta las tácticas (objetivos del ataque), las técnicas (cómo se logran esos objetivos) y los procedimientos (las implementaciones específicas de las técnicas) utilizados contra dispositivos móviles, tanto del sistema operativo Android como iOS. Su propósito es ayudar a las organizaciones a comprender el comportamiento de los atacantes, mejorar la detección de amenazas y validar la efectividad de sus defensas de seguridad.

A continuación, se listan algunas de las tácticas y técnicas más relevantes en el marco ATT&CK for Mobile:

- **Acceso a Credenciales (Credential Access):** Tácticas para robar nombres de usuario y contraseñas.

    - **Técnica: Superposición de Pantalla (Screen Overlay):** Una aplicación maliciosa detecta cuándo el usuario abre una aplicación legítima (como una app de redes sociales o bancaria) y superpone

una ventana de inicio de sesión falsa, idéntica a la real, para capturar el usuario y la contraseña ingresados.

- **Recolección de Datos (Collection):** Tácticas para recopilar información del dispositivo y del usuario.
  - **Técnica: Rastreo de Ubicación (Location Tracking):** Una aplicación, a menudo bajo la apariencia de una utilidad inofensiva, accede a los datos del GPS del dispositivo en segundo plano para monitorear y registrar los movimientos del usuario sin su consentimiento explícito.
- **Impacto (Impact):** Tácticas diseñadas para dañar o interrumpir el uso del dispositivo.
  - **Técnica: Ransomware:** Un tipo de software malicioso que cifra los archivos personales del usuario (fotos, documentos) o bloquea completamente el acceso al dispositivo, exigiendo un pago (rescate) para devolver el control.

A diferencia de otros marcos como OWASP, que se centra en las vulnerabilidades del código, **ATT&CK** se enfoca en el comportamiento del atacante, ofreciendo un modelo de amenazas que describe el cómo de los ataques. Esto permite a los defensores entender las motivaciones y pasos de un adversario, facilitando la creación de defensas proactivas y la emulación de adversarios para validar la postura de seguridad.

NIST Mobile Threat Catalogue (MTC)

El **National Institute of Standards and Technology** (NIST) es una agencia del gobierno de EE. UU. dedicada a la promoción de la innovación y la competitividad industrial mediante el desarrollo de estándares (National Institute of Standards and Technology, 2023). El **Mobile Threat Catalogue (MTC)** es un proyecto clave que busca organizar las amenazas a la seguridad de dispositivos móviles de manera coherente y sistemática (National Institute of Standards and Technology, 2018). Su enfoque es centrado en las amenazas y está diseñado para ser agnóstico a la tecnología, lo que significa que se aplica a cualquier tipo de dispositivo y sistema operativo móvil.

A diferencia de otros marcos de trabajo, que se centran en vulnerabilidades del código (OWASP) o en el comportamiento del atacante (MITRE), el enfoque de NIST es más granular y holístico. Su modelo se basa en la idea de "puntos de amenaza", que son combinaciones específicas de amenazas, vulnerabilidades, ataques y contramedidas. Este modelo proporciona una visión completa de la problemática y es una herramienta valiosa para profesionales de seguridad, ya que les permite realizar un análisis de riesgos estructurado, identificar amenazas potenciales e implementar controles de seguridad a lo largo de todo el ciclo de vida de un sistema móvil.

La taxonomía del MTC se arma a través de una estructura jerárquica que va desde las amenazas de alto nivel hasta los puntos de amenaza específicos. Para leer y comprender las categorías del MTC, es necesario seguir un enfoque descendente. Se comienza con una de las categorías principales (por ejemplo, "Amenazas a la Aplicación") y se desciende a través de las subcategorías para identificar los "puntos de amenaza" concretos. Cada punto de amenaza combina elementos como la descripción del ataque, las vulnerabilidades explotadas, las técnicas utilizadas y las posibles contramedidas. Esto proporciona una visión completa de la amenaza, permitiendo a los analistas de seguridad no solo entender el riesgo, sino también planificar su mitigación de manera efectiva.

Las categorías principales son:

- **Amenazas a la Aplicación:** Enfocadas en las debilidades del software, como la validación insuficiente de datos o el uso de librerías vulnerables.
- **Amenazas al Sistema Operativo y al *Stack*:** Cubren fallas en el sistema base del dispositivo, como configuraciones predeterminadas inseguras o problemas de permisos.
- **Amenazas a las Redes Celulares, LAN/PAN y GPS:** Relacionadas con la interceptación de comunicaciones en redes inalámbricas o la manipulación de señales de localización.
- **Amenazas del Ecosistema y la Cadena de Suministro:** Se centran en los riesgos derivados de servicios de terceros o del proceso de distribución de software.

La diferencia principal con los otros marcos es que NIST ofrece una visión más completa y granular del ciclo de vida de una amenaza, lo que lo hace particularmente útil para el diseño de políticas de seguridad y la evaluación de la postura defensiva en un entorno móvil.

Capítulo 3

Análisis y Diseño de la Solución

Este capítulo presenta el proceso de análisis y diseño de la solución para integrar la emulación de dispositivos móviles en Tectonic.

Se comienza con la **taxonomía de ataques móviles**, desarrollada a partir del estudio de los marcos de referencia de la industria, que permite identificar y organizar el panorama completo de amenazas en el ecosistema móvil. Esta taxonomía sirve como guía para definir los **requerimientos del proyecto**, asegurando que la solución cubra la mayor cantidad posible de escenarios representativos, priorizando aquellas funcionalidades que permitan simular un espectro amplio de las categorías de ataque identificadas.

A continuación, se justifica la **selección de Android** como plataforma móvil objetivo, fundamentada en criterios técnicos relacionados con la infraestructura de Tectonic. Posteriormente, se evalúan las **alternativas de solución** consideradas, describiendo las opciones exploradas y las decisiones técnicas tomadas. Finalmente, se presenta la **solución propuesta**, detallando los componentes seleccionados y su integración con el ecosistema de Tectonic.

3.1. Taxonomía de Ataques Móviles

La taxonomía que se presenta a continuación es el resultado de un proceso de síntesis y consolidación de los marcos teóricos más relevantes en ciberseguridad móvil. Los principales frameworks de la industria, como *OWASP Mobile Top 10*, *MITRE ATT&CK Mobile*, y el *NIST Mobile Threat Catalogue*, abordan la problemática desde perspectivas distintas, con un enfoque que no se alinea completamente con lo que necesitamos en este proyecto.

- **OWASP Mobile Top 10:** Se enfoca en las vulnerabilidades más críticas en el código de las aplicaciones, desde una perspectiva de desarrollo y auditoría.

- **MITRE ATT&CK Mobile:** Se centra en el comportamiento de los atacantes, mapeando sus tácticas y técnicas en un contexto de defensa y emulación de adversarios.
- **NIST Mobile Threat Catalogue:** Clasifica las amenazas según el punto del ecosistema móvil que comprometen, facilitando una visión holística de los riesgos.

A diferencia de estos marcos, que están orientados principalmente al análisis descriptivo, la presente taxonomía ha sido diseñada específicamente para el diseño de escenarios de entrenamiento en cyber ranges. Su principal valor radica en su enfoque pedagógico, que la convierte en una hoja de ruta práctica para la construcción e implementación de laboratorios de seguridad móvil. Mediante la agrupación de amenazas en categorías funcionales, centradas en el punto de entrada o en el objetivo del ataque, se facilita la construcción de escenarios que simulan un flujo de ataque completo y aseguran que cada práctica se base en un modelo de amenaza sistemático y validado.

3.1.1. Categorías de la Taxonomía

La división en estas categorías se centra en la replicabilidad de los escenarios de ataque dentro del entorno de Tectonic. Cada sección agrupa un conjunto de vulnerabilidades y técnicas de ataque que comparten un punto de entrada o un objetivo común en el ecosistema móvil. Esta metodología garantiza que los escenarios diseñados no solo tengan un propósito teórico, sino que también sean viables para su emulación y experimentación en un *cyber range*. Además, cada categoría incluye referencias cruzadas a los marcos de trabajo principales, lo que permite una validación y contextualización más profunda de cada tipo de amenaza.

1. Aplicación y Binarios

- **Descripción:** Las amenazas en esta categoría explotan debilidades directamente relacionadas con el código y el empaquetado de la aplicación. Los atacantes utilizan técnicas de ingeniería inversa para analizar el binario de la aplicación (el APK en Android), lo que les permite descubrir secretos, como claves de API, credenciales o algoritmos de cifrado. Una vez analizado, el atacante puede realizar un *code tampering*, que implica modificar el código de la aplicación. Por ejemplo, podría alterar la lógica de autenticación para evadir el inicio de sesión o inyectar código malicioso que le otorgue un punto de control en el dispositivo. Finalmente, la aplicación es recompilada y firmada con una nueva clave para su redistribución. Estas técnicas se ven facilitadas por la falta de ofuscación o el uso de métodos de protección deficientes en el binario.
- **Referencias de Marcos:** OWASP Mobile Top 10 (M07) (OWASP Foundation, 2023h), MITRE ATT&CK Mobile (T1577, T1406) (MITRE, 2023g, 2023a) y NIST Mobile Threat Catalogue (APP-3) (NIST, 2023f)

2. Autenticación y Autorización

- **Descripción:** Esta categoría se centra en los ataques que comprometen los mecanismos de control de acceso y privilegios. Las amenazas pueden manifestarse en una autenticación débil o ausente, lo que permite a un adversario eludir el proceso de inicio de sesión, a menudo explotando credenciales predecibles o embebidas en el código. También se incluyen los fallos de autorización, donde el atacante logra acceder a funcionalidades o datos que no le corresponden, como información de otros usuarios o funciones administrativas, a pesar de haberse autenticado correctamente. Esto puede lograrse mediante el uso de tokens comprometidos o fallos en la validación del backend.
- **Referencias de Marcos:** OWASP Mobile Top 10 (M01, M03, M04) (OWASP Foundation, 2023b, 2023d, 2023e), MITRE ATT&CK Mobile (T1635) (MITRE, 2023j) y NIST Mobile Threat Catalogue (AUT-0, AUT-12) (NIST, 2023h, 2023i)

3. Comunicación

- **Descripción:** Los ataques en esta categoría se centran en la interceptación o manipulación del tráfico de red para obtener datos sensibles. Las vulnerabilidades más comunes incluyen la falta de cifrado (por ejemplo, el uso de HTTP en lugar de HTTPS), la aceptación de certificados inválidos o el uso de protocolos de comunicación obsoletos. Un escenario típico es el ataque de *Man-in-the-Middle (MITM)*, donde el atacante se posiciona entre el dispositivo móvil y un servidor para interceptar y leer la comunicación.
- **Referencias de Marcos:** OWASP Mobile Top 10 (M05) (OWASP Foundation, 2023f), MITRE ATT&CK Mobile (T1631, T1638) (MITRE, 2023h, 2023k) y NIST Mobile Threat Catalogue (LPN-2, APP-1) (NIST, 2023j, 2023d)

4. Almacenamiento de Datos

- **Descripción:** Las amenazas en esta categoría se centran en la exposición de información sensible y datos de usuario, que pueden ser aprovechados tanto por la propia aplicación como por un atacante. Los riesgos se materializan cuando credenciales, tokens o información personal se guardan en el sistema de archivos de la aplicación sin el cifrado adecuado o utilizando algoritmos criptográficos débiles. Un adversario con acceso al dispositivo (a través de ADB, por ejemplo) puede extraer y descifrar esta información, exponiendo los datos.
- **Referencias de Marcos:** OWASP Mobile Top 10 (M09, M10) (OWASP Foundation, 2023i, 2023a), MITRE ATT&CK Mobile (T1533) (MITRE, 2023f) y NIST Mobile Threat Catalogue (APP-10) (NIST, 2023a)

5. Privacidad

- **Descripción:** Los ataques en esta categoría se centran en la recolección y exposición no autorizada de datos personales del usuario. Esto incluye la fuga de información sensible a través de mecanismos de registro (logs), el rastreo de ubicación, la recolección de identificadores de dispositivo persistentes o cualquier otra forma de acceso a datos sin consentimiento del usuario.

- **Referencias de Marcos:** OWASP Mobile Top 10 (M06) (OWASP Foundation, 2023g), MITRE ATT&CK Mobile (T1430, T1426) (MITRE, 2023d, 2023c) y NIST Mobile Threat Catalogue (APP-2, APP-12) (NIST, 2023e, 2023b)

6. Ecosistema y Cadena de Suministro

- **Descripción:** Esta categoría se enfoca en amenazas que no se originan en la aplicación o el dispositivo en sí, sino en el entorno circundante. Esto incluye el compromiso de las tiendas de aplicaciones, la inyección de código malicioso en dependencias o librerías de terceros (ataques a la cadena de suministro) y el uso de servicios del sistema operativo con configuraciones predeterminadas inseguras. Los ataques de suplantación de pantalla, como la vulnerabilidad StrandHogg (Android, 2024), también entran en esta categoría, ya que explotan fallos del sistema para replicar la interfaz de una aplicación legítima.

- **Referencias de Marcos:** OWASP Mobile Top 10 (M02) (OWASP Foundation, 2023c), MITRE ATT&CK Mobile (T1632.001, T1407, T1661) (MITRE, 2023i, 2023b, 2023l) y NIST Mobile Threat Catalogue (APP-6, APP-14) (NIST, 2023g, 2023c)

7. Acceso Físico

- **Descripción:** Las amenazas de esta categoría requieren acceso físico al dispositivo para ser llevadas a cabo. Los atacantes pueden explotar vulnerabilidades en la pantalla de bloqueo para obtener acceso al sistema, inyectar comandos o datos a través de una conexión USB o comprometer el dispositivo mediante un *jailbreak* o *rooting* para obtener privilegios elevados.

- **Referencias de Marcos:** MITRE ATT&CK Mobile (T1461) (MITRE, 2023e) y NIST Mobile Threat Catalogue (PHY-2, STA-6) (NIST, 2023k, 2023l)

La Figura 3.1 presenta una visualización de la taxonomía de ataques móviles, mostrando las siete categorías identificadas junto con ejemplos de técnicas de ataque específicas asociadas a cada categoría.

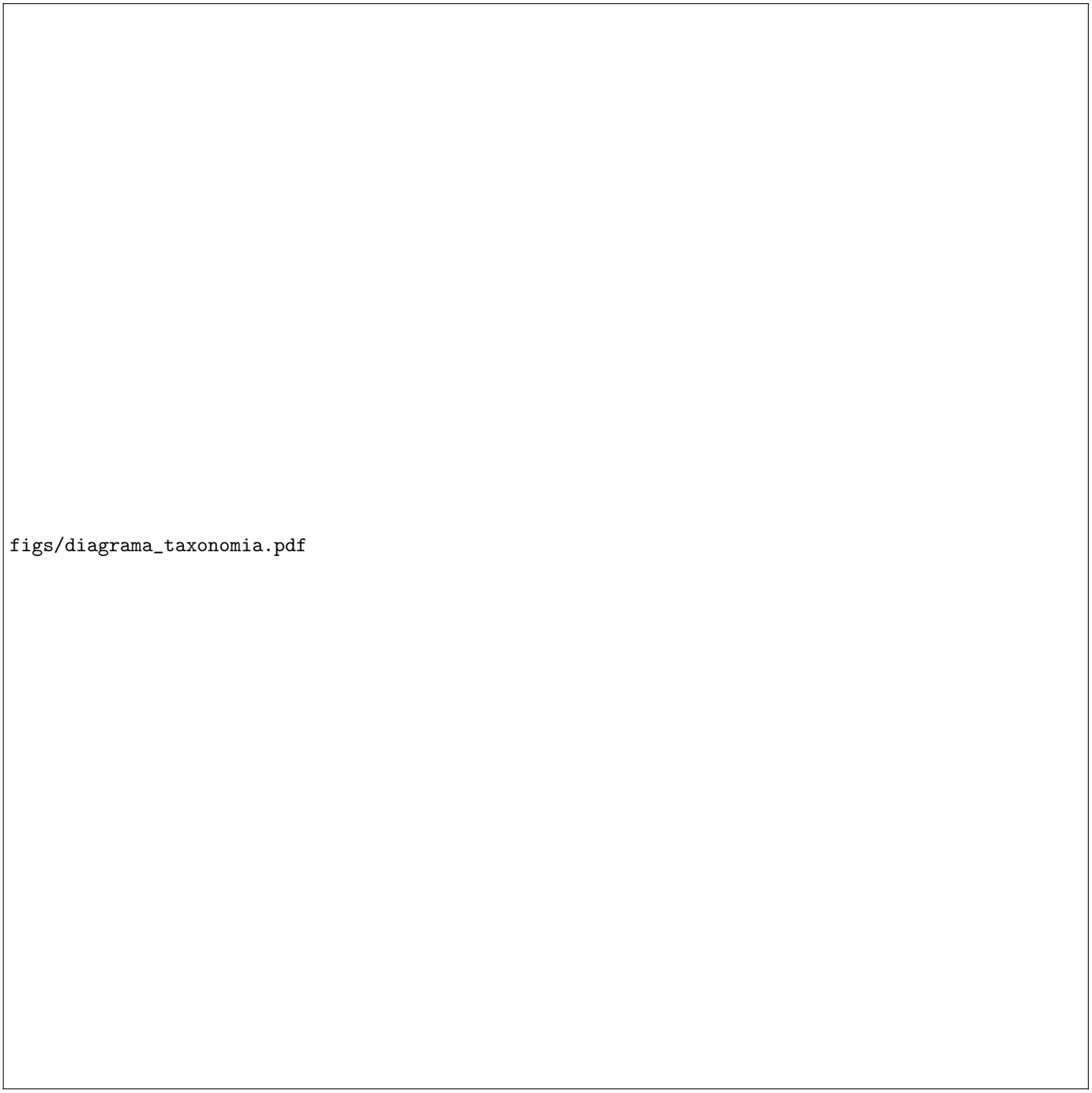

Figura 3.1: Taxonomía de ataques móviles

3.2. Requerimientos del Proyecto

A partir de la taxonomía de ataques móviles presentada en la sección anterior, se realizó un análisis sistemático para determinar qué funcionalidades son necesarias para reproducir cada categoría en un entorno de cyber range. El proceso siguió un enfoque estructurado: para cada una de las siete categorías de la taxonomía, se identificaron tres capacidades técnicas específicas necesarias para reproducir los diferentes tipos de ataques que esa categoría engloba. Este enfoque permite descomponer cada categoría amplia en capacidades técnicas concretas y medibles, facilitando la definición de requerimientos funcionales precisos. Este análisis sirvió como base para definir los requerimientos funcionales del proyecto.

3.2.1. De la Taxonomía a los Requerimientos

Para cada categoría de la taxonomía, se identificaron las capacidades técnicas necesarias para su reproducción:

1. Aplicación y Binarios

- **Instalación y Modificación:** Capacidad de instalar archivos APK desde una fuente externa (no una tienda) y reinstalar versiones modificadas.
- **Extracción de Binarios:** Capacidad de extraer el archivo APK de una aplicación ya instalada para su análisis (ingeniería inversa).
- **Análisis en Ejecución:** Capacidad de adjuntar herramientas de instrumentación a un proceso en ejecución para inspeccionar o modificar su comportamiento.

2. Autenticación y Autorización

- **Interacción con Interfaz Gráfica:** Capacidad de visualizar y manipular la interfaz de usuario para probar flujos de autenticación.
- **Manipulación de Tokens y Sesiones:** Capacidad de acceder y modificar tokens de autenticación almacenados en el dispositivo para evadir controles de autorización o suplantar identidades de usuario.
- **Análisis de Lógica de Autorización:** Capacidad de inspeccionar y modificar la lógica de control de acceso mediante herramientas de análisis estático y dinámico.

3. Comunicación

- **Interceptación de Tráfico de Red:** Capacidad de capturar y analizar el tráfico de red entre la aplicación y servicios externos.

- **Manipulación de Certificados:** Capacidad de instalar certificados personalizados y modificar la validación de certificados SSL/TLS para realizar ataques Man-in-the-Middle (MITM).
- **Análisis de Protocolos:** Capacidad de inspeccionar y modificar paquetes de red para identificar vulnerabilidades en la implementación de protocolos de comunicación.

4. Almacenamiento de Datos

- **Acceso al Sistema de Archivos:** Capacidad de explorar y acceder al sistema de archivos del dispositivo para examinar cómo y dónde se almacena información sensible.
- **Análisis de Bases de Datos:** Capacidad de acceder y examinar bases de datos locales para identificar datos almacenados sin cifrado adecuado.
- **Extracción de Datos Sensibles:** Capacidad de extraer y analizar credenciales, tokens y otros datos sensibles almacenados en el dispositivo.

5. Privacidad

- **Monitoreo de Acceso a Recursos:** Capacidad de monitorear qué permisos y recursos del sistema (ubicación, cámara, micrófono, contactos) accede una aplicación.
- **Análisis de Fugas de Información:** Capacidad de detectar y analizar fugas de información personal a través de logs, tráfico de red o almacenamiento local.
- **Análisis de Recolección de Datos:** Capacidad de detectar y analizar qué datos personales (identificadores de dispositivo, ubicación, información del usuario) recolecta una aplicación y cómo los transmite o almacena.

6. Ecosistema y Cadena de Suministro

- **Análisis de Dependencias:** Capacidad de examinar las librerías y componentes de terceros incluidos en una aplicación para identificar vulnerabilidades en la cadena de suministro.
- **Instalación desde Fuentes Externas:** Capacidad de instalar aplicaciones desde fuentes no oficiales para simular escenarios de distribución comprometida.
- **Análisis de Configuración del Sistema:** Capacidad de examinar y modificar configuraciones del sistema operativo que puedan ser explotadas por aplicaciones maliciosas.

7. Acceso Físico

- **Explotación de Pantalla de Bloqueo:** Capacidad de interactuar físicamente con la pantalla del dispositivo para explotar vulnerabilidades en los mecanismos de bloqueo y obtener acceso no autorizado al sistema.
- **Inyección de Comandos por USB:** Capacidad de inyectar comandos o datos directamente a través de una conexión USB física, incluyendo acceso a puertos de depuración y modos especiales del dispositivo.
- **Manipulación de Hardware:** Capacidad de realizar modificaciones físicas al dispositivo, incluyendo extracción de componentes de memoria, manipulación de elementos del hardware, y técnicas de compromiso mediante *jailbreak* o *rooting* que requieren acceso físico directo.

Con base en este análisis, se definió como objetivo de diseño cubrir la mayor cantidad posible de categorías de la taxonomía, logrando así representar un amplio espectro de ataques del ecosistema móvil.

3.2.2. Requerimientos Funcionales (RF)

Los requerimientos funcionales especifican las capacidades que la solución debe proporcionar para alcanzar el objetivo de máxima cobertura de la taxonomía. Cada requerimiento responde directamente a las necesidades identificadas en el análisis anterior.

RF1 - Emulación de Dispositivos Móviles

- **Descripción:** El sistema debe proveer al menos un dispositivo móvil emulado por laboratorio, con el sistema operativo instalado, configuración de red funcional y servicios de acceso expuestos.
- **Justificación:** Es un requisito transversal que habilita todas las categorías de la taxonomía. Sin un dispositivo móvil disponible, no es posible reproducir ningún ataque del ecosistema móvil.

RF2 - Acceso Gráfico

- **Descripción:** Se debe proveer acceso a la interfaz gráfica del dispositivo emulado, permitiendo al usuario ver e interactuar con la pantalla del dispositivo.
- **Justificación:** Muchos ataques requieren interacción visual con la aplicación: probar flujos de autenticación, detectar suplantación de interfaz, o navegar por la aplicación como lo haría un usuario final.

RF3 - Control por Línea de Comandos

- **Descripción:** Se debe garantizar acceso completo al dispositivo a través de una interfaz de línea de comandos. Esto incluye la capacidad de instalar aplicaciones, modificar archivos del sistema y controlar el dispositivo.

- **Justificación:** Es el requerimiento con mayor impacto en la cobertura, habilitando prácticamente todas las categorías: permite instalar aplicaciones modificadas, explorar el sistema de archivos, configurar proxies, manipular credenciales y automatizar análisis. Es la herramienta principal para la interacción técnica con el dispositivo.

RF4 - Soporte de Múltiples Versiones del Sistema

- **Descripción:** La solución debe ser capaz de soportar múltiples versiones del sistema operativo móvil, incluyendo diferentes variantes y configuraciones del mismo.
- **Justificación:** Aumenta significativamente la variedad de escenarios posibles. Las vulnerabilidades y mecanismos de seguridad evolucionan entre versiones del sistema operativo, por lo que poder elegir la versión específica permite recrear amenazas históricas, comparar evolución de protecciones y adaptar laboratorios a contextos específicos.

RF5 - Instalación y Uso de Herramientas de Análisis

- **Descripción:** El sistema debe permitir la instalación y uso de herramientas externas de análisis y seguridad móvil dentro del entorno del laboratorio.
- **Justificación:** Muchas de las capacidades técnicas identificadas requieren herramientas especializadas para su implementación. El análisis en ejecución, la interceptación de tráfico, el análisis de binarios y otras técnicas de seguridad móvil dependen de herramientas específicas que deben estar disponibles en el entorno del laboratorio.

3.2.3. Requerimientos No Funcionales (RNF)

Además de las capacidades funcionales, la solución debe cumplir con restricciones técnicas y arquitecturales que aseguran su viabilidad dentro del ecosistema de Tectonic y garantizan su operación eficiente en un entorno de producción.

RNF1 - Orquestación Declarativa

- **Descripción:** El despliegue debe realizarse exclusivamente a través de archivos de configuración y playbooks de Ansible, sin intervención manual.
- **Justificación:** Esto garantiza la reproducibilidad, consistencia entre escenarios y alineación con la filosofía de Infraestructura como Código (IaC) de Tectonic. Es una restricción arquitectural fundamental de la plataforma.

RNF2 - Rendimiento y Uso Eficiente de Recursos

- **Descripción:** Cada instancia del emulador debe ofrecer un rendimiento fluido y estable, utilizando de forma eficiente los recursos disponibles (CPU, memoria y almacenamiento).

- **Justificación:** Garantiza que varios laboratorios puedan ejecutarse de manera simultánea sin afectar la estabilidad ni el desempeño global de la infraestructura de Tectonic. Es crítico para la escalabilidad de la plataforma.

RNF3 - Independencia de Plataforma

- **Descripción:** La solución debe ser compatible con las diferentes plataformas de infraestructura que soporta Tectonic: Libvirt, Docker y proveedores de servicios en la nube como AWS.

- **Justificación:** La independencia de plataforma asegura que los laboratorios de seguridad móvil puedan ejecutarse consistentemente sin importar la infraestructura subyacente, maximizando la portabilidad y flexibilidad de los laboratorios.

3.2.4. Cobertura de la Taxonomía

Para demostrar que los requerimientos funcionales definidos son suficientes para cubrir las capacidades técnicas necesarias, se realizó un análisis sistemático de mapeo entre cada capacidad técnica identificada en la sección anterior y los requerimientos funcionales que la habilitan. Este proceso consistió en evaluar, para cada capacidad técnica de cada categoría de la taxonomía, qué combinación de requerimientos funcionales (RF1-RF5) permite su implementación práctica.

Cabe destacar que RF1 (Emulación de Dispositivos Móviles) es un requisito base transversal que está implícito en todas las capacidades técnicas, ya que sin un dispositivo emulado no es posible implementar ninguna de ellas. Por esta razón, RF1 no aparece como columna en la Tabla 3.1, que se enfoca en mostrar los requerimientos adicionales específicos necesarios para cada capacidad técnica.

La Tabla 3.1 presenta los resultados de este análisis de mapeo. Cada fila corresponde a una capacidad técnica específica, y cada columna RF indica si ese requerimiento es necesario para implementar la capacidad. Los símbolos utilizados son: ✓ indica que el requerimiento es necesario y suficiente para habilitar la capacidad, los espacios en blanco indican que ese requerimiento funcional no es necesario para implementar esa capacidad técnica específica, ∼ indica que el requerimiento proporciona soporte parcial pero no completo, permitiendo una simulación limitada de la capacidad técnica, y — indica que la capacidad no puede ser implementada en un entorno de emulación de software.

Para ilustrar este razonamiento, consideremos la capacidad técnica “Análisis en Ejecución” de la categoría Aplicación y Binarios. Esta capacidad requiere adjuntar herramientas de instrumentación a un proceso en ejecución. Para implementarla, se necesita RF3 (Control por Línea de Comandos) porque es necesario acceder al dispositivo mediante ADB para adjuntar la herramienta al proceso, y RF5 (Instalación y Uso de Herramientas de Análisis) porque se requiere instalar y ejecutar la herramienta de instrumentación. Por el contrario, no se necesita

RF2 (Acceso Gráfico) porque el análisis en ejecución se realiza mediante comandos y no requiere interacción visual con la interfaz, ni RF4 (Soporte de Múltiples Versiones del Sistema) porque esta capacidad técnica puede implementarse en cualquier versión del sistema operativo sin requerir una versión específica.

El análisis revela que los requerimientos funcionales definidos cubren completamente la mayoría de las capacidades técnicas identificadas. La única excepción es la capacidad de "Manipulación de Hardware"de la categoría Acceso Físico, que requiere interacción directa con componentes físicos del dispositivo que no pueden ser emulados.

En síntesis, estos requerimientos y su relación con la taxonomía definen los criterios que guiaron la evaluación de alternativas y la selección de la solución presentada en las secciones siguientes.

3.3. Selección de la Plataforma Móvil

Con los requerimientos del proyecto definidos en términos del ecosistema móvil en general, es necesario decidir qué plataforma específica implementar. En el mercado actual, Android e iOS de Apple son los dos sistemas operativos que dominan el ecosistema de dispositivos móviles a nivel mundial, representando en conjunto más del 99 % de la cuota de mercado (StatCounter Global Stats, 2024). Android posee aproximadamente el 72 % del mercado global, mientras que iOS representa el 27 % restante.

Ambas plataformas presentan características distintivas que impactan en su viabilidad para integración en un cyber range:

- **Compatibilidad de emulación:** El emulador de Android funciona en múltiples sistemas operativos Linux, Windows y macOS. El iOS Simulator es exclusivo de macOS y requiere obligatoriamente hardware de Apple, sin alternativas para otros sistemas operativos.
- **Herramientas de análisis y depuración:** Para trabajar con iOS se requiere obligatoriamente un Mac con Xcode (el entorno de desarrollo integrado de Apple, que solo funciona en hardware de Apple), lo que limita las opciones de desarrollo y análisis. Android, en contraste, puede analizarse y depurarse desde cualquier sistema operativo sin requisitos de hardware específico.

Basándonos en los puntos anteriores y bajo el contexto de que Tectonic opera sobre infraestructura basada en Linux, integrar iOS implicaría virtualizar macOS completo —con las complejidades técnicas que esto conlleva—, adquirir y mantener hardware específico de Apple, o rediseñar la arquitectura de despliegue de Tectonic. Cualquiera de estas alternativas escaparía significativamente del alcance de este proyecto y contravendría el principio de compatibilidad con la infraestructura existente.

Por estas razones, Android se presenta como la elección natural: su compatibilidad nativa con Linux, su flexibilidad de integración, y su alineación con la

Tabla 3.1: Cobertura de las capacidades técnicas de la taxonomía mediante los requerimientos funcionales

Categoría	**Capacidad técnica**	**RF2**	**RF3**	**RF4**	**RF5**
Aplicación y Binarios	Instalación y Modificación		✓		
	Extracción de Binarios		✓		✓
	Análisis en Ejecución		✓		✓
Autenticación y Autorización	Interacción con Interfaz Gráfica	✓			
	Manipulación de Tokens y Sesiones		✓		✓
	Análisis de Lógica de Autorización		✓		✓
Comunicación	Interceptación de Tráfico de Red		✓		✓
	Manipulación de Certificados		✓		✓
	Análisis de Protocolos		✓		✓
Almacenamiento de Datos	Acceso al Sistema de Archivos		✓		
	Análisis de Bases de Datos		✓		✓
	Extracción de Datos Sensibles		✓		✓
Privacidad	Monitoreo de Acceso a Recursos	✓			∼
	Análisis de Fugas de Información		✓		✓
	Análisis de Recolección de Datos		✓		✓
Ecosistema y Cadena de Suministro	Análisis de Dependencias		✓	✓	✓
	Instalación desde Fuentes Externas		✓		
	Análisis de Configuración del Sistema		✓		
Acceso Físico	Explotación de Pantalla de Bloqueo	∼			
	Inyección de Comandos por USB		✓		
	Manipulación de Hardware	—	—	—	—

Leyenda: ✓ = cubierto; ∼ = parcialmente cubierto; — = no cubierto.

arquitectura de Tectonic lo convierten en la opción técnicamente viable dentro del alcance del proyecto.

3.4. Alternativas de Solución

Con los requerimientos funcionales y no funcionales definidos, el desafío principal fue encontrar una solución de emulación de Android que los satisficiera. Para asegurar la toma de la mejor decisión, se exploraron y compararon tres enfoques diferentes, analizando en cada caso su documentación técnica y realizando pruebas de concepto para cada una. La solución implementada es el resultado de este análisis, dado que las demás opciones presentaron limitaciones que comprometían la reproducibilidad o la escalabilidad que requiere el laboratorio.

3.4.1. Contenedores Docker

La primera opción de análisis se centró en el uso de contenedores Docker. Esta elección fue natural, ya que Tectonic se apoya en contenedores para su ciclo de desarrollo. Las soluciones evaluadas, como las imágenes `budtmo/docker-android` (budtmo, 2024) y `HQarroum/docker-android` (HQarroum, 2024), consistían internamente en una instalación ligera de Linux que ejecutaba un emulador de Android con una versión predefinida. Este contenedor, además de la instalación base de Linux y el propio emulador de Android, incluía herramientas adicionales como un servidor para la visualización remota de la interfaz gráfica y, en algunos casos, un proxy para Android Debug Bridge (ADB), lo que permitía acceder y controlar el emulador Android de forma remota.

Ventajas.

Esta aproximación presenta varias ventajas significativas en el contexto de un entorno basado en contenedores. Por un lado, la integración y facilidad de orquestación: al estar alineadas con la filosofía de Infraestructura como Código de Tectonic, estas soluciones ofrecen una integración directa que facilita la reproducibilidad de los escenarios. Por otro lado, los contenedores son inherentemente ligeros y portables, ofreciendo una vía rápida para generar y distribuir réplicas de los entornos de laboratorio.

Limitaciones.

Aunque Tectonic utiliza contenedores Docker para su ciclo de desarrollo, en el contexto de un cyber range de producción, el modelo de aislamiento de contenedores no garantiza la separación de recursos ni el aislamiento estricto necesarios para los escenarios de entrenamiento, comprometiendo la integridad del laboratorio. Además, el emulador de Android requiere el uso de aceleración por hardware —capacidad del sistema para utilizar la GPU en lugar de la CPU— para lograr un buen rendimiento. La activación de aceleración por hardware en

contenedores presenta limitaciones de estabilidad y configuración que impiden que el contenedor acceda a las extensiones del procesador necesarias. Esto fuerza al emulador a un modo de emulación por software, resultando en un bajo rendimiento.

3.4.2. Android-x86 en Máquina Virtual

Se exploró la instalación de una imagen de `Android-x86` (Android-x86 Project, 2024) directamente dentro de una máquina virtual. `Android-x86` es un proyecto que permite portar el sistema operativo Android para que se ejecute de forma nativa en máquinas con arquitectura x86, resolviendo así el problema de rendimiento de la emulación por software. Al ejecutarse directamente como una máquina virtual, esta solución evita capas de virtualización adicionales como las que requiere Docker, y típicamente corre una versión optimizada de Android en la arquitectura x86 con una interfaz tipo tablet.

Ventajas.

Ejecutar `Android-x86` directamente sobre el hipervisor permite acceder a la aceleración por hardware. Este acceso directo al procesador huésped garantiza un rendimiento fluido que, con poca latencia, ofrece una experiencia de usuario similar a la de un dispositivo físico. Esta solución también proporciona control completo sobre el sistema, incluyendo la conexión y depuración mediante la herramienta Android Debug Bridge (ADB). Para visualizar la interfaz gráfica de forma remota, se utiliza `scrcpy` (Genymobile, 2024), una herramienta ligera que proyecta la pantalla del dispositivo a través de la conexión ADB, proporcionando baja latencia en la transmisión de video.

Limitaciones.

A pesar del buen rendimiento, esta solución está limitada en cuanto a las versiones de Android funcionales para este proyecto (RF4). `Android-x86` solo permite usar imágenes que vienen precompiladas por terceros para la arquitectura x86, lo cual complica la tarea de elegir y automatizar el uso de versiones específicas dependiendo del escenario. Adicionalmente, esta solución ofrece menos flexibilidad para configurar parámetros específicos del emulador, como resolución, características de hardware virtual o configuraciones de red.

3.4.3. Máquina Virtual con Emulador Android y Virtualización de Display

Esta solución utiliza el SDK oficial de Android instalado dentro de una máquina virtual base.

A diferencia de las soluciones anteriores, no se emplea una imagen precompilada ni un contenedor existente, sino que el emulador se instala y configura programáticamente durante la creación del laboratorio. Esta capacidad permite

que las herramientas se instalen y configuren dinámicamente, permitiendo definir la versión exacta del sistema operativo Android a utilizar y garantizando un entorno completamente reproducible y adaptable a distintos escenarios.

Ventajas.

La principal ventaja de esta solución es el control directo sobre el emulador y el entorno operativo. Gracias a la configuración de la máquina virtual, el sistema descarga e instala las herramientas y las versiones de Android directamente desde los repositorios oficiales de Google. Esta capacidad de poder elegir las imágenes oficiales garantiza el cumplimiento del requerimiento Soporte de Múltiples Versiones del Sistema (RF4). Además, el SDK de Android incluye la herramienta Android Debug Bridge (ADB), permitiendo acceder al dispositivo emulado mediante la terminal y manipularlo, cumpliendo así con el RF3. En esta solución, el acceso gráfico se logra mediante la implementación de un conjunto de herramientas de virtualización de pantalla, lo cual permite visualizar la interfaz del emulador y, a su vez, interactuar con el emulador de forma fluida, lo que satisface directamente el RF2.

Limitaciones.

La principal limitación de esta solución se origina en que el emulador se ejecuta dentro de una máquina virtual, lo que introduce un costo computacional adicional en comparación con la ejecución directa sobre hardware, ya que el emulador debe ejecutarse sobre una capa de virtualización adicional. Por otro lado, esta solución requiere una configuración más compleja que las alternativas anteriores, involucrando múltiples pasos de instalación y configuración de dependencias.

3.5. Arquitectura de la Solución Propuesta

Tras evaluar las tres alternativas presentadas en la sección anterior, se selecciona la tercera opción: el SDK oficial de Android ejecutándose dentro de una máquina virtual con virtualización de display. Esta solución cumple con todos los requerimientos funcionales identificados —emulación de dispositivos (RF1), acceso gráfico (RF2), control por línea de comandos (RF3), soporte de múltiples versiones (RF4) e instalación de herramientas de análisis (RF5)— y supera las limitaciones observadas en las otras alternativas, ofreciendo el equilibrio adecuado entre fidelidad, flexibilidad y control programático.

La arquitectura propuesta se integra en el ciclo de vida de Tectonic aprovechando sus capacidades de orquestación y configuración.

La solución se despliega mediante un proceso automatizado que, partiendo de una definición de escenario, crea y configura una máquina virtual con las capacidades necesarias para ejecutar el emulador de Android con un rendimiento adecuado. A continuación, se instala y configura el emulador Android según las especificaciones del escenario, permitiendo seleccionar la versión exacta del

sistema operativo requerida. Posteriormente, se configura el acceso remoto mediante visualización gráfica (VNC) y control por línea de comandos (ADB), habilitando la interacción con el dispositivo emulado sin requerir acceso directo a la máquina virtual. El requerimiento de instalación de herramientas de análisis (RF5) se cumple aprovechando las capacidades de automatización que Tectonic ya proporciona, permitiendo instalar cualquier herramienta necesaria (como mitmproxy u otras) durante el despliegue del escenario. Una vez completados estos pasos, el entorno queda listo para su uso.

La arquitectura está compuesta por tres componentes principales que trabajan en conjunto para proporcionar un entorno de emulación Android funcional. El emulador de Android es el núcleo de la solución, proporcionando la ejecución de dispositivos Android virtuales con todas las capacidades necesarias para reproducir escenarios de seguridad. La visualización remota permite exponer la interfaz gráfica del emulador de forma remota, habilitando la interacción visual con el dispositivo emulado sin requerir acceso directo a la máquina virtual. Por último, un sistema de gestión de servicios coordina el inicio y monitoreo de los componentes, garantizando un arranque ordenado, el manejo adecuado de dependencias entre servicios y la capacidad de reinicio automático ante fallos, asegurando así la disponibilidad y estabilidad del entorno de laboratorio.

Esta arquitectura permite que cada escenario de laboratorio defina sus propios requisitos —versión de Android, características del dispositivo, herramientas de análisis necesarias— y el sistema genere automáticamente un entorno completamente funcional y reproducible que satisface esos requisitos, cumpliendo así con los objetivos de flexibilidad y reproducibilidad establecidos para el proyecto.

La Figura 3.2 ilustra la arquitectura de componentes de la solución, mostrando los elementos principales y sus interconexiones.

Los detalles técnicos de implementación de esta arquitectura, incluyendo la configuración específica de los componentes, los mecanismos de automatización y la integración con Tectonic, se presentan en detalle en el siguiente capítulo.

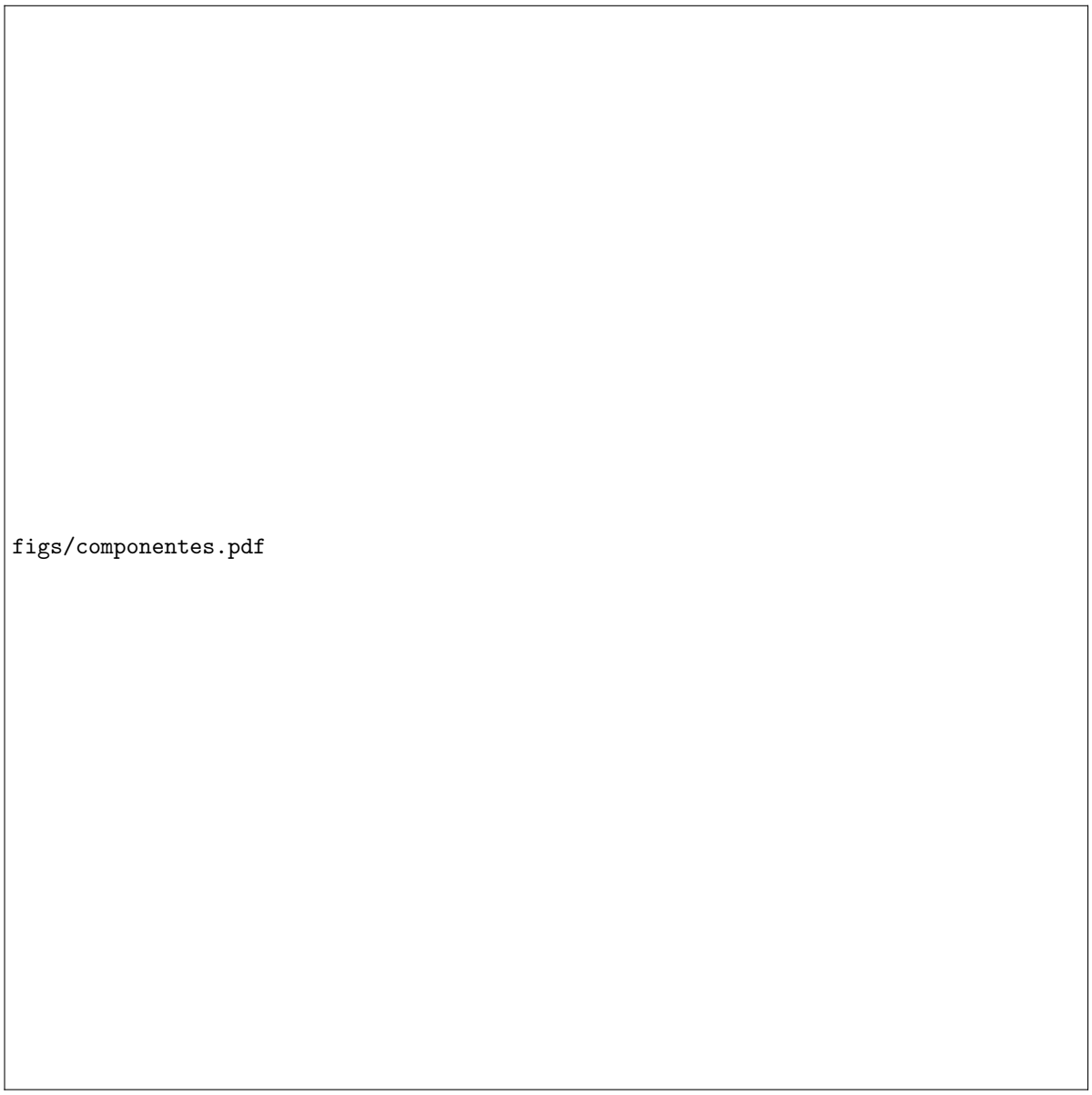

Figura 3.2: Componentes de la solución.

Capítulo 4

Implementación

Este capítulo detalla los aspectos técnicos de la implementación de la solución diseñada en el capítulo anterior para integrar dispositivos Android emulados en Tectonic.

El capítulo se organiza en dos partes principales que reflejan la arquitectura modular de la solución. En primer lugar, se presenta la **solución base de emulación Android**, una implementación independiente y portable que consta del playbook de Ansible, la configuración de virtualización anidada, y la parametrización del entorno. Esta solución puede ejecutarse en cualquier máquina virtual sin depender de Tectonic. Posteriormente, se describe la **integración con el ecosistema de Tectonic**, explicando cómo la solución base se orquesta mediante los archivos de configuración de la plataforma y se integra en su ciclo de vida.

4.1. Implementación del Entorno de Emulación Android

El núcleo de la solución es el playbook de Ansible `android.yml`. Este artefacto transforma una máquina virtual genérica en un entorno de emulación Android funcional de manera automática. El playbook ejecuta una secuencia de tareas idempotentes, garantizando que el estado final sea siempre el mismo sin importar cuántas veces se ejecute. El playbook completo se puede consultar en el Anexo A.

4.1.1. Instalación de Dependencias del Sistema

La fase inicial garantiza la presencia de todas las dependencias de software mediante el uso del módulo `ansible.builtin.apt`. Las tareas correspondientes actualizan primero la caché de paquetes y luego instalan los siguientes componentes necesarios para el funcionamiento del emulador:

- `openjdk-21-jdk`: Entorno de ejecución Java requerido por las herramientas del SDK de Android.
- `xvfb`: Servidor de visualización en memoria para crear displays virtuales.
- `x11vnc`: Servidor VNC para exponer el display virtual de forma remota.
- `libpulse0`: Librería de audio necesaria para el emulador.
- `libgl1-mesa-dev`: Librerías de OpenGL necesarias para el renderizado gráfico.
- `unzip`: Utilidad para descomprimir archivos ZIP del SDK.

```
- name: "APT update"
  become: true
  ansible.builtin.apt:
    update_cache: yes
    cache_valid_time: 3600

- name: "Instalar dependencias"
  become: true
  ansible.builtin.apt:
    name:
      - openjdk-21-jdk
      - x11vnc
      - unzip
      - xvfb
      - libpulse0
      - libgl1-mesa-dev
    state: present
```

Listado 4.1: Instalación de dependencias del sistema

4.1.2. Instalación del Android SDK

Una vez instaladas las dependencias del sistema, se procede con la configuración del SDK oficial de Android. El playbook descarga el Android Command Line Tools desde los repositorios oficiales de Google, descomprime el archivo ZIP y organiza la estructura de directorios en `cmdline-tools/latest`, que es el formato requerido por las herramientas.

Para que las herramientas del SDK sean accesibles, se configuran variables de entorno permanentes mediante el módulo `ansible.builtin.blockinfile`. Estas variables se añaden al archivo `.bashrc` del usuario, incluyendo `ANDROID_SDK_ROOT`, `ANDROID_HOME`, `ANDROID_AVD_HOME`, y la actualización de la variable `PATH` con las rutas necesarias.

Un paso crítico para una ejecución desatendida es la aceptación no interactiva de las licencias del SDK. En lugar de un proceso manual, se utiliza el módulo `ansible.builtin.shell` para automatizar la respuesta afirmativa:

```yaml
- name: "Aceptar las licencias de Android SDK"
  ansible.builtin.shell:
    cmd: "yes | {{ android_sdk_root }}/cmdline-tools/latest/bin/
    sdkmanager --licenses"
  environment: "{{ sdk_environment }}"
  args:
    chdir: "{{ android_base_dir }}"
  register: license_acceptance_result
  changed_when: >
    ('All SDK package licenses accepted' not in
    license_acceptance_result.stdout and
     'licenses were accepted' not in license_acceptance_result.
    stderr)
  failed_when: >
    (license_acceptance_result.rc != 0 and
    'All SDK package licenses accepted' not in
    license_acceptance_result.stdout and
    'licenses were accepted' not in license_acceptance_result.
    stderr)
```

Listado 4.2: Aceptación automática de licencias del Android SDK

Como se observa, se define una condición de fallo personalizada (`failed_when`) para prevenir falsos negativos cuando las licencias ya estaban aceptadas.

Finalmente, se invoca a `sdkmanager` para instalar los componentes del emulador, como `platform-tools` y la imagen del sistema seleccionada. El uso del argumento `creates` en las tareas de Ansible asegura la idempotencia: la descarga solo se ejecuta si el componente objetivo no existe, evitando operaciones redundantes en ejecuciones posteriores del playbook.

```yaml
- name: "Instalar platform-tools, emulator, e imagen del sistema"
  ansible.builtin.shell:
    cmd: >
      sdkmanager
      "platform-tools"
      "emulator"
      "{{ system_image_package }}"
  environment: "{{ sdk_environment }}"
  args:
    creates: "{{ android_sdk_root }}/platforms/android-{{ api_level
     }}"
```

Listado 4.3: Instalación idempotente de componentes del emulador

4.1.3. Creación del Dispositivo Virtual

Una vez configurado el SDK y los componentes necesarios, se invoca a `avdmanager` para crear el Android Virtual Device (AVD). Se utiliza la opción `--force` para asegurar que cada despliegue comience desde un estado limpio. La pregunta interactiva sobre perfiles de hardware se responde automáticamente:

```yaml
- name: Crear Android Virtual Device (AVD)
  ansible.builtin.shell:
    cmd: |
      echo "no" | avdmanager create avd \
```

```
        --name "{{ avd_name }}" \
        --package "{{ system_image_package }}" \
        --force
```

Listado 4.4: Creación automatizada del Android Virtual Device

4.1.4. Configuración de Permisos

Una vez creado el AVD, se configuran los permisos necesarios para que el emulador pueda ejecutarse correctamente. La configuración de permisos incluye dos componentes principales. En primer lugar, el usuario de Android es añadido al grupo del sistema `kvm` (Kernel-based Virtual Machine), que gestiona el acceso al módulo de kernel que expone las capacidades de virtualización del procesador. Esta operación es crítica porque sin acceso a KVM, el emulador Android no puede utilizar aceleración por hardware y caería en modo de emulación por software, resultando en un bajo rendimiento.

En segundo lugar, se ajustan los permisos de los archivos del AVD creado mediante el módulo `ansible.builtin.file`. Específicamente, se establecen los permisos de propiedad sobre dos componentes: el archivo `.ini` que contiene la configuración del dispositivo virtual, y el directorio `.avd` que almacena los archivos de datos persistentes del emulador (snapshots, imágenes de disco, etc.). Ambos son asignados al usuario de Android con permisos de lectura para el grupo y otros, garantizando que el emulador pueda acceder a sus archivos sin restricciones de permisos.

```
- name: "Add user '{{ android_user }}' to group kvm"
  ansible.builtin.user:
    name: '{{ android_user }}'
    groups: kvm
    append: yes

- name: "Cambiar permisos del .ini AVD"
  become: true
  ansible.builtin.file:
    path: "{{ android_sdk_root }}/{{ avd_name }}.ini"
    owner: "{{ android_user }}"
    group: "{{ android_user }}"
    mode: '0644'

- name: "Cambiar permisos del .avd AVD"
  become: true
  ansible.builtin.file:
    path: "{{ android_sdk_root }}/{{ avd_name }}.avd"
    state: directory
    owner: "{{ android_user }}"
    group: "{{ android_user }}"
    recurse: yes
    mode: '0755'
```

Listado 4.5: Configuración de permisos del AVD y grupo KVM

4.1.5. Configuración de Virtualización de Display

Para permitir el acceso gráfico remoto al emulador Android, se implementa un sistema de virtualización de display basado en dos componentes: Xvfb (X Virtual Framebuffer) y x11vnc (LibVNC, 2024).

X11 es el sistema de ventanas usado en sistemas Unix (como Linux) que permite que las aplicaciones rendericen su interfaz gráfica en un display, que puede ser un monitor físico o virtual. Los displays en X11 son abstracciones que representan un conjunto de dispositivos de entrada y salida. Xvfb (X Virtual Framebuffer) es un servidor de visualización en memoria para sistemas tipo Unix que permite ejecutar aplicaciones gráficas sin necesidad de un monitor físico. Xvfb crea un display virtual identificado por un número (en este caso, el display número 2, configurado mediante la variable `display_num`), que simula un monitor físico con dimensiones y profundidad de color específicas. La resolución 1080x1920x24 representa 1080 píxeles de ancho por 1920 de alto con 24 bits de profundidad de color, emulando así las características típicas de un smartphone moderno. El emulador Android renderiza su interfaz gráfica en este display virtual, que actúa como si fuera la pantalla del dispositivo móvil.

Para exponer este display a través de la red, se utiliza x11vnc, un servidor VNC (Virtual Network Computing) que toma el display virtual creado por Xvfb y lo expone mediante el protocolo RFB (Remote Frame Buffer). El protocolo RFB es un protocolo de red que permite el acceso y control remoto de una interfaz gráfica de usuario. VNC es un protocolo de visualización remota ampliamente utilizado que permite compartir el escritorio completo de un sistema con clientes remotos. Los usuarios pueden conectarse al servidor mediante cualquier cliente VNC especificando el puerto configurado (por defecto 5902) y la contraseña definida en las variables de configuración.

La configuración del sistema de virtualización de display se parametriza mediante las siguientes variables:

```
# --- Configuracion de Display Virtual ---
display_num: 2
display_res: "1080x1920x24"

# --- Configuracion de Servidor VNC ---
vnc_port: 5902
vnc_pass: "1234"
```

Listado 4.6: Variables de configuración de virtualización de display

La variable `display_num` especifica el número de display virtual, mientras que `display_res` define las dimensiones y profundidad de color. Por su parte, `vnc_port` establece el puerto TCP donde x11vnc escuchará conexiones, y `vnc_pass` define la contraseña de autenticación requerida para acceder.

4.1.6. Configuración de Servicios

Para asegurar un arranque robusto y resiliente del entorno de emulación, se definen tres servicios `systemd` gestionados por Ansible. `systemd` es el sistema de inicialización y gestión de servicios en la mayoría de las distribuciones

modernas de Linux. Permite definir servicios como unidades de configuración que se inician automáticamente en el arranque del sistema y se gestionan de forma centralizada, proporcionando control sobre dependencias, políticas de reinicio y monitoreo de estado. Las unidades de servicio establecen dependencias explícitas entre los componentes y configuran políticas de reinicio automático (`Restart=on-failure`) para garantizar la estabilidad del entorno.

La arquitectura de servicios sigue una cascada de dependencias donde cada componente requiere que los anteriores estén activos. El primer servicio en arrancar es `xvfb.service`, que inicia el display virtual en el display número configurado con las dimensiones especificadas. Una vez configurado, introduce un delay de 2 segundos para asegurar que el display esté listo:

```
- name: "Install Xvfb service file"
  ansible.builtin.copy:
    content: |
      [Unit]
      Description=Run Xvfb on {{ display_num }}

      [Service]
      Type=exec
      ExecStart=Xvfb :{{ display_num }} -screen 0 {{ display_res }}
      ExecStartPost=sleep 2

      [Install]
      WantedBy=default.target
    dest: /etc/systemd/system/xvfb.service
```

Listado 4.7: Configuración del servicio Xvfb

Una vez que el display virtual está listo, se inicia `x11vnc.service`, que expone el display a través del protocolo VNC en el puerto configurado. Este servicio incluye una verificación explícita mediante `ExecStartPost` que espera hasta 30 segundos a que el puerto VNC esté efectivamente escuchando conexiones antes de considerar el servicio iniciado:

```
- name: "Install x11vnc service file"
  ansible.builtin.copy:
    content: |
      [Unit]
      Description=Run x11vnc
      After=xvfb.service

      [Service]
      Type=exec
      ExecStart=x11vnc -display :{{ display_num }} -forever -
    rfbport {{ vnc_port }} -passwd {{ vnc_pass }}
      ExecStartPost=/usr/bin/timeout 30 sh -c 'while ! ss -H -t -l
    -n sport = :{{ vnc_port }} | grep -q "^LISTEN.*:{{ vnc_port
    }}"; do sleep 1; done'

      [Install]
      WantedBy=default.target
    dest: /etc/systemd/system/x11vnc.service
```

Listado 4.8: Configuración del servicio x11vnc

Finalmente, `android_emulator.service` se encarga de lanzar el emulador Android. Este servicio establece explícitamente la variable de entorno `DISPLAY` para que el emulador renderice en el display virtual, y se ejecuta con el usuario de Android configurado:

```
- name: "Install Android emulator service file"
  ansible.builtin.copy:
    content: |
      [Unit]
      Description=Run Android Emulator
      After=x11vnc.service

      [Service]
      Type=exec
      WorkingDirectory={{ android_base_dir }}
      User={{ android_user }}

      # Explicit environment variables for robustness
      Environment="ANDROID_SDK_ROOT={{ android_sdk_root }}"
      Environment="ANDROID_HOME={{ android_home }}"
      Environment="ANDROID_AVD_HOME={{ android_sdk_root }}"
      Environment="ANDROID_SDK_HOME=/home/{{ android_user }}"
      Environment="PATH={{ android_sdk_root }}/cmdline-tools/latest
    /bin:{{ android_sdk_root }}/platform-tools:{{ android_sdk_root
    }}/emulator:/usr/bin:/bin"
      Environment="DISPLAY=:{{ display_num }}"

      ExecStart={{ android_sdk_root }}/emulator/emulator -avd {{
    avd_name }} {{ emulator_options }}
      Restart=on-failure

      [Install]
      WantedBy=default.target
    dest: /etc/systemd/system/android_emulator.service
```

Listado 4.9: Configuración del servicio Android Emulator

La política de reinicio `on-failure` garantiza que si cualquier servicio falla, será reiniciado automáticamente por `systemd`, proporcionando resiliencia al sistema completo. Una vez instaladas las unidades, Ansible recarga la configuración de `systemd` mediante `daemon_reload` y habilita los tres servicios para que se inicien automáticamente en el arranque del sistema.

4.1.7. Parametrización del Entorno

Toda la configuración específica del entorno de emulación se encuentra definida en un archivo de variables de Ansible. Este diseño permite modificar por completo el comportamiento del despliegue sin alterar la lógica del *playbook*, separando así la configuración de la implementación.

El archivo incluye variables que cubren múltiples aspectos del despliegue: configuración de directorios y rutas donde se instala el SDK, enlaces de descarga de las herramientas oficiales, versión de Android a emular, perfil de hardware del dispositivo, nombre del AVD, opciones del emulador, y la configuración de

acceso remoto (número de display, resolución, puerto VNC y contraseña). El archivo completo se puede consultar en el Anexo A.

4.2. Virtualización Anidada

Para que el emulador de Android funcione con un rendimiento aceptable, la aceleración por hardware es un requisito indispensable. Dado que el emulador se ejecuta dentro de una máquina virtual, se requiere virtualización anidada. Esta capacidad se habilita aplicando una transformación XSLT (`smartphone.xsl`) sobre la definición XML del dominio de Libvirt antes de su creación.

La transformación XSLT modifica la configuración de la CPU al modo `host-passthrough`, instruyendo a Libvirt para exponer las extensiones de virtualización del procesador físico directamente a la máquina virtual huésped. Este modo permite que la máquina virtual huésped acceda directamente a las capacidades de virtualización (como VT-x o AMD-V) del procesador, posibilitando así que el emulador Android utilice aceleración por hardware para emular el dispositivo de forma eficiente.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL
    /Transform">
    <xsl:output method="xml" indent="yes"/>

    <xsl:template match="@*|node()">
        <xsl:copy>
            <xsl:apply-templates select="@*|node()"/>
        </xsl:copy>
    </xsl:template>

    <xsl:template match="/domain/cpu">
        <cpu mode="host-passthrough" check='none' migratable='on'/>
    </xsl:template>
</xsl:stylesheet>
```

Listado 4.10: Transformación XSLT para habilitar virtualización anidada

La transformación utiliza un patrón común en XSLT: un template de identidad que copia todos los elementos del documento original, y un template específico que coincide con el elemento `<cpu>` y lo reemplaza con la configuración de `host-passthrough`. Esta configuración establece que la máquina virtual reciba todas las características del CPU físico (`mode="host-passthrough"`), deshabilita la verificación estricta de compatibilidad (`check='none'`) y permite que la máquina virtual sea migrable (`migratable='on'`).

4.3. Integración con Tectonic

Como se describió en el Capítulo 2, Tectonic gestiona el ciclo de vida completo de los escenarios mediante Infraestructura como Código. La solución de emulación Android se integra en este ecosistema siguiendo el mismo paradigma,

permitiendo que los instructores definan escenarios con dispositivos Android emulados de manera declarativa y reproducible.

4.3.1. Integración en el ciclo de vida de Tectonic

El despliegue de escenarios en Tectonic sigue un proceso de dos fases, tal como se explicó en el Capítulo 2. Para integrar el dispositivo Android emulado, es fundamental entender en qué momento del ciclo de vida se ejecuta cada componente:

Configuración durante la creación de imágenes base

Durante la fase de **creación de imágenes base** (comando `create-images`), Tectonic utiliza Packer para generar imágenes base que contienen el sistema operativo y las configuraciones iniciales. En este momento se ejecuta el playbook `base_config.yml`, ubicado en el directorio `ansible/` del escenario. Este playbook se ejecuta **una única vez** durante la creación de la imagen y tiene acceso a internet.

Para integrar el emulador Android, el instructor debe incluir o referenciar el playbook `android.yml` desde `base_config.yml`. Esto garantiza que el SDK de Android, el emulador y todos los componentes necesarios se instalen y configuren durante la creación de la imagen base. El Listado 4.11 muestra un ejemplo de cómo integrar el playbook:

```
1 - name: "Include android emulation playbook"
2   ansible.builtin.import_playbook: android.yml
```

Listado 4.11: Ejemplo de `base_config.yml` incluyendo el playbook de emulación Android

Personalización durante el despliegue de instancias

Una vez creadas las imágenes base, Tectonic utiliza Terraform para clonar las instancias del escenario. Después de clonar cada instancia, Tectonic ejecuta automáticamente el playbook `after_clone.yml`, también ubicado en el directorio `ansible/`. Este playbook se ejecuta **para cada instancia clonada** y, por defecto, no tiene acceso a internet, ya que las máquinas están aisladas en la red interna del laboratorio.

Es en esta fase donde los instructores pueden personalizar el dispositivo Android emulado para cada instancia, ya que el emulador estará en ejecución. Las tareas de personalización se agregan a `after_clone.yml` y utilizan ADB para interactuar con el dispositivo emulado.

4.3.2. Definición de la máquina virtual en `description.yml`

Para declarar la máquina virtual que hospedará el emulador Android, el instructor agrega una entrada en `description.yml` especificando los recursos necesarios. El Listado 4.12 muestra la configuración típica:

```yaml
guest_settings:
  smartphone:
    disk: 40
    memory: 8192
    vcpu: 4
```

Listado 4.12: Ejemplo de `description.yml` para un escenario con dispositivo Android

Esta configuración define una máquina virtual con 8 GB de RAM, 4 CPUs virtuales y 40 GB de disco, recursos suficientes para ejecutar el emulador Android con un rendimiento adecuado.

Además de `description.yml`, los instructores necesitan crear un archivo `lab_edition.yml` que especifica parámetros de la ejecución del escenario, como la cantidad de instancias a desplegar, identificadores de la edición y parámetros de acceso. Este archivo se utiliza como parámetro en los comandos de Tectonic (`create-images` y `deploy`).

4.3.3. Personalización del dispositivo emulado mediante ADB

Una vez que el emulador está en ejecución, los instructores pueden personalizar completamente el dispositivo Android utilizando comandos ADB desde playbooks de Ansible. Las siguientes subsecciones muestran ejemplos prácticos de cómo realizar las tareas más comunes.

Instalación de aplicaciones (APKs)

Para instalar aplicaciones en el dispositivo emulado, el instructor agrega tareas al playbook `after_clone.yml` que utilizan ADB. El APK debe estar disponible dentro de la máquina virtual (por ejemplo, copiado mediante tareas previas de Ansible o incluido en la imagen base). Una vez que el archivo está disponible, se puede instalar utilizando su ruta. El Listado 4.13 muestra un ejemplo:

```yaml
- name: "Install APK"
  ansible.builtin.shell: "{{ android_sdk_root }}/platform-tools/adb
     -e install -r /tmp/app-debug.apk"
```

Listado 4.13: Instalación de una aplicación mediante ADB en `after_clone.yml`

La instalación se realiza mediante el comando `adb install` especificando la ruta del APK dentro de la máquina virtual. Los instructores pueden incluir aplicaciones vulnerables, aplicaciones de ejemplo, o cualquier APK necesario para el escenario, asegurándose de que el archivo esté disponible en la máquina virtual antes de la instalación.

Modificación del sistema de archivos

Los instructores pueden realizar modificaciones en el sistema de archivos del dispositivo emulado utilizando comandos ADB. Esto permite crear archivos

de configuración, modificar bases de datos SQLite, o colocar archivos de datos específicos. El Listado 4.14 muestra ejemplos:

```
- name: "Create directory in device"
  ansible.builtin.shell: "{{ android_sdk_root }}/platform-tools/adb
     -e shell mkdir -p /sdcard/lab_data"

- name: "Push file to device"
  ansible.builtin.shell: "{{ android_sdk_root }}/platform-tools/adb
     -e push {{ item.src }} {{ item.dest }}"
  loop:
    - { src: "/tmp/config.json", dest: "/sdcard/lab_data/config.
    json" }
    - { src: "/tmp/database.db", dest: "/sdcard/lab_data/database.
    db" }
```

Listado 4.14: Modificación del sistema de archivos mediante ADB

4.3.4. Creación de escenarios

El proceso completo para crear un escenario con un dispositivo Android emulado se ilustra en la Figura 4.1 y consta de los siguientes pasos:

1. **Definir la máquina virtual:** Agregar la entrada `smartphone` en `description.yml` con los recursos necesarios (memoria, CPU, disco).

2. **Configurar el emulador base:** Crear o modificar `base_config.yml` para incluir el playbook `android.yml`, asegurando que el emulador se instale durante la creación de la imagen base.

3. **Personalizar el dispositivo:** Crear o modificar `after_clone.yml` para agregar las tareas de personalización (instalación de APKs, modificación de archivos, configuración del sistema) que se ejecutarán automáticamente en cada instancia.

4. **Crear imágenes base:** Ejecutar `tectonic <lab_edition.yml> create-images` para generar las imágenes base con el emulador configurado.

5. **Desplegar el escenario:** Ejecutar `tectonic <lab_edition.yml> deploy` para desplegar las instancias. Tectonic ejecutará automáticamente `after_clone.yml` en cada instancia, personalizando el dispositivo según lo especificado.

Adicionalmente, Tectonic proporciona el comando `run-ansible` que permite ejecutar playbooks personalizados en instancias ya desplegadas, facilitando la modificación o actualización de escenarios sin necesidad de recrear las imágenes base.

Figura 4.1: Flujo del proceso de creación y despliegue de escenarios con dispositivos Android emulados en Tectonic.

4.3.5. Acceso del estudiante al dispositivo emulado

Una vez desplegado el escenario, los estudiantes necesitan acceder al dispositivo Android emulado para realizar las actividades del laboratorio. El acceso se realiza a través de la máquina virtual que hospeda el emulador, utilizando las capacidades de acceso remoto de Tectonic.

Acceso gráfico mediante VNC

Para acceder visualmente al emulador Android, los estudiantes utilizan un cliente VNC conectándose al servidor x11vnc configurado en la máquina virtual. Sin embargo, dado que las máquinas virtuales están aisladas en la red interna del laboratorio, el acceso VNC no es directo. En su lugar, los estudiantes establecen un túnel SSH mediante port forwarding a través del punto de entrada (entry point) del escenario, redirigiendo el puerto VNC configurado (por defecto 5902) desde la máquina virtual hacia su máquina local.

Tectonic proporciona acceso SSH para estudiantes mediante usuarios con el formato `traineeXX`, donde `XX` es el número de instancia asignado. Una vez establecido el túnel SSH, el estudiante puede conectarse al emulador utilizando cualquier cliente VNC estándar (por ejemplo, Remmina, TigerVNC o RealVNC) apuntando a `localhost` en el puerto redirigido y utilizando la contraseña VNC configurada en las variables de Ansible durante el despliegue.

Acceso mediante ADB

Para interactuar con el dispositivo Android mediante línea de comandos, los estudiantes acceden mediante SSH a la máquina virtual que hospeda el emulador. El acceso SSH se realiza a través del punto de entrada del escenario, utilizando las credenciales proporcionadas por Tectonic. Una vez conectados a la máquina virtual, los estudiantes tienen acceso completo a las herramientas de ADB instaladas junto con el SDK de Android.

Las herramientas ADB están disponibles en la ruta configurada durante el despliegue (especificada en `android_sdk_root`), permitiendo a los estudiantes realizar todas las operaciones estándar: listar dispositivos conectados, ejecutar comandos en el dispositivo mediante `adb shell`, instalar aplicaciones mediante `adb install`, transferir archivos, ver logs del sistema, y cualquier otra operación que ADB permite. El emulador aparece automáticamente como un dispositivo conectado cuando está en ejecución.

Tectonic proporciona todas las credenciales de acceso necesarias (direcciones IP de las máquinas virtuales, nombres de usuario, contraseñas SSH y contraseñas VNC) mediante el comando `tectonic <lab_edition.yml>info`, que muestra toda la información necesaria para que los estudiantes accedan a sus instancias del escenario de forma independiente.

Actualmente, el equipo de Tectonic está trabajando en agregar soporte a una interfaz web basada en Guacamole que permita acceso SSH, VNC y RDP directo desde un navegador. Una vez implementada, esta funcionalidad permitiría a los

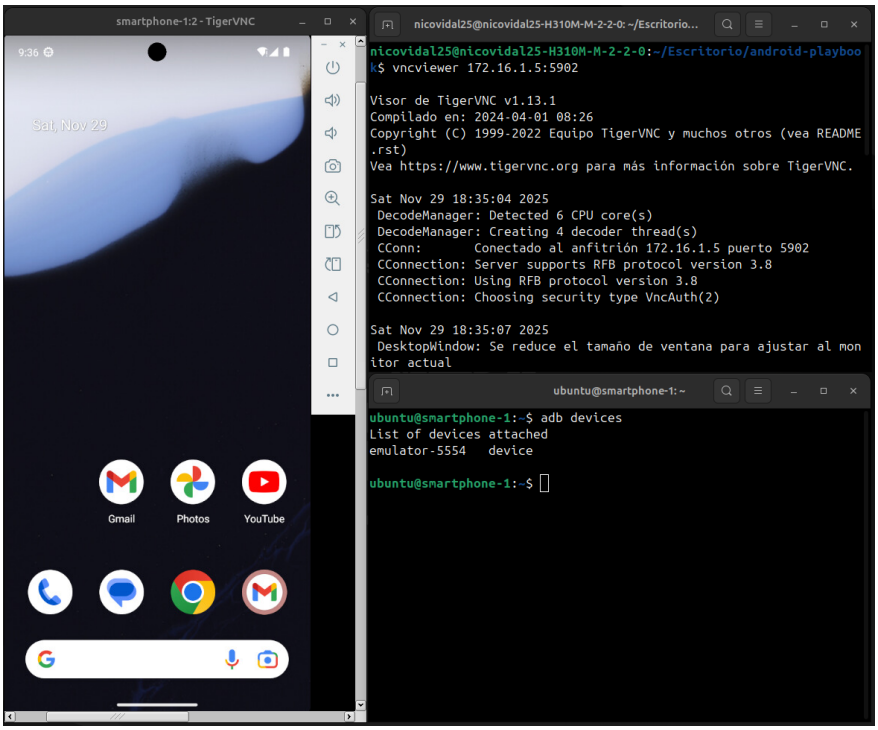

Figura 4.2: Emulador Android en ejecución accesible mediante VNC y verificación de conexión mediante ADB.

estudiantes acceder al emulador sin configurar túneles SSH ni instalar clientes VNC locales, simplificando significativamente el proceso de acceso.

La Figura 4.2 muestra el emulador Android en ejecución accesible mediante VNC y la verificación de ambos métodos de acceso. En la ventana izquierda se observa la interfaz gráfica del dispositivo emulado accesible a través de TigerVNC, mostrando la pantalla de inicio de Android. En la ventana superior derecha se muestra la terminal confirmando la conexión VNC exitosa al servidor x11vnc en la dirección `10.10.1.5:5902`. Finalmente, en la ventana inferior derecha se ejecuta el comando `adb devices`, que detecta correctamente el emulador como `emulator-5554 device`, demostrando que tanto el acceso gráfico (VNC) como el acceso por línea de comandos (ADB) funcionan correctamente y de forma simultánea.

Capítulo 5

Caso de Estudio: Laboratorio de Ejecución Remota de Código

Este capítulo documenta la validación de la solución mediante el diseño y despliegue de un escenario práctico de Ejecución Remota de Código. Se explica la motivación del laboratorio y su encuadre dentro de la taxonomía propuesta, se repasa el fundamento teórico de la vulnerabilidad y se describe la aplicación vulnerable desarrollada. Luego se detallan los componentes del escenario, la automatización del despliegue mediante Tectonic y Ansible, y la ejecución del ataque desde la perspectiva del estudiante. Finalmente, se presentan las consideraciones de mitigación y se documenta la experimentación realizada para validar el cumplimiento de los requerimientos no funcionales de la plataforma mediante mediciones de performance y evaluación de estabilidad y portabilidad.

5.1. Elección del Caso de Estudio

El objetivo era elegir un caso que cubriera al menos dos categorías de la taxonomía definida en el Capítulo 3 y que, al mismo tiempo, exigiera a la plataforma mostrar todas sus capacidades. Se optó por un escenario de Ejecución Remota de Código que, en la práctica, abarca las siguientes categorías:

- **Aplicación y binarios**. El ataque manipula los componentes ejecutables de la aplicación hasta que realiza acciones no autorizadas, mostrando cómo un adversario puede alterar su comportamiento previsto.

- **Comunicación**. La explotación depende de interceptar y modificar el tráfico de red bajo un ataque *Man-in-the-Middle* controlado, lo que evidencia la necesidad de supervisar las comunicaciones del dispositivo.

- **Ecosistema y cadena de suministro**. El ataque se apoya en una dependencia habitual del ecosistema Android cuya falla permite comprometer al dispositivo, ilustrando el impacto de confiar en componentes externos vulnerables.

Así, el escenario funciona como prueba técnica y como primera validación de nuestra taxonomía, demostrando su utilidad para diseñar laboratorios reproducibles centrados en amenazas reales. Al mismo tiempo, confirma que la plataforma es capaz de soportar un flujo de ataque complejo de extremo a extremo y que los requerimientos funcionales definidos sirven como guía práctica para trasladar esa taxonomía a escenarios móviles realistas dentro de Tectonic.

5.2. Fundamento Teórico de la Vulnerabilidad

La vulnerabilidad que tomamos como referencia fue documentada por NowSecure (NowSecure, 2017) como la combinación peligrosa de tres comportamientos inseguros en aplicaciones Android: (i) descargar archivos ZIP sobre HTTP sin cifrar (sin TLS), permitiendo la interceptación del tráfico mediante ataques *Man-in-the-Middle*; (ii) extraer archivos ZIP sin validar las rutas internas, habilitando el path traversal; y (iii) depender de la librería MultiDex 1.0.1, que carga automáticamente cualquier archivo `classes*.dex` que encuentre en su directorio de trabajo. En el caso analizado por NowSecure, la aplicación Talking Tom incorporaba el SDK publicitario Vungle, responsable de realizar esas descargas y extracciones de ZIP sin controles adicionales.

5.2.1. Ataques Man-in-the-Middle

Los ataques *Man-in-the-Middle* (MITM) permiten que un atacante intercepte y modifique el tráfico de red entre dos partes que se comunican, posicionándose entre ellas sin que ninguna de las dos detecte la presencia del intermediario. Cuando las comunicaciones utilizan protocolos sin cifrar como HTTP, el atacante puede leer y modificar el contenido en tránsito sin restricciones. Herramientas como `mitmproxy` facilitan la realización de estos ataques al actuar como un proxy HTTP/HTTPS que intercepta, inspecciona y modifica el tráfico de red. En el contexto de este escenario, `mitmproxy` permite interceptar las solicitudes HTTP de la aplicación y reemplazar el contenido legítimo con un payload malicioso, aprovechando la falta de cifrado.

La Figura 5.1 ilustra cómo el atacante intercepta la comunicación entre el emulador y el servidor, bloqueando el flujo directo y respondiendo con el payload malicioso.

5.2.2. Path traversal mediante archivos ZIP

Muchos descompresores, incluida la implementación por defecto en librerías Java y Android, no filtran los nombres de las entradas de un ZIP. Esto permite rutas como `../../../../etc/passwd` o `/data/data/com.app/../evil.dex`.

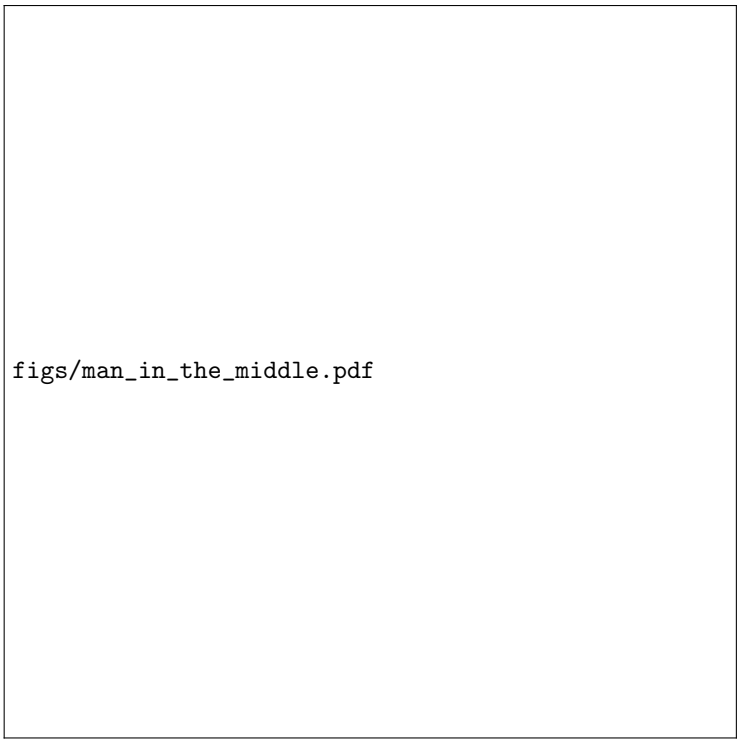

Figura 5.1: Ataque Man-in-the-Middle en el escenario de ejecución remota de código.

Si la aplicación extrae el ZIP sin sanitizar, esas secuencias `../` o rutas absolutas habilitan un *directory traversal*: el contenido termina escribiéndose fuera del directorio previsto.

5.2.3. Archivos DEX y MultiDex

Los archivos `.dex` son el "ejecutable" de una app Android: contienen las clases y métodos que el sistema corre en el dispositivo. Cada APK trae al menos un `classes.dex`; si la aplicación es muy grande y supera el límite de métodos, se generan archivos extra como `classes2.dex` o `classes3.dex`. Al instalarse o arrancar, la librería MultiDex extrae esos archivos adicionales del APK y los guarda en la carpeta interna `secondary-dexes` antes de cargarlos, de modo que la app pueda funcionar con normalidad.

5.2.4. Carga automática en MultiDex 1.0.1

La versión 1.0.1 de MultiDex asumía que sólo la propia aplicación podía crear esos archivos extra, por lo que cargaba automáticamente cualquier `classes*.dex` o `classes*.zip` que encontrara en la carpeta interna de la app. Si un ZIP malicioso lograba colocar un archivo allí —por ejemplo, por culpa de un SDK que extrae ZIP sin validarlos—, MultiDex lo agregaba al inicio junto con el código legítimo y el atacante terminaba ejecutando su código dentro de la aplicación.

5.2.5. Caso de referencia: Talking Tom

NowSecure mostró el flujo completo con la aplicación Talking Tom, que integraba el SDK publicitario Vungle. Este SDK descargaba el archivo ZIP con recursos publicitarios sobre HTTP sin cifrar (sin TLS), lo que permitía que un atacante en la misma red interceptara o modificara el contenido en tránsito mediante un ataque *Man-in-the-Middle*. Además, la biblioteca extraía el ZIP sin validar rutas internas; el *payload* malicioso, nombrado como `classes2.zip`, quedaba almacenado en la carpeta `secondary-dexes`. En el siguiente inicio, MultiDex 1.0.1 lo detectaba como legítimo, lo optimizaba y lo cargaba. El código adversario se ejecutaba en el contexto de la app, confirmando la ejecución remota de código: los investigadores evidenciaron la intrusión creando el archivo `pwned.txt` dentro del sandbox como evidencia del compromiso. El artículo original de NowSecure demostró el concepto con un simple mensaje de logging, indicando que "este logging puede reemplazarse fácilmente por cualquier payload malicioso". El laboratorio desarrollado replica fielmente este patrón de vulnerabilidades, manteniendo las tres condiciones esenciales: comunicación HTTP sin cifrar, extracción ZIP vulnerable con path traversal, y carga automática insegura por MultiDex 1.0.1.

La Figura 5.2 muestra el análisis realizado por NowSecure: en la ventana izquierda se observa mitmproxy interceptando peticiones HTTP sin cifrado TLS realizadas por la aplicación My Talking Tom (visible en el emulador de la de-

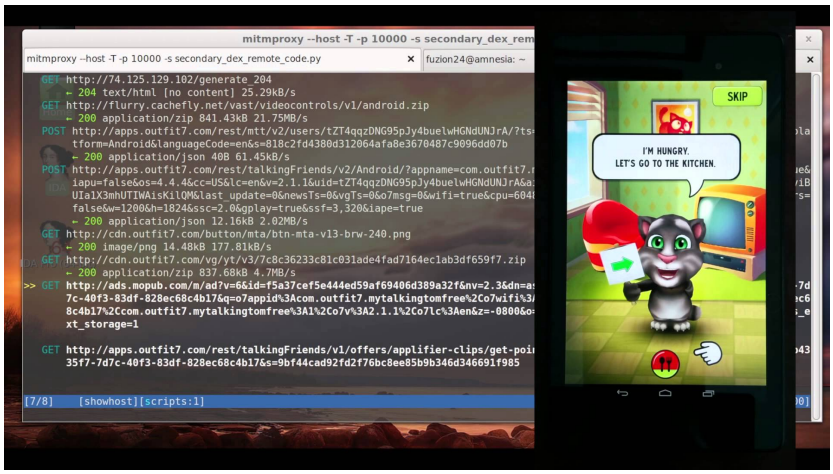

Figura 5.2: Interceptación de tráfico HTTP mediante mitmproxy y aplicación My Talking Tom ejecutándose en el emulador Android durante el análisis realizado por NowSecure(NowSecure, 2017).

recha), demostrando cómo un atacante puede interceptar y modificar el tráfico mediante un ataque Man-in-the-Middle para inyectar un payload malicioso.

5.3. Aplicación Vulnerable del Escenario

La idea inicial fue reutilizar la misma versión vulnerable de *MyTalkingTom* que NowSecure analizó en 2017. Esa aplicación fue compilada para arquitectura ARM, mientras que los emuladores que utilizamos en Tectonic son x86, por lo que el instalador ni siquiera permitía cargarla: rechazaba el paquete por incompatibilidad de arquitectura. Una opción sería utilizar una máquina base con arquitectura ARM, pero esto no es compatible con la infraestructura actual de Tectonic, que está limitada a arquitectura x86. Investigamos alternativas como crear un emulador con imagen ARM sobre el host x86, pero esta opción requiere una capa de virtualización para emular la arquitectura ARM, lo que introduce una degradación de rendimiento considerable que hace la solución poco práctica. Posteriormente, probamos ejecutarla en un emulador x86 con imagen de Android 11, que incluye soporte para ejecutar algunas aplicaciones ARM mediante traducción binaria en tiempo de ejecución. Aunque esta opción permitió ejecutar la aplicación, falló por depender de un archivo de Android que contiene recursos adicionales como gráficos, audio y video que no caben en el APK principal, llamado OBB, con todos los recursos del juego: en cada intento la aplicación se bloqueaba debido a la falta de estos archivos. Ante ese panorama optamos por crear nuestra propia aplicación vulnerable. Esta decisión nos dio tres ventajas claras: eliminamos cualquier dependencia de OBB o de paquetes de terceros, controlamos completamente el código que expone la vulnerabili-

dad y garantizamos la compatibilidad con los emuladores x86 que utilizamos en Tectonic.

5.3.1. Características de la aplicación desarrollada

La aplicación de validación del escenario se diseñó para replicar las tres vulnerabilidades principales del caso de la aplicación MyTalkingTom: la comunicación HTTP sin cifrar (sin TLS), la extracción de archivos ZIP sin validar rutas internas, generando path traversal, y la carga dinámica insegura de archivos DEX por la librería MultiDex 1.0.1. La aplicación incorpora estas condiciones de la siguiente manera:

- Solicita periódicamente (cada 10 segundos) paquetes ZIP sobre HTTP sin cifrar, replicando el comportamiento del SDK vulnerable original. Esto permite interceptar y modificar el tráfico mediante un ataque Man-in-the-Middle.
- Los archivos ZIP descargados se extraen dentro del almacenamiento interno mediante un extractor vulnerable que no valida ni sanitiza las rutas internas antes de extraer los archivos.
- El extractor vulnerable procesa las entradas del ZIP sin validación de rutas, permitiendo que secuencias de path traversal como `../` se resuelvan automáticamente. Esto permite que un payload malicioso con rutas que apuntan fuera del directorio de extracción previsto termine escribiéndose directamente en ubicaciones como `secondary-dexes`, fuera del directorio esperado.
- Aprovecha el comportamiento de MultiDex 1.0.1 para permitir la carga automática del archivo DEX malicioso.
- Incluye un indicador en la interfaz que permite confirmar visualmente que la aplicación ha sido vulnerada.

El código fuente de la aplicación y de los artefactos de explotación puede consultarse en el repositorio público disponible en https://gitlab.fing.edu.uy/gsi/android-tectonic/android-multidex-rce-lab.

5.3.2. Artefactos del Laboratorio

Para materializar el ataque descrito teóricamente, el laboratorio proporciona dos artefactos principales que el estudiante debe compilar y configurar:

- **Payload malicioso (`exploit_bundle.zip`):** Es un archivo ZIP que contiene código malicioso diseñado para aprovechar las vulnerabilidades del escenario. El archivo se genera mediante el script `build_payload.sh` que compila la clase `Shell.java` a formato DEX y lo empaqueta con una ruta

de path traversal que permite escribir el archivo directamente en el directorio `secondary-dexes` donde MultiDex busca componentes adicionales. El extractor vulnerable procesa esta entrada y resuelve el path traversal automáticamente, escribiendo el archivo en la ubicación especificada fuera del directorio de extracción previsto. La clase `Shell` se ejecuta automáticamente cuando MultiDex carga el componente. Una vez cargado por MultiDex, el payload crea el archivo `pwned.txt` como evidencia visible del compromiso, demostrando que se logró ejecutar código no autorizado dentro de la aplicación.

- **Script de interceptación (`inject_payload.py`):** Es un script de `mitm proxy` que replica el comportamiento del SDK publicitario vulnerable original. Intercepta las solicitudes HTTP de la aplicación, específicamente aquellas dirigidas a `api.vungle.com`, y responde con el `exploit_bundle .zip` malicioso en lugar del contenido legítimo, permitiendo al estudiante controlar el momento exacto en que se inyecta el payload.

Cabe señalar que, si bien estos artefactos se proporcionan precompilados para facilitar la demostración del escenario, la creación de ambos componentes (creación del payload y configuración del script de interceptación) podría constituir por sí misma objetivos de aprendizaje del laboratorio, requiriendo que los estudiantes comprendan en profundidad tanto el mecanismo de la vulnerabilidad como las técnicas de interceptación de tráfico necesarias para explotarla.

5.4. Componentes y Proceso de Ataque

Esta sección describe cómo se arma el escenario de ataque en torno a la aplicación vulnerable. El objetivo es que el estudiante observe el recorrido completo del ataque: desde la descarga interceptada, pasando por la escritura del DEX malicioso, hasta los indicadores que confirman el compromiso del dispositivo.

5.4.1. Componentes del Escenario

El escenario está compuesto por tres elementos principales. La **máquina víctima** consiste en un emulador Android con la aplicación del escenario instalada y configurada. Esta aplicación incluye un servicio interno que solicita periódicamente (cada 10 segundos) paquetes ZIP desde un servidor remoto y los extrae mediante un extractor vulnerable que no valida rutas internas, replicando el comportamiento del SDK publicitario vulnerable original.

La **máquina atacante** contiene las herramientas necesarias para ejecutar el ataque: `mitmproxy` con un script personalizado que intercepta y modifica el tráfico de red, permitiendo inyectar el payload malicioso en el tráfico HTTP de la aplicación.

En el laboratorio, el emulador Android está configurado automáticamente durante el despliegue para utilizar esta máquina como proxy HTTP, lo que permite la interceptación transparente sin requerir configuración manual adicional.

Esta configuración automática es una simplificación que facilita enfocarse en la explotación de la vulnerabilidad. En un escenario real, un atacante en la misma red (por ejemplo, una red WiFi pública o comprometida) utilizaría técnicas como *ARP spoofing* para redirigir el tráfico hacia su proxy, posicionándose como intermediario sin control directo sobre la configuración del dispositivo víctima. Este escenario es realista cuando las aplicaciones realizan comunicaciones sin cifrar sobre HTTP, como fue el caso del SDK de Vungle en la aplicación My Talking Tom.

El **payload malicioso** es un archivo ZIP diseñado para ser inyectado en el tráfico de red de la aplicación y, una vez procesado, ejecutar código no autorizado dentro del contexto de la aplicación víctima, demostrando así el compromiso del dispositivo.

5.4.2. Secuencia del Ataque

A continuación se describe, paso a paso, cómo se desarrolla el ataque dentro del escenario:

1. Se prepara el entorno: emulador y aplicación vulnerables listos, proxy configurado hacia `mitmproxy`.
2. Cuando el servicio de la aplicación solicita un ZIP, el proxy responde con el paquete malicioso, tal como se muestra en la Figura 5.1.
3. La app descomprime el ZIP sin validar rutas, escribe el DEX inyectado en `secondary-dexes` y lo deja listo para la siguiente carga.
4. Al reiniciar la app, MultiDex 1.0.1 incorpora el DEX malicioso y ejecuta el código adversario, que crea el archivo `pwned.txt` como evidencia del compromiso.

La Figura 5.3 ilustra el flujo completo del ataque desde la perspectiva del estudiante.

5.4.3. Indicadores de Compromiso

Una vez que el ataque se ejecuta exitosamente, varios indicadores confirman que el dispositivo ha sido comprometido. El primer indicador visual es el cambio en la interfaz de la aplicación, que pasa de mostrar el estado “SECURE” en color verde a “COMPROMISED” en color rojo. Este cambio se produce cuando la aplicación detecta la presencia del archivo `pwned.txt` en su directorio de almacenamiento interno, proporcionando una confirmación inmediata y visual del éxito del ataque desde la perspectiva del estudiante.

Para una validación técnica más profunda, el estudiante puede verificar la evidencia escrita dentro del sandbox de la aplicación accediendo al archivo `pwned.txt` mediante el comando:

```
adb shell run-as com.app.lab.rcelab_nowsecure cat files/pwned.txt
```

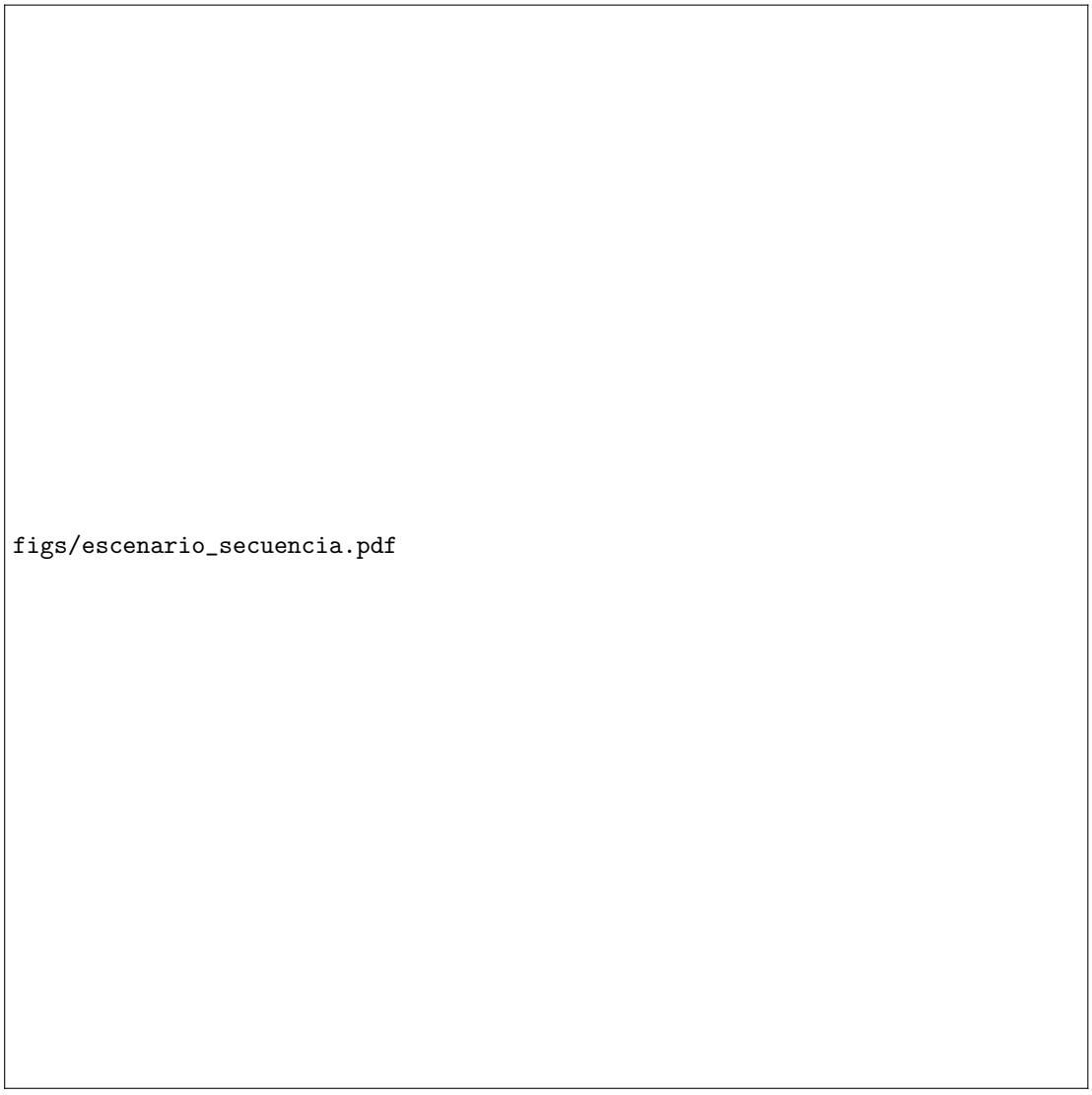

Figura 5.3: Secuencia del ataque de ejecución remota de código.

Este archivo contiene una marca temporal generada por el código malicioso al ejecutarse, demostrando que el payload se ejecutó con los permisos de la aplicación y que MultiDex cargó exitosamente el DEX malicioso.

Estos indicadores, combinados con la revisión de los registros de `logcat` que muestran los mensajes emitidos por el payload malicioso, proporcionan evidencia completa de que el ataque se completó exitosamente y que se logró ejecutar código no autorizado dentro del contexto de la aplicación.

5.5. Automatización con Tectonic y Ansible

Una vez definidos los componentes y el flujo del ataque, la siguiente etapa es automatizar el despliegue mediante Tectonic. La integración del escenario RCE con la plataforma permite desplegar automáticamente toda la infraestructura necesaria mediante Infraestructura como Código. El proceso combina la reutilización de componentes genéricos (emulador Android) con la configuración específica del laboratorio (aplicación vulnerable, herramientas de ataque).

5.5.1. Definición de la infraestructura

El archivo `description.yml` declara los componentes del escenario: una máquina atacante que actúa como punto de entrada y una máquina víctima que hospeda el emulador Android. La configuración establece recursos suficientes para ejecutar el emulador con rendimiento adecuado:

```yaml
guest_settings:
  attacker:
    entry_point: yes
  smartphone:
    disk: 40
    memory: 8192
    vcpu: 4

topology:
  - name: internal
    members:
      - attacker
      - smartphone
```

Listado 5.1: Definición de infraestructura del escenario RCE

5.5.2. Configuración base del escenario

El playbook `base_config.yml` orquesta la preparación de ambos nodos durante la creación de imágenes base. Reutiliza el playbook `android.yml` descrito en el capítulo anterior para configurar el emulador, y añade las herramientas específicas del atacante. El playbook incluye la instalación de `mitmproxy`, la creación de directorios de trabajo y la copia de los artefactos de explotación necesarios. El contenido completo del playbook se puede consultar en el Anexo A.

5.5.3. Preparación del emulador

Una vez desplegadas las instancias, el playbook `after_clone.yml` personaliza cada emulador con la aplicación vulnerable del escenario. Este proceso incluye la instalación del APK desarrollado, la configuración del proxy para habilitar el ataque Man-in-the-Middle, y el lanzamiento automático de la aplicación para que el estudiante encuentre el dispositivo listo para comenzar el laboratorio. El playbook utiliza ADB para interactuar con el emulador, esperando a que el dispositivo esté completamente iniciado antes de proceder con la instalación y configuración. La configuración del proxy se realiza dinámicamente obteniendo la dirección IP de la máquina atacante mediante variables de Ansible. El contenido completo del playbook se puede consultar en el Anexo A.

5.5.4. Resultado de la automatización

Al completarse el despliegue, cada estudiante dispone de:

- Un emulador Android con la aplicación vulnerable instalada y en ejecución
- Acceso VNC para observar la interfaz gráfica del dispositivo
- Una máquina atacante con `mitmproxy` y scripts listos para el ejercicio
- Configuración automática del proxy en el emulador para dirigir el tráfico hacia la máquina atacante

5.6. Ejecución del Ataque

Esta sección describe el desarrollo del ataque desde la perspectiva del estudiante, mostrando paso a paso cómo se reproduce la vulnerabilidad utilizando los artefactos del laboratorio.

5.6.1. Preparación y Ejecución

El ataque se ejecuta siguiendo estos pasos secuenciales, tal como se muestra en la Figura 5.3:

1. **Acceso al entorno:** Una vez desplegado el escenario mediante Tectonic, el estudiante se conecta a la máquina atacante por SSH y accede a la estación de trabajo del laboratorio que contiene los artefactos necesarios. El estudiante establece también una conexión VNC al emulador Android para observar visualmente el comportamiento de la aplicación vulnerable.

2. **Configuración de la interceptación:** El estudiante navega al subdirectorio de scripts dentro de la estación de trabajo e inicia el proxy de interceptación mediante `mitmdump -s mitmproxy/inject_payload.py -p 8090`, donde `-s` especifica el script de interceptación a ejecutar y `-p 8090` define el puerto donde se configuró el proxy. Gracias a la configuración

automática del proxy durante el despliegue, todas las solicitudes HTTP del emulador pasan a través de `mitmproxy`, permitiendo la interceptación automática cuando la aplicación solicite contenido.

3. **Interceptación del tráfico:** Con el proxy activo, el estudiante observa en la consola de `mitmproxy` la solicitud GET del archivo ZIP por parte de la aplicación. En este momento, el proxy responde automáticamente con el payload malicioso.

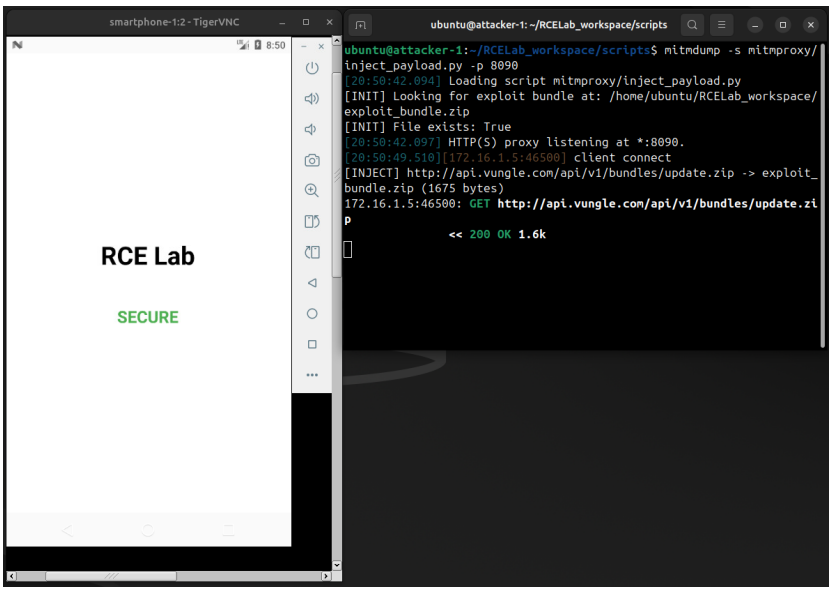

Figura 5.4: Interceptación del tráfico HTTP mediante mitmproxy mostrando la solicitud GET del archivo ZIP.

4. **Activación de la vulnerabilidad:** Para activar la carga del DEX malicioso, el estudiante debe cerrar y reabrir la aplicación en el emulador Android a través de la interfaz VNC. Este reinicio es necesario porque MultiDex 1.0.1 ejecuta su rutina de carga automática durante la inicialización de la aplicación.

5.6.2. Verificación y Análisis del Compromiso

La aplicación monitorea periódicamente (cada 5 segundos) la presencia del archivo `pwned.txt` y cambia su interfaz de "SECURE" (verde) a "COMPROMISED" (rojo) cuando lo detecta, proporcionando un primer indicador visual de compromiso.

Para confirmarlo, el estudiante se conecta a la máquina `smartphone` y valida la evidencia escrita dentro del sandbox de la aplicación ejecutando el comando:

```
adb shell run-as com.app.lab.rcelab_nowsecure cat files/pwned.txt
```

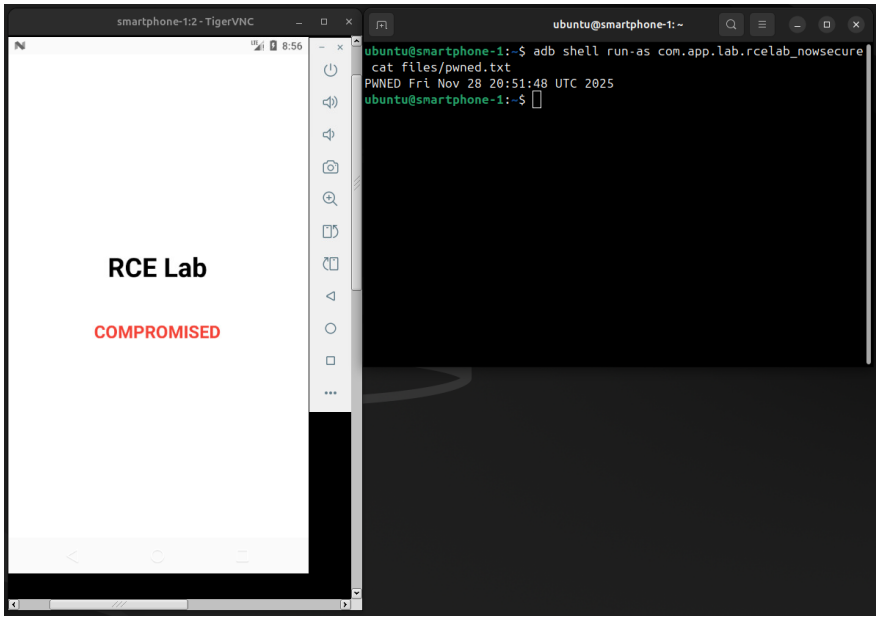

Figura 5.5: Verificación del compromiso mostrando la aplicación en estado COMPROMISED (izquierda) y la evidencia técnica obtenida mediante ADB (derecha).

La salida de la instrucción incluye la marca temporal generada por la clase `Shell`, corroborando que el código malicioso se ejecutó con los permisos de la app. El estudiante puede también consultar los registros de `logcat` en busca de los mensajes emitidos por el payload para obtener evidencia adicional del compromiso.

Estos indicadores confirman que la cadena completa del ataque se manifiesta de extremo a extremo:

- La extracción insegura del ZIP permitió colocar el payload en `secondary-dexes`.
- MultiDex 1.0.1 cargó el DEX sin validarlo y ejecutó automáticamente la lógica de inicialización definida en la clase maliciosa.
- El resultado fue la ejecución de código no autorizado dentro del contexto de la aplicación.

5.7. Consideraciones de Mitigación

NowSecure documentó este patrón de ataque en su artículo "A Pattern for Remote Code Execution using Arbitrary File Writes and MultiDex Applications" (NowSecure, 2017) junto con un conjunto de medidas para mitigar o prevenir su explotación:

- Utilizar HTTPS/TLS para toda descarga de archivos ZIP o DEX, evitando que un atacante intercepte o modifique el contenido en tránsito.
- Validar y sanitizar los archivos comprimidos antes de extraerlos, rechazando rutas que contengan `../` o rutas absolutas, y asegurando que todos los archivos se escriban únicamente dentro del directorio previsto por la aplicación.
- Evitar la extracción o carga dinámica de código en carpetas que el runtime pueda ejecutar automáticamente (como `code_cache/secondary-dexes`), moviendo estos contenidos a directorios que no sean cargados de manera implícita.
- Actualizar a MultiDex 1.0.2 o utilizar el soporte nativo de Android 5.0+, que incorpora validaciones adicionales y elimina la carga automática de `classes*.dex` sin verificación.

A estas recomendaciones se suma la verificación de integridad o firmas digitales sobre los archivos descargados, así como la auditoría periódica de bibliotecas de terceros.

5.8. Experimentación

El escenario desarrollado se utilizó como herramienta de validación para realizar mediciones de performance y evaluar los aspectos no funcionales de la solución propuesta. A través de la ejecución del escenario, se pudieron verificar empíricamente el cumplimiento de los requerimientos no funcionales establecidos, proporcionando evidencia práctica de la capacidad de la plataforma para soportar escenarios móviles.

5.8.1. Configuración del Hardware de Pruebas

Las pruebas de performance y validación se realizaron en un entorno de pruebas con las siguientes especificaciones de hardware:

- **Procesador:** Intel i5 9400
- **Memoria RAM:** 16 GB DDR4
- **Almacenamiento:** HDD 500GB
- **Hipervisor:** KVM/QEMU (Libvirt)

El escenario RCE requiere recursos específicos para cada máquina virtual según la configuración definida en `description.yml`: la máquina `smartphone` que hospeda el emulador Android requiere 8 GB de RAM, 4 vCPU y 40 GB de almacenamiento, mientras que la máquina `attacker` utiliza recursos mínimos (2 GB RAM, 2 vCPU, 20 GB disco). Estas especificaciones fueron determinadas empíricamente para garantizar un rendimiento fluido del emulador mientras se mantiene un uso eficiente de recursos del host.

5.8.2. Mediciones de Performance

Se midieron los tiempos de dos etapas del ciclo de vida del escenario: la creación de imágenes base (`create-images`) y el despliegue (`deploy`). La etapa de `create-images` incluye la instalación del sistema operativo, la configuración del emulador Android mediante el playbook `android.yml` y la instalación de herramientas base. Por su parte, la etapa de `deploy` abarca la clonación de la máquina virtual y la ejecución de las tareas de personalización definidas en `after_clone.yml`.

Las mediciones se realizaron mediante un script automatizado que ejecutó cada etapa de forma independiente. La creación de imágenes se midió ejecutando `create-images`, mientras que el despliegue se midió ejecutando `deploy` cuando las imágenes base ya existían. La Tabla 5.1 resume los tiempos registrados en dos ejecuciones de cada etapa.

Los resultados muestran una variabilidad mínima entre ejecuciones, con tiempos de creación de imágenes entre 13 y 15 minutos, y tiempos de despliegue consistentes alrededor de 3 minutos cuando las imágenes base ya existen. El tiempo total representa la suma de ambas etapas ejecutadas secuencialmente.

Etapa	Ejecución 1	Ejecución 2
Creación de imágenes	14m 38.956s	13m 24.568s
Despliegue	3m 7.489s	3m 2.069s
Total	**17m 46.445s**	**16m 26.637s**

Tabla 5.1: Tiempos de creación de imágenes base y despliegue del escenario RCE

El tiempo de creación de imágenes (aproximadamente 14 minutos) es el componente más costoso del proceso, pero es una inversión que se realiza una única vez. Una vez creadas las imágenes base, los despliegues subsecuentes son significativamente más rápidos (aproximadamente 3 minutos), ya que solo requieren clonar las imágenes existentes y ejecutar las tareas de personalización definidas en `after_clone.yml`. Todo el proceso de despliegue se realiza mediante archivos declarativos de configuración (`description.yml`) y playbooks de Ansible (`base_config.yml`, `after_clone.yml`). Esta característica valida el cumplimiento del RNF1 (Orquestación Declarativa), demostrando que el escenario se despliega completamente mediante Infraestructura como Código, garantizando reproducibilidad y consistencia entre ejecuciones.

5.8.3. Estabilidad y Uso de Recursos

Durante la ejecución del escenario, la emulación mantiene un funcionamiento estable y fluido. Tanto la instalación de la aplicación como la ejecución de las herramientas ofensivas se realizan sin degradar el rendimiento del sistema, conservando los márgenes de CPU, memoria y almacenamiento dentro de los límites previstos (8 GB RAM, 4 vCPU para la máquina smartphone), lo que permite que múltiples instancias del laboratorio puedan ejecutarse simultáneamente sin afectar la estabilidad de la infraestructura. Este comportamiento valida el cumplimiento del RNF2 (Rendimiento y Uso Eficiente de Recursos).

5.8.4. Portabilidad y Consideraciones de Despliegue en Nube

La construcción del escenario se apoya exclusivamente en archivos declarativos de IaC y en la operación automatizada de los comandos de Tectonic. El laboratorio se despliega sin modificaciones en las plataformas Libvirt, Docker y AWS reutilizando los mismos playbooks y plantillas, manteniendo el mismo funcionamiento y comportamiento en cada plataforma. Esta portabilidad comprueba el RNF3 (Independencia de Plataforma).

Sin embargo, para el caso específico de **AWS**, existen consideraciones importantes de performance:

- **Virtualización Anidada:** Cuando se despliega un emulador Android dentro de una instancia EC2 estándar, se crea una situación de *virtualización anidada*. Esta doble capa de virtualización introduce una penalización

significativa de performance, resultando en latencia alta y una experiencia de usuario degradada.

- **Instancias Bare-Metal:** Para mitigar lo anterior, AWS ofrece instancias *bare-metal* que proporcionan acceso directo al hardware físico, eliminando la penalización de la virtualización anidada. Esto permite un rendimiento óptimo del emulador, similar al obtenido *on-premise*. No obstante, estas instancias tienen un costo significativamente mayor que las instancias virtuales estándar.

Por estas razones, para despliegues de producción se recomienda utilizar las plataformas Libvirt o Docker en infraestructura *on-premise*, donde el emulador puede acceder directamente al hardware del host sin penalizaciones de virtualización anidada y sin los costos adicionales asociados con instancias bare-metal.

5.8.5. Conclusiones de la Experimentación

Los resultados obtenidos durante esta fase experimental demuestran que la solución propuesta funciona correctamente en un entorno real. La consistencia en los tiempos de despliegue, la estabilidad del sistema bajo carga con uso eficiente de recursos, y la capacidad de adaptación a distintas infraestructuras permiten validar el cumplimiento de los requerimientos no funcionales definidos para el proyecto: orquestación declarativa (RNF1), rendimiento y uso eficiente de recursos (RNF2), e independencia de plataforma (RNF3).

Capítulo 6

Conclusiones y Trabajo Futuro

Este capítulo presenta las principales conclusiones del proyecto y las líneas de trabajo futuro que permitirían continuar y mejorar los desarrollos realizados.

6.1. Conclusiones

Este proyecto fue desarrollado con el objetivo de ampliar las capacidades de Tectonic para incluir emulación de dispositivos móviles, llenando así el vacío existente en la formación práctica de ciberseguridad para el ecosistema móvil.

En primer lugar, se realizó una investigación del estado del arte en seguridad móvil, estudiando los marcos de referencia más relevantes de la industria —OWASP Mobile Top 10, MITRE ATT&CK Mobile y NIST Mobile Threat Catalogue— y sintetizando sus aportes desde distintos enfoques. Como resultado, se desarrolló una taxonomía de ataques propia de siete categorías, centrada en el punto de entrada u objetivo del ataque, que facilita la construcción de escenarios replicables y pedagógicamente valiosos dentro de cyber ranges.

Para abordar el problema de la implementación, el desafío principal fue encontrar una solución de emulación de Android que cumpliera con los requerimientos definidos. Se exploraron tres alternativas —contenedores Docker, Android-x86 en máquina virtual y el SDK oficial de Android ejecutándose dentro de una máquina virtual con virtualización de display—, analizando en cada caso su documentación técnica y realizando pruebas de concepto. La solución implementada utiliza el SDK oficial de Android con virtualización anidada, seleccionada por ofrecer el mejor equilibrio entre fidelidad, flexibilidad y control programático.

La implementación se realizó utilizando tecnologías robustas y de código abierto: el SDK oficial de Android, KVM con virtualización anidada, y Ansible para la orquestación declarativa. El playbook `android.yml` transforma automáticamente una máquina virtual genérica en un entorno de emulación

Android funcional, instalando dependencias, configurando el SDK, creando el dispositivo virtual y desplegando los servicios de visualización remota. Esta implementación cumple con el paradigma de Infraestructura como Código de Tectonic, garantizando reproducibilidad y escalabilidad.

Para validar la solución, se construyó y desplegó exitosamente un laboratorio de Ejecución Remota de Código (RCE) que implementa tres categorías de la taxonomía: Aplicación y Binarios, Comunicación, y Ecosistema y Cadena de Suministro. Este escenario, basado en una vulnerabilidad documentada de 2017, demuestra no solo la viabilidad técnica de la solución, sino también la utilidad del marco taxonómico propuesto para el diseño estructurado de laboratorios de seguridad móvil.

El proyecto logró cumplir con el objetivo general y los cuatro objetivos específicos planteados, entregando una solución funcional, documentada y validada que establece las bases para futuros desarrollos en seguridad móvil dentro de Tectonic.

Más allá de los resultados técnicos, el desarrollo de este proyecto nos permitió adquirir conocimientos en seguridad móvil, emulación de Android y automatización mediante Infraestructura como Código. Este trabajo también permitió comprender mejor la importancia de la ciberseguridad y la necesidad de formar profesionales capaces de proteger los sistemas digitales que sustentan nuestra sociedad. La evaluación sistemática de alternativas técnicas demostró la importancia de un análisis exhaustivo para llegar a una solución robusta y bien fundamentada.

6.2. Trabajo Futuro

A continuación se presentan las líneas de trabajo que permitirían continuar y mejorar los desarrollos realizados.

6.2.1. Escenarios adicionales para completar la taxonomía

Sería relevante implementar escenarios adicionales para cubrir todas las categorías de la taxonomía, validando así la versatilidad de la plataforma para abordar los distintos tipos de ataques móviles. Cada escenario demostraría la capacidad de emular diferentes vectores de ataque y contribuiría a una formación más completa en seguridad móvil.

6.2.2. Viabilidad de Ataques de acceso físico

Actualmente la solución excluye la categoría de Acceso Físico dado que el entorno se basa en emulación de software y estos ataques requieren interacción directa con hardware físico. Sin embargo, sería valioso explorar alternativas para extender la solución y dar soporte a aspectos específicos de esta categoría, estableciendo límites claros sobre qué puede y qué no puede emularse.

6.2.3. Mejoras en rendimiento y optimización

Una línea de investigación prometedora sería evaluar el rendimiento en diferentes configuraciones de hardware —cantidad de núcleos, memoria RAM, tipo de procesador—, identificando los requisitos mínimos y óptimos para diferentes escenarios. Adicionalmente, se podría explorar la optimización de configuraciones del emulador, ajustes del entorno y otras mejoras para reducir el tiempo de despliegue y mejorar la asignación de recursos. Esta caracterización permitiría mejorar la capacidad de la plataforma para ejecutar múltiples laboratorios simultáneamente.

Como se mencionó en la sección de experimentación, los despliegues en AWS presentan penalizaciones de performance debido a la virtualización anidada, requiriendo instancias bare-metal con costos elevados para obtener rendimiento óptimo. Una línea de trabajo futuro sería investigar técnicas de optimización para mejorar el rendimiento del emulador Android en instancias EC2 estándar, reduciendo así la necesidad de instancias bare-metal y haciendo más viable económicamente el despliegue en AWS.

6.2.4. Evaluación cuantitativa y métricas

Para caracterizar mejor el rendimiento y la usabilidad de la solución, se requiere realizar una evaluación cuantitativa exhaustiva que incluya: tiempo de despliegue de laboratorios, consumo de recursos (CPU, memoria, almacenamiento) en diferentes configuraciones, latencia en la visualización remota, y estabilidad durante sesiones prolongadas de uso. Además, sería relevante realizar pruebas de estrés con múltiples laboratorios concurrentes para determinar los límites de escalabilidad de la infraestructura actual.

6.2.5. Integración declarativa con Tectonic

Actualmente, la configuración de dispositivos Android en los escenarios requiere especificar comandos concretos en los playbooks de Ansible (por ejemplo, comandos `adb` específicos para instalar APKs o configurar el emulador), lo que limita la flexibilidad de la solución. Este enfoque requiere que el desarrollador del escenario sepa cómo realizar cada tarea técnica en lugar de simplemente declarar qué desea lograr. Resulta conveniente incorporar soporte nativo para que la máquina Android del escenario reciba parámetros y artefactos de manera declarativa desde la especificación del laboratorio —como ya sucede con servicios como Caldera o Elastic—, permitiendo que el desarrollador especifique sus necesidades (por ejemplo, “instalar esta aplicación” o “configurar este proxy”) sin preocuparse por los detalles técnicos de implementación. Esto eliminaría pasos manuales y mejoraría la usabilidad y mantenibilidad de los escenarios.

6.2.6. Soporte para aplicaciones ARM

Actualmente la solución está limitada a aplicaciones compiladas para arquitectura x86, debido a que las máquinas base de Tectonic utilizan arquitectura

x86 y los emuladores Android se ejecutan de forma nativa sobre esta plataforma. Una línea de trabajo futuro sería explorar el despliegue de máquinas base con arquitectura ARM para permitir la ejecución nativa de aplicaciones Android ARM sin degradación de rendimiento ni dependencias de emulación o traducción binaria. Esto facilitaría el análisis de aplicaciones legacy compiladas exclusivamente para ARM y ampliaría el rango de escenarios que pueden implementarse dentro de la plataforma.

Referencias

Android. (2024). *Ataque de strandhogg/vulnerabilidad de afinidad de la tarea.* https://developer.android.com/privacy-and-security/risks/strandhogg?hl=es-419. (Consultado en 2025)

Android Developers. (2024). *Run apps on the android emulator.* https://developer.android.com/studio/run/emulator. (Consultado en 2025)

Android-x86 Project. (2024). *Android-x86 - porting android to x86.* https://www.android-x86.org/. (Consultado en 2025)

Ansible. (2023). *Ansible playbooks.* https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_intro.html. (Consultado en 2025)

budtmo. (2024). *docker-android.* https://github.com/budtmo/docker-android. (Consultado en 2025)

Genymobile. (2024). *scrcpy - display and control your android device.* https://github.com/Genymobile/scrcpy. (Consultado en 2025)

Grupo de Seguridad Informática, FIng-Udelar. (2024). *Tectonic: An academic cyber range.* https://www.fing.edu.uy/inco/proyectos/tectonic. (Consultado en 2025)

GSI-Fing-Udelar. (2024). *Tectonic - an academic cyber range.* https://github.com/GSI-Fing-Udelar/tectonic. (Consultado en 2025)

Guerrero, G., Betarte, G., y Campo, J. D. (2024). *Tectonic: An academic cyber range.* https://ieeexplore.ieee.org/document/10735713. (Consultado en 2025)

Hashicorp. (2013). *Packer.* https://developer.hashicorp.com/packer#what-is-packer. (Consultado en 2025)

Hashicorp. (2014). *Terraform.* https://developer.hashicorp.com/terraform. (Consultado en 2025)

HQarroum. (2024). *docker-android.* https://github.com/HQarroum/docker-android. (Consultado en 2025)

IBM. (2024). *¿qué es un cyber range?* https://www.ibm.com/mx-es/think/topics/cyber-range. (Consultado en 2025)

ISC2. (2024). *2024 isc2 cybersecurity workforce study.* https://www.isc2.org/Insights/2024/10/ISC2-2024-Cybersecurity-Workforce-Study. (Consultado en 2025)

Kivva, A. (2024). *Mobile malware evolution in 2024.* https://securelist.com/mobile-threat-report-2024/115494/. (Consultado en 2025)

LibVNC. (2024). *x11vnc: a vnc server for real x displays.* https://github.com/LibVNC/x11vnc. (Consultado en 2025)

Michael DeHaan. (2012). *Ansible.* https://docs.ansible.com/. (Consultado en 2025)

MITRE. (2020). *Att&ck for mobile.* https://attack.mitre.org/matrices/mobile/. (Consultado en 2025)

MITRE. (2023a). *Mitre att&ck mobile - t1406: Obfuscated files or information.* https://attack.mitre.org/techniques/T1406/. (Consultado en 2025)

MITRE. (2023b). *Mitre att&ck mobile - t1407: Runtime code modification.* https://attack.mitre.org/techniques/T1407/. (Consultado en 2025)

MITRE. (2023c). *Mitre att&ck mobile - t1426: System information discovery.* https://attack.mitre.org/techniques/T1426/. (Consultado en 2025)

MITRE. (2023d). *Mitre att&ck mobile - t1430: Location tracking.* https://attack.mitre.org/techniques/T1430/. (Consultado en 2025)

MITRE. (2023e). *Mitre att&ck mobile - t1461: Lockscreen bypass.* https://attack.mitre.org/techniques/T1461/. (Consultado en 2025)

MITRE. (2023f). *Mitre att&ck mobile - t1533: Data from local system.* https://attack.mitre.org/techniques/T1533/. (Consultado en 2025)

MITRE. (2023g). *Mitre att&ck mobile - t1577: Compromise application executable.* https://attack.mitre.org/techniques/T1577/. (Consultado en 2025)

MITRE. (2023h). *Mitre att&ck mobile - t1631: Process injection.* https://attack.mitre.org/techniques/T1631/. (Consultado en 2025)

MITRE. (2023i). *Mitre att&ck mobile - t1632.001: Code signing policy modification.* https://attack.mitre.org/techniques/T1632/001/. (Consultado en 2025)

MITRE. (2023j). *Mitre att&ck mobile - t1635: Steal application access token.* https://attack.mitre.org/techniques/T1635/. (Consultado en 2025)

MITRE. (2023k). *Mitre att&ck mobile - t1638: Network traffic manipulation.* https://attack.mitre.org/techniques/T1638/. (Consultado en 2025)

MITRE. (2023l). *Mitre att&ck mobile - t1661: Malicious app update.* https://attack.mitre.org/techniques/T1661/. (Consultado en 2025)

National Institute of Standards and Technology. (2018). *Mobile threat catalogue.* https://pages.nist.gov/mobile-threat-catalogue/. (Consultado en 2025)

National Institute of Standards and Technology. (2023). *The cyber range: A guide* (Inf. Téc.). NIST. Descargado de https://www.nist.gov/system/files/documents/2023/09/29/The%20Cyber%20Range_A%20Guide.pdf (Consultado en 2025)

NIST. (2023a). *Nist mobile threat catalogue - app-10: Insufficient cryptography.* https://pages.nist.gov/mobile-threat-catalogue/application-threats/APP-10.html. (Consultado en 2025)

NIST. (2023b). *Nist mobile threat catalogue - app-12: Device information collection.* https://pages.nist.gov/mobile-threat-catalogue/application-threats/APP-12.html. (Consultado en 2025)

NIST. (2023c). *Nist mobile threat catalogue - app-14: Malicious application.* https://pages.nist.gov/mobile-threat-catalogue/application-threats/APP-14.html. (Consultado en 2025)
NIST. (2023d). *Nist mobile threat catalogue - app-1: Network traffic compromise.* https://pages.nist.gov/mobile-threat-catalogue/application-threats/APP-1.html. (Consultado en 2025)
NIST. (2023e). *Nist mobile threat catalogue - app-2: Sensitive information exposure.* https://pages.nist.gov/mobile-threat-catalogue/application-threats/APP-2.html. (Consultado en 2025)
NIST. (2023f). *Nist mobile threat catalogue - app-3: Sensitive information in system logs.* https://pages.nist.gov/mobile-threat-catalogue/application-threats/APP-3.html. (Consultado en 2025)
NIST. (2023g). *Nist mobile threat catalogue - app-6: Vulnerable third-party library.* https://pages.nist.gov/mobile-threat-catalogue/application-threats/APP-6.html. (Consultado en 2025)
NIST. (2023h). *Nist mobile threat catalogue - aut-0: Use of stolen credentials.* https://pages.nist.gov/mobile-threat-catalogue/authentication-threats/AUT-0.html. (Consultado en 2025)
NIST. (2023i). *Nist mobile threat catalogue - aut-12: Insecure credential storage.* https://pages.nist.gov/mobile-threat-catalogue/authentication-threats/AUT-12.html. (Consultado en 2025)
NIST. (2023j). *Nist mobile threat catalogue - lpn-2: Insecure networks.* https://pages.nist.gov/mobile-threat-catalogue/lan-pan-threats/LPN-2.html. (Consultado en 2025)
NIST. (2023k). *Nist mobile threat catalogue - phy-2: Insecure usb connection to computer.* https://pages.nist.gov/mobile-threat-catalogue/physical-threats/PHY-2.html. (Consultado en 2025)
NIST. (2023l). *Nist mobile threat catalogue - sta-6: Malicious app installation via usb.* https://pages.nist.gov/mobile-threat-catalogue/stack-threats/STA-6.html. (Consultado en 2025)
NowSecure. (2017). *A pattern for remote code execution using arbitrary file writes and multidex applications.* https://www.nowsecure.com/blog/2017/06/15/a-pattern-for-remote-code-execution-using-arbitrary-file-writes-and-multidex-applications/. (Consultado en 2025)
OWASP Foundation. (s.f.). *Owasp mobile top 10.* https://owasp.org/www-project-mobile-top-10/. (Consultado en 2025)
OWASP Foundation. (2020). *Owasp mobile security testing guide.* https://owasp.org/www-project-mobile-security-testing-guide/. (Consultado en 2025)
OWASP Foundation. (2023a). *Owasp mobile top 10 2023 - m10: Insufficient cryptography.* https://owasp.org/www-project-mobile-top-10/2023-risks/m10-insufficient-cryptography. (Consultado en 2025)
OWASP Foundation. (2023b). *Owasp mobile top 10 2023 - m1: Improper credential usage.* https://owasp.org/www-project-mobile-top-10/2023-risks/m1-improper-credential-usage. (Consultado en 2025)

OWASP Foundation. (2023c). *Owasp mobile top 10 2023 - m2: Inadequate supply chain security.* https://owasp.org/www-project-mobile-top-10/2023-risks/m2-inadequate-supply-chain-security. (Consultado en 2025)

OWASP Foundation. (2023d). *Owasp mobile top 10 2023 - m3: Insecure authentication/authorization.* https://owasp.org/www-project-mobile-top-10/2023-risks/m3-insecure-authentication-authorization. (Consultado en 2025)

OWASP Foundation. (2023e). *Owasp mobile top 10 2023 - m4: Insufficient input/output validation.* https://owasp.org/www-project-mobile-top-10/2023-risks/m4-insufficient-input-output-validation. (Consultado en 2025)

OWASP Foundation. (2023f). *Owasp mobile top 10 2023 - m5: Insecure communication.* https://owasp.org/www-project-mobile-top-10/2023-risks/m5-insecure-communication. (Consultado en 2025)

OWASP Foundation. (2023g). *Owasp mobile top 10 2023 - m6: Inadequate privacy controls.* https://owasp.org/www-project-mobile-top-10/2023-risks/m6-inadequate-privacy-controls. (Consultado en 2025)

OWASP Foundation. (2023h). *Owasp mobile top 10 2023 - m7: Insufficient binary protection.* https://owasp.org/www-project-mobile-top-10/2023-risks/m7-insufficient-binary-protection. (Consultado en 2025)

OWASP Foundation. (2023i). *Owasp mobile top 10 2023 - m9: Insecure data storage.* https://owasp.org/www-project-mobile-top-10/2023-risks/m9-insecure-data-storage. (Consultado en 2025)

Russo, E., Merla, A., y Verderame, L. (2020). Enabling next-generation cyber ranges with mobile security components. En *Ictss 2020.* Descargado de https://www.researchgate.net/publication/347303681 (Consultado en 2025)

StatCounter Global Stats. (2024). *Mobile operating system market share worldwide.* https://gs.statcounter.com/os-market-share/mobile/worldwide. (Consultado en 2025)

Threat Defence. (s.f.). *Cyber range resource guide.* https://www.threatdefence.com/cyber-range-resource-guide. (Consultado en 2025)

Anexo A: Playbooks de Ansible

Este anexo contiene los playbooks completos de Ansible que implementan la solución de emulación Android descrita en el Capítulo 4 y su aplicación en el escenario RCE del Capítulo 5. Se incluyen el playbook base `android.yml`, el archivo de variables `android-vars.yml`, y los playbooks del escenario (`base_config.yml` y `after_clone.yml`).

A.1. Playbook Principal: android.yml

El siguiente listado muestra el playbook completo que automatiza la instalación y configuración del emulador Android:

```
---
- name: Android Emulator with VNC
  hosts: smartphone
  become: false
  vars_files:
    - android-vars.yml
  vars:
    sdk_environment:
      ANDROID_SDK_ROOT: "{{ android_sdk_root }}"
      ANDROID_HOME: "{{ android_home }}"
      ANDROID_AVD_HOME: "{{ android_sdk_root }}"
      ANDROID_SDK_HOME: "/home/{{ android_user }}"
      PATH: "{{ android_sdk_root }}/cmdline-tools/latest/bin:{{
    android_sdk_root }}/emulator:{{ android_sdk_root }}/platform-
    tools:{{ ansible_env.PATH }}"

  tasks:
    - name: "APT update"
      become: true
      ansible.builtin.apt:
        update_cache: yes
        cache_valid_time: 3600

    - name: "Install dependencies"
      become: true
      ansible.builtin.apt:
        name:
```

```
        - openjdk-21-jdk
        - x11vnc
        - unzip
        - xvfb
        - libpulse0
        - libgl1-mesa-dev
      state: present

  - name: "Create Android SDK root directory"
    ansible.builtin.file:
      path: "{{ android_sdk_root }}"
      state: directory
      mode: "0755"

  - name: "Download Android Command Line Tools"
    ansible.builtin.get_url:
      url: "{{ cli_tools_url }}"
      dest: "{{ cli_tools_zip_dest }}"
      mode: "0644"

  - name: "Create parent directory for 'latest'"
    ansible.builtin.file:
      path: "{{ android_sdk_root }}/cmdline-tools"
      state: directory
      mode: "0755"

  - name: "Unzip Command Line Tools"
    ansible.builtin.unarchive:
      src: "{{ cli_tools_zip_dest }}"
      dest: "{{ android_sdk_root }}/cmdline-tools/"
      remote_src: yes
      creates: "{{ android_sdk_root }}/cmdline-tools/latest/bin/
sdkmanager"

  - name: "Move cmdline-tools to latest"
    ansible.builtin.command:
      cmd: "mv {{ android_sdk_root }}/cmdline-tools/cmdline-tools
 {{ android_sdk_root }}/cmdline-tools/latest"
    args:
      creates: "{{ android_sdk_root }}/cmdline-tools/latest/bin"

  - name: "Configure environment variables"
    ansible.builtin.blockinfile:
      path: "/home/{{ android_user }}/.bashrc"
      create: yes
      block: |
        export ANDROID_SDK_ROOT="{{ android_sdk_root }}"
        export ANDROID_HOME="{{ android_home }}"
        export ANDROID_SDK_HOME="/home/{{ android_user }}"
        export ANDROID_AVD_HOME="{{ android_sdk_root }}"
        export PATH="$ANDROID_HOME/cmdline-tools/latest/bin:
$ANDROID_SDK_ROOT/platform-tools:$ANDROID_SDK_ROOT/emulator:
$PATH"
      mode: "0644"

  - name: "Accept Android SDK licenses"
    ansible.builtin.shell:
```

```
      cmd: "yes | {{ android_sdk_root }}/cmdline-tools/latest/bin/sdkmanager --licenses"
    environment: "{{ sdk_environment }}"
    args:
      chdir: "{{ android_base_dir }}"
    register: license_acceptance_result
    changed_when: >
      ('All SDK package licenses accepted' not in license_acceptance_result.stdout and
       'licenses were accepted' not in license_acceptance_result.stderr)
    failed_when: >
      (license_acceptance_result.rc != 0 and
      'All SDK package licenses accepted' not in license_acceptance_result.stdout and
      'licenses were accepted' not in license_acceptance_result.stderr)

  - name: "Install platform-tools, emulator, platform and system image"
    ansible.builtin.shell:
      cmd: >
        yes | sdkmanager
        "platform-tools"
        "emulator"
        "{{ platform_package }}"
        "{{ system_image_package }}"
    environment: "{{ sdk_environment }}"
    args:
      creates: "{{ android_sdk_root }}/platforms/android-{{ api_level }}"
      chdir: "/home/{{ android_user }}"

  - name: "Create Android Virtual Device (AVD)"
    ansible.builtin.shell:
          cmd: |
            echo "no" | {{ android_sdk_root }}/cmdline-tools/latest/bin/avdmanager create avd \
              --name "{{ avd_name }}" \
              --package "{{ system_image_package }}" \
              --device "{{ avd_device }}" \
              --force
    environment: "{{ sdk_environment }}"
    args:
        chdir: "{{ android_base_dir }}"
        creates: "{{ android_sdk_root }}/{{ avd_name }}.ini"
    changed_when: true

  - name: Determine available groups
    getent:
      database: group

  - name: "Add user '{{ android_user }}' to group kvm"
    become: true
    ansible.builtin.user:
      name: "{{ android_user }}"
      groups: kvm
```

```
        append: yes
      when: '"kvm" in ansible_facts.getent_group'

  - name: "Change permissions for .ini AVD"
    become: true
    ansible.builtin.file:
      path: "{{ android_sdk_root }}/{{ avd_name }}.ini"
      state: file
      owner: "{{ android_user }}"
      group: "{{ android_user }}"
      mode: "0644"

  - name: "Change permissions for .avd AVD"
    become: true
    ansible.builtin.file:
      path: "{{ android_sdk_root }}/{{ avd_name }}.avd"
      state: directory
      owner: "{{ android_user }}"
      group: "{{ android_user }}"
      recurse: yes
      mode: "0755"

  - name: "Install Xvfb service file"
    become: true
    ansible.builtin.copy:
      content: |
        [Unit]
        Description=Run Xvfb on {{ display_num }}

        [Service]
        Type=exec
        ExecStart=/usr/bin/Xvfb :{{ display_num }} -screen 0 {{ display_res }}
        ExecStartPost=/bin/sleep 2

        [Install]
        WantedBy=default.target
      dest: /etc/systemd/system/xvfb.service
      mode: "0644"

  - name: "Install x11vnc service file"
    become: true
    ansible.builtin.copy:
      content: |
        [Unit]
        Description=Run x11vnc
        After=xvfb.service

        [Service]
        Type=exec
        ExecStart=/usr/bin/x11vnc -display :{{ display_num }} -forever -rfbport {{ vnc_port }} -passwd {{ vnc_pass }}
        # Wait for port to listen (Health Check)
        ExecStartPost=/usr/bin/timeout 30 sh -c 'while ! ss -H -t -l -n sport = :{{ vnc_port }} | grep -q "LISTEN.*:{{ vnc_port }}"; do sleep 1; done'

```

```
        [Install]
        WantedBy=default.target
      dest: /etc/systemd/system/x11vnc.service
      mode: "0644"

  - name: "Install Android emulator service file"
    become: true
    ansible.builtin.copy:
      content: |
        [Unit]
        Description=Run Android Emulator
        After=x11vnc.service

        [Service]
        Type=exec
        WorkingDirectory={{ android_base_dir }}
        User={{ android_user }}

        # Explicit environment variables for robustness
        Environment="ANDROID_SDK_ROOT={{ android_sdk_root }}"
        Environment="ANDROID_HOME={{ android_home }}"
        Environment="ANDROID_AVD_HOME={{ android_sdk_root }}"
        Environment="ANDROID_SDK_HOME=/home/{{ android_user }}"
        Environment="PATH={{ android_sdk_root }}/cmdline-tools/
latest/bin:{{ android_sdk_root }}/platform-tools:{{
android_sdk_root }}/emulator:/usr/bin:/bin"
        Environment="DISPLAY=:{{ display_num }}"

        ExecStart={{ android_sdk_root }}/emulator/emulator -avd
{{ avd_name }} {{ emulator_options }}
        Restart=on-failure

        [Install]
        WantedBy=default.target
      dest: /etc/systemd/system/android_emulator.service
      mode: "0644"

  - name: "Reload daemon configs"
    become: true
    ansible.builtin.systemd:
      daemon_reload: true

  - name: "Enable and start Xvfb service"
    become: true
    ansible.builtin.systemd:
      name: xvfb
      state: started
      enabled: true

  - name: "Enable and start x11vnc service"
    become: true
    ansible.builtin.systemd:
      name: x11vnc
      state: started
      enabled: true

  - name: "Enable and start Android Emulator service"
```

```
      become: true
      ansible.builtin.systemd:
        name: android_emulator
        state: started
        enabled: true

    - name: "Wait for VNC to be accessible (Final check)"
      ansible.builtin.wait_for:
        port: "{{ vnc_port }}"
        timeout: 60
        state: started
```

Listado A.1: Playbook completo de Ansible para emulación Android (`android.yml`)

A.2. Archivo de Variables: android-vars.yml

El playbook anterior hace referencia a un archivo de variables `android-vars.yml` que contiene toda la configuración parametrizable del despliegue. El siguiente listado muestra el contenido completo de este archivo:

```
---
android_user: "{{ ansible_user }}"
android_sdk_root: "/home/{{ android_user }}/Android"
android_base_dir: "/home/{{ android_user }}"
android_home: "{{ android_sdk_root }}"
cli_tools_url: "https://dl.google.com/android/repository/
    commandlinetools-linux-11076708_latest.zip"
cli_tools_zip_dest: "/tmp/commandlinetools-{{ android_user }}.zip"

api_level: "33"
avd_device: "pixel"
system_image_package: "system-images;android-{{ api_level }};
    google_apis;x86_64"
platform_package: "platforms;android-{{ api_level }}"
avd_name: "miEmulador"
emulator_options: "-no-audio -no-snapshot-load -no-snapshot-save -
    no-boot-anim"
display_num: 2
display_res: "1080x1920x24"
vnc_port: 5902
vnc_pass: "1234"
```

Listado A.2: Archivo de variables de Ansible para emulación Android (`android-vars.yml`)

A.3. Descripción de Variables

Las variables definidas en `android-vars.yml` permiten personalizar completamente el comportamiento del despliegue:

- `android_user`: Usuario del sistema que ejecutará el emulador (por defecto, el usuario de Ansible).

- `android_sdk_root`: Directorio raíz donde se instala el SDK de Android.
- `android_base_dir`: Directorio base de trabajo para el usuario de Android.
- `android_home`: Variable de entorno ANDROID_HOME (normalmente igual a `android_sdk_root`).
- `cli_tools_url`: URL de descarga de las Android Command Line Tools desde los repositorios oficiales de Google.
- `cli_tools_zip_dest`: Ruta temporal donde se descarga el archivo ZIP de las herramientas.
- `api_level`: Nivel de API de Android a emular (por ejemplo, "33" corresponde a Android 13).
- `avd_device`: Perfil de hardware del dispositivo virtual (por ejemplo, "pixel").
- `system_image_package`: Identificador del paquete de imagen del sistema a instalar.
- `platform_package`: Identificador del paquete de plataforma Android a instalar.
- `avd_name`: Nombre del Android Virtual Device (AVD) a crear.
- `emulator_options`: Opciones de línea de comandos para el emulador (por ejemplo, deshabilitar audio, snapshots, etc.).
- `display_num`: Número del display virtual X11 (por defecto, 2).
- `display_res`: Resolución del display virtual en formato "ancho x alto x profundidad" (por ejemplo, "1080x1920x24").
- `vnc_port`: Puerto TCP donde x11vnc escuchará conexiones VNC (por defecto, 5902).
- `vnc_pass`: Contraseña de autenticación para acceder al servidor VNC.

A.4. Playbooks del Escenario RCE

Esta sección contiene los playbooks completos utilizados en el escenario de Ejecución Remota de Código descrito en el Capítulo 5. Estos playbooks complementan el playbook base `android.yml` con la configuración específica del escenario.

A.4.1. Playbook: base_config.yml

El playbook `base_config.yml` se ejecuta durante la creación de imágenes base y configura tanto el emulador Android como la máquina atacante. Reutiliza el playbook `android.yml` para configurar el emulador y añade las herramientas y archivos necesarios para el atacante:

```yaml
---
- name: "Include android emulation playbook"
  ansible.builtin.import_playbook: android.yml

- name: "Attacker configuration"
  hosts: attacker
  become: true

  tasks:
    - name: "Update apt cache"
      ansible.builtin.apt:
        update_cache: true

    - name: "Install attacker tools"
      ansible.builtin.package:
        state: present
        name:
          - python3-pip

    - name: "Install mitmproxy via pip"
      ansible.builtin.pip:
        name: mitmproxy

    - name: "Create RCELab_workspace directory"
      become: false
      ansible.builtin.file:
        path: "/home/ubuntu/RCELab_workspace"
        state: directory
        mode: "0755"

    - name: "Copy scripts directory"
      become: false
      ansible.builtin.copy:
        src: ../files/scripts
        dest: /home/ubuntu/RCELab_workspace/
        owner: ubuntu
        group: ubuntu
        mode: "0755"

    - name: "Copy exploit bundle"
      become: false
      ansible.builtin.copy:
        src: ../files/exploit_bundle.zip
        dest: /home/ubuntu/RCELab_workspace/exploit_bundle.zip
        owner: ubuntu
        group: ubuntu
        mode: "0644"
```

Listado A.3: Playbook completo de configuración base del escenario RCE (`base_config.yml`)

A.4.2. Playbook: after_clone.yml

El playbook `after_clone.yml` se ejecuta después de clonar cada instancia del escenario y personaliza el emulador Android con la aplicación vulnerable. Este proceso incluye la instalación del APK, la configuración del proxy HTTP para habilitar el ataque Man-in-the-Middle, y el lanzamiento automático de la aplicación:

```
---
- name: "Gather information from all machines"
  hosts: all
  tasks: []

- name: "Install and Launch Lab APK"
  hosts: smartphone
  become: false

  vars:
    adb_path: "/home/ubuntu/Android/platform-tools/adb"
    apk_name: "app-debug.apk"

  tasks:
    - name: "Copy the APK"
      ansible.builtin.copy:
        src: "../files/{{ apk_name }}"
        dest: "/tmp/{{ apk_name }}"

    - name: "Wait for emulator to be fully booted"
      ansible.builtin.shell:
        cmd: "{{ adb_path }} wait-for-device shell 'while [[ -z $(
    getprop sys.boot_completed) ]]; do sleep 1; done;'"
      changed_when: false
      retries: 10
      delay: 15

    - name: "Install the APK using ADB"
      ansible.builtin.shell:
        cmd: "{{ adb_path }} install -r /tmp/{{ apk_name }}"
      changed_when: true

    - name: "Configure emulator proxy to point to the attacker
    machine"
      vars:
        attacker_host_name: "{{ hostvars.keys() | select('search',
    'attacker') | first }}"
      ansible.builtin.shell:
        cmd: "{{ adb_path }} shell settings put global http_proxy
    {{ hostvars[attacker_host_name]['ansible_host'] }}:8090"
      changed_when: true

    - name: "Launch the application"
      ansible.builtin.shell:
        cmd: "{{ adb_path }} shell am start -n com.app.lab.
    rcelab_nowsecure/.MainActivity"
      changed_when: true

    - name: "Clean up the temporary APK file"
```

```
45      ansible.builtin.file:
46        path: "/tmp/{{ apk_name }}"
47        state: absent
```

Listado A.4: Playbook completo de personalización del emulador (`after_clone.yml`)